Справочное руководство Guile

Edition 2.2.3, revision 1, for use with Guile 2.2.3

The Guile Developers

This manual documents Guile version 2.2.3.

Оглавление

Предисловие

В этом руководстве описывается, как использовать Guile, GNU's Ubiquitous Intelligent Language for Extensions. Оно относится, в частности к Guile версии 2.2.3.

Авторы данного руководства

Like Guile itself, the Guile reference manual is a living entity, cared for by many people over a long period of time. As such, it is hard to identify individuals of whom to say "yes, this person, she wrote the manual."

Still, among the many contributions, some caretakers stand out. First among them is Neil Jerram, who has been working on this document for ten years now. Neil's attention both to detail and to the big picture have made a real difference in the understanding of a generation of Guile hackers.

Next we should note Marius Vollmer's effect on this document. Marius maintained Guile during a period in which Guile's API was clarified—put to the fire, so to speak—and he had the good sense to effect the same change on the manual.

Martin Grabmueller made substantial contributions throughout the manual in preparation for the Guile 1.6 release, including filling out a lot of the documentation of Scheme data types, control mechanisms and procedures. In addition, he wrote the documentation for Guile's SRFI modules and modules associated with the Guile REPL.

Ludovic Courtes and Andy Wingo, the Guile maintainers at the time of this writing (late 2010), have also made their dent in the manual, writing documentation for new modules and subsystems in Guile 2.0. They are also responsible for ensuring that the existing text retains its relevance as Guile evolves. См. Раздел 2.6 [Reporting Bugs], страница 13, for more information on reporting problems in this manual.

The content for the first versions of this manual incorporated and was inspired by documents from Aubrey Jaffer, author of the SCM system on which Guile was based, and from Tom Lord, Guile's first maintainer. Although most of this text has been rewritten, all of it was important, and some of the structure remains.

The manual for the first versions of Guile were largely written, edited, and compiled by Mark Galassi and Jim Blandy. In particular, Jim wrote the original tutorial on Guile's data representation and the C API for accessing Guile objects.

Significant portions were also contributed by Thien-Thi Nguyen, Kevin Ryde, Mikael Djurfeldt, Christian Lynbech, Julian Graham, Gary Houston, Tim Pierce, and a few dozen more. You, reader, are most welcome to join their esteemed ranks. Visit Guile's web site at `http://www.gnu.org/software/guile/` to find out how to get involved.

Лицензия Guile

Guile is Free Software. Guile is copyrighted, not public domain, and there are restrictions on its distribution or redistribution, but these restrictions are designed to permit everything a cooperating person would want to do.

- The Guile library (libguile) and supporting files are published under the terms of the GNU Lesser General Public License version 3 or later. See the files `COPYING.LESSER` and `COPYING`.

- The Guile readline module is published under the terms of the GNU General Public License version 3 or later. See the file `COPYING`.
- The manual you're now reading is published under the terms of the GNU Free Documentation License (см. Приложение A [GNU Free Documentation License], страница 893).

C code linking to the Guile library is subject to terms of that library. Basically such code may be published on any terms, provided users can re-link against a new or modified version of Guile.

C code linking to the Guile readline module is subject to the terms of that module. Basically such code must be published on Free terms.

Scheme level code written to be run by Guile (but not derived from Guile itself) is not restricted in any way, and may be published on any terms. We encourage authors to publish on Free terms.

You must be aware there is no warranty whatsoever for Guile. This is described in full in the licenses.

1 Введение

Guile - это реализация языка программирования Scheme. Scheme (`http://schemers.org/`) - это элегантный и концептуально простой диалект Lisp, созданный Guy Steele и Gerald Sussman, а также из-за серии отчетов известных как RnRS (the Revised^n Reports on Scheme).

В отличии от, например, Python или Perl, Scheme не имеет доброжелательного диктатора. Здесь очень много реализаций Scheme, с различными характеристиками и с сообществами и академическим действиями вокруг них, и язык развивается в результате взаимодействия между ними. Особенностью Guile является то, что:

- он легко сочектается с другим кодом, написанным на Си
- он имеет историческую и постоянную связь с проектом GNU
- он подчеркивает интерактивное и инкрементальное программирование
- он фактически поддерживает несколько языков, а не только Scheme.

Следующие несколько разделов объясняют, что мы подразумеваем под этими пунктами. После этого раздела идет информация о том как в может получить и установить Guile, и типографские соглашения, которые мы используем в этом руководстве.

1.1 Guile и Scheme

Guile реализует Scheme, как описано в Revised^5 Report on the Algorithmic Language Scheme (известная как R5RS), обеспечивающая чистые и общие данные и структуры управления. Guile выходит за рамки довольно сурового языка, представленного в R5RS, расширяя его с помощью модульной системы, полного доступа к системным вызовам POSIX, сетевой поддержки, многопоточности, динамического связывания, интерфейса вызова внешних функций, мощной обработкой строк, и множества других функций, необходимых для программирования в реальном мире.

Сообщество Scheme недавно приняло и опубликовало R6RS, последнее дополнение в серии RnRS. R6RS значительно расширяет основной язык Scheme, и стандартизирует многие неосновные функции, которые реализуются, в том числе и в Guile, ранее разными путями. Guile был обновлен с учетом некоторых особенностей R6RS, и скорректировал некотоыре существующие функции, чтобы соответствовать спецификации R6RS, оно это ни коим образом не означает полную реализацию R6RS. См. Раздел 7.6 [R6RS Support], страница 677.

Между R5RS и R6RS, идет процесс SRFI (`http://srfi.schemers.org/`) стандартизации интерфейсов для многих практических задач, таких как многопоточное программирование и многомерные массивы. Guile поддерживает множество SRFI, как подробно описано в Раздел 7.5 [SRFI Support], страница 605.

Резюмируя, что касается отношения к стандартам Scheme, то Guile является реализацией R5RS со многими расширениями, некоторые из которых соответствуют SRFIs или соответствуют части R6RS.

1.2 Объединение с Си кодом

Подбно shell, Guile может запускаться интерактивно читая выражения от пользователя, производя их вычисления, отоброжать результаты — или как интерпретатор сценариев, читающий и исполняющий код Scheme из файла. Guile также прдоставляет библиотеку объектов *libguile*, которая позволяет другим приложениям легко включать полный интерпретатор Scheme. Затем приложение может использоать Guile как язык расширения, чистый и мощный язык настройки, или как многоцелевой "клей(glue)", соединяющий примитивы, предоставляемые приложением. Код Scheme легко вызывать из кода на Си и наоборот, что дает разработчику приложения полный контроль над тем, как и когда вызывать интерпретатор. Приложения могут добавлять новые функции, типы данных, структуры управления, и даже синтаксис для Guile, создавая специфичный языковой домен, адаптированный к задаче, но на основе надежного языкового дизайна.

Этому сочетанию помогают четыре аспекта дизайна и истории Guile. Сначала это то что Guile всегда был ориентирован как язык расширения. Следовательно его Си API всегда был очень важен и был соответствующим образом разработан. Второе и третье, технические моменты — что Guile использует консервативную сборку мусора и что она реализует концепцию Scheme продолжений(continuations) путем копирования и востановления стека Си, но чьим практическим последствием является то, что большинство существующего Си кода можно склеить в Guile как есть, без небохдимости модификации, что бы справиться со странным выполнения Scheme. Последним является модуль системы, который помогает расширениям сосуществовать, не мешая друг другу.

Модульная система Guile позволяет разбить большую программу на управляемые секции с четко определенными интерфейсами между ними. Модули могут содержать смесь интерпретируемого и скомпилированного кода; Guile может использовать статическую или динамическую компоновку для включения скомпилированного кода. Модули также поощряют разработчиков накапливать полезные коллекции подпрограмм для общего распространения; на момент написания этой статьи можно найти интерфесы Emacs, подпрограммы доступа к базам данных, компиляторы, интерфейсы инстурментария GUI и функции клиенты HTTP, среди прочих.

1.3 Guile и проект GNU

Guile был задуман проектом GNU после фантастического успеха Emacs Lisp в качестве языка расширения Emacs. Так же, как Emacs Lisp допускал завершенные и непредвиденные приложения, которые написаны в среде Emacs, идея заключалась в том, что Guile должне сделать тоже самое для других приложений проекта GNU. Эта цель остается актуальной и сегодня.

Идея расширяемости тесно связана с основной задачей проекта GNU: продвижение свободы программного обеспечения. Свобода программного обеспечения означает, что люди, получающие программное обеспечение могут менять или улучшать его по собственному желанию, в том числе способами, о которых даже не подозревали разработчики. Для программ написанных и скомпилированных на языках подобных Си, эта свобода охватывает модификацию и востановление Си кода, но если программа также предоставляет язык расширений, который обычно является гораздо более дру-

жественным, он будет более низким барьером, чтобы пользователь начал создавать свои собственные изменения.

Guile теперь используется проектами GNU, такими как AutoGen, Lilypond, Denemo, Mailutils, TeXmacs и Gnucash, и мы надеемся, что в будущем их будет намного больше.

1.4 Интерактивное Программирование

Несвободное программное обеспечение не заинтересовано в том, чтобы пользователи могли видиеть, как оно работает. Они должны просто принять его или сообщить о проблемах и надеятся, что владельцы исходного кода будут решать их проблему.

Свободное программное обеспечение направлено на то, чтобы работать надежно, как и несвободное ПО, но оно должно также расширять возможности своих пользователей, делая свою работу доступной. Это полезно по многим причинам, включая образование, аудит и усовершенствования, а также проблемы отладки.

Идеальная свободная программная система достигает этого, позволяя заинтересованным пользователям видеть исходный код функций, которые они используют, и следить за этим исходным кодом шаг за шагом, по мере его выполнения. В Emacs, хорошими поимерами этого являются гиперссылки исходного кода в справочной системе и `edebug`. Затем, в качестве бонуса и максимизации возможностей пользователь может провести эксперименты с изменением исходного кода, система должна разрешать изменять часть исходного кода и перезагружать эти изменения в исходную программу, что бы получить немедленный эффект.

Guile разработан для такого рода интерактивного программирования, и это отличает его от многих реализаций Scheme, которые запускают фиксированную программу Scheme как можно быстрее — потому что есть компромисс между производительностью и способностью изменить части уже запущенной программы. Есть более быстрые Schemes чем Guile, но Guile это проект GNU, поэтому он ставит приоритет в отношении концепции свободы программирования и эксперементирования.

1.5 Поддержка множества языков

Начиная с версии 2.0 архитектура Guile поддерживает компиляцию любого языка в его основной виртуальный машинный байт-код, а Scheme это только один из поддерживаемых языков. Другими поддерживаемыми языками являются Emacs Lisp, ECMAScript (обычно известный как Javascript) и Brainfuck, и ведется обсуждение для Lua, Ruby и Python.

Это означает, что пользователи могут программировать в приложениях, которые используют Guile выбирая свой язык, а не язык навязываемый им автором приложения.

1.6 Obtaining and Installing Guile

Guile can be obtained from the main GNU archive site `ftp://ftp.gnu.org` or any of its mirrors. The file will be named guile-*version*.tar.gz. The current version is 2.2.3, so the file you should grab is:

```
ftp://ftp.gnu.org/gnu/guile/guile-2.2.3.tar.gz
```

To unbundle Guile use the instruction

```
zcat guile-2.2.3.tar.gz | tar xvf -
```

which will create a directory called `guile-2.2.3` with all the sources. You can look at the file `INSTALL` for detailed instructions on how to build and install Guile, but you should be able to just do

```
cd guile-2.2.3
./configure
make
make install
```

This will install the Guile executable `guile`, the Guile library `libguile` and various associated header files and support libraries. It will also install the Guile reference manual.

Since this manual frequently refers to the Scheme "standard", also known as R5RS, or the "Revised[5] Report on the Algorithmic Language Scheme", we have included the report in the Guile distribution; see Раздел "Introduction" в *Revised(5) Report on the Algorithmic Language Scheme*. This will also be installed in your info directory.

1.7 Организация данного Руководства

Остальная часть этого руководства состоит из следующих разделов.

Chapter 2: Hello Guile!
: Быстрый тур показывающий, как Guile можно использовать интерактивно и как интерпретатор скриптов, как внедрить Guile в ваше приложение и как писать модули интерпретируемого и компилируемого кода для использования с Guile. Все рассказанное здесь документируется и полностью описывается в последующих частях руководства.

Chapter 3: Hello Scheme!
: Для читателей не знакомых со Scheme, данная глава содержит введение в идеи языка Scheme. Данный материал применим к любой реализации Scheme и поэтому не ссылается ни на что, что связано со спецификой Guile.

Chapter 4: Программирование в Scheme
: Предоставляет обзор программирования на Scheme используя Guile. В нем рассказывается, как вызывать программу `guile` из командной строки и как писать скрипты на Scheme. В нем также представлены расширения, которые Guile предлагает за пределами стандартной Scheme.

Chapter 5: Programming in C
: Предоставляет обзор того, как использовать Guile в Си программах. В нем обсуждаются фундаментальные концепции, которые необходимо понять для доступа к функциям Guile, такие как динамические типы и сборщик мусора. Это объясняется в учебнике как определить новые типы данных и функции для использования в программе Scheme.

Chapter 6: Guile API Reference
: В этой части руководства содержиться описание API Guile по функциональным группам с интерфейсами Scheme и Си, представленными бок о бок.

Chapter 7: Guile Modules
: Описывыает некоторые важные модули, распространяемые как часть дистрибутива Guile, которые расширяют функциональность, предоставляемую ядром Guile Scheme.

Chapter 8: GOOPS
: Описывает GOOPS, объектно-ориентированное расширение Guile, которое предоставляет классы, множественное наследование и общие функции.

1.8 Типографические соглашения

In examples and procedure descriptions and all other places where the evaluation of Scheme expression is shown, we use some notation for denoting the output and evaluation results of expressions.

The symbol '⇒' is used to tell which value is returned by an evaluation:

```
(+ 1 2)
⇒ 3
```

Some procedures produce some output besides returning a value. This is denoted by the symbol '⊣'.

```
(begin (display 1) (newline) 'hooray)
⊣ 1
⇒ hooray
```

As you can see, this code prints '`1`' (denoted by '⊣'), and returns `hooray` (denoted by '⇒').

2 Hello Guile!

В этой главе представлена краткая экскурисия по всем способам использования Guile. Есть дополнительные примеры в каталоге `examples/` дистрибутива Guile. Она также объясняет, как лучше сообщить о любых проблемах, которые вы найдете.

В следующих примерах предполагается что Guile установлен в `/usr/local/`.

2.1 Запуск интерактивной сессии Guile

В простейшей форме, Guile выступает в качестве интерактивного интерпретатора языка программирования Scheme, читает и выполняет выражения Scheme, которые пользователь вводит в терминале. Вот пример взаимодействия между Guile и пользователем; Ввод пользователя осуществляется после приглашения `$` и `scheme@(guile-user)>`:

```
$ guile
scheme@(guile-user)> (+ 1 2 3)                ; add some numbers
$1 = 6
scheme@(guile-user)> (define (factorial n)    ; define a function
                       (if (zero? n) 1 (* n (factorial (- n 1)))))
scheme@(guile-user)> (factorial 20)
$2 = 2432902008176640000
scheme@(guile-user)> (getpwnam "root")        ; look in /etc/passwd
$3 = #("root" "x" 0 0 "root" "/root" "/bin/bash")
scheme@(guile-user)> C-d
$
```

2.2 Запуск скрипта Guile

Подобно AWK, Perl, или любой другой оболочке, Guile может интерпретировать файлы сценариев. Сценарий Guile — это просто файл кода Scheme с некоторой дополнительной информацией в начале, которая сообщает операционной системе как вызвать Guile, а затем говорит Guile как обработать код Scheme.

Вот тривиальный сценарий Guile. См. Раздел 4.3 [Guile Scripting], страница 44, для более подробной информации.

```
#!/usr/local/bin/guile -s
!#
(display "Hello, world!")
(newline)
```

2.3 Встраивание Guile в программу(статическая линковка)

Интерпретатор Guile доступен в виде библиотеки объектов, которая должна быть присоединена(подлинкована_ к приложению, использующему Scheme как язык конфигурации или расширения.

Вот `simple-guile.c`, исходный код программы, которая создаст полноценный интерпрепретатор Guile. В дополнение к обычным функциям, предоставляемыми Guile, он так же будет предлагать функцию `my-hostname`.

```
#include <stdlib.h>
#include <libguile.h>

static SCM
my_hostname (void)
{
  char *s = getenv ("HOSTNAME");
  if (s == NULL)
    return SCM_BOOL_F;
  else
    return scm_from_locale_string (s);
}

static void
inner_main (void *data, int argc, char **argv)
{
  scm_c_define_gsubr ("my-hostname", 0, 0, 0, my_hostname);
  scm_shell (argc, argv);
}

int
main (int argc, char **argv)
{
  scm_boot_guile (argc, argv, inner_main, 0);
  return 0; /* never reached */
}
```

Когда Guile правильно установлен в вашей системе, вышеуказанная программа может быть скомпилирована и связана со статическми библиотеками(слинкована) так:

```
$ gcc -o simple-guile simple-guile.c \
    `pkg-config --cflags --libs guile-2.2`
```

Когда она запускается, она ведет себя точно так же, как программа `guile`, за исключением того, что вы также можете вызвать новую функцию `my-hostname`.

```
$ ./simple-guile
scheme@(guile-user)> (+ 1 2 3)
$1 = 6
scheme@(guile-user)> (my-hostname)
"burns"
```

2.4 Написание расширений Guile

Вы можете связать Guile с вашей программой и сделать Scheme доступным для пользователей вашей программы. Вы также можете связать свою библиотеку с Guile и сделать ее функциональность доступной для всех пользователей Guile.

Библиотека, которая связана с Guile, называется *расширением(extension)*, но на самом деле это просто обычная объектная библиотека.

В следующем примере показано, как написать простое расширение для Guile которое делает функцию `j0` доступной из кода Scheme.

```
#include <math.h>
#include <libguile.h>

SCM
j0_wrapper (SCM x)
{
  return scm_from_double (j0 (scm_to_double (x)));
}

void
init_bessel ()
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
}
```

Этот исходный Си файл должен быть скомпилирован в разделяемую библиотеку. Вот как это делается в GNU/Linux:

```
gcc `pkg-config --cflags guile-2.2` \
  -shared -o libguile-bessel.so -fPIC bessel.c
```

Для создания переносимых разделяемых библиотек, мы рекомендуем использовать GNU Libtool (см. Раздел “Introduction” в *GNU Libtool*).

Разделяемая библиотека может быть загружена в запущенный процесс Guile функцией `load-extension`. Затем функция `j0` становиться доступной для выполнения:

```
$ guile
scheme@(guile-user)> (load-extension "./libguile-bessel" "init_bessel")
scheme@(guile-user)> (j0 2)
$1 = 0.223890779141236
```

Подробнее о том, как установить расширение, см. Раздел 4.7 [Installing Site Packages], страница 61.

2.5 Использование модульной системы Guile

Guile поддерживает разделение программа на модули*modules*. Используя модули, вы можете группировать и управлять содержанием программ, состоящих, в основном, из независимых частей.

Для получения дополнительной информации о модульной системе сверх вводного материала, см. См. Раздел 6.20 [Modules], страница 431.

2.5.1 Использование Модулей

Guile поставляется со множеством полезных модулей, например, для обработки строк или синтаксического анализа командной строки. Кроме того, существует множество модулей Guile написанных другими хакерами Guile, но которые необходимо устанавливать в ручную.

Вот пример интерактивного сеанса, который показывает, как использовать модуль `(ice-9 popen)` который обеспечивает средства связи с другими процессами посред-

ством pipes совместно с модулем `(ice-9 rdelim)` предоставляющим функцию чтения строки `read-line`.

```
$ guile
scheme@(guile-user)> (use-modules (ice-9 popen))
scheme@(guile-user)> (use-modules (ice-9 rdelim))
scheme@(guile-user)> (define p (open-input-pipe "ls -l"))
scheme@(guile-user)> (read-line p)
$1 = "total 30"
scheme@(guile-user)> (read-line p)
$2 = "drwxr-sr-x    2 mgrabmue mgrabmue     1024 Mar 29 19:57 CVS"
```

2.5.2 Написание новых Модулей

Вы можете создавать новые модули с помощью синтаксической формы `define-module`. Все определения следующие за данной формой, будут помещаться в этот модуль, пока не будет размещена новая форма `define-module` обозначающая новый модуль.

Один модуль обычно размещается в один файл, и этот файл устанавливается в том месте, где Guile может автоматически найти его. В следующем сеансе показан простой пример:

```
$ cat /usr/local/share/guile/site/foo/bar.scm

(define-module (foo bar)
  #:export (frob))

(define (frob x) (* 2 x))

$ guile
scheme@(guile-user)> (use-modules (foo bar))
scheme@(guile-user)> (frob 12)
$1 = 24
```

Подробнее о том, как устанавливать модули, см. см. Раздел 4.7 [Installing Site Packages], страница 61.

2.5.3 Вставка расширений в Модуль

В дополнении к коду Scheme вы также можете помещать в модуль объекты, которые определены в Си модулях.

ВЫ сделаете это, написав небольшой файл Scheme, который определяет модуль и вызывает `load-extension` непосредственно из тела модуля.

```
$ cat /usr/local/share/guile/site/math/bessel.scm

(define-module (math bessel)
  #:export (j0))

(load-extension "libguile-bessel" "init_bessel")

$ file /usr/local/lib/guile/2.2/extensions/libguile-bessel.so
... ELF 32-bit LSB shared object ...
$ guile
scheme@(guile-user)> (use-modules (math bessel))
scheme@(guile-user)> (j0 2)
$1 = 0.223890779141236
```

См. Раздел 6.21.4 [Modules and Extensions], страница 452, за дальнейшей информацией.

2.6 Сообщение об Ошибках

О любых проблемах с установкой следует сообщать на `bug-guile@gnu.org`.

Если вы обнаружите ошибку в Guile, пожалуйста сообщите об том разработчикам Guile, чтобы они могли ее исправить. Они так же могут предложить обходные пути, когда вам не удасться применить bug-fix или установить новую версию Guile самостоятельно.

Перед отправкой отчетов об ошибке, пожалуйста ознакомьтесь со следующим списком, который подтвердит что вы действительно нашли ошибку.

- Всякий раз, когда документация и фактическое поведение отличаются, вы наверняка обнаружили ошибку, либо в документации, либо в программе.
- Когда Guile "падает", это ошибка.
- Когда Guile зависает или берет навсегда выполнение задачи, это ошибка.
- Когда вычисления выдают неправильные результаты, это ошибка.
- Когда Guile сообщает об ошибке в корректных программх Scheme, это ошибка.
- Когда Guile не сигнализирует об ошибке для некорректных программ Scheme, это может быть ошибка, если это явно не задокументировано.
- Когда какая-то часть документации не ясна и не имеет для вас смысла даже после повторного прочтения раздела, это ошибка.

Прежде чем сообщать об ошибке, проверьте, загружены ли какие-либо программы в Guile, включая ваш файл `.guile`, устаровлены ли какие либо переменные, способные повлиять на работу Guile. Также, смотрите, возникнет ли проблема в чисто загруженном Guile без загрузки вашего файла `.guile` (старт Guile с переключателем `-q` предотвращает загрузку вашего инициализационного файла). Если проблема не устраняется, тогда вы должны сообщить точное содержание любых программ, которые вы загружаете в Guile, чтобы вызвать проблему.

Когда вы пишете отчет об ошибке, обязательно указывайте как можно больше информации в отчете описанной ниже. Если вы не можете понять некоторые элементы, это не проблема, но чем больше информации мы получаем, тем более вероятно, что мы сможем диагностировать и исправить ошибку.

- Номер версии Guile. Вы можете получить эту инофрмацию при вызове '`guile --version`' в вашей оболочке или вызвав `(version)` изнутри Guile.
- Тип вашей машины, определяемый скриптом оболочки `config.guess`. Если вы проверите Guile, этот файл находиться в директории `build-aux`; иначе вы его можете получить его полную версию по ссылке: `http://git.savannah.gnu.org/gitweb/?p=config.git;a=blob_plain;f=config.guess;hb=HEAD`.

  ```
  $ build-aux/config.guess
  x86_64-unknown-linux-gnu
  ```

- Если вы установили Guile из двоичного пакета, версию этого пакета. В системах которые используют, используйте `rpm -qa | grep guile`. В системах использующих DPKG, `dpkg -l | grep guile`.

- Если вы собирали Guile самостоятельно, то необходима конфигурация которую вы использовали:

  ```
  $ ./config.status --config
  '--enable-error-on-warning' '--disable-deprecated'...
  ```

- Полное описание того как воспроизвести ошибку.

  Если у вас есть программа Scheme которая генериует ошибку, пожалуйста включите ее в отчет об ошибке. Если ваша программа слишком велика, чтобы ее включить, попробуйте уменьшить свой код до минимального тестового примера.

  Если вы можете воспроизвести свою проблему в REPL, это лучше всего. Дайта расшифровку введенных вами выражений в REPL.

- Описание неправильного поведения. Например, "Процесс Guile получает сметрельный сигнал," или, "Результат получается следующим образом, что я считаю не правильным."

  Если появление ошибки является сообщением об ошибке Guile, важно сообщить полный текст сообщения об ошибке и обратную трассировку, показывающую как програмаа Scheme пришла к ошибке. Это можно сделать используя команду `,backtrace` в отладчике Guile.

Если ваша ошибка приводит к собою Guil, дополнительная информация от низко-уровневого отлидчика, такого как GDB может быть полезна. Если вы сами собрали Guile, вы можете запустить Guile под GDB с помощью сценария `meta/gdb-uninstalled-guile` . Вместо вызова Guile как обычно, вызовите сценарий обертку, введите `run` для запуска процесса, затем `backtrace`, когда произойдет сбой. Включите эту трассировку в свой отчет.

3 Hello Scheme!

В этой главе мы введем основные понятия, лежащие в основе элегантности и силы языка Scheme.

Читатели, которые уже владеют базовым знанием Scheme, могут с радостью пропустить данную главу. Однако для читателей, которые являются новичками в этом языке, следующие обсуждения по данным, процедурам, выражениям и замыканиям, предназначенны для обеспечения минимального уровня понимания Scheme, которое более или менее подразумевается в последующих главах.

Стиль этого вводного материала стоит на полпути между точностью R5RS и дискурсивностью существующих обучающих программ Scheme. Мы укажем на полезные ресурсы по Scheme в интернете, см Раздел 3.5 [Further Reading], страница 36.

3.1 Типы данных, Значения и Переменные

В этом разделе обсуждается представление типо данных и значений, что означает для Scheme быть скрыто-типизированным языком, и роль переменных. В заключении Мы заключим введение в синтаксис Scheme определением новых переменных и изменением значений существующей переменной.

3.1.1 Скрытая типизация

Термин *скрытая типизация* используется для описания компьютерного языка, такого как Scheme, дл которого вы не можете, в общем случаее, просто взглянув на исходный код программы, определить, какой тип данных будет связан с конкретной переменной или с результатом конкретного выражения.

Иногда, конечно, вы *можете* указать из кода, какой тип будет у выражения. Если у вас есть строка программы, в которой устанавливается значение переменной `x` в числовое значение равное 1, вы можете быть уверенными, что сразу после выполнения этой строки(при отсутствии многопоточности), `x` примет числовое значение 1. Или если вы пишите процедуру, которая разработана чтобы объединить две строки, вполне вероятно, что остальная часть вашего приложения будет всегда ссылаться на эту процедуру с двумя строковыми параметрами, и вполне вероятно что эта процедура будет работать не правильно, если ее вызывать с параметрами, которые не являются одновременно оба строками.

Тем не менее, дело в том, что в Scheme нет ниего, что регулирует тип параметров процедур, что может потребовать, чтобы параметры процедуры были строками, или чтобы `x` всегда содержало числовое значение, и нет способа объявить в вашей программе, что такие ограничения всегда должны выполняться. В тоже время, нет способа объявить ожидаемый тип возращаемого процедурой значения.

Вместо этого типы переменных и выражений известны — в общем — только во время выполнения. Если вам нужно в какой-то момент проверить, что значение имеет ожидаемый тип, Scheme предоставляет процедуру времени выполнения, которую вы можете использовать для этого. Но в равной степени, это может быть справедливо для двух отдельных вызовов той же процедуры для указания аргументов с разными типами и для возврата значений разных типов.

В следующем подразделе объясняется, что это означает на практике, способы для программ Scheme использовать типы данных, значения и переменные.

3.1.2 Значения и Переменные

Scheme предоставляет множество типов данных, которые можно использовать для представления ваших данных. Примитивные типы, включают символы, строки, числа и процедуры. Составные типы, которые позволяют сгруппировать примитивные типы и составные значения, включают в себя списки, пары, векторы и многомерные массивы. Кроме того, Guile позволяет приложениям определять свои собственные типы данных, имеющие тот же статус, что и встроенные стандартные типы Scheme.

По мере запуска Scheme программ, значения всех типов появляются и исчезают. Иногда значения хранятся в переменных, но чаще они легко передаются как результат из одного вычисления, в качестве одного из параметров, для следующего.

Рассмотрим пример. Строковое значение создается, потому что интерпретатор читает литералы строк из исходного кода вашей программы. Затем в результате создается числовое значение вычисления длины строки. Второее числовое значение создается путем удвоения рассчитанной длины. Наконец, программа создает список с двумя элементами - удвоенной длиной и самой исходной строкой - и сохраняет этот список в программной переменной.

Все используемые здесь значения — фактически, все значения в Scheme – имеют свой тип. Другими словами, каждое значение "знает," во время выполнения, какое значение оно имеет. Число, строка, список или что угодно.

С другой стороны, переменная не имеет фиксированного типа. Переменная `x`, скажем – это просто имя местоположения – поле, в котором вы можете сохранить любое значение Scheme. Та же самая переменная в программе может содержать число в один момент, скисок процедур в следующий момент, а затем пару строк. "Тип" переменной — насколько эта идея имеет смысл — это просто тип любого значения, которое хранит переменная в определенный момент времени.

3.1.3 Определение и Установка Переменных

Чтобы определить новую переменную, используйте синтаксис `define` Scheme следующим образом:

```
(define variable-name value)
```

Оно создает новую переменную с именем *variable-name* и сохраняет в ней значение *value* — начальное значение переменной. Например:

```
;; Make a variable `x' with initial numeric value 1.
(define x 1)

;; Make a variable `organization' with an initial string value.
(define organization "Free Software Foundation")
```

(В Scheme, точка с запятой означает начало коментария, который продолжается до конца линии. Так начинающиеся строки с `;;` являются коментариями.)

Изменение значения уже существующей переменной очень похоже, за исключением того что синтаксис `define` заменяется в Scheme синтаксисом `set!`, вот так:

```
(set! variable-name new-value)
```

Помните, что переменные не имеют фиксиксированных типов, поэтому новое значение *new-value* может иметь полностью другой тип, чем тот который был ранее сохранен в переменной названной *variable-name*. Поэтому оба следующих примера являются правильными.

```
;; Change the value of `x' to 5.
(set! x 5)

;; Change the value of `organization' to the FSF's street number.
(set! organization 545)
```

В этих примерах, *value* и *new-value* являются буквально числовыми и строковыми значениями. В целом, однако, *value* и *new-value* могут быть любыми выражениями Scheme. Хотя мы еще не рассматривали формы, которые могут принимать выражения Scheme (см. Раздел 3.3 [About Expressions], страница 21), вы вероятно, можете догадаться, что делает следующий `set!` например does. . .

```
(set! x (+ x 1))
```

(Примечание: это не полное описание `define` и `set!`, потому что нам нужно ввести некоторые другие аспекты Scheme до того, как недостающие части могут быть заполнены. Если, однако вы уже знакомы со структурой Scheme, вам может понравиться читать эти недостающие части сразу, перескакивая вперед по следующим ссылкам.

- Раздел 3.2.4 [Lambda Alternatives], страница 20, что бы прочитать об альтернативной форме синтаксиса `define`, который можно использовать при определении новых процедур.
- Раздел 6.9.8 [Procedures with Setters], страница 274, чтобы прочитать об альтернативной форме сисинтаксиса `set!`, который помогает с изменением единственного значения в глубине составной структуры данных.)
- См. Раздел 6.12.3 [Internal Definitions], страница 315, чтобы прочитать об использовании `define` другого чем на высшем уровне программы Scheme, включая обсуждение того, как оно работает, чтобы использовать `define` а не `set!` для изменения значения существующей переменной.

3.2 Представление и Использование Процедур

В этом разделе описываются основы использования и создания процедур Scheme. В нем обсуждается предоставление процедур как еще один вид значения Scheme, и показывает, как строиться вызов процедур. Затем мы объясняем, как `лямбда(lambda)` используется для создания новых процедур и в заключении представляем различные сокращенные формы `define`, которые можно использовать, вместо того, чтобы писать явное `lambda` выражение.

3.2.1 Процедуры как Значения

Одно из великих упрощений Scheme заключается в том, что процедура — это еще один тип значения, и эти значения—процедуры могут быть переданы и сохранены в переменных точно также как, например строики и списки. Когда мы говорим о встроенной стандартной процедуре Scheme, такой как `open-input-file`, на самом деле мы имеем в виду, что существует предопределенная переменная верхнего уровня, назы-

ваемая `open-input-file`, значением которой является процедура, которая реализует то что говорит R5RS о том что должна делать функция `open-input-file`.

Обратите внимание, что это сильно отличается от многих диалектов Lisp — включая Emacs Lisp — в котором программа может использовать одно и тоже имя с двумя совершенно отдельными значениями: одно значение идентифицирует функцию Lisp, другое значение идентифицирует переменную Lisp, значение которой не имеет ничего общего с с функцией, которая связана с первой переменной. В этих диалектах, функции и переменные, как гворят, живут в разных пространствах имен(*namespaces*).

С другой стороны, в Scheme, все имена принадлежат единому пространству имен, а переменные которые идентифицируют эти имена, могут содержать любое значение Scheme, включая процедурные значения.

Одним из последствий идеи "процедуры как значения" является то, что если вам не нравиться стандартное имя процедуры Scheme, вы можете изменить его.

Например, `call-with-current-continuation` является очень важной стандартной процедурой Scheme, но она также имеет очень длинное имя! Таким образом, многие программисты используют следующий прием, назначить значение процедуры переменной с более удобным именем, например `call/cc`.

```
(define call/cc call-with-current-continuation)
```

Понятно, как это работает. Определение создает новую перменную `call/cc`, и затем устанавливает ее значение в значение переменной `call-with-current-continuation`; значение последней — это процедура, которая реализует поведение, которое R5RS определеяет в соответствии с именем "call-with-current-continuation". Таким образом `call/cc` сохраняет это значение.

Теперь, когда `call/cc` содержит требуемое значение процедуры, вы можете использовать переменную с именем `call-with-current-continuation` для совершенно другой цели или просто изменить ее значение, чтобы вы получили сообщение об ошибке, если вы случайно используете `call-with-current-continuation` как процедуру в вашей программе, не вызов `call/cc`. Например:

```
(set! call-with-current-continuation "Not a procedure any more!")
```

Или вы могли бы просто оставить `call-with-current-continuation` как было. Это прекрасно для более чем одной переменной удерживать одно и тоже значение процедуры.

3.2.2 Простой Вызов Процедур

Вызов процедур в Scheme записывается следующим образом:

```
(procedure [arg1 [arg2 ...]])
```

В этом выражении, *procedure* может быть любым выражением Scheme, значение которого является процедурой. Чаще всего, однако, процедура — это просто имя переменной, значением которой является процедура.

Например, `string-append` — это стандартная процедура Scheme, задача которой объединить все аргументы, которые, как ожидается, будут строками, в единую строку. Таким образом, выражение

```
(string-append "/home" "/" "andrew")
```

является вызовом процедуры, результатом действия которой будет строковое значение `"/home/andrew"`.

Аналогично, `string-length` является стандартной процедурой Scheme которая возвращает длину передаваемого ей аргумента-строки, поэтому

```
(string-length "abc")
```

является вызовом процедуры, результатом которого является числовое значение 3.

Каждый из параметров в вызове процедуры может быть любым выражением Scheme. Поскольку вызов процедуры сам по себе является выражением, мы можем свести эти два примера вместе, получив:

```
(string-length (string-append "/home" "/" "andrew"))
```

— вызов процедуры, результатом которого является числовое значение 12.

(Вам может быть интересно, что произойдет, если два примера будут объединены другим способом. Если мы это сделаем так, мы создадим синтаксически верное выражение вызова:

```
(string-append "/home" (string-length "abc"))
```

но когда это выражение выполниться, оно вызовет ошибку, так как результат `(string-length "abc")` это числовое значение, а `string-append` не предназанчена для приема числовых значений как одного из типов аргументов.)

3.2.3 Создание и использование новых Процедур

Scheme имеет множество стандартных процедур, и Guile предоставляет все эти возможности с помощью предопределений переменных верхнего уровняр. Все эти стандартные процедуры описаны в следующих главах этого справочного руководства.

Однако, очень скоро, вы захотите создать новые процедуры, которые инкапсулируют функциональные аспекты ваших собственных приложений. Для этого вы можете испльзовать знаменитый синтаксис - лямбда `lambda`.

Например, значением следующего выражения Scheme

```
(lambda (name address) expression ...)
```

является новой процедурой, которая принимает два аргумента `name` и `address`. Поведение новой процедуры определяется последовательностью выражений*expression* в теле определения процедуры. (Как правило, эти выражения *expression* каким-то образом используют аргументы, иначе не было бы смысла передавать их в процедуру) При вызове новая процедура возвращает значение, которое является значением последнего выражения в теле процедуры.

Чтобы сделать пример более конкретным, давайте предположим, что оба аргумента – это две строки, и что целью этой процедуры является формирование объединенной строки, которая включает в себя эти аргументы. Тогда полное лямбда выражение может выглядеть так:

```
(lambda (name address)
  (string-append "Name=" name ":Address=" address))
```

В предыдущем разделе мы отметили, что процедурой при вызове процедуры может быть любое выражени Scheme возвращающее значение типа процедуры. Но это точ-

но то, что возвращает лямбда выражение! Таким образом, мы можем использовать лямбда-выражение непосредственно в месте вызова процедуры, например:

```
((lambda (name address)
   (string-append "Name=" name ":Address=" address))
 "FSF"
 "Cambridge")
```

Это допустимое выражение вызова процедуры и его резальтатом является строка:

```
"Name=FSF:Address=Cambridge"
```

Однако более частым является сохранение значения процедуры в переменной —

```
(define make-combined-string
  (lambda (name address)
    (string-append "Name=" name ":Address=" address)))
```

— и затем использование имени переменной в вызове процедуры:

```
(make-combined-string "FSF" "Cambridge")
```

Что имеет точно такой же результат.

Важно отметить, что процедуры, созданные с использванием `lambda`, имеют точно такой же статус как и стандартные встроенные процедуры Scheme и могут быть вызваны, переданы и сохранены в переменные точно так же.

3.2.4 Альтернативы лямбда(lambda)

Поскольку в программах Scheme так часто возникает необходимость создать процедуру, а затем сохранить ее в переменной есть альтернативная форма синтаксиса `define`, которая позволяет вам это сделать.

Форма выражения `define`

```
(define (name [arg1 [arg2 ...]])
  expression ...)
```

точно эквивалентна более длинной форме:

```
(define name
  (lambda ([arg1 [arg2 ...]])
    expression ...))
```

Так, например, определение `make-combined-string` в предыдущем подразделе может быть написано так:

```
(define (make-combined-string name address)
  (string-append "Name=" name ":Address=" address))
```

Такое определение процедуры создает процедуру, котрая требует точно ожидаемого количества аргументов. Существуют еще две формы лямбда- выражений, которые создают процедуры, которые могут принимать переменное количество аргументов:

```
(lambda (arg1 ... . args) expression ...)

(lambda args expression ...)
```

Соотвествующие формы альтернативного синтаксиса `define` определяют:

```
(define (name arg1 ... . args) expression ...)
```

```
(define (name . args) expression ...)
```

Подробнее о том, как работают эти формы, см. См. Раздел 6.9.1 [Lambda], страница 262.

Для Guile 2.0, Guile предоствляет расширение для определения синтксиса, которое позволяет вам вставлять предыдущее расширение до произвольной глубины. По умолчанию это расширение больше не предоставляется, и вместо этого были перенесены в раздел: Раздел 7.18 [Curried Definitions], страница 756,

(Можно утверждать, что альтернативные формы `define` довольно запутывают, осоебнно для новичков в языке Scheme, поскольку они скрывают как роль `lambda`, так и тот факт, что процедуры - это значения, которые храняться в переменных так же, как и любой другой тип данных. С другой стороны, они очень удобны, они также являются хорошим примером других мощных возможностей Scheme: возможность определить произвольные синтаксические преобразования во время работы, которые могут быть применены для последующего чтения ввода.

3.3 Выражения и Вычисления

До сих пор мы встречали выражения, которые выполняют такие вещи, как выражения `define`, которые создают и инициализируют новые переменные, и мы также говорили о выражениях, которые имеют значение *values*, например значение выражения вызова процедуры:

```
(string-append "/home" "/" "andrew")
```

но мы еще не уточнили, что вызывает выражение, подобное вызову этой процедуры, сокращающееся до значения “value” возвращаемого после вызова процедуры, или как обработка таких выражений относится к выполнению программы Scheme в целом.

В этом разделе разъясняется, что мы подразумеваем под значением выражения, вводя идею Вычисления*evaluation*. В нем обсуждаются побочные эффекты, которые могут быть при вычиселении, объясняется, как вычисляются каждое из различных выражений Scheme и описываются поведение и использование Guile REPL как механизма изучения вычислений. Раздел завершается очень кратким изложением общих синтаксических выражений Scheme.

3.3.1 Вычисление Выражений и Выполнение Программ

В Scheme, процесс выполнения выражения известен как Вычисление (точный перевод Оценка)*evaluation*. Вычисление имеет два вида результата:

- Значение*value* вычисленного выражения
- побочные эффекты*side effects* вычисления, которые состоят из любых эффектов вычисления выражения которые не представлены значением.

Из выражений, которые мы встречали до сих пор, выражения `define` и `set!` имеют сторонние эффекты — создание или изменение переменной — не отражающиеся на значении выражения; выражение `lambda` имеет значение — новая построенная процедура — но не имеет побочных эффектов; процедура выражения вызова, как правило имеет либо значение, либо побочный эффект, либо и то, и другое.

Заманчиво пытаться более интуитивно определить, что мы подразумеваем под значением “value” и сторонним эффектом“side effects”, и какова разница между ними. В целом, это чрезвычайно сложно. Это даже не нужно; вместо этого мы с радостью можем определить поведение программы Scheme указав, как Scheme выполняет программу в целом, а затем описать значения и побочные эффекты вычилслений для каждого типа выражений индивидуально.

Итак, некоторые[1] определения ...

- Программа Scheme состоит из последовательности выражений.
- Интерпретатор Scheme выполняет программу, вычисляя эти выражения по порядку по одному.
- Выражение может быть
  - частью литеральных данных, таких как число `2.3` или строка `"Hello world!"`
  - именем переменной
  - выражением вызова процедуры
  - одним из специальных синтаксических выражений Scheme.

В последующих разделах описывается, как вычисляется каждый из типов выражений.

3.3.1.1 Вычисление Литеральных Данных

Когда вычисляется выражение состоящее из литеральных данных, значение выражения представляет собой просто значение того, что описывает это выражение. Вычисление литерального выражения данных не имеет побочных эффектов.

Так, например,

- значение выражения `"abc"` представляет собой строковое значение `"abc"`
- вычисление выражения `3+4i` является значением комплексноым числом 3 + 4i
- вычисление выражения `#(1 2 3)` представляет собой значение трехэлементный вектор, содержащий числовые значения 1, 2 и 3.

Для любого типа данных, который может быть выражен литералами подобно указанным выше, существует синтаксис записи выражений литеральных данных. Другими словами, то что вам нужно записать в свой код для указания литерального значения этого типа — известно как синтаксис чтения типа данных*read syntax*. Это руководство указывает синтаксис чтения для каждого такого типа данных в разделе, который описывает эти типы данных.

Некоторые типы данных не имеют синтаксиса чтения. Процедуры, например, не могут быть выражены литеральными данными; они должны быть созданы с использованием выражения лямбда `lambda`(см. Раздел 3.2.3 [Creating a Procedure], страница 19) или неявного использования короткой формы `define` (см. Раздел 3.2.4 [Lambda Alternatives], страница 20).

[1] Это определение является приблизительным. Полную и детальную информацию, см. Раздел “Formal syntax and semantics” в *The Revised(5) Report on the Algorithmic Language Scheme*.

3.3.1.2 Вычисление ссылок на переменные

Когда вычисляется выражение, состоящее просто из имени переменной, значение выражения представляет собой значение именованной переменной. Вычисление выражения ссылки переменной не имеет побочных эффектов.

Так, после

```
(define key "Paul Evans")
```

значение выражения `key` это строковое значение `"Paul Evans"`. Если *key* изменить, с помощью:

```
(set! key 3.74)
```

значение выражения `key` будет числовым значением 3.74.

Если нет переменной с указанным именем, вычисление выражения ссылки на переменную сообщит об ошибке.

3.3.1.3 Вычисление выражения Вызова Процедуры

Здесь начинаются интересные вычисления! Как уже отмечалось, выражение вызова процедуры имеет вид:

```
(procedure [arg1 [arg2 ...]])
```

где *procedure* должна быть выражением, значением которого, когда производиться вычисление, является процедурой.

Вычисление выражения вызова процедуры продолжается дальше:

- вычисляются индивидуально выражения *procedure*, *arg1*, *arg2*, и т.д.
- выполняется вызов процедуры, которая является значением вычисления выражения *procedure* со списком значений полученных из вычисления *arg1*, *arg2* и т.д. в качестве параметров.

Для процедуры, определенной в Scheme, “вызов процедуры со списком значений как ее параметров” означает привязку значений к формальным параметрам процедуры, а затем вычислением последовательности выражений составляющих тело процедуры. Значение выражения вызова процедуры — это значение последнего вычисленного выражения в процедуре. Побочные эффекты вызова процедуры – это сочетание побочных эффектов последовательности вычислений выражаний в теле процедуры.

Для встроенных процедур значение и побочные эффекты вызова процедуры лучше всего описаны в документации по этой процедуре.

Обратите внимание, что побочные эффекты вычисления выражения вызова процедуры заключаются не только в побочных эффектах вызова процедуры, но и в любых побочных эффектах предшествующих вычислению выражений *procedure*, *arg1*, *arg2*, и т.д.

Чтобы проилюстрировать это, давайте снова посмотрим на выражение вызова процедуры:

```
(string-length (string-append "/home" "/" "andrew"))
```

В самом внешнем выражении, *procedure* `string-length` и аргументом *arg1* равным `(string-append "/home" "/" "andrew")`.

- Вычисление `string-length`, с переданной в процедуру переменной, позволяет процедуре реализовать поведение для “string-length”.

- Вычисление `(string-append "/home" "/" "andrew")`, которая является вызовом другой процедуры, означает вычисление каждого:
  - `string-append`, выражение дает значение процедуры, которая реализует поведение для “string-append”
  - `"/home"`, выражение которое дает строковое значение `"/home"`
  - `"/"`, выражение дает строковое значение `"/"`
  - `"andrew"`, выражение которое дает строковое значение `"andrew"`

  и затем вызвает значение procedure со списком строковых значений в качестве своих аргументов. Результирующее значение предстваляет собой однострочное значение, которое является конкатенацией всех аргументов, а именно `"/home/andrew"`.

При вычислении внешнего выражения интерпретатор теперь может вызвать значение процедуры, полученное из *procedure* со значением полученным из *arg1* в качестве аргумента. Результирующее значение представляет собой длину строки аргумента, которой является число 12.

3.3.1.4 Вычисление Специальных Синтаксических Выражений

Когда вычисляется выражение вызова процедуры, процедура и все аргументы выражения должны быть вычислены до до вызова процедуры. Специальные синтаксические выражения являются особенными, поскольку они могут манипулировать своими аргументами в невычисленной форме и могут выбирать, следует ли вычислять некоторые или все аргументы выражения.

Зачем это необходимо? Рассмотрим фрагмент программы, который запрашивает у пользователя, удаление файла; а затем удаляет его если пользователь ответил да.

```
(if (string=? (read-answer "Should I delete this file?")
              "yes")
    (delete-file file))
```

Если внешнее выражение `(if ...)` здесь бы было выражением вызова процедуры, выражение `(delete-file file)`, чей побочный эффект состоит в том, чтобы фактически удалить файл, было бы вычислено до вычисления процедуры `if`! Ясно, что это бесполезно — целая позиция в выражении `if` называемая следствием(*consequent*) вычисляется только только тогда, когда условие выражения `if` является истинным(“true”).

Поэтому `if` должен быть специальным синтаксисом, а не процедурой. Другие специальные синтаксисы, с которыми мы уже встречались: `define`, `set!` и `lambda`. `define` и `set!` являются синтаксисами поскольку им надо знать имя*name* переменной, которое задается в качестве первого аргумента в выражениях `define` или `set!`, а не значение этой переменной. `lambda` это синтаксис, потому что не сразу оценивает выражения, определяющие тело процедуры; вместо этого он создает процедуру объект, которая включает эти выражения, чтобы их можно было вычислить в будущем, когда будет вызвана эта процедура.

Правила вычисления каждого специального синтаксического выражения идивидуальны для каждого специального синтаксиса. Для краткого описания стандартного специального синтаксиса см. См. Раздел 3.3.4 [Syntax Summary], страница 26.

3.3.2 Хвостовые вызовы

Scheme "оптимизирует хвостовую рекурсию", что означает, что хвостовые вызовы или рекурсии из определенных контекстов не потребляют пространство стека или других ресурсов и поэтому могут использовать произвольно большие данные или для произвольно больших вычислений. Рассмотрим например,

```
(define (foo n)
  (display n)
  (newline)
  (foo (1+ n)))

(foo 1)
⊣
1
2
3
...
```

`foo` печатает числа бесконечно, начиная с данного n. Она реализуется путем печати n затем рекурсивно вызывает себя для печати $n + 1$ и т.д. Эта рекурсия содержит хвостовой вызов, и в Scheme подобные хвостовые вызовы могут быть сделаны без ограничений.

Или рассмотрим случай когда возвращается значение, из SRFI-1 вариант функции `last` (см. Раздел 7.5.3.3 [SRFI-1 Selectors], страница 609) возвращающей последний элемент списка,

```
(define (my-last lst)
  (if (null? (cdr lst))
      (car lst)
      (my-last (cdr lst))))

(my-last '(1 2 3)) ⇒ 3
```

Если список содержит более одного элемента, `my-last` применяется к `cdr`. Эта рекурсия является хвостовым вызовом, после него нет кода, а возвращаемое значение является возвращаемым значением этого вызова. В Scheme эту функцию можно использовать для аргумента с произвольно длинным списком.

Правильный хвостовой вызов доступен только из определенных контекстов, а именно следующие специальные формы позиций,

- `and` — последнее выражение
- `begin` — последнее выражение
- `case` — последнее выражение в каждом пункте
- `cond` — последнее выражение в каждом пункте, а вызов процедуры `=>` это хвостовой вызов
- `do` — последний результат выражения
- `if` — "истина(true)" и "ложь(false)" выражения `"ног"`
- `lambda` — последнее выражение в теле

- `let`, `let*`, `letrec`, `let-syntax`, `letrec-syntax` — последнее выражение в теле
- `or` — последнее выражение

Следующие основные функции выполняют хвостовой вызов,

- `apply` — хвостовой вызов к данной процедуре
- `call-with-current-continuation` — хвостовой вызов процедуры принимающей новое продолжение(continuation)
- `call-with-values` — хвостовой вызов процедуры получающей значения
- `eval` — хвостовой вызов вычисления формы
- `string-any`, `string-every` — хвостовой вызов предиката на последнем символе(если эта точка достижима)

Выше приведенны основные функции и специальные формы. Хвостовой вызов в других модулях описан в соответствующей документации, например SRFI-1 `any` и `every` (см. Раздел 7.5.3.7 [SRFI-1 Searching], страница 614).

Следует отметить, что существует много мест, которые потенциально могут быть хвостовыми вызовами, например последний вызов в `for-each`, но только те из них гарантированы которые имеют явное описание.

3.3.3 Использование Guile REPL

Если вы запустите Guile без указания конкретной программы для выполнения, Guile входит в стандартный цикл Чтение-Выполнение-Печать(Read Evaluate Print Loop или коротко *REPL*). В этом режиме, Guile неоднократно читает выражения Scheme, которые вводит пользователь, вычисляет их и печатает результирующие значения.

REPL — полезный механизм для изучения описанного поведения вычислений описанного в предыдущем разделе. Если вы набираете `string-append`, например, REPL отвечает `#<primitive-procedure string-append>`, илюстрирующая взаимосвязь между переменными `string-append` и процедурой, хранящейся в этой переменной.

В этом руководстве, обозначение ⇒ используется для обозначения результа вычисления “evaluates to”. Где бы вы не увидели пример формы:

expression
⇒
result

не стесняйтесь попробовать себя, набрав выражение в REPL и проверить, что он выдает ожидаемый результат.

3.3.4 Итоги о Общем Синтаксисе

В этом подразделе перечислены наиболее часто используемые синтаксические выражения Scheme, настолько простые, что как только вы их увидите вы сможете разпознать общий специальный синтакс. Для полного описания каждого из этих синтаксисов дана соответствующая гиперссылка.

`lambda` (см. Раздел 6.9.1 [Lambda], страница 262) используется для создания объектов процедур.

`define` (см. Раздел 6.12.1 [Top Level], страница 312) используется для создания новой переменной и установки ее первоначального значения.

`set!` (см. Раздел 6.12.1 [Top Level], страница 312) используется для изменения значения существующей переменной.

`let`, `let*` и `letrec` (см. Раздел 6.12.2 [Local Bindings], страница 314) создают внутреннюю лексическую среду для вычисления последовательности выражений, в которой задано множество локальных переменных связываемых со множеством значений соответствующего набора выражений. Для введения в окружение(среду), см. См. Раздел 3.4 [About Closure], страница 27.

`begin` (см. Раздел 6.13.1 [begin], страница 317) выполняет последовательность вражений в порядке их следования и возвращаетж последнее выражение. Обратите внимание, что это не тоже самое, что процедура которая возвращает совй последний аргумент, потому что оценка выражения вызова процедуры не гарантирует вычисления аргументов в порядке их следования.

`if` and `cond` (см. Раздел 6.13.2 [Conditionals], страница 318) обеспечивают условное вычисление выражений аргументов в зависимости от того, является ли результатом вычисления условия истина или ложь.

`case` (см. Раздел 6.13.2 [Conditionals], страница 318) обеспечивают условное вычисление выражений аргументов в зависимости от того, совпадает ли переменная условия с одной из заданных групп значений.

`and` (см. Раздел 6.13.3 [and or], страница 320) выполняет последовательность выраженей в порядке их следования, пока не останется никаких выражений, или в одном из них результом будет значение “false”.

`or` (см. Раздел 6.13.3 [and or], страница 320)выполняет последовательность выражений в порядке их следования, пока не останется выражений или в одном из них результатом не будет значение “true”.

3.4 Концепция Замыкания

Концепция замыкания *closure* состоит в том, что выражение лямбда захватывает“captures” свяазнные переменные лексической области в точке, где определяется лямбда выражение. Процедура создания лямбда выражений, может ссылаться или изменять захваченные связанные переменные, но значение этих связанных переменных будет сохраняться между вызовами процедур.

В этом разделе более подробно рассматриваются и анализируются различные аспекты этой идеи.

3.4.1 Имена, Местоположение, Значения и Среды

Ранее мы говорили, что имя переменной в программе Scheme связано с местоположением в котором может храниться любое значение Scheme. (Кстати, термин “vcell” часто используется в кругах Lisp и Scheme в качестве альтернативы местоположению“location”.) Таким образом когда мы говорим о “создании переменной”, фактически мы устанавливаем связь между именем или идентификатором, который используется программным кодом Scheme, и местоположением переменной которое относиться к этому имени. Хотя значение, которое храниться в этом местоположении, может измениться, место,на которое ссылается данное имя, всегда одно и тоже.

Мы можем проилюстрировать это, разбив синтаксис определения `define` на три части: `define`

- создает новое местоположение
- устанавливает связь между этим местоположением и именем, обозначенным как первый аргумент выражения `define`
- сохраняет в этом местоположении, значение, полученное путем вычисления второго аргумента выражения `define`.

Набор ассоциаций между именами и местоположениями называется окружением (средой)*environment*. Когда вы создаете переменную верхнего уровня в программе, используя `define`, ассоциация имя-местоположение для этой переменной добавляется в окружение "верхнего уровня". Кроме того, окружение "верхнего уровня" включает ассоциации имя-местоположение для всех процедур предоставляемых стандартной Scheme.

Также возможно создавать среды, отличные от верхнего уровня, и создавать связанные переменные или ассоциации имя-местоположение в этих средах. Эта способность является ключевым инградиентом в концепции замыкания; В следующем подразделе показывается как это делается.

3.4.2 Локальные Переменные и Окружающая Среда

Мы видели, как создавать переменные верхнего уровня с помощью синтаксиса `define` см. (см. Раздел 3.1.3 [Definition], страница 16). Часто бывает полезно создавать переменные, которые были бы ограничены в своем конексте, как правило, частью тела процедуры. В Scheme это делается с использованием синтаксиса `let`, или его модифицированной формы `let*` или `letrec`. Эти синтаксисы описаны полностью дальше в руководстве см. (см. Раздел 6.12.2 [Local Bindings], страница 314). Здесь наша цель - проилюстрировать их использование, чтобы мы могли видеть, как работают локальные переменные.

Например, следующий код использует локальную перменную `s`, чтобы упростить вычисление площади треугоальника по трем сторонам.

```
(define a 5.3)
(define b 4.7)
(define c 2.8)

(define area
  (let ((s (/ (+ a b c) 2)))
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

Эфектом от выражения `let` является создание новой окружающей среды и в рамках этой среды устанавливается связь межуд именем `s` и новым местоположением, начальное значение которого получено вычислением `(/ (+ a b c) 2)`. Выражения в теле `let`, а именно `(sqrt (* s (- s a) (- s b) (- s c)))`, затем вычисляются в контексте новой окружающей среды и значение последнего вычисленного выражения становяться значением всего выражения `let`, и следовательно, значением всей вычисленной переменной `area`.

3.4.3 Цепочка Окружений

В примере предыдущего подраздела мы скрыли важный момент. Тело выражения `let` в этом примере, относится не только к локальной переменной `s`, но также к переменным верхнего уровня `a`, `b`, `c` и `sqrt`. (`sqrt` стандартная процедура Scheme для вычисления квадратного корня.) Если тело выражения `let` вычислятеся в контексте локальной среды `let`, как вычислитель получает значения этих переменных верхнего уровня?

Ответ заключается в том. что локальная среда, созданная автоматически выражением `let` содержит ссылку на среду его содержащую - в данном случае, среду верхнего уровня — и интерпрететор Scheme автоматически ищет связанные переменные в среде верхнего уровня, если он не находит их в локальном окружении. В более общем плане, каждая среда, за исключением верхнего уровня, имеет ссылку на содержащую ее среду, а интерпретатор осущетствляет обратный поиск по цепочке окружений из локального уровня до верхнего уровня, пока не найдет связанную переменную для требуемого идентификатора или не пройдет всю цепь.

Это описание также определяет, что происходит, когда имеется более одной переменной связанных с одинаковым имеменем. предположим, что (продолжая пример предыдущего подраздела) уже существовала пермеменная верхнего уровня `s` созданная выражением:

```
(define s "Some beans, my lord!")
```

Тогда и среда верхнего уровня, и среда локального `let` будут содержать связанные значения с именем `s`. При вычислении кода внутри тела `let`, интерпретатор смотрит сначала локальную среду `let`, и поэтому находит привязку для `s` созданную синтаксисом `let` Несмотря на то, что среда имеет ссылку на среду верхнего уровня, которая также имеет связанную переменную с именем `s`, интерпретатор не доходт до того чтобы искать ее там. При вычислении кода вне тела `let`, интерпретатор ищет имена переменных в среде верхнего уровня, поэтому имя переменной `s` относится к переменной верхнего уровня.

Говорят, что внутри тела `let`, связывание для переменной `s` в локальной среде скрывает *shadow* связанную переменную `s` в среде верхнего уровня.

3.4.4 Лексическая Область видимости

Правила, которые мы только что описали — это детали того, как Scheme реализует “лексическую область видимости”. В этом подразделе делается краткое введение, чтобы объяснить, что означает лексическая область видимости вообще и представляется пример не лексической области видимости.

“Лексическая область видимости” в целом — это идея о том, что

- идентификатор в определенном месте в программе “всегда” ссылается на одно и тоже местоположение переменной — где “всегда” означает “каждый раз, когда выполняется содержащее его выражение”, и что
- местоположение переменной, на которое она ссылается, может быть определено статическим исследованием контекста исходного кода, в котором отображается этот идентификатор, без необходимости рассматривать поток исполнения программы в целом.

На практике лексическая область видимости является нормой для большинства языков программирования и, вероятно, соответствует тому, что вы интуитивно считаете “нормальным”. Вы даже может быть задаться вопросмо, как может возникнуть иная ситуация и чем она может быть полезна. Поэтому мы продемонстрируем возможную ситуацию иной области видимости и сравним ее с лексической областью видимости. В следующем разделе представлен пример нелексической области видимости и подробно рассматривается как его поведение отличается от соответствующего кода с лексической областью видимости.

3.4.4.1 Пример не Лексической области видимости

Чтобы продемонстрировать, что нелексическая область видимости существует и может быть полезна, мы приводим следующий пример из языка Emacs Lisp, который является языком с “динамической областью видмости”.

```
(defvar currency-abbreviation "USD")

(defun currency-string (units hundredths)
  (concat currency-abbreviation
          (number-to-string units)
          "."
          (number-to-string hundredths)))

(defun french-currency-string (units hundredths)
  (let ((currency-abbreviation "FRF"))
    (currency-string units hundredths)))
```

Здесь нужно сосредоточиться на вопросе: что означает идентификатор `currency-abbreviation` в функции `currency-string`? Ответ в Emacs Lisp заключается в том, что все связанные переменные переходят в единый стек, и `currency-abbreviation` означается как верхняя привязка из этого стека, которая имеет название “currency-abbreviation”. Связывание, которое создается формой `defvar` с значением `"USD"`, имеет значение только в том случае, если ни один операто из кода, который вызвал `currency-string` не переопределит “currency-abbreviation”.

Вторая функция `french-currency-string` работает точно так используя это поведение. Она создает новую привязку для имени “currency-abbreviation” которая переопределяет ту, которая была установлена формой `defvar`.

```
;; Замечание!!! Выполнение Emacs Lisp, а не Scheme!
(french-currency-string 33 44)
⇒
"FRF33.44"
```

Теперь давайте посмотрим на соотвествующий код, выполняющийся в соответствии с лекической областью видимостью в Scheme:

```
(define currency-abbreviation "USD")

(define (currency-string units hundredths)
  (string-append currency-abbreviation
                 (number->string units)
```

```
                    "."
                    (number->string hundredths)))

(define (french-currency-string units hundredths)
  (let ((currency-abbreviation "FRF"))
    (currency-string units hundredths)))
```

Согласно правилам лексической области видимости, имя `currency-abbreviation` в `currency-string` ссылается на местоположение переменной в самом ближайшем окружении к точке кода, который выполняет связываение для `currency-abbreviation`, данное расположение переменной задано на верхнем уровне окружения выражением: `(define currency-abbreviation ...)`.

Поэтому в Scheme, процедура `french-currency-string` не будет работать должным образом. Связываение переменных создаваемое для "currency-abbreviation" является чисто локальным кодом, который формирует тело выражения `let`. Поскольку этот код напрямую не использует имя "currency-abbreviation" нигде, это связывание бессмысленно.

```
(french-currency-string 33 44)
⇒
"USD33.44"
```

Это ставит вопрос, как поведение Emacs Lisp может быть реализовано на Scheme. В общем, это вопрос дизайна, ответ на которй зависит от адресата. В даном случае, лучшим ответом могбы быть тот, что `currency-string` должна принимать необязательный третий аргумент. Этот третий аргумент, если он представлен, интерпретируется как абревиатура валюты, которая отменяет значение по умолчанию.

Можно изменить `french-currency-string` чтобы она изменяла основную `currency-string`, но эти изменения не будут элегантными и устойчивыми к прерываниям, которые могут перевести `currency-abbreviation` в некорректное состояние:

```
(define (french-currency-string units hundredths)
  (set! currency-abbreviation "FRF")
  (let ((result (currency-string units hundredths)))
    (set! currency-abbreviation "USD")
    result))
```

Ключевым моментом здесь является то, что код не создает никакой локальной привязки для идентификатора `currency-abbreviation`, так что все изменения относятся к пермеменной верхнего уровня.

3.4.5 Замыкания

Рассмотрим выражение а `let`, которое не содержит никаких `lambda`s:

```
(let ((s (/ (+ a b c) 2)))
  (sqrt (* s (- s a) (- s b) (- s c))))
```

Когда интерпретатор Scheme вычисляет это, он

- создает новую окружающую среду со ссылкой на среду, которая была до того как ему встретилось выражение `let`

- создает связанную пермеменную для `s` в новом окружении, со значением задаваемым выражением `(/ (+ a b c) 2)`
- вычисляет выражение в теле `let` в контексте нового локального окружения и запоминает значение `V`, последнего выполненого выражения.

  забывает локальное окружение.
- продолжает выполнять выражения, содержащее `let`, используя значение `V` как значение выражения `let`, в контексте содержащей его среды.

После того, как выражение `let` было вычислено, созданная локальная среда просто забывается, и пропадает доступ ко всем связанным в нем переменным. Если один и тот же код будет снова выполнен, он будет повторять те же шаги, создавая вторую новую локальную среду, не с связанную с первой, а затем снова забывая о ней.

Однако, если тело `let` содержит `lambda` выражение, то локальная среда не является забытой. Вместо этого, она становиться связанной с процедурой, созданной `lambda` выражением, и востанавливается каждый раз, когда вызывается эта процедура. В деталях это работает следующим образом...

- Когда интерпретатор Scheme вычисляет `lambda` выражение, чтобы создать процедурный объект, он сохраняет текущую среду как часть определения процедуры.
- Затем, всякий раз, когда процедура вызывается, интерпретатор восстанавливает среду которая храниться в определении процедуры и вычисляет тело процедуры в пределах контекста этой среды.

В результате тело процедуры вычисляется в контексте среды которая была актуальна, когда эта процедура создавалась.

Это и есть то что мы называем замыкание *closure*. В следущих нескольких подразделах представлены примеры, которые исследуют полезность этой концепции.

3.4.6 Пример 1: Генератор последовательности чисел

В этом примере используется замыкание для создания процедуры которая является внутренней для данной процедуры, как локальная переменная, но значение которой сохраняется между вызовами процедур.

```
(define (make-serial-number-generator)
  (let ((current-serial-number 0))
    (lambda ()
      (set! current-serial-number (+ current-serial-number 1))
      current-serial-number)))

(define entry-sn-generator (make-serial-number-generator))

(entry-sn-generator)
⇒
1

(entry-sn-generator)
⇒
2
```

Когда вызывается `make-serial-number-generator`, он создает локальную среду со связанной переменной с именем `current-serial-number` начальное значение которой равно 0, в этой среде создается процедура. Локальная среда храниться в созданном объекте процедуры и поэтому сохраняется на протяжении жизни созданной процедуры.

Каждый раз, когда вызывается созданная процедура, она увеличивает значение связанной переменной `current-serial-number` в захваченной среде, а затем возвращает текущее значение.

Обратите внимание, что `make-serial-number-generator` можно вызвать снова, чтобы создать второй генератор последовательных чисел, который независит от первого. Каждый новый вызов `make-serial-number-generator` создает новую локальную среду `let` и возвращает новый объект процедуры ассоциированный с этой средой.

3.4.7 Пример 2: Общая статическая переменная.

В этом примере используется замыкание, чтобы создать две процедуры `get-balance` и `deposit`, которые обращаются к одной и той же локальной среде, чтобы обеспечить доступ к связанной перменной `balance` определенной внутри это локальной среды. Значение этой связанной переменной сохраняется между вызовами этих процедур.

Обратите внимание, что связанная перемеменная `balance` является частной для этих двух процедур: она на прямую не доступна для любого другого кода. Доступ к ней можно получить косвенно только через `get-balance` или `deposit`, как это показано в процедуре `withdraw`.

```
(define get-balance #f)
(define deposit #f)

(let ((balance 0))
  (set! get-balance
        (lambda ()
          balance))
  (set! deposit
        (lambda (amount)
          (set! balance (+ balance amount))
          balance)))

(define (withdraw amount)
  (deposit (- amount)))

(get-balance)
⇒
0

(deposit 50)
⇒
50
```

```
(withdraw 75)
⇒
-25
```

Важная деталь здесь заключается в том, что должны быть установлены переменные `get-balance` и `deposit` определенные на верхнем уровне с помощью `define`, а затем установлены с помощью `set!` внутри тела `let`. Использование `define` внутри тела `let` работать не будет: это создало бы связанные переменные внутри локальной среды `let`, которая будет недоступной на верхнем уровне.

3.4.8 Пример 3: Проблема замыкания обратного вызова

Часто используемая модель программирования для библиотечного кода — позволить приложению регистрировать функции обратного вызова для библиотеки вызываемые при возникновении определенного события. Для приложения зачастую полезно выполнить несколько таких регистраций, используя одну и туже функцию обратного вызова, для например если несколько похожих событий библиотеки могут обрабатываться с использованием одного и того же кода приложения, но тогда возникает необходимость различать вызовы функций обратного вызова, которые связаны с одной регистрацией обратного вызова, от тех которые связаны с другой регистрацией обратного вызова.

В языках, не имеющих возможности динамически создавать функции, эта проблема обычно решается путем передачи параметра `user_data` при регистрации вызова и включения значения этого параметра в качестве одного из параметров функции обратного вызова. Вот пример объявления с использованием этого решения на Си:

```
typedef void (event_handler_t) (int event_type,
                                 void *user_data);

void register_callback (int event_type,
                        event_handler_t *handler,
                        void *user_data);
```

В Scheme, замыкание может использоваться для достижения такой же функциональности, не используя библиотечный код для хранения пользовательских данных `user-data` для регистрации обратного вызова.

```
;; In the library:

(define (register-callback event-type handler-proc)
  ...)

;; In the application:

(define (make-handler event-type user-data)
  (lambda ()
    ...
    <code referencing event-type and user-data>
    ...))

(register-callback event-type
```

```
(make-handler event-type ...))
```

Что касается библиотеки, `handler-proc` это процедура без аргументов, и все что нужно сделать библиотеке, это вызвать его когда произойдет соответствующее событие. В приложения однако, однако процедура обработчика использует замыкание для захвата среды, которая включает в себя весь контекст, который требуется коду обработчика — `event-type` и `user-data` — для правильной обработки события.

3.4.9 Пример 4: Объектная Ориентация

Замыкание — это захыват среды, содержащей постоянные связанные переменные, внутри определения процедуры или набора связанных процедур. Это довольно похоже на идею в некоторых объектно-ориентированных языках, инкапсуляции набора связанных переменных внутри "объекта", вместе с набором "методов" которые работают с инкапсулированными данными. Следующий пример показывает, как замыкание можно использовать для эмуляции идеи объектов, методов и инкапусуляции в Scheme.

```
(define (make-account)
  (let ((balance 0))
    (define (get-balance)
      balance)
    (define (deposit amount)
      (set! balance (+ balance amount))
      balance)
    (define (withdraw amount)
      (deposit (- amount)))

    (lambda args
      (apply
        (case (car args)
          ((get-balance) get-balance)
          ((deposit) deposit)
          ((withdraw) withdraw)
          (else (error "Invalid method!")))
        (cdr args)))))
```

Каждый вызов `make-account` создает и возвращает новую процедуру, созданную выражением в коде примера, который начинается с: "(lambda args".

```
(define my-account (make-account))

my-account
⇒
#<procedure args>
```

Эта процедура действует как объект учетная запись(СЧЕТ) с методами `get-balance`, `deposit` и `withdraw`. Чтобы применить один из методов к текущему СЧЕТу, вы вызываете процедуру с помощью символа указывая требуемый метод в качестве первого параметра, за которым следуют любые другие параметры, которые требуются этому методу.

```
(my-account 'get-balance)
```

```
⇒
0

(my-account 'withdraw 5)
⇒
-5

(my-account 'deposit 396)
⇒
391

(my-account 'get-balance)
⇒
391
```

Обратите внимание, в этом примере, как текущий баланс, так и вспомогательные процедуры `get-balance`, `deposit` и `withdraw`, используемые для реализации методов работающих с внутренним балансом, храняться в переменных связанных в закрытой локальной среде, захваченной выражением `lambda`, которе создает процедуру-объект — СЧЕТ.

3.5 Что еще почитать

- Веб сайт `http://www.schemers.org/` является хорошей отправной точкой для всех вещей касающихся Scheme.
- Онлайн учебник Dorai Sitaram's по Scheme, *Обучай себе сам Scheme за ограниченной число дней Teach Yourself Scheme in Fixnum Days*, см. `http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html`. Включает хорошее объяснение продолжений(continuations).
- Полный текст *Структура и интерпертация компьютерных программ*, классическое введение в компьютерные науки и язык Scheme от Hal Abelson, Jerry Sussman и Julie Sussman, теперь она доступна онлайн `http://mitpress.mit.edu/sicp/sicp.html`. Этот сайт также содержит учебные материалы, связанные с книгой, и весь исходный код, испльзуемый в книге, в форме подходящей для загрузки и запуска.

4 Programming in Scheme

Основной язык Guile это Scheme, и многого можно добиться простым использованием Guile для написания и запуска программ Scheme — в отличие от необходимости погружаться в код Си. В этой части руководства мы объясняем, как использовать Guile в этом режиме и описываем инструменты которые предоставляет Guile, чтобы помочь вам писать сценарии, отлаживать и упаковывать ваши программы для распространения.

Подробную информацию о переменных, функциях и т.д., которые предоставляет программный интерфейс(API) Guile, см. Глава 6 [API Reference], страница 107.

4.1 Guile реализация Scheme

Основной язык Guile это Scheme, которая описана в серии отчетов известной как *RnRS*. *RnRS* является сокращением от *Revised^n Report on the Algorithmic Language Scheme*. Guile полностью соответствует R5RS (см. Раздел "Introduction" в *R5RS*), и реализует некоторые аспекты R6RS.

Guile также имеет множество расширений, выходящих за рамки этих отчетов. Некоторые из областей, где Guile расширяет R5RS такие:

- Интерактивная система документации Guile
- Guile поддерживает POSIX-совместиоме сетевое программирование
- GOOPS – Guile фреймворк для объектно ориентированного программирования.

4.2 Вызов Guile

Многие функции Guile зависят и могут быть изменены информацией, которую предоставляет пользователь либо до, либо во время запуска Guile. Ниже приводиться описание того, какую информацию предоставлять и как это сделать.

4.2.1 Command-line Options

Здесь мы подробно опишем обработку командной строки Guile. Guile обрабатываепт свои аргументы слева на право, распознавая переключатели описаные ниже. Например, см Раздел 4.3.4 [Scripting Examples], страница 47.

script arg...
`-s` *script arg...*

По умолчанию, Guile будет читать файл с именем в командной строке как скрипт. Любые аргументы командной строки *arg...* следующие за *script* становяться аргументами скрипта; функция `command-line` возвращает список строк вида (*script arg...*).

Возможно называть файла используя начальный дефис, например, `-myfile.scm`. Этом случае имени файла должна предшествовать `-s` чтобы сообщить Guile что файл (скрипт) имеет такое название.

Скрипты читаются и выполняются как исходный код Scheme так же как если бы использовалась функция `load`. После выполнения загруженного скрипта *script*, Guile завершает работу.

`-c` *`expr arg...`*

Вычисляет *expr* как код Scheme, и завершает работу. Любые аргументы командной строки *arg...* следующие за *expr* становяться аргументами командной строки; функция `command-line` возвращает список строк вида (*`guile arg...`*), где *guile* это путь исполняемого файла Guile.

`--` *`arg...`* Запускается в интерактивном режиме, выдавая пользователю приглашение для ввода выражений и вычисляя их. Любые аргументы командной строки *arg...* следующие за `--` становяться аргументами командной строки для интерактивного сеанса; функция `command-line` возвращает список строк вида (*`guile arg...`*), где *guile* это путь к исполняемому файлу Guile.

`-L` *`directory`*

Добавляет *directory* в начало пути загрузки Guile. Данные каталоги используютя для поиска в порядке, указанном в командной строке и перед любыми каталогами из переменной среды `GUILE_LOAD_PATH`. Пути, добавленные здесь, не оказывают эффекта во время исполнения пользовательского файла `.guile`.

`-C` *`directory`*

Подобно `-L`, но корректирует путь загрузки для скомпилированных(*compiled*) файлов.

`-x` *`extension`*

Добавляет расришения *extension* в начало списка загружаемых расширений Guile (см. Раздел 6.18.7 [Load Paths], страница 414). Указанные расширения пробуются в порядке указанном в командной строке, и до расширений загружаемых по умолчанию. Добавленные здесь расширения не действуют во время выполнения пользовательского файла `.guile`.

`-l` *`file`* Загружает исходный код Scheme из *file*, и продолжает обработку командной строки.

`-e` *`function`*

Делает функцию *function точкой входа* скрипта. После загрузки файла скрипта (с `-s`) или вычисляя выражение (с `-c`), применяет функцию *function* к списку, содержащему имя программы и аргументы командной строки —списку предоставляемому функцией `command-line`.

Клюя `-e` может появиться в любом месте списка аргументов, но Guile всегда вызывает функцию *function* как *последнее* действие которое она выполняет. Это странно, но так как вызов скрипта работает в POSIX, опция `-s` всегда должна быть последней в списке.

Функция *function* чаще всего является простым символом, котоырй именует определенную функцию в скрипте. Он также может иметь форму (`@` *`module-name symbol`*), и в этом случае символ ищется в модуле с именем *module-name*.

В качестве сокращения вы можете использовать форму `(symbol ...)`, то есть списко, только символы, котолого не начинаются с `@`. Это эквивалентно (`@` *`module-name`* `main`), где *module-name* имеет вид `(symbol ...)`. См.

Раздел 6.20.2 [Using Guile Modules], страница 432, и Раздел 4.3.4 [Scripting Examples], страница 47.

`-ds` Обрабатывает последний параметр `-s` как если бы она была в этой точке командной строки; загружая скрипт здесь.

Этот переключатель необходим, потому что, хотя механизм вызова сценария POSIX эффективно требует, чтобы опция `-s` появлялась последней, программисту может потребоваться запустить скрипт перед другими действиями, запрошенными в командной строке. Пример, см. Раздел 4.3.4 [Scripting Examples], страница 47.

`\` Читать больше аргументов командной строки, начиная со второй строки файла скрипта. См. Раздел 4.3.2 [The Meta Switch], страница 44.

`--use-srfi=`*`list`*

Опция `--use-srfi` ожидает разделенный запятыми список чисел, какждое из которых представляет модуль SRFI для загрузки в интерпретатор перед выполнением файла скрипта или запуска REPL. Кроме того, идентификаторы функции для загружаемых SRFIs распознаются процедурой `cond-expand` когда эта опция используется.

Вот пример, который загружает модули SRFI-8 ('receive') и SRFI-13 ('string library') перед запуском интерпретатора GUILE:

```
guile --use-srfi=8,13
```

`--debug` Стартует guile с отладочной виртуальной машиной (VM). Использование отладочной VM включает поддержку хуков(hooks) VM, которые необходимы для трассировки, устновки точек прерывания, и точного подсчета количества вызовов при профилировании. Отладочная VM, медленнее обычной VM, приблизительно на десять процентов. См. Раздел 6.26.4.1 [VM Hooks], страница 508, для получения дополнительной информации.

По умолчанию, движок отладочной виртуальной машины VM используется только при входе в интерактивную сессию. При выполнении сценария с `-s` или `-c`, исползуется обычная, более быстрая виртуальная машина VM по умолчанию.

`--no-debug`

Не использовать движок отладочной VM, даже при входе в интерактивный сеанс.

Обратите внимание, что несмотря на название, Guile рабтающая с опцией `--no-debug` *дает* поддержку обычным средствам отладки, таким как распечатка подробной трассировки при ошибке. Единственное отличие от `--debug` отсутствие поддержки VM hooks и средств которые опираются на них (см выше).

`-q` Не загружать файл инициализации `.guile`. Эта опция влияет только при работе в интерактивном режиме; Запуск скриптов не загружает файл `.guile`. См. Раздел 4.4.1 [Init File], страница 51.

`--listen[=`*`p`*`]`

Во время работы программы, прослушивает локальный порт или путь для клиентов REPL. Если *p* начинается с числа, предполагается, что это ло-

кальный порт для прослушивания. Если он начинается с прямой косой черты(слеша), предполагается, что это имя файла домена UNIX сокетов для прослушивания.

Если *p* не задано, по умолчанию используется локальный порт 37146. Если вы посмотрите на него перевернув, это будет почти “Guile”. Если у вас установлен netcat, вы имеете возможность дать команду *`nc localhost 37146`* и получить приглашение Guile. После чего, вы можете запустить Emacs и подсоединиться к процессу; см. Раздел 4.5 [Using Guile in Emacs], страница 60, для получения подробной информации.

> **Примечание:** Открытие порта позволяет любому, кто может подключиться к этому порту, делать все, что может делать Guile, как пользователь запустивший процесс Guile. Не используйте опцию `--listen` на многопользовательских машинах. Конечно, если вы не передаете `--listen` в Guile, порт не будет открыт.
>
> Guile защищается от *HTTP inter-protocol exploitation attack* (`https://en.wikipedia.org/wiki/Inter-protocol_exploitation`), сценария, при котором атакующий может, *через* HTML страницу, заставить веб-браузер отправлять данные на TCP сервер прослушивающий loopback интерфейс или частную сеть. Тем не менее, вам рекомендуется использовать сокеты домена UNI, как в `--listen=/some/local/file`, когда это возможно.

Тем не менее, `--listen` отлично подходит для интерактивной отладки и разработки.

`--auto-compile`
: Автоматическая компиляция исходных файлов (поведение по умолчанию).

`--fresh-auto-compile`
: Обозначить кеш автокомпиляции, как недействительный, вызывая перекомпиляцию.

`--no-auto-compile`
: Отменяет автоматическую компиляцию исходного файла.

`--language=`*`lang`*
: Для оставшихся аргументов командной строки, предполагает, что упомянутые файлы с `-l` и выражения переданные с `-c` являются написаными на языке *lang*. *lang* должно быть одним из языков, поддерживаемых компилятором (см. Раздел 9.4.1 [Compiler Tower], страница 870). При запуске в интерактивном режиме устанавливает язык REPL в *lang* (см. Раздел 4.4 [Using Guile Interactively], страница 51).

  По умолчанияю язык `scheme`; другие интересные значения включают `elisp` (для Emacs Lisp), и `ecmascript`.

  Пример ниже показывает вычисление выражения на Scheme, Emacs Lisp и ECMAScript:

  ```
  guile -c "(apply + '(1 2))"
  ```

```
guile --language=elisp -c "(= (funcall (symbol-function '+) 1 2) 3)"
guile --language=ecmascript -c '(function (x) { return x * x; })(2);'
```

Чтобы загрузить файл, написаный на Scheme, а другой написаный на Emacs Lisp, и затем запустить Scheme REPL, наберите:

```
guile -l foo.scm --language=elisp -l foo.el --language=scheme
```

`-h`, `--help`
: Показывает справку по вызовам Guile, и завершает работу.

`-v`, `--version`
: Показывает текущую версию Guile, и затем завершает работу.

4.2.2 Environment Variables

Окружающая среда(*environment*) это функция(свойство) предоставляемая операционной системой; она состоит из набора переменных с именами и значениями. Каждая переменная называется переменной среды(*environment variable*) (или, иногда, переменной оболочки ("shell variable"); Имена переменных среды чувствительны к регистру и обычно используют только заглавные буквы. Значения это текстовые строки, даже те, которые записаны как числа. (Обратите внимание, что здесь мы имеем в виду имена и значения, которые определены в оболочке операционной системы, из которой вызывается Guile. Это не тоже самое. что среда Scheme определяемая в работающем экземпляре Guile. Для описания среды Scheme, см. см. Раздел 3.4.1 [About Environments], страница 27.)

Как установить переменные среды перед запуском Guile, зависит от операционной системы и особенно от используемой вами оболочки. Например, вот как сказать Guile предоставлять подробные сообщения об устаревших функциях, путем установки `GUILE_WARN_DEPRECATED` используя Bash:

```
$ export GUILE_WARN_DEPRECATED="detailed"
$ guile
```

Или, подробные предупреждения могут быть включены для одного вызова, с помощью:

```
$ env GUILE_WARN_DEPRECATED="detailed" guile
```

Если вы хотите получить или изменить значение переменных среды облочки, которые влияют на поведение Guile во время выполнения из запущенного экземпляра Guile, см. Раздел 7.2.6 [Runtime Environment], страница 540.

Вот переменные окружения, которые влияют на поведение Guile во время выполнения:

`GUILE_AUTO_COMPILE`
: Этот флаг, который может использоваться для указания Guile, компилировать или нет исходные файлы Scheme автоматически. Начиная с Guile 2.0, исходные файлы Scheme будут компилироваться автоматически, по умолчанию.

  Если скомпилированный файл (`.go`) соответствующий файлу `.scm` не найден, или не новее чем файл `.scm`, файл `.scm` будет скомпилирован на лету, и результирующий файл `.go` сохранен. Консультирующее уведомление будет напечатано на консоли.

Скомпилированные файлы будут храниться в каталоге `$XDG_CACHE_HOME/guile/ccache`, где `XDG_CACHE_HOME` по умолчанию это директория `$HOME/.cache`. Этот каталог будет создан, если он до этого не существовал.

Обратите внимание, что механизм зависит от временной отметки файла `.go`, являющейся более новой, чем у файла `.scm`; если файлы `.scm` или `.go` перемещаются полсе установки, следует соблюдать осторожность, чтобы сохранить их оригинальные метки времени.

Установите `GUILE_AUTO_COMPILE` в ноль (0), чтобы предотвратить автоматическую компиляцию файлов Scheme. Установите эту переменную в "fresh", чтобы сообщить Guile о компиляции файлов Scheme, в не зависимости от того, являются ли они новее скомпилированных или нет.

См. Раздел 6.18.5 [Compilation], страница 411.

`GUILE_HISTORY`

Эта переменная именует файл, содержащий историю команд Guile REPL. Вы можете указать другой файл истории, установив эту переменную среды. По умолчанию файл истории это `$HOME/.guile_history`.

`GUILE_INSTALL_LOCALE`

Это флаг, который может использоваться, чтобы сообщить Guile, устанавливать или нет текущую локаль при запуске через вызов `(setlocale LC_ALL "")`[1]. См. Раздел 7.2.13 [Locales], страница 571, для дальнейшей информации о локалях.

Вы можете явно указать, что не хотите устанавливать локаль, установив `GUILE_INSTALL_LOCALE` в `0`, или явно включить ее, установив переменную в `1`.

Обычно, установка текущей локали это правильная вещь. Это позволяет Guile правильно анализировать и печатать строки с не-ASCII символьными знаками. Следовательно по умолчанию эта опция включена.

`GUILE_STACK_SIZE`

Guile в настоящее время имеет ограниченный размер стека для вычислений Scheme. Попытка вызова слишком большого количества вложенных функций будет сигнализировать об ошибке. Это хорошо обнаруживает бесконечную рекурсию, но иногда предел достигается и для обычных вычислений. Эта переменная среды, если задано положительное целое числ, указывает число слотов значений Scheme размещаемых в стеке.

В будущем мы будем внедрять стеки, которые могут расти и уменьшаться, но пока этот хак придется делать вам.

`GUILE_LOAD_COMPILED_PATH`

Эта переменная может использоваться для дополнения пути, по которому ищутся скомпилированные файлы Scheme (`.go`) при загрузке. Его значение должно быть списком каталогов, разделенным двоеточиями. Если он

[1] Переменная среды `GUILE_INSTALL_LOCALE` была проигнорирована в версиях Guile до 2.0.9.

содержит специальный компонент пути `...` (многоточия), то вместо многоточия указывается путь по умолчанию в конце. Результат сохраняется в `%load-compiled-path` (см. Раздел 6.18.7 [Load Paths], страница 414).

Вот пример использования оболочки Bash, которая доббавляет текущий каталог `.`, и относительный каталог `../my-library` к `%load-compiled-path`:

```
$ export GUILE_LOAD_COMPILED_PATH=".:../my-library"
$ guile -c '(display %load-compiled-path) (newline)'
(. ../my-library /usr/local/lib/guile/2.2/ccache)
```

`GUILE_LOAD_PATH`

Эта переменная может быть использована для дополнения пути, по которому Scheme ищет файлы для загрузки. Ее значение должно быть разделенным двоеточиями списком каталогов. Если она содержит специальный компонент пути `...` (многоточие), тогда путь по умолчанию подставляется вместо многоточия, в противном случае путь по умолчанию находитсья в конце. результат сохраняется в `%load-path` (см. Раздел 6.18.7 [Load Paths], страница 414).

Вот пример использования оболочки Bash, который добавляет текущий каталог в `%load-path`, и добавляет относительный каталог `../srfi` в конец:

```
$ env GUILE_LOAD_PATH=".:...:../srfi" \
guile -c '(display %load-path) (newline)'
(. /usr/local/share/guile/2.2 \
/usr/local/share/guile/site/2.2 \
/usr/local/share/guile/site \
/usr/local/share/guile \
../srfi)
```

(Примечание: Разрывы строк, приведенные выше, предназначены только для документирования, и не требуются в реальном примере.)

`GUILE_WARN_DEPRECATED`

По мере развития Guile, некоторые функции будут удалены или заменены новыми. Чтобы помочь пользователям перенести свой код по мере развития, Guile выдаст предупреждения о коде, который исползует функции, которые были отмечены для возможного отключения. `GUILE_WARN_DEPRECATED` может быть установлен в нет“no”, чтобы сказать Guile не отображать эти предупреждающие сообщения, или установлен “detailed”(подробно), чтобы сообщить Guile об отображении более длинных сообщений с описанием предупреждения. См. ⟨undefined⟩ [Deprecation], страница ⟨undefined⟩.

`HOME` Guile использует переменную среды окружения `HOME`, имя вашего домашнего каталога, чтобы найти различные файлы, такие как `.guile` или `.guile_history`.

4.3 Скрипты Guile

Подобно AWK, Perl, или другим оболочкам, Guile может интерпретировать файлы сценариев. Скрипт(Сценарий) Guile это просто файл с кодом Scheme с дополнительной информацией в начале, которая сообщает операционной системе, как вызывать Guile, а затем говорит Guile как обрабатывать код Scheme.

4.3.1 Начало Файла скрипта

Первая строка скрипта Guile должна указывать операционной системе использовать Guile для вычисления скрипта, а затем указать Guile как это сделать. Вот самый простой случай:

- Первые два символа должны быть '`#!`'.

  Операционная система интерпретирует это, так что оставшаяся часть строки является именем исполняемого файла, который может интерпретировать скрипт. Guile, однако, интерпретирует эти символы как начало многострочного комментария, заканчивающегося символьными знаками '`!#`' в своей последней строке. (Это расширение синтаксиса описанного в R5RS, добавлено для поддержки скриптов оболочки(shell).)
- Сразу после этих двух знаков должен идти полный путь к интерпретатору Guile. В большинстве систем он будет '`/usr/local/bin/guile`'.
- Затем должен идти пробел, за которым следует аргумент командной строки для передачи в Guile; это должен быть '`-s`'. Этот переключатель говорит Guile запустить скрипт, вместо того чтобы просить пользователя вводить команды в окне терминала. Есть более сложные вещи, которые можно сделать здесь; см. Раздел 4.3.2 [The Meta Switch], страница 44.
- Далее следует знак новой строки.
- Вторая строка скриптам должна содержать только символьные знаки '`!#`' — просто как верхняя строка файла, но в обратном порядке. Операционная система никогда не читает так далеко, но Guile относится к этому как к концу комментария, начатому в первой строке символьными знаками '`#!`'.
- Если файл исходного кода не является закодированным в ASCII или ISO-8859-1, объявление кодировки, как `coding: utf-8` должно появиться в комментарии где-то в первых пяти строках файла: см Раздел 6.18.8 [Character Encoding of Source Files], страница 416.
- Остальная часть файла должна быть программой Scheme.

Guile читает программу, вычисляет выражения в порядке их появления. Достигнув конца файла Guile завершает работу.

4.3.2 Мета переключатель

Переключатели командной строки Guile позволяют программисту описывать достаточно сложные действия в скриптах. К сожалению, механизм вызова скрипта POSIX допускает только один аргумент, который может появиться в строке '`#!`' после пути к исполняемому файлу Guile, и накладывает произвольные ограничения на длину этого аргумента. Предположим вы написали скрипт, начинающийся так:

```
#!/usr/local/bin/guile -e main -s
```

```
!#
(define (main args)
  (map (lambda (arg) (display arg) (display " "))
       (cdr args))
  (newline))
```

Предполагаемое назначение ясно: грузим файл, и вызываем `main` как аргумент командной строки. Однако система будет относиться ко всему после пути Guile как одному аргументу — строка `"-e main -s"` — это не то что мы хотим.

В качестве обходного пути, мета переключатель \ позволяет программисту Guile определять произвольное количество опций без исправления ядра. Если первым аргументом Guile является \, Guile откроет файл скрипта имя которого следует за аргументом \, и разберет аргументы для старта из файла начиная со второй строки(в соответствии с правилами описанными ниже), и заменит на них переключатель \.

Работая вместе с мета-переключателем, Guile рассматривает символьные знаки '`#!`' как начало комментария, который простирается до следующей строки, содержащей только знаки '`!#`'. Комментарии такого рода могут появиться в любом месте программы Guile, но они наиболее полезны в верхней части файла, волшебным образом объединяясь с механизмом вызова сценариев POSIX.

Итак, рассмотрим скрипт с именем `/u/jimb/ekko` который начинается так:

```
#!/usr/local/bin/guile \
-e main -s
!#
(define (main args)
        (map (lambda (arg) (display arg) (display " "))
             (cdr args))
        (newline))
```

Предположим, что пользователь вызывает этот скрипт следующим образом:

```
$ /u/jimb/ekko a b c
```

Вот что здесь происходит:

- операционная система распознает токен the '`#!`' в верху файла и переписывает командную строку так:

  ```
  /usr/local/bin/guile \ /u/jimb/ekko a b c
  ```

  Это обычное поведение, предписанное POSIX.

- Когда Guile видит первые два аргумента, `\ /u/jimb/ekko`, он открывает `/u/jimb/ekko`, анализирует три аргумента `-e`, `main`, and `-s` из него, и подставляет их вместо переключателя \. Таким образом, командная строка Guile теперь выглядит так:

  ```
  /usr/local/bin/guile -e main -s /u/jimb/ekko a b c
  ```

- Затем Guile обрабатывает ключи: он загружает файл `/u/jimb/ekko` как файл с кодом Scheme (обрабатывая первые три строки как комментраий), и затем выполняет приложение `(main "/u/jimb/ekko" "a" "b" "c")`.

Когда Guile видит метаперключатель \, он анализирует аргументы командной строки из файла скрипта в соответствии со следующими правилами:

- Каждый символ пробела завершает аргумент. Это означает что два пробела подряд вводят аргумент(пустую строку) `""`.
- Знак табуляции не допускается (если вы не заключите его в кавычки, как описано ниже), чтобы избежать путаницы.
- Знак новой строки завершает последовательность аргументов, а также завершает последний не пустой аргумент. (Однако, новая строка после пробела не будет вводить последний аргумент пустой строки; он только завершает список аргументов.)
- Знак обратной косой черты являлется экранирующим(escape) символьным знаком. Он экранирует обратную косую черту, пробел, знак табуляции и новой строки. Экранирующий последовательности ANSI C такие как `\n` и `\t` также поддерживаются. Они дают составляющие аргументов; комбинация из двух знаков `\n` не действует как завершающий перевод строки. Экранируемая последовательность `\`*`NNN`* для ровно трех восмеричных цифр читает символ с ASCII кодом равным *NNN*. Как и выше, знаковый символ созданный таким образом является составляющим аргумента. Обратная косая черта, за которой следуют другие знаки не допускается.

4.3.3 Обработка командной строки

Способность принимать и обрабатывать аргументы командной строки очень важна при написании скриптов Guile для решения определенных задач, таких как извлечение информации из текстовых файлов или взаимодействие с существующими приложениями командной строки. Эта глава описывает, как Guile делает аргументы командной строки доступными для скрипта Guile, и утилит которые Guile предоставляет в помощь в обработке аргументов командной строки.

Когда вызывается скрипт Guile, Guile делает аргументы командной строки доступными через процедуру `command-line`, которая возвращает аргументы в виде списка строк.

Например, если скрипт

```
#! /usr/local/bin/guile -s
!#
(write (command-line))
(newline)
```

сохранить в файл `cmdline-test.scm` и вызывать используя командную строку `./cmdline-test.scm bar.txt -o foo -frumple grob`, вывод будет

```
("./cmdline-test.scm" "bar.txt" "-o" "foo" "-frumple" "grob")
```

Если вызов скрипта включает опцию `-e`, он указывает процедуру вызываемую после загрузки скрипта, Guile вызовет эту процедуру с `(command-line)` в качестве аргументов. Итак, скрипту при исползовании `-e` нет необходимости явно ссылаться на `command-line` в своем коде. Например, скрипт выше будет иметь идентичное поведение, если он будет написан так:

```
#! /usr/local/bin/guile \
-e main -s
!#
```

```
(define (main args)
  (write args)
  (newline))
```

(Обратите внимание на использование мета переключателя \ чтоыб вызов скрипта мог включать более одной опции Guile: См. Раздел 4.3.2 [The Meta Switch], страница 44.)

Эти скрипты испольуют `#!` соглашение POSIX, чтобы они могли быть выполнены с использованием их собственного имени файла напрямую, как в примере командной строки `./cmdline-test.scm bar.txt -o foo -frumple grob`. Но он также может быть выполнен, напечатав подразумеваемую командную строку Guile полностью, как в:

```
$ guile -s ./cmdline-test.scm bar.txt -o foo -frumple grob
```

or

```
$ guile -e main -s ./cmdline-test2.scm bar.txt -o foo -frumple grob
```

Даже когда сценарий вызывается с использованием этой более длинной формы, аргументы этого скрипта получаются так же, как если бы они были вызваны с использованием краткой формы. Guile гарантирует, что `(command-line)` или аргументы `-e` не зависят от того, как вызывается скрипт, путем удаления аргументов, которые Guile обрабатывает сам.

Скрипт может анализировать и обрабатывать аргументы командной строки любым способом, какой он выберет. Там где набор возможных опций и аргументов сложен, он может получить сложное извлечение всех опций, проверить правильность задания аргументов и так далее. Эта задача может быть значительно упрощена использованием модуля `(ice-9 getopt-long)`, который распространяется вместе с Guile, См. Раздел 7.4 [getopt-long], страница 600.

4.3.4 Примеры Скриптов

Для начала приведем несколько примеров прямого вызова Guile:

`guile -- a b c`
: Запуск Guile в интерактивном режиме; `(command-line)` должен вернуть `("/usr/local/bin/guile" "a" "b" "c")`.

`guile -s /u/jimb/ex2 a b c`
: Загрузить файл `/u/jimb/ex2`; `(command-line)` должен вернуть `("/u/jimb/ex2" "a" "b" "c")`.

`guile -c '(write %load-path) (newline)'`
: Записывает значение переменной `%load-path`, печатает новую строку и выходит.

`guile -e main -s /u/jimb/ex4 foo`
: Загружает файл `/u/jimb/ex4`, и затем вызвает функцию `main`, передавая в нее список `("/u/jimb/ex4" "foo")`.

`guile -e '(ex4)' -s /u/jimb/ex4.scm foo`
: Загружает файл `/u/jimb/ex4.scm`, и затем вызывает функцию `main` из модуля '(ex4)', передавая в нее список `("/u/jimb/ex4" "foo")`.

`guile -l first -ds -l last -s script`

Загружает файлы `first`, `script`, и `last`, в указанном порядке. Переключатель `-ds` говорит когда обрабатывать ключ the `-s`. Для более мотивированного примера, смотри скрипты ниже.

Вот очень простой скрипт Guile:

```
#!/usr/local/bin/guile -s
!#
(display "Hello, world!")
(newline)
```

Первая строка помечает файл как скрипт Guile. Когда пользователь вызывает его система запускает интерпретатор скрипта `/usr/local/bin/guile` передавая ему параметры `-s`, с именем файла, и любые другие аргументы переданные сценарию в качестве аргументов командной строки. Когда Guile видит `-s` ***`script`***, он загружает *script*. Таким образом, запуск этой программы приводит к выводу:

```
Hello, world!
```

Вот скрипт, который печатает факториал своего аргумента:

```
#!/usr/local/bin/guile -s
!#
(define (fact n)
  (if (zero? n) 1
    (* n (fact (- n 1)))))

(display (fact (string->number (cadr (command-line)))))
(newline)
```

В действии:

```
$ ./fact 5
120
$
```

Однако, предположим, что мы хотим использовать определение `fact` в этом файле из другого скрипта. Мы не можем просто загрузить(`load`) файл сценария, а затем использовать определение `fact`, потому что сценарий попытается вычислить и отобразить факториал, когда мы его загрузим. Чтобы избежать этой проблемы мы могли бы написать скрипт так:

```
#!/usr/local/bin/guile \
-e main -s
!#
(define (fact n)
  (if (zero? n) 1
    (* n (fact (- n 1)))))

(define (main args)
  (display (fact (string->number (cadr args))))
  (newline))
```

Эта версия упаковывает действия, которые должен выполнять скрипт в функцию `main`. Это позволяет нам загружать файл исключительно ради его определений, без

выполнения каких либо посторонних вычислений. Для чего мы использжуем метаперключатель \ и переключатель указывающий точку входа `-e`, чтобы сообщить Guile что нужно вызвать `main` после загрузки.

```
$ ./fact 50
30414093201713378043612608166064768844377641568960512000000000000
```

Предположим, что теперь мы хотим написать скрипт, который вычисляет функцию `choose`: давая множество m различных объектов, (`choose` n m) это число различных подмножеств содержащих n объектов каждое. Это просто написать `choose` используя данное определение `fact`, поэтому мы можем написать скрипт следующим образом:

```
#!/usr/local/bin/guile \
-l fact -e main -s
!#
(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))

(define (main args)
  (let ((n (string->number (cadr args)))
        (m (string->number (caddr args))))
    (display (choose n m))
    (newline)))
```

Аргументы командной строки здесь говорят Guile сначала загрузить файл `fact`, и затем запустить сценарийt, с точкой входа в `main`. Другими словами, скрипт `choose` может использовать определения в скрипте `fact`. Вот примеры некоторых запусков:

```
$ ./choose 0 4
1
$ ./choose 1 4
4
$ ./choose 2 4
6
$ ./choose 3 4
4
$ ./choose 4 4
1
$ ./choose 50 100
100891344545564193334812497256
```

Чтобы вызвать определенную процедуру из данного модуля, мы можем использовать специальную форму (`@` (*`module`*) *`procedure`*):

```
#!/usr/local/bin/guile \
-l fact -e (@ (fac) main) -s
!#
(define-module (fac)
  #:export (main))

(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))
```

```
(define (main args)
  (let ((n (string->number (cadr args)))
        (m (string->number (caddr args))))
    (display (choose n m))
    (newline)))
```

Мы можем использовать `@@` для вызова неэкспортируемых процедур. Для экспортируемых процедур, мы можем упростить этот вызов с помощью сокращения (*module*):

```
#!/usr/local/bin/guile \
-l fact -e (fac) -s
!#
(define-module (fac)
  #:export (main))

(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))

(define (main args)
  (let ((n (string->number (cadr args)))
        (m (string->number (caddr args))))
    (display (choose n m))
    (newline)))
```

Для максимальной переносимости, мы можем вместо этого использовать оболочку для выполнения `guile` с указанными аргументами командной строки. Здесь нужно позаботиться о правильном квотировании аргументов команды:

```
#!/usr/bin/env sh
exec guile -l fact -e '(@ (fac) main)' -s "$0" "$@"
!#
(define-module (fac)
  #:export (main))

(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))

(define (main args)
  (let ((n (string->number (cadr args)))
        (m (string->number (caddr args))))
    (display (choose n m))
    (newline)))
```

Наконец, опытные сценаристы, вероятно упускают упоминание о подпроцессах. В Bash, например, большинство сценариев оболочки запускают другие программы, такие как `sed` или ему подобные для выполнения реальной работы.

В Guile часто можно сделать все внутри самого Guile, так что пробуйте первое. Но если вам просто необходимо запустить программу и дождаться ее завершения, ис-

пользуйте `system*`. Если вам необходимо запустить подпрограмму и захватить ее выходные данные или вывести их, используйте `open-pipe`. См. Раздел 7.2.7 [Processes], страница 542, и См. Раздел 7.2.10 [Pipes], страница 552, для получения подробной информации.

4.4 Интерактивное Использование Guile

Когда вы запускаете Guile, просто набирая `guile`, без аргумента `-c` или имени скрипта для выполнения, вы получаете интерактивный интерпретатор, где вы можете вводить выражения Scheme, и Guile вычислит их и распечатает результаты для вас. Вот несколько простых примеров.

```
scheme@(guile-user)> (+ 3 4 5)
$1 = 12
scheme@(guile-user)> (display "Hello world!\n")
Hello world!
scheme@(guile-user)> (values 'a 'b)
$2 = a
$3 = b
```

Этот режим использования называется *REPL*, что является сокращением от (Читать-Выполнять-Печатать в Цикле) “Read-Eval-Print Loop”, потому что Guile интерпретатор сначала читает набранное вами выражение, заем вычисляет его, а затем печатает результат.

В подсказке отображается язык и модуль, в котором вы находитесь. В нашем случае, текущий язык это `scheme`, и текущий модуль это `(guile-user)`. См. Раздел 6.24 [Other Languages], страница 482, для получения дополнительной информации о поддерживаемых Guile языках отличных от Scheme.

4.4.1 Файл инициализации, `~/.guile`

При интерактивном запуске, Guile загружает локальный файл инициализации из `~/.guile`. Этот файл должен содержать выражения Scheme для выполнения.

Это средство позволяет пользователю настраивать свою интерактивную среду Guile, добавляя дополнительные модули или параметризацию реализации REPL.

Чтобы запустить Guile без загрузки файла инициализации используйте параметр командной строки `-q`.

4.4.2 Readline

Чтобы проще было повторять и изменять ранее введенные выражения и редактировать набранные выражения в Guile можно использовать библиотеку GNU Readline. Эта библиотека не включена по умолчанию, по причинам лицензирования, но все что вам нужно сделать для активации Readline, это набрать следующую пару строк.

```
scheme@(guile-user)> (use-modules (ice-9 readline))
scheme@(guile-user)> (activate-readline)
```

Рекомендуется поместить эти две строки (без приглашений `scheme@(guile-user)>`) в ваш файл `.guile`. См. Раздел 4.4.1 [Init File], страница 51, для получения дополнительной информации о `.guile`.

4.4.3 История Значений(Value)

Подобно тому как Readline помогает вам повторно использовать предыдущую строку ввода, история значений(*value history*) позволяет вам использовать результаты *result* предедыдущих вычислений в новых выражениях. Когда история значений включена, каждый результат вычисления автоматически присваивается следующей переменной в последовательности переменных `$1`, `$2`, ... Вы можете использовать эти переменные в последующих выражениях.

```
scheme@(guile-user)> (iota 10)
$1 = (0 1 2 3 4 5 6 7 8 9)
scheme@(guile-user)> (apply * (cdr $1))
$2 = 362880
scheme@(guile-user)> (sqrt $2)
$3 = 602.3952191045344
scheme@(guile-user)> (cons $2 $1)
$4 = (362880 0 1 2 3 4 5 6 7 8 9)
```

История значений включена по умолчанию, поскольку Guile REPL импортирует модуль `(ice-9 history)`. История значений может быть отключена или включена в repl, используя опции интерфейса:

```
scheme@(guile-user)> ,option value-history #f
scheme@(guile-user)> 'foo
foo
scheme@(guile-user)> ,option value-history #t
scheme@(guile-user)> 'bar
$5 = bar
```

Обратите внимание, что ранее записанные значения по прежнему доступны, даже если история значений отключена. В редких случаях эти ссылки на прошлые вычисления могут привести к тому, что Guile будет использовать слишком много памяти. Можно очистить эти значения, возможно включив сборку мусора, через процедуру `clear-value-history!`, описанную ниже.

Программный интерфейс для истории значений находится в модуле:

```
(use-modules (ice-9 history))
```

`value-history-enabled?` [Scheme Procedure]

Возвращает истину, если история значений включена, или ложь в противном случае.

`enable-value-history!` [Scheme Procedure]

Включает запись истории значений, если она выключена.

`disable-value-history!` [Scheme Procedure]

Выключает запись истории значений, если она включена.

`clear-value-history!` [Scheme Procedure]

Очищает историю значений. Если сохраненные значения не фиксируются другими структурами данных или замыканиями, они могут быть затем утилизированы сборщиком мусора.

4.4.4 Команды REPL

REPL существует чтобы читать выражения, вычислять их и распечатывать их результы. Но иногда кто-то хочет сказать REPL, что бы он оценивал выражение по другому или сделал что-то вообще. Пользователь может повлиять на работу REPL с помощью команд REPL (*REPL command*).

В предыдущем разделе был приведен пример команды, в виде `,option`.

```
scheme@(guile-user)> ,option value-history #t
```

Команды отличаются от выражений наличием в них начальной запятой (‘`,`’). С запятой не может начинаться выражение в большинстве языков, это эффективный показатель для REPL что следующий текст формирует команду, а не выражение.

Команды REPL удобны, потому что они всегда есть. Даже если текущий модуль не имеет привязки для `pretty-print`, всегда можно сделать `,pretty-print`.

В следующих разделах описаны различные команды, сгруппированные по функциональности. Многие команды имеют сокращения; см. онлайн-справку (`,help`) для получения дополнительной информации.

4.4.4.1 Команды помощи(Help)

Когда Guile запускается в интерактивном режиме, он уведомляет пользователя о том, что справку можно получить набрав ‘`,help`’. Действительно, `help` это команда особенно полезная, поскольку позволяет пользователю обнаружить остальные команды.

`help` [`all` | *group* | [`-c`] *command*] [REPL Command]
Показывает справку(помощь).

С одним аругментом, пытается найти аргумент как имя группы, давая помощь если это группа. В противном случае, пытается найти аргумент в качестве команды, давая помощь если это команда.

Если есть команда, имя которой также является именем группы, используйте фаорму ‘`-c` *command*’ для получения помощи по команде, а не по группе.

Без каких либо аргументов отображается список команд справки и групп команд.

`show` [*topic*] [REPL Command]
Дает информацию о Guile.

С одним аргументом, пытается показать конкретную часть информации; в настоящее время поддеживаются темы: гарантии(‘warranty’) (или ‘w’), "копирование"(‘copying’) (или ‘c’) и версия(‘version’) (или ‘v’).

Без каких либо аргументов отображает список тем.

`apropos` *regexp* [REPL Command]
Ищет привязки/модули/пакеты

`describe` *obj* [REPL Command]
Показывает описание(description)/документацию(documentation).

4.4.4.2 Команды Модулей

`module` [*module*] [REPL Command]
Изменяет текущий модуль / Показывает текущий модуль.

`import` *module . . .* [REPL Command]
Импортирует модуль / Показывает список имротрированных модулей(текущим модулем).

`load` *file* [REPL Command]
Загружает файл в текущий модуль.

`reload` [*module*] [REPL Command]
Перезагружает данный модуль или текущий модуль, если он не был указан.

`binding` [REPL Command]
Список текущих привязок.

`in` *module expression* [REPL Command]
`in` *module command arg . . .* [REPL Command]
Вычисляет выражение, или альтернативно, выполняет другую мета-команду в контексте модуля. Например, '`,in (foo bar) ,binding`' показывает привязки в модуле `(foo bar)`. а так интереснее: '`,in (value-history) ,binding`'

4.4.4.3 Команды Языка

`language` *language* [REPL Command]
Изменяет текущий язык.

4.4.4.4 Команды Компиляции

`compile` *exp* [REPL Command]
Создает скомпилированный код выражения.

`compile-file` *file* [REPL Command]
Компилирует файл.

`expand` *exp* [REPL Command]
Расширяет макрос в форму Scheme(а возможно и другого языка). '`,expand (define-once a 3)`' или '`,expand (when (defined? a) (display "a defined!!!"))`'

`optimize` *exp* [REPL Command]
Запускает оптимизатор на куске кода и распечатывает результат.

`disassemble` *exp* [REPL Command]
Дизассемблирует скомпилированную процедуру. '`(define a (lambda () (display "Test disassm\n"))) ,disassemble a`'

`disassemble-file` *file* [REPL Command]
Дизассемблирует файл.

4.4.4.5 Команды профилирования(измерения скорости работы)

`time` *exp* [REPL Command]
Выдает время выполнения выражения.

`profile` *exp* [*#:hz hz=100*] [*#:count-calls? count-calls?=#f*] [*#:display-style display-style=list*] [REPL Command]
Профилирование выполнения выражения. Эта команда компилирует выражение *exp* и затем запускае его в профилировщике statprof, передавая все ключевые слова в процедуру `statprof`. Подробнее о statprof и доступных опциях для этой команды смотри См. Раздел 7.19 [Statprof], страница 757.

`trace` *exp* [*#:width w*] [*#:max-indent i*] [REPL Command]
Трассировка исполнения

По умолчанию, trace ограничена шириной вашего терминала, или указанием *width* если он определен. Вложенные вызовы процедур будут печататься дальше в право, хотя если ширина отступа соответствует *max-indent*, максимальный отступ сокращается.

4.4.4.6 Команды Отладки

Эти команды отладки доступын только в рамках рекурсивного REPL; они не работают на верхнем уровне.

`backtrace` [*count*] [*#:width w*] [*#:full? f*] [REPL Command]
Печатает обратную трассу(стек вызовов).

Выводит обратную трассу всех кадров стека, или верхние *count* кадров. Если *count* отритцательный, будут показаны последние *count* кадров.

`up` [*count*] [REPL Command]
Выбирает кадр стека вызова.

Выбирает и распечатывает кадры стека, которые вызвали это. Аргумент говорит, сколько кадров надо пройти вверх.

`down` [*count*] [REPL Command]
Выбирает кадр стека вызовов.

Выбирает и распечатывает кадры стека, которые вызвали это. Аргумент говорит, сколько кадров надо пройти вниз.

`frame` [*idx*] [REPL Command]
Показывает кадр.

Показывает выбранный кадр. С аргументом, выбирает кадр по индексу, а затем показывает его.

`locals` [REPL Command]
Показывает локальные переменные.

Показывает локально связанные переменные в выбранном кадре.

`error-message` [REPL Command]
`error` [REPL Command]

Показывает сообщение об ошибке.

Отображает сообщение, связанное с ошибкой с которой началась текущая сессия отладки REPL.

`registers` [REPL Command]

Показывает регистры VM свяазнные с текущим кадром.

См. Раздел 9.3.3 [Stack Layout], страница 846, для получения дополнительной информации о кадрах стека VM.

`width` [*cols*] [REPL Command]

Устанавливает количество отображаемых столбцов в выходных данных `,backtrace` и `,locals` в значение *cols*. Если *cols* не указан, используется ширина терминала.

Следующие 3 команды работают в любом REPL.

`break` *proc* [REPL Command]

Установить точку останова на процедуре *proc*.

`break-at-source` *file line* [REPL Command]

Установить точку останова в указанном местоположении исходного файла.

`tracepoint` *proc* [REPL Command]

Устанавливает точку трассировки для данной процедуры. Это приведет к тому, что все вызовы процедуры распечатают сообщение трассировки. См. Раздел 6.26.4.4 [Tracing Traps], страница 512, для получения дополнительной информации.

Все остальные команды данной подсекции применяются только тогда, когда стек является продолжаемым (*continuable*) — другими словами, когда для программы стек которой получен возможно продолжить работу. Обычно это означает, что программа остановлена из за прерывания или точки останова.

`step` [REPL Command]

Говорит отлаживаемой программе перейти к следующему положению в исходнике.

`next` [REPL Command]

Говорит отлаживаемой программе перейти к следуюещему положению в исходнике в том же кадре стека (См. Раздел 6.26.4 [Traps], страница 507, для получения детальной информации как это работает.)

`finish` [REPL Command]

Говорит отлаживаемой программе, что бы она продолжала работать до завершения текущего кадра стека, после чего распечатать результат и повторнно войти в REPL.

4.4.4.7 Проверяющие(Инспектирующие) Команды

`inspect` *exp* [REPL Command]
Проверяет результат вычисления *exp*.

`pretty-print` *exp* [REPL Command]
Достаточно красиво печатает результат вычисления выражения *exp*.

4.4.4.8 Системные команды

`gc` [REPL Command]
Сборка мусора.

`statistics` [REPL Command]
Отображение статистики.

`option` [*name*] [*exp*] [REPL Command]
Без аргументов, перечисляет все варианты. С одним аргументом, показывает текущее значение опции с именем *name*. С двумя аргументами, устанавливает для опции с именем *name* результат вычисления Scheme выражения *exp*.

`quit` [REPL Command]
Выходит из сессии.

Текущие опции REPL включают:

`compile-options`
Параметры используемые при компиляции введенных выражений в REPL. См. Раздел 6.18.5 [Compilation], страница 411, для подробной информации о опциях компиляции.

`interp` Следует ли интерпретировать или компилировать выражения предоставляемые REPL, если такой выбор доступен. По умолчанию Off (indicating compilation).

`prompt` Настроенное приглашение REPL. `#f` по умолчанию, указывает на приглашение по умолчанию.

`print` Процедура с двумя аргументами используемая для печати результата вычисления каждого выражения. Аргументы это текущий REPL и печатаемое значение. По умолчанию, `#f`, использовать процедуру по умолчанию.

`value-history`
Включена ли история значений или нет. См. Раздел 4.4.3 [Value History], страница 52.

`on-error` Что делать при возникновении ошибки. По умолчанию, `debug`, то есть вход в отладчик. Другие значения включают `backtrace`, для показа трассировки стека без входа в отладчик, или `report`, просто выводит сообщение об ошибке.

Значения по умолчанию для опций REPL могут быть установлены использованием `repl-default-option-set!` из `(system repl common)`:

`repl-default-option-set!` *key value* [Scheme Procedure]
Устанавливает значения по умолчанию опций REPL. Эта функция особенно полезна в файле инициализации пользователя. См. Раздел 4.4.1 [Init File], страница 51.

4.4.5 Обработка ошибок

Когда в коде выполняющимся из REPL обнаруживаеся ошибка, Guile вводит новое приглашение, позволяя вам проверить контекст ошибки.

```
scheme@(guile-user)> (map string-append '("a" "b") '("c" #\d))
ERROR: In procedure string-append:
ERROR: Wrong type (expecting string): #\d
Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
scheme@(guile-user) [1]>
```

Новое приглашение запускается внутри старого, в динамическом контексте ошибки. Это рекурсивный REPL, дополненный усовершенствованным представлением стека, готовым к отладке.

`,backtrace` (abbreviated `,bt`) отображает стек вызовов Scheme в точке, где произошла ошибка:

```
scheme@(guile-user) [1]> ,bt
           1 (map #<procedure string-append _> ("a" "b") ("c" #\d))
           0 (string-append "b" #\d)
```

В приведенном выше примере, обратная трассировка(backtrace) не имеет информации об исходном тексте обоих примитивов `map` и `string-append`. Но в общем случае пространство слева, от backtrace указывает строку и столбец, в которых данная процедура вызывает другую.

Вы можете выйти из рекусрсивного REPL так же как вы выходите из любого REPL: через '`(quit)`', '`,quit`' (сокращенно '`,q`'), или другой вариант *C-d*.

4.4.6 Интерактивная Отладка

Рекурсивный отладочный REPL представляет ряд других метакоманд, которые проверяют состояние вычислений на момент ошибки. Эти команды позовляют вам

- отобразить стек вызовов Scheme в точке, где произошла ошибка;
- перемещаться вверх и вниз по стеку вызовов, чтобы детально увидеть вычисляемое выражение или применяемую процедуру в каждом кадре(*frame*); а также
- изучить значения переменных и выражений в контексте каждого фрейма(кадра стека).

См. Раздел 4.4.4.6 [Debug Commands], страница 55, для документации по отдельным командам. Этот раздел призван дать более детальное описание типичного сеанса отладки.

Первым делом, нам понадобиться "хорошая" ошибка. Давайте попробуем macroexpand - расширить выражение `(unquote foo)`, за пределами формы `quasiquote`, и посмотрим, как макрорасширитель сообщит об этой ошибке.

```
scheme@(guile-user)> (macroexpand '(unquote foo))
ERROR: In procedure macroexpand:
```

```
ERROR: unquote: expression not valid outside of quasiquote in (unquote foo)
Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
scheme@(guile-user) [1]>
```

Команда `backtrace`, которая также может быть вызвана как `bt`, отображает стек вызовов (иначе читается как backtrace) в том месте, где управление процессом вычисления перешло к отладчику:

```
scheme@(guile-user) [1]> ,bt
In ice-9/psyntax.scm:
  1130:21  3 (chi-top (unquote foo) () ((top)) e (eval) (hygiene #))
  1071:30  2 (syntax-type (unquote foo) () ((top)) #f #f (# #) #f)
  1368:28  1 (chi-macro #<procedure de9360 at ice-9/psyntax.scm...> ...)
In unknown file:
           0 (scm-error syntax-error macroexpand "~a: ~a in ~a" # #f)
```

Стек вызовов состоит из последовательности кадров стека(*frames*), причем каждый кадр описывает одну процедуру, которая ждет, чтобы чтото сделать со значениями, возвращаемыми другой процедурой. Здесь мы видим, что в стеке четыре кадра.

Обратите внимание, что `macroexpand` нет в стеке — в действительности он выполняет хвостовой вызов `chi-top`, это можно найти в его определении в `ice-9/psyntax.scm`.

Когда вы входите в отладчик, выбирается самый внутренний кадр, что означает, что команды для получения информации о текущем кадре (“current” frame), или для вычисления выражений в контексте текущего кадра, будут делать это по умолчанию относительно самого внутреннего кадра. Чтобы выбрать другой кадр и применить к нему эти операции, используйте команды: `up`, `down` и `frame` вот так:

```
scheme@(guile-user) [1]> ,up
In ice-9/psyntax.scm:
  1368:28  1 (chi-macro #<procedure de9360 at ice-9/psyntax.scm...> ...)
scheme@(guile-user) [1]> ,frame 3
In ice-9/psyntax.scm:
  1130:21  3 (chi-top (unquote foo) () ((top)) e (eval) (hygiene #))
scheme@(guile-user) [1]> ,down
In ice-9/psyntax.scm:
  1071:30  2 (syntax-type (unquote foo) () ((top)) #f #f (# #) #f)
```

Возможно, нам интересно, что происходит во втором кадре, поэтому мы посмотрим его локальные переменные:

```
scheme@(guile-user) [1]> ,locals
  Local variables:
  $1 = e = (unquote foo)
  $2 = r = ()
  $3 = w = ((top))
  $4 = s = #f
  $5 = rib = #f
  $6 = mod = (hygiene guile-user)
  $7 = for-car? = #f
  $8 = first = unquote
  $9 = ftype = macro
```

```
$10 = fval = #<procedure de9360 at ice-9/psyntax.scm:2817:2 (x)>
$11 = fe = unquote
$12 = fw = ((top))
$13 = fs = #f
$14 = fmod = (hygiene guile-user)
```

Все значения доступны по именам истории значений (`$n`):

```
scheme@(guile-user) [1]> $10
$15 = #<procedure de9360 at ice-9/psyntax.scm:2817:2 (x)>
```

Мы даже можем напрямую вызвать процедуру в REPL:

```
scheme@(guile-user) [1]> ($10 'not-going-to-work)
ERROR: In procedure macroexpand:
ERROR: source expression failed to match any pattern in not-going-to-work
Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
```

И в этой точке, мы вызвали ошибку внутри ошибки. Давайте просто вернемся на верхний уровень(к стеку первой ошибки):

```
scheme@(guile-user) [2]> ,q
scheme@(guile-user) [1]> ,q
scheme@(guile-user)>
```

Наконец: как совет для мудрых, хакеры закрывают свои приглашения REPL с помощью `C-d`.

4.5 Использование Guile в Emacs

Любой текстовый редактор может редактировать код Scheme, но некоторые лучше чем другие. Emacs – конечно лучший, и не только потому, что это хороший текстовый редактор. Emacs хорошо поддерживает Scheme сразу после установки, с разумными правилами отступов, сопоставлением скобок, подсветкой синтаксиса, и даже набором сочетаний горячих клавишь для структурного редактирования, позволяющего осуществлять навигацию, вырезать и вставлять, и выполнять операции транспонирования, которые работают на сбалансированных S-выражениях.

Как бы то нибыло, две вещи значительно улучшат ваш опыт работы с Emacs и Guile.

Первой является Paredit (`http://www.emacswiki.org/emacs/ParEdit`) от Taylor Campbell-a. Вы не должны кодировать на любом диалекте Lisp без Paredit. (Они говорят, что неуверенно, писать скучно, отсюда этот тон, но не зависимо от этого) Paredit это пчела в колене.(фигня какая то, непереводимая игра английских слов)

Второй это Jose Antonio Ortega Ruiz's Geiser (`http://www.nongnu.org/geiser/`). Geiser дополняет режимы Emacs' `scheme-mode` тесной интеграцией с запущенным Guile процессом через а `comint-mode` буфер REPL.

Конечно, есть комбинации горячих клавиш для переключения в REPL, и среда REPL хороша, но Geiser выходит за рамки этого, обеспечивая:

- Вычисление формы в контексте модуля текущего файла.
- Макрорасширение.
- Загрузку файла/модуля и/или его компиляцию.

- Завершение идентификатора с учетом пространства имен(включая локальные привязки, имена, видимые в текущем модуле и имена модулей)
- Autodoc: область эха показывает информацию о сигнатуре процедуры/макроса вокруг текущей точки - автоматически.
- Переход к определению текущего идентификатора.
- Доступ к документации (включая docstrings когда их обеспечивает реализация).
- Списки идентификаторов, экспортируемых данным модулем.
- Список вызывающих/вызываемых процедур.
- Зачаточную поддержку для отладки и отслеживания ошибки.
- Одновременную поддержку нескольких REPL.

См веб страницу Geiser `http://www.nongnu.org/geiser/`, для получения подробной информации.

4.6 Использование Инструментов Guile

Guile также поставляется с растущим числом утилит командной строки: компилятор, дизассемблер, некоторые модули инспекторы, и в будущем, система для установки пакетов Guile из интернета. Эти инструменты могут быть вызваны с помощью программы the `guild`.

```
$ guild compile -o foo.go foo.scm
wrote `foo.go'
```

Эта программа раньше называлась `guile-tools` вплоть до версии Guile 2.0.1, и для обратной совместимости ее еще можно назвать таковой. Однако мы изменили название на `guild`, не только потому что оно приятно короче и легче читается, но и потому, что этот инструмент будет служить чтобы связать мастеров Guile вместе, позволяя хакерам обмениваться кодом друг с другом, используя CPAN-подобную систему.

См. Раздел 6.18.5 [Compilation], страница 411, для получения дополнительной информации о `guild compile`.

Полный список guild скриптов можно получить, вызвав `guild list`, или просто `guild`.

4.7 Установка Местных(Site) Пакетов

В какой-то, момент вы возможно, захотите поделиться своим кодом с другими людьми. Чтобы это сделать, важно следовать ряду общих соглашений, чтобы облегчить пользователям процесс установки и использования вашего пакета.

Первое, что нужно сделать, это установить файлы Scheme, туда где Guile сможет их найти. Когда Guile отправляется на поиск файлов Scheme, он ищет используя путь загрузки(*load path*) для поиска файла: сначала по собственному Guile пути, затем в пути для местных пакетов(*site packages*). Местный пакет это любой код Scheme, который установлен, и не является частью самого Guile. См. Раздел 6.18.7 [Load Paths], страница 414, для получения дополнительной информации о путях загрузки.

Есть несколько местных путей(site paths), по историческим причинам, но тот который обычно должен быть использован можно получить вызвав процедуру `%site-dir`.

См. Раздел 6.23.1 [Build Config], страница 477. Если Guile 2.2 установлен в вашей системе в `/usr/`, то `(%site-dir)` будет `/usr/share/guile/site/2.2`. Файлы Scheme должны быть установлены там.

Если вы не установили скомпилированные `.go` файллы, Guile скомпилирует ваши модули и программы когда они будут впервые использованы, и кеширует их в домашнем каталоге пользователя. См. Раздел 6.18.5 [Compilation], страница 411, для получения дополнительной информации о авто-компиляции. Тем не менее, лучше скомпилировать файлы до их установки, и просто скопировать файлы в то место, где Guile сможет их найти.

Как и в случае с файлами Scheme, Guile ищет путь для поиска скомпилированных файлов `.go` используя путь `%load-compiled-path`. По умолчанию этот путь содержит две записи: путь для Guile файлов и путь для местных(site) пакетов. Вы должны установить ваши файлы `.go` в последний каталог, значение которого возвращается путем вызова процедуры `%site-ccache-dir`. Как и в предыдущем примере, если Guile 2.2 установлен в вашей системе в каталог `/usr/`, тогда `(%site-ccache-dir)` содержащая путь к местным пакетам будет `/usr/lib/guile/2.2/site-ccache`.

Обратите внимание, что файл `.go` будет загружен только вместо файла `.scm` если тот не новее. По этой причине, вы должны сначала установить файлы Scheme, а потом ваши скомпилированные файлы. См. Раздел 6.18.7 [Load Paths], страница 414, для получения дополнительной информации о процессе загрузки.

Наконец, хотя этот раздел посвящен только Scheme, иногда вам необходимо устанавливать и расширения Си. Разделяемые(общие) библиотеки должны быть установлены в директории *extensions dir*. Это значение можно узнать из конфигурации сборки. (см. Раздел 6.23.1 [Build Config], страница 477). Опять же, если Guile 2.2 установлен в вашей системе в `/usr/`, тогда директория для расширений будет `/usr/lib/guile/2.2/extensions`.

5 Программирование на Си

В этой части руководства объясняются общие концепции, которые необходимо понять, для написания программ взаимодействия с Guile из Си. Вы узнаете, как встроена скрытая типизация Scheme в статическую типизацию Си, как сборщик мусора Guile становиться доступным для Си кода и как продолжения (continuations) влияют на поток управления в программе на Си.

Эти знания должны сделать простыми добавление новых функций в Guile, которые могут быть вызваны из Scheme. Добавление новых типов данных также возможно и выполняется путем определения *внешних объектов(foreign objects)*.

Раздел Раздел 5.7 [Programming Overview], страница 90, этой части содержит общие размышления и рекомендации по программированию на Guile. В нем рассматриваются различные способы проектирования программ, связанных с Guile, или то как встраивать Guile в существующие программы.

Для педагогического, но подробного объяснения того, какое представление данных реализует Guile, смж См. Раздел 9.2 [Data Representation], страница 834. Вам не нужно знать подробности, данные там, чтобы использовать Guile из Си, но они полезны, когда вы хотите изменить Guile или когда вам будет интересно узнать, как все это делается.

Подробную справочную информацию о переменных, функциях и т.д. Которые составляют интефрейс прикладного программирования (API) Guile, см. См. Глава 6 [API Reference], страница 107.

5.1 Параллельная(Независимая) инсталяция

Guile обеспечивает надежные гарантии стабильности API и ABI для стабильной версии, так что если пользователь пишет программу для Guile версии 2.2.3, она будет совместима с некоторыми будущими версиями 2.2.7. В этом случае, мы говорим, что 2.2 является эффективной версией(*effective version*), состоящей из главной и второстепенной версии, в этом случае 2 и 2.

Пользователи могут устанавиливать несколько эффективных версий Guile, с заголовками для каждой из них, библиотеками и файлами Scheme в своих собственных каталогах. Это обеспечивает необходимую стабильность гарантий для пользователей, а также позволяет разработчикам Guile развивать язык и его реализацию.

Тем не менее, паралельная инсталяция имеет и скрытую сторону, поскольку пользователи должны знать какую версию Guile необходимо использовать, когда они повторно создают приложения с Guile. Guile решает эту проблему установкой файла для чтения утилитой `pkg-config`, инструмент для опроса установленных пакетов по имени. Guile кодирует версию в имени для pkg-config, чтобы пользователи могли в зависимости от ситуации обращаться к `guile-2.0` или `guile-2.2`.

Например, для эффективной версии 2.2, вы будете вызывать пакет `pkg-config --cflags --libs guile-2.2`, чтобы получить флаги компиляции и связывания. необходимые для ссылки на версию 2.2 Guile. Обычно вы запускаете `pkg-config` на этапе конфигурации вашей программы и используете полученную информацию в Makefile.

Файл `pkg-config` Guile `guile-2.2.pc`, определяет дополнительные полезные переменные:

`sitedir` Каталог по умолчанию, в котором Guile ищет исходны код Scheme и скомпилированные файлы (см. Раздел 4.7 [Installing Site Packages], страница 61). Запустите `pkg-config guile-2.2 --variable=sitedir` чтобы увидет его значение. См. Раздел 5.8.2 [Autoconf Macros], страница 102, для дополнительной информации о том как использовать его с Autoconf.

`extensiondir`
: Каталог по умолчанию, где Guile ищет расширения—т.е., разделяемые библиотеки предоставляющие дополнительрные функции (см. Раздел 6.21.4 [Modules and Extensions], страница 452). Запустите `pkg-config guile-2.2 --variable=extensiondir`, чтобы увидеть его значение.

`guile`
`guild` Абсолютное имя файла `guile` и `guild` commands[1]. запустите `pkg-config guile-2.2 --variable=guile` или `--variable=guild` чтобы увидеть его значение.

Эти переменные позволяют пользователям обрабатывать преобразование имен программ, которые можно указать при настройке Guile с именем `--program-transform-name`, `--program-suffix`, или `--program-prefix` (см. Раздел "Transformation Options" в *GNU Autoconf Manual*).

См справочник man по `pkg-config`, для получения дополнительной информации или веб сайт, `http://pkg-config.freedesktop.org/`. См. Раздел 5.8 [Autoconf Support], страница 102, для получения дополнительной информации о проверке Guile из файла `configure.ac`.

5.2 Связывание программ с Guile

В этом разделе описывается механизм связывания вашей программы с Guile на типичной POSIX системе.

Заголовочный файл `<libguile.h>` содедржит объявления для всех функций и констант Guile. Вы должне его подключить `#include` в начале любого исходного файла Си, который использует описанные в этом руководстве идентификаторы. После того как вы скомпилировали свои исходные файлы, вам нужно связать их с библиотекой объектного кода Guile, `libguile`.

Как отмечалось в предыдущем разделе, `<libguile.h>` не находиться в пути поиска по умолчанию для заголовков. Следующие командные строки дают соответствующие флаги компиляции Си и флаги для связывания необходимые для создания программ с использованием Guile 2.2:

```
pkg-config guile-2.2 --cflags
pkg-config guile-2.2 --libs
```

[1] Переменные `guile` и `guild` определенные начиная с версии Guile 2.0.12.

5.2.1 Функции Инициализации Guile

Чтобы инициализировать Guile, вы можете использовать одну из нескольких функций. Первая, `scm_with_guile`, это самый переносимый способ инициализации Guile, а затем функция, которую вы можете указать. Много поточное приложение может вызвать `scm_with_guile` одновременно и она может быть вызывана более одного раза в данном потоке. Глобальное состояние Guile будет выживать от одного вызова `scm_with_guile` до следующего. Ваша функция вызывается изнутри `scm_with_guile` поскольку сборщик мусора в Guile должен знать где здесь находиться стек каждой нити.

Вторая функция, `scm_init_guile`, инициализирует Guile для текущего потока. Когда она возвращается, вы можете использовать Guile API в текущем потоке. Эта функция использует некую непереносимую магию, чтобы узнать о границе стека и следовательно, может быть не доступна на всех платформах.

Одним из распространных способов использования Guile является написание набора функций Си, которые выполняют несколько полезных задач, сделать их вызываемыми из Scheme, а затем связать программу с Guile. Это делает интерпретатор Scheme как и `guile`, но дополненным дополнительными функциями для некоторых конкретных приложений — специализированного языка сценариев.

В этой ситуации приложение вероятно должно обрабатывать аргументы командной строки, таким же образом какк стандартный интерпретатор Guile. Чтобы сделать это просто, Guile предоставляет функции `scm_boot_guile` и `scm_shell`.

Подробнее об этих функциях, см Раздел 6.4 [Initialization], страница 109.

5.2.2 Пример Guile программы Main

Вот `simple-guile.c`, исходный код для `main` и функции `inner_main` которая будет создавать полный интерпретатор Guile.

```
/* simple-guile.c --- Start Guile from C.  */

#include <libguile.h>

static void
inner_main (void *closure, int argc, char **argv)
{
  /* preparation */
  scm_shell (argc, argv);
  /* after exit */
}

int
main (int argc, char **argv)
{
  scm_boot_guile (argc, argv, inner_main, 0);
  return 0; /* never reached, see inner_main */
}
```

Функция `main` вызывает `scm_boot_guile` для инициализации Guile, передавая ей `inner_main`. Когда `scm_boot_guile` будет готов, он вызывает `inner_main`, который вызывает `scm_shell` для обработки аргументов команднойо строки обычным способом.

5.2.3 Сборка примера с помощью Make

Вот файл Makefile который вы можете использовать для компиляции примера программы. Он использует `pkg-config` чтобы узнать о необходимых флагах компиляции и связывания.

```
# Use GCC, if you have it installed.
CC=gcc

# Tell the C compiler where to find <libguile.h>
CFLAGS=`pkg-config --cflags guile-2.2`

# Tell the linker what libraries to use and where to find them.
LIBS=`pkg-config --libs guile-2.2`

simple-guile: simple-guile.o
        ${CC} simple-guile.o ${LIBS} -o simple-guile

simple-guile.o: simple-guile.c
        ${CC} -c ${CFLAGS} simple-guile.c
```

5.2.4 Сборка примера с использованием Autoconf

Если вы используете пакет GNU Autoconf, чтобы сделать ваше приложение более переносимым, Autoconf автоматически установит многие детали в в Makefile, делая его намного проще и переносимее; Мы рекомендуем использовать Autoconf с Guile. Вот файл `configure.ac` для `simple-guile` который использует стандартный макрос `PKG_CHECK_MODULES` для проверки Guile. Autoconf обработает этот файл в скрипте `configure`. Мы рекомендуем использовать вызов Autoconf через утилиту `autoreconf`.

```
AC_INIT(simple-guile.c)

# Find a C compiler.
AC_PROG_CC

# Check for Guile
PKG_CHECK_MODULES([GUILE], [guile-2.2])

# Generate a Makefile, based on the results.
AC_OUTPUT(Makefile)
```

Запускаем `autoreconf -vif` для создания `configure`.

Вот шаблон `Makefile.in`, из которого скрипт `configure` создаст Makefile настроеный на текущую машину:

```
# The configure script fills in these values.
```

```
CC=@CC@
CFLAGS=@GUILE_CFLAGS@
LIBS=@GUILE_LIBS@

simple-guile: simple-guile.o
        ${CC} simple-guile.o ${LIBS} -o simple-guile
simple-guile.o: simple-guile.c
        ${CC} -c ${CFLAGS} simple-guile.c
```

Разработчик должен использовать Autoconf для генерации скрипта `configure` из шаблона `configure.ac`, и распространять `configure` вместе с приложением. Вот как пользователь может пойти при создании приложения:

```
$ ls
Makefile.in     configure*      configure.ac    simple-guile.c
$ ./configure
checking for gcc... ccache gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether ccache gcc accepts -g... yes
checking for ccache gcc option to accept ISO C89... none needed
checking for pkg-config... /usr/bin/pkg-config
checking pkg-config is at least version 0.9.0... yes
checking for GUILE... yes
configure: creating ./config.status
config.status: creating Makefile
$ make
[...]
$ ./simple-guile
guile> (+ 1 2 3)
6
guile> (getpwnam "jimb")
#("jimb" "83Z7d75W2tyJQ" 4008 10 "Jim Blandy" "/u/jimb"
  "/usr/local/bin/bash")
guile> (exit)
$
```

5.3 Связывание Guile с Библиотеками

Предыдущий раздел кратко объяснил, как писать программы использующие встроенный интерпретатор Guile. Но иногда, все что вы хотите сделать, это создать новую примитивную процедуру или тип данных доступным для программиста Scheme. Написание новой версии `guile` в этом случае неудобно, и это сделает жизнь пользователей, вашей новой функции, без необходимости, трудной.

Например, предположим есть программа `guile-db` которая являлется версией Guile с дополнительной функцией доступа к базам данных. Людям, которые хотят писать программы Scheme для использования этих функций придеся использовать `guile-db` вместо обычной программы `guile`. Теперь предположим, что есть также программа `guile-gtk` которая расширяет Guile доступок к популярному инструментарию Gtk+ для разработки графических интерфейсов. Людям которые хотят писать GUI на Scheme придется использовать `guile-gtk`. Теперь, что произойдет когда вы захотите написать приложение Scheme которое использует GUI чтобы предоставить пользователю доступ к базе данных? Вы должны были бы написать третью(*third*) программу, которая включает в себя как базы данных, так и GUI. Это может быть не легко (поскольку, скажем, `guile-gtk` может быть довольно непонятной программой) и этот пример дополнительно позволяет легко увидеть, что этот подход не может работать на практике.

Было бы намного лучше, если бы функции базы данных, и функции GUI были предоставлены в виде библиотек которые можно просто подсоединить к `guile`. Guile позволяет легко сделать это, и мы рекомендуем вам делать ваши расширения Guile доступными в виде библиотек, всякий раз, когда это возможно.

Вы пишите новые примитивные процедуры и типы данных обычным способом, и связываете их в общую(разделяемую) библиотеку, а не в отдельную программу. Общая библиотека может затем динамически загружаться с помощью Guile.

5.3.1 Пример Guile Расширения

В этом разделе объясняется, как сделать функцию Bessel из библиотеки на Си доступной в Scheme. Сначала нам нужно написать склеивающий код для преобразования аргументов и возвращаемых значений из Scheme в Си и обратно. Кроме того, нам нужна функция которая добавит наши функции к набору примитивов Guile. Поскольку это всего лишь пример, мы будем реализовывать это только для функции `j0`.

Рассмотрим следующий файл `bessel.c`.

```
#include <math.h>
#include <libguile.h>

SCM
j0_wrapper (SCM x)
{
  return scm_from_double (j0 (scm_to_double (x)));
}

void
init_bessel ()
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
}
```

Этот исходный Си файл необходимо скомпилировать в разделяемую библиотеку. Вот как это сделать на GNU/Linux:

```
gcc `pkg-config --cflags guile-2.2` \
  -shared -o libguile-bessel.so -fPIC bessel.c
```

Для создания переносимых разделяемых библиотек, мы рекомендуем использовать GNU Libtool (см. Раздел “Introduction” в *GNU Libtool*).

Разделяемая библиотека может быть загружена в работающий процесс Guile с помощью функции `load-extension`. В дополнении к имени библиотеки для загрузки, эта функция также ожидает имя функции из этой библиотеки, которая будет вызвана для ее инициализации. Для нашего примера, мы собираемся вызвать функцию `init_bessel` которая сделает `j0_wrapper` доступной для программ Scheme с именем `j0`. Обратите внимание, что мы не указываем расширение в имени файла, такое как `.so` при вызове `load-extension`. Правильное расширение для базовой платформы будет предоставлено автоматически.

```
(load-extension "libguile-bessel" "init_bessel")
(j0 2)
⇒ 0.223890779141236
```

Конечно, чтобы это работало, `load-extension` должно быть в состоянии найти `libguile-bessel`. Оно будет искать в тех местах, которые являются обчными для вашей операционной системы, и будет дополнительно просматривать директории перечисленные в переменной среды `LTDL_LIBRARY_PATH`.

Чтобы увидеть, как эти расширения Guile через разделяемые библиотеки относятся к модульной системе, смотрите, См. Раздел 2.5.3 [Putting Extensions into Modules], страница 12.

5.4 Общие понятия для использования libguile

Когда вы хотите встроить интерпретатор Guile Scheme в свою программу или библиотеку, вам необходимо связать их с библиотекой `libguile` (см. Раздел 5.2 [Linking Programs With Guile], страница 64). После того, как вы это сделали, ваш код Си получает доступ к нескольким типам данных и функциям, которые могут использоваться для вызова интерпретатора или создания новых функций, которые вы хотите написать на Си и сделать их доступными для вызова из кода Scheme, среди прочего.

Scheme отличаетс от Си несколькими существенными моментами, и Guile пытается сделать преимущества Scheme доступными для Си. Таким образом, в дополнение к интерпретатору Scheme, libguile ткже предоставляет динамические типы, сборщика мусора, продолжения, арифметику с числами произвольной длины и другие вещи.

Двумя основными понятиями являются динамические типы и сборка мусора. Вам необходимо понять, как libguile предлагает их программам Си, чтобы использовать остальную часть libguile. Также, должен быть рассмотрен общий поток управления Scheme вызыванный продолжениями.

Запуск асинхронных обработчиков сигналов и многопоточность известны Си коду, но есть несколько дополнительных правил при их использовании совместно с libguile.

5.4.1 Динамические типы

Scheme — динамически типизированный язык, это означает. что система вообще не может определить тип данного выражения во время компиляции. Типы становятся известными только во время выполнения. Переменные не имеют фиксированных типов; переменная может содержать пару в один момент времени; целое число в следующий и тысячный элемент позже. Вместо этого значения имеют фиксированные типы, а не переменные.

В порядке реализации стандартных функций Scheme, таких как `pair?` и `string?` и обеспечения сборки мусора, представление каждого значения должно содержать достаточно информации для точного определения его типа во время выполнения. Часто системы Scheme также используют эту информацию для определения, пыталась ли программа применить операцию к неуместному типу значения (например, получения первого элемента `car` от строки(string)).

Поскольку переменные, пары и вектора могут содержать значения любого типа, реализации Scheme используют единое представление для значений — один тип, достаточно большой. чтобы удерживать либо полное значение, либо указатель на полное значение, а также необходимую информацию для информации о типе.

В Guile, данное единое представление всех значений Scheme является Си типом `SCM`. Это непрозрачный тип и его размеры обычно эквивалентны указателю на `void`. Таким образом, значения `SCM` могут эффективно передаваться, и они занимают мало достаточно мало места.

Самое важное правило: Вы никогда не получаеете доступ к значению `SCM` напрямую; Вы передаете его только функции и ли макросу определенным в libguile.

В качестве очевидного примера, хотя переменная `SCM` может содержать целые числа, вы конечно не можете сложить их используя Си оператор `+`. Вы должны использовать функцию libguile `scm_sum`.

Менее очевидно и поэтому важно помнить, что вы также не можете непосредственно проверять значения `SCM` на истинность. В Scheme, значение `#f` считается ложным и конечно, переменная `SCM` может представлять это значения. Но нет никакой гарантии, что `SCM` представление `#f` выглядит ложным и для Си кода. Вам нужно использовать `scm_is_true` или `scm_is_false` для проверки значения `SCM` на истинность или ложность, соответственно.

Вы также не можете на прямую сравнить два значения `SCM`, чтобы выяснить являются ли они идентичными (т.е. являются ли они эквивалентными(`eq?`) в терминах Scheme). Вам необходимо использовать для этого `scm_is_eq`.

Единственное исключение состоит в том, что вы можете напрямую назначить значение `SCM` переменной `SCM` используя Си оператор `=`.

Следующий (надуманный) пример показывает, как сделать это правильно. Он реализует функцию двух аргументов (*a* и *flag*) которая возвращает *a*+1 если *flag* является истиной, иначе она вернет неизменненное *a*.

```
SCM
my_incrementing_function (SCM a, SCM flag)
{
  SCM result;

  if (scm_is_true (flag))
    result = scm_sum (a, scm_from_int (1));
  else
    result = a;

  return result;
}
```

Часто вам нужно преобразовать значения `SCM` в соответствующие значения Си и наоборот. Например, нам нужно преобразовать целое `1` в представление `SCM` чтобы добавить его в *a*. Libguile предоставляет множество функций для этих преобразований, как из Си в `SCM`, таки из `SCM` в Си.

Функции преобразования следуют общей схеме именования: те, которые создают значение `SCM` из значения Си имеют имена в форме `scm_from_`*`type`*` (...)` и те которые конвертируют из занчений `SCM` в значения Си используют форму `scm_to_`*`type`*` (...)`.

Однако лучше избегать преобразования значений, когда можете. Когда вы должны объединить Си и `SCM` значения в вычислении, часто лучше преобразовать Си значения в `SCM` значения и выполнить вычисления с использованием функций libguile, а не наоборот (преобразовывать `SCM` в Си и выполнять вычисления другим способом).

В качестве простого примера рассмотрим эту версию `my_incrementing_function`:

```
SCM
my_other_incrementing_function (SCM a, SCM flag)
{
  int result;

  if (scm_is_true (flag))
    result = scm_to_int (a) + 1;
  else
    result = scm_to_int (a);

  return scm_from_int (result);
}
```

Эта версия гораздо менее общая, чем оригинальная: она будет работать только для значений *A*, которые могуть вписаться в `int`. Оригинальная функция работала для значений, которые может пердставлять Guile и что `scm_sum` может понять, включая целые числа, большие чем длинные, с плавающей запятой числа, комплексные числа и новые числовые типы, которые были добавлены в Guile сторонними библиотеками.

Кроме того, вычисления с помощью `SCM` не обязательно не эффективны. Малые целые числа будут закодированы непосредственно в значение `SCM`, например, и не нуждаются в дополнительной памяти в куче. См. Раздел 9.2 [Data Representation], страница 834, чтобы узнать подробности.

Некоторые специальные значения `SCM` доступны для Си кода без необходимости их преобразования из Си значений:

Scheme value	C representation
`#f`	`SCM_BOOL_F`
`#t`	`SCM_BOOL_T`
`()`	`SCM_EOL`

В дополнении к `SCM`, Guile также определяет родственный тип `scm_t_bits`. Это беззнаковый интегральный тип достаточного размера для хранения всей информации, которая содержиться непосредственно в значении `SCM`. Тип `scm_t_bits` используется внутри Guile все объяснения бит есть в Раздел 9.2 [Data Representation], страница 834, но вы будете сталкиваться с ним иногда в низкоуровневом коде пользователя.

5.4.2 Сборщик Мусора

Как объяснено выше, тип `SCM` может представлять все занчения Scheme. Некоторые значения полностью содержаться в значениях `SCM` (такие как небольшие целые числа), но для других значений требуется дополнительное хранилище в куче(например, строки и векторы). Это дополнительное хранилище управляется автоматически Guile. Вам не нужно явно освобождать его. когда значение `SCM` больше не используется.

Необходимо гарантировать две вещи, чтобы Guile мог управлять хранилищем автоматически: он должен знать обо всех блоках памяти, которые когда либо были выделены для значений Scheme, и он должен знать обо всех значениях Scheme которые еще используются. Учитывая эти знания, Guile может переодически освобождать все блоки, которые были выделены, но не используются никакими активными значениями Scheme. Эта деятельность называется сборкой мусора (*garbage collection*).

Сборщик мусора Guile автоматически обнаружит ссылки на объекты `SCM` которые возникают в глобальных переменных, статических разделах данных, аргументах функции или локальных переменных в стеках Си и Scheme и значения в машинных регистрах. Другие ссылки на объекты `SCM`, такие как в других произвольных структурах в Си куче которая содержит поля типа `SCM`, могут быть видимы сборщику мусора, вызовом функций `scm_gc_protect_object` или `scm_permanent_object`. В совокупности эти знания образуют "корневой набор" сбора мусора; любое значение в куче, которое прямо или косвенно ссылается на члена корневого набора сохраняется, и все остальные объекты имеют право на возврат.

В Guile, сбор мусора имеет две логические фазы: фазу пометки(*mark phase*), в которой сборщик мусора обнаруживает набор всех живых объектов и фазу уборки(*sweep phase*), в которой сборщик мусора возвращает ресурсы системе, связанные с мертвыми объектами. Фаза пометки приостанавливает программу и отслеживает все ссылки на объекты `SCM` начиная с корневого набора. Фаза уборки фактически выполняется вместе с основной программой, поэтапно востанавливая память по мере необходимости.

В фазе пометки, сборщик мусора отслеживает стек Scheme и кучу в точности(*precisely*). Поскольку стек и куча Scheme управляются Guile, Guile может точно знать, где в этих структурах данных сборщик может найти сслылки на другие объекты кучи. Однако это не так, для указателей в Си стеке и статическом сегменте данных Си. Вместо того чтобы требовать от пользователей необоходимости сообщать Guile обо всех переменных в Си, которые могут указывать на объекты кучи, Guile последовательно проверяет Си стек и статический сегмент данных (*conservatively*). Т.е., Guile просто проверяет каждое слово Си стека и любую глобальную переменную Си в качестве потенциальной ссылки на кучу Scheme[2]. Любое значение, которое выглядит как указатель на объект, управляемый сборщиком мусора рассматривается например, является ли оно ссылкой или нет. Таким образом, сканирование стека Си и сегмента статических данных гарантированно найдет все фактические сслыки, но оно также может найти слова которые только случайно выглядят как ссылки. Эти "ложные срабатывания" могут сохранять объекты `SCM` живыми, котоыре иначе бы считались

[2] Обратите внимание, что Guile не сканирует Си кучу для поиска ссылок на объект `SCM` из памяти сегмента, выделенного с помощью `malloc`. необходимо использовать некоторые другие средства для сохранения объекта `SCM` живым. См. Раздел 6.19.1 [Garbage Collection Functions], страница 425.

мертвыми. Хотя это может повлечь потерю памяти, сохранять объект дольше, чем это нужно, это не так страшно. Вот почему эта техника называется “консреативная сборка мусора”. На практике потерянная память, не проблема, поскольку статическое множество Си корней почти вегда конечное и малое, учитывая что стек Scheme отделен от Си стека.

Стек каждого потока сканируется таким образом, а регистры процессора и всех други мест памяти, где могут появиться локальные переменные или параметры функции также включены в это сканирование.

Следствием консервативного сканирования является то, что вы можете просто объявить локальные переменные и параметры функций типа `SCM` и убедиться что сборщик мусора не освободит соответствующие объекты.

Однако локальная переменная или параметр защищены только при условиии, что они находятся реально в стеке(или в каком либо регистре). В качесте оптимизации компилятор Си может повторно использовать их местоположение для другого значения, и объект `SCM` больше не будет защищен от удаления. Как обычно, это приводит к верному поведению: компилятор перезаписывает ссылку, когда она больше не требуется, и таким образом, объект становиться незащищенным, когда ссылка исчезает, что нам и надо.

Однако существуют ситуации, когда объект `SCM` должен существовать дольше, чем ссылка на него из локальной переменной или параметра функции. Это происходит например, когда вы извлекаете некоторый указатель из внешнего объекта и работаете с этим указателем. Ссылка на внешний объект `SCM` может быть мертва после того как указатель был извлечен, но сам указатель(и указатель на память) все еще испльзуется и следовательно внешний объект должен быть защищен. Компилятор не знает об этой связке и может перезаписать ссылка `SCM` слишком рано.

Чтобы обойти эту проблему, вы можете использовать `scm_remember_upto_here_1` и его аналоги. Это заставить сохранить ссылку. См. Раздел 5.5.4 [Foreign Object Memory Management], страница 82.

5.4.3 Управление потоком выполнения программы

Scheme имеет более общий вид потока выполнения программы, чем Си, как локально, так и не локально.

Управление локальным потоком выполнения включает в себя такие вещи, как переходы(gotos), циклы(loops), вызовы функций(calling) и возвраты из них(returning). Нелокальный поток выполнения относится к ситуациям, в которых программа перескакивает через один или несколько уровней активации функций без использования обычных операций вызова(call) или возврата(return).

Примитивным средством Си для локального управления потоком выполнения является операторы `goto` вместе с `if`. Циклы выполняющиеся с помощью конструкций `for`, `while` и `do` могут быть переписаны с помощью операторов `goto` и `if`. В Scheme, примитивным средством для управления локальным потоком выполнения является вызов функции(*function call*) (вместе с оператором `if`). Таким образом, повторение некоторых вычислений в цикле в конечном счете реализуется функцией которая вызывает сама себя, то есть рекурсией.

Этот подход теоретически более сильный, поскольку формально рассуждать о рекурсии, более проще чем о переходах(gotos). В Си, использование рекурсии исключительно не практично, т.к. она быстро съедает весь стек. В Scheme, однако это практично: вызовы функций, которые появляются в хвостовой позиции(*tail position*) не используют дополнительное пространство в стеке(см. Раздел 3.3.2 [Tail Calls], страница 25).

Вызов функции находиться в хвостовой позиции, когда это последнее дейтсвия, которое выполняет вызывающая функция. Значение, возвращаемое вызываемой функцией немедленно возвращается из вызывающей функции. В следующем примере, вызов `bar-1` находиться в хвостовой позиции, а вызов `bar-2` нет. (Однако вызов `1-` в `foo-2` находиться в хвостовой позиции.)

```
(define (foo-1 x)
  (bar-1 (1- x)))

(define (foo-2 x)
  (1- (bar-2 x)))
```

Таким образом, вы получаете чистую рекурсию только в хвостовой позиции, такая рекурсия будет использовать постоянное пространство стека и будет такой же хорошей как цикл, построенный из переходов gotos.

Scheme предлагает несколько синтаксических абстракций (`do` и *named* `let`), которые создают циклы намного легче.

Но только функции Scheme могут вызывать другие функции в хвостовой позиции: Си функции этого не могут. Это имеет значение, если у вас есть, скажем, две функции, которые рекурсивно вызывают друг друга, образуя общий цикл. Следующий (нереалиситчный) пример показывает как можно определить, является ли неотритцательное целое число n четным или нечетным.

```
(define (my-even? n)
  (cond ((zero? n) #t)
        (else (my-odd? (1- n)))))

(define (my-odd? n)
  (cond ((zero? n) #f)
        (else (my-even? (1- n)))))
```

Поскольку вызовы `my-even?` и `my-odd?` находятся в хвостовой позиции, эти две процедуры могут применяться к произвольно большим целым, не переполняя стек. (Конечно, они все равно займут много времени.)

Однако, когда одна или обе из этих двух процедур будут переписаны на Си, они не смогут больше вызывать своего компаньона в хвостовой позиции (поскольку Си не имеет этого понятия). Вам необходимо учитывать это соображение при принятии решения о том, какие части вашей программы писать на Scheme а какие на Си.

В дополнение к вызовам функций и возврату из них, программа Scheme также может выходить из функций не локально, так что поток выполения возвращается непосредственно на внешний уровень. Это означает, что некотоыре функции могут вообще не иметь возврата.

Более того, невозможно не только перейти на некоторый внешний уровень контроля, Scheme программа может также перепрыгнуть назад в середину функции и повторно выйти. Это может вызвать возврат из некоторых функций более одного раза.

В общем, эти не локальные переходы выполняются путем вызова продолжений(*continuations*), которые ранее были перехвачены с использованием `call-with-current-continuation`. Guile также предлагает ограниченный набор функций, `catch` и `throw`, котоыре могут использоваться для нелокальных выходов. Это ограничение делает их более эффективными. Отчет об ошибках (с функцией `error`) реализован путем вызова например `throw`. Функции `catch` и `throw` входят в тему исключений(*exceptions*).

Поскольку функции Scheme могут вызывать Си функции и наоборот, код Си может более широко управлять потоком выполнения Scheme. Возможно Си функция никогда не возвращается, или возвращается более одного раза. Хотя Си пердлагает `setjmp` и `longjmp` для нелокальных выходов., это все еще необычная вещь для кода Си. Напротив, нелокальные выходы очень распространены в Scheme, в основном для сообщения об ошибках.

Вы должны быть готовыми к нелокальным скачкам в потоке выполнения всякий раз, когда вы используете функции из `libguile`: лучше предположить, что любая функция `libguile` может выдать сигнал ошибки ли запустить обработки сигнала (который, в свою очередь, может выполнять произвольные действия).

Часто бывает необходимо предпринять действия по очистке, когда управление покидает функцию нелокально. Кроме того, когда управление возвращается нелокально, могут потребоваться некоторые действия по настройке. Например, функция Scheme `with-output-to-port` нуждается в изменении глобального состояния, так что `current-output-port` вернул порт переданный `with-output-to-port`. Глобальный выходной порт должен быть сброшен до своего предыдущего значения когда `with-output-to-port` возвращется нормально или когда она возвращается не локально. Аналогично, порт необходимо установить снова, когда управление возващается нелокально.

Код Scheme может использовать функцию `dynamic-wind` для настройки и сброса глобальных настроек. Си код может использовать соответствующую функцию `scm_internal_dynamic_wind` или пару `scm_dynwind_begin`/`scm_dynwind_end` вместе с подходящими 'dynwind действиями' (см. Раздел 6.13.10 [Dynamic Wind], страница 339).

Вместо того, чтобы справляться с нелокальным потоком управления, вы также можете предотвратить его, установив барьер продолжения(*continuation barrier*), См. Раздел 6.13.14 [Continuation Barriers], страница 351. Например, функция `scm_c_with_continuation_barrier`, гарантированно возвращается ровно один раз.

5.4.4 Асинхронные Сигналы

Вы не можете вызывать функции libguile из обработчиков POSIX сигналов, но вы можете зарегистрировать Scheme обработчики для POSIX сигналов, таких как `SIGINT`. Эти обработчики не запускаются во время фактической доставки сигнала. Вместо этого они запускаются, когда программа(точнее поток который зарегистрировал обработчик) достигает следующей безопасной точки (*safe point*).

У самих функций libguile есть много таких безопасных точек. Следовательно, вы должны быть готовы к произвольным действиям при вызове функции libguile. Например, даже `scm_cons` может содержать безопасную точку, и когда обработчик сигнала ожидает для вашего потока, вызов `scm_cons` будет запускать этот обработчик, и все может произойти, включая нелокальный выход, хотя сам `scm_cons` обычно так не делает.

Если вы не хотите разрешать работу асинхронных обработчиков сигналов, вы можете временно заблокировать их с помощью например `scm_dynwind_block_asyncs` См. Раздел 6.22.3 [Asyncs], страница 466.

Поскольку обработка сигналов в Guile зависит от безопасных точек, вам необходимо убедиться, что ваши функции действительно предлагают их в достаточном количестве. Обычно, вызов функции libguile в обычном режиме, это все что необходимо. Но когда поток можт долгое время находиться в секции кода, которая не вызывает функций libguile, полезно влючать в код явные безопасные точки. Это позволит, например, пользователю прервать ваш код с помощью `C-c`.

Вы можете сделать это с помощью макроса `SCM_TICK`. Этот макрос синтаксически является выражением. То есть, вы можете использовать его так:

```
while (1)
  {
    SCM_TICK;
    do_some_work ();
  }
```

Частое выполнение безопасной точки еще более важно в многопоточных программах, См. Раздел 5.4.5 [Multi-Threading], страница 76.

5.4.5 МногоПоточность

Guile можно использовать в многопоточных программах, так же как и в однопоточных.

Каждый поток, который хочет использовать функции из libguile, должен поместить себя в режим *guile mode* и должен следовать нескольким правилам. Если он не хочет соблюдать эти правила в определенных ситуациях, поток может покинуть режим guile (но больше, конечно, не сможет использовать функции libguile в это время).

Поток входит в режим guile вызывая `scm_with_guile`, `scm_boot_guile`, или `scm_init_guile`. Как поясняется в справочной документации для этих функций, Guile тогда узнает о границах стека потока и может защитить значения `SCM` сохраненные в локальных переменных. Когда поток впервые попадает в режим guile, он получает представление Scheme и например, список `all-threads`.

Потоки в режиме guile могут блокироваться (например., выполняя блокирующий I/O) не вызывая никаких проблем[3]; временно покидать режим guile с `scm_without_guile` перед блокировкой немного улучшая производительность GC. Для некоторых общих операций блокировки, Guile предоставляет удобные функции. Например, если

[3] В Guile 1.8, блокировка потока в режиме guile прдотвращала сборку мусора. Таким образом приходилось покидать режим guile когда он мог блокироваться. Это больше не нужно в Guile 2.*x*.

вы хотите заблокировать мьютексы pthread в режиме guile, вы можете воспользоваться `scm_pthread_mutex_lock` который похож на `pthread_mutex_lock` за исключением того, что он покидает режим guile при блокировке.

Все функции libguile (должны быть) надежны перед лицом нескольких потоков использующих их одновременно. Это означает, что нет никаких рисков для внутренней структуры данных libguile стать поврежденными таким образом, что процесс завершиться крахом.

Тем не менее, программа может по прежнему давать бессмысленные результаты. Принимая хэш-таблицу например, Guile гарантирует, что вы можете использовать ее из нескольких потоков одновременно и хеш-таблица всегда будет оставаться допустимой хэш-таблицей и Guile не даст сбой при доступе к ней. Однако это не гарантирует, что вставка в него одновременно из двух потоков будет давать полезные результаты: может произойти только одна вставка, ни одна из вставок не может произойти, или таблица вообще может быть изменена совершенно произвольно. (Она все равно будет действительной хеш-таблицей, но не той, которой вы могли ожидать.) Guile также может сигнализировать об ошибке когда он обнаруживает состояние гонки.

Таким образом, вам нужно добавлять дополнительную синхронизацию, когда несколько потоков хотят использовать одну хеш-таблицу или любой другой изменяемый объект Scheme.

При написании кода используемого с libguile, вы должны попытаться сделать его надежным. Пример, который преобразует список в вектор, поможет это проилюстрироать. Вот правильная версия:

```
SCM
my_list_to_vector (SCM list)
{
  SCM vector = scm_make_vector (scm_length (list), SCM_UNDEFINED);
  size_t len, i;

  len = scm_c_vector_length (vector);
  i = 0;
  while (i < len && scm_is_pair (list))
    {
      scm_c_vector_set_x (vector, i, scm_car (list));
      list = scm_cdr (list);
      i++;
    }

  return vector;
}
```

Прежде всего следует отметить, что хранение в местоположении `SCM` одновременно из нескольких потоков гарантирует надежность: вы не знаете, какое значение выиграет, но оно будет в любом случаее действительным значением `SCM`.

Но нет никакой гарантии, что список на который ссылается переменная *list* не изменяется в другом потоке, в то время когда по нему проходит цикл итерации. Таким образом, копируя свои элементы в вектор, список может быть длиннее или короче.

По этой причине цикл должен проверять, что он не перегрузит вектор и что он не выйдет за рамки списка. В противном случае `scm_c_vector_set_x` вызовет ошибку, если индекс будет вне допустимого диапазона, а `scm_car` и `scm_cdr` вызовет ошибку если значение не будет парой.

Безопасно использовать `scm_car` и `scm_cdr` с локальной переменной *list*, когда будет известно что переменная содержит пару. Содержание пары может измениться спонтанно, но это всегда будет действительная пара(и локальная переменная, конечно, не будет спонтанно указывать на другой объект Scheme).

Аналогично, вектор, такой как тот, который возвращается `scm_make_vector` всегда гарантированно сохраняет туже длину, что бы было безопасно использовать только scm_c_vector_length и сохранить результат. (В этом примере вектор(*vector*) является безопасным, так как он является новым объектом, и нет другого потока который может знать о нем, до тех пор пока он не будет возвращен из `my_list_to_vector`.)

Конечно, поведение `my_list_to_vector` субоптимально когда список (*list*) действительно асинхронно удлинняется или укорачивается в другом потоке. Но он рабочий: он всегда будет возвращать действительный вектор. Этот вектор может быть короче чем ожидалось, или его последние элементы будут не указаны, но это действительный вектор, и если программа хочет исключить эти случаи, она должна избегать асинхронного изменения списка.

Вот еще одна правильная версия:

```
SCM
my_pedantic_list_to_vector (SCM list)
{
  SCM vector = scm_make_vector (scm_length (list), SCM_UNDEFINED);
  size_t len, i;

  len = scm_c_vector_length (vector);
  i = 0;
  while (i < len)
    {
      scm_c_vector_set_x (vector, i, scm_car (list));
      list = scm_cdr (list);
      i++;
    }

  return vector;
}
```

Эта версия основана на проверке ошибок `scm_car` и `scm_cdr`. Когда список сокращается(т.е когда список(*list*) содержит не пары), `scm_car` выдает ошибку. В этом случае предпочтительнее просто вернуть полуинициализированный вектор.

API для доступа к векторам и массивам различных типов из Си имеет несколько иной подход к обеспечению потоковой устойчивости. Чтобы получить необработанную(сырую) память, в которой храняться элементы массива, вам нужно зарезервировать этот массив(*reserve*), если вам нужна необработанная память. В течении времени хранения массива, его элементы могут спонтанно изменять свои значения, но сама

память и другие вещи, такие как размер массива, гарантированно остаются фиксированными. Любая операция, которая изменяет эти параметры массива, который в настоящее время зарезервирован, будет сигнализировать об ошибке. Чтобы измежать этих ошибок, программа должна, конечно иметь механизмы синхронизации. Как вы можете видеть, сам Guile беспокоиться о надежности, а не о правильности: без правильной синхронизации ваша программа скорее всего не будет правильной, но худшим последствием будет сообщение об ошибке.

Реальная потоко-безопасность часто требует, чтобы критический раздел кода выполнялся в определенной ограничивающей манере. Общим требованием является то, что в раздел кода не входит второй поток управления во то время когда он уже выполняется. Блокировка мьютекса в этом разделе обеспечивает, что никакаой другой поток не начнет ее выполнять, блокировка асинхронности(asyncs) гарантирует блокировку асинхронного кода снова входящего в раздел текущего потока, а проверка ошибок мьютексов Guile гарантирует, что ошибка сигнализируется, когда текущий поток случайно возвращается в критический раздел через рекурсивные вызовы функций.

Guile предоставляет два механизма для поддержки критических разеделов, как описано выше. Вы можете использовать макросы `SCM_CRITICAL_SECTION_START` и `SCM_CRITICAL_SECTION_END` для очень простых разделов; или использовать контекст dynwind вместе с вызовом `scm_dynwind_critical_section`.

Макросы работают надежно только для критических секций, которые гарантированно не содержат нелокальный выход. Они также не обнаруживают случайный повторный вход в текущем потоке. Таким образом, вы, вероятно должны использовать их только для разделения критических разделов, которые не содержат вызовов функций libguile или других внешних функций, которые могут выполнять сложные вещи.

С другой стороны, функция `scm_dynwind_critical_section` будет правильно решать работу с нелокальными выходами, поскольку они требуют контекста dynwind. Кроме того, используя отдельный мьютекс для каждого критического участка он может обнаруживать случайные повторные попытки входа в критическую секцию.

5.5 Определение новых типов внешних объектов

Тип внешних объектов(*foreign object type*) является механизмом Guile для импорта объектов и типов из Си или других языков в систему Guile. Если у вас есть Си тип `struct foo`, например, вы можете определить соответствующий тип внешнего объекта Guile, который позволит обрабатывать коду Scheme объекты указывающие `struct foo *`.

Чтобы определить новый тип внешних объектов, программист предоставляет Guile некоторую существенную информацию о типе — какое у него имя, сколько у него полей, и есть ли финализатор (если есть) — и Guile выделяет для него новый тип для него. Внешние объекты могут быть доступны из Scheme или из Си.

5.5.1 Определение типов Внешних Объектов

Чтобы создать новый тип внешних объектов из Си, вызовите `scm_make_foreign_object_type`. Он возвращает значение типа `SCM`, которое идентифицирует новый тип.

Вот как можно объявить новый тип, представляющий восьмибитные черно-белые изображения:

```
#include <libguile.h>

struct image {
  int width, height;
  char *pixels;

  /* The name of this image */
  SCM name;

  /* A function to call when this image is
     modified, e.g., to update the screen,
     or SCM_BOOL_F if no action necessary */
  SCM update_func;
};

static SCM image_type image_type;

void
init_image_type (void)
{
  SCM name, slots;
  scm_t_struct_finalize finalizer;

  name = scm_from_utf8_symbol ("image");
  slots = scm_list_1 (scm_from_utf8_symbol ("data"));
  finalizer = NULL;

  image_type =
    scm_make_foreign_object_type (name, slots, finalizer);
}
```

Результатом является инициализированное значение `image_type`, которое идентифицирует новый тип внешних объектов. В следующем разделе описано, как создавать внешние объекты и как обращаться к их слотам.

5.5.2 Создание Внешних Объектов

Внешние объекты содержат ноль и более “слотов” данных. Слот может содержать указатель, целое число которое вписывается в `size_t` или `ssize_t`, или значение `SCM`.

Все объекты данного типа внешних объектов имеют одинаковое количество слотов. В примере из предыдущего раздела, тип изображение(`image`) имеет один слот, потому что список слотов переданый в `scm_make_foreign_object_type` имеет длину равную одному. (Фактические имена , присвоенные слотам, несущественны для большинства пользователей интерфейса Си, но могут быть испльзованы на стороне Scheme, чтобы осмотреть внешний объект.)

Чтобы создать внешний объект и инициализировать его первый слот, вызовите `scm_make_foreign_object_1 (`*`type`*`, `*`first_slot_value`*`)`. Аналогично вызываются конструкторы для инициализции 0, 1, 2, или 3 слотов, или инициализация n слотов через массив. См. Раздел 6.7 [Foreign Objects], страница 257, для получения полной информации. Любые поля, которые явно не инициализированы, устанавливаются в 0.

Чтобы получить или установить значение слота по индексу, вы можете использовать функции `scm_foreign_object_ref` и `scm_foreign_object_set_x`. Эти функции принимают и возвращают значения как указатели `void *`; существуют процедуры для доступа к слотам целых чисел со знаком и без знака, соответственно, такие как `_signed_ref` и `_unsigned_set_x`

Поля внешних объектов, которые являются указателями, могут быть сложными для управления. Если возможно, лучше всего что бы вся память, на которую ссылается внешний объект, управлялась сборщиком мусора. Такимо образом, сборщик мусора может автоматически гарантировать доступность памяти, когда это необходимо, и освобождать память, когда она становиться недоступной. Если это не так для вашей программы - например, если вы предоставляете объект Scheme который был выделен какой-либо другой не Guile частью вашей программы – тогда вам, вероятно, понадобиться реализовать финализатор. См. Раздел 5.5.4 [Foreign Object Memory Management], страница 82, для получения дополнительной информации.

Продолжая пример из предыдущего раздела, если глобальная переменная `image_type` содержит тип возвращаемый `scm_make_foreign_object_type`, вот как мы могли бы построить внешний объект, чье поле “data” содержит указатель на только что выделенную структуру `struct image`:

```
SCM
make_image (SCM name, SCM s_width, SCM s_height)
{
  struct image *image;
  int width = scm_to_int (s_width);
  int height = scm_to_int (s_height);

  /* Allocate the `struct image'.  Because we
     use scm_gc_malloc, this memory block will
     be automatically reclaimed when it becomes
     inaccessible, and its members will be traced
     by the garbage collector.  */
  image = (struct image *)
    scm_gc_malloc (sizeof (struct image), "image");

  image->width = width;
  image->height = height;

  /* Allocating the pixels with
     scm_gc_malloc_pointerless means that the
     pixels data is collectable by GC, but
     that GC shouldn't spend time tracing its
     contents for nested pointers because there
```

```
     aren't any.  */
  image->pixels =
    scm_gc_malloc_pointerless (width * height, "image pixels");

  image->name = name;
  image->update_func = SCM_BOOL_F;

  /* Now wrap the struct image* in a new foreign
     object, and return that object.  */
  return scm_make_foreign_object_1 (image_type, image);
}
```

Мы используем `scm_gc_malloc_pointerless` для буфера пикселей, чтобы сообщить сборщику мусора не сканировать его на указатели. Вызов `scm_gc_malloc`, `scm_make_foreign_object_1`, и `scm_gc_malloc_pointerless` вызывает исключение в условиях отсутствия памяти; сборщик мусора может вернуть ране выделенную память, если это призойдет.

5.5.3 Проверка Типа Внешнего Объекта

Функции, работающие с внешними объектами, должны проверять, что принятое `SCM` значение действительно правильного типа перед доступом к его данным. Они могут сделать это с помощью `scm_assert_foreign_object_type`.

Например, вот простая функция, которая работает с объектом изображения и проверяет тип своего аргумента.

```
SCM
clear_image (SCM image_obj)
{
  int area;
  struct image *image;

  scm_assert_foreign_object_type (image_type, image_obj);

  image = scm_foreign_object_ref (image_obj, 0);
  area = image->width * image->height;
  memset (image->pixels, 0, area);

  /* Invoke the image's update function.  */
  if (scm_is_true (image->update_func))
    scm_call_0 (image->update_func);

  return SCM_UNSPECIFIED;
}
```

5.5.4 Управление памятью Внешних Объектов

После того как внешний объект был освобожден для системы Scheme, он должен быть готов к сборке мусора. В приведенном выше примере, вся память связанная с внешним объектом, управляется сборщиком мусора, поскольку мы использовали функции

распределения памяти `scm_gc_`. Таким образом, особого внимания не требуется: сборщик мусора автоматически сканирует ее и восстанавливает неиспользуемую память.

Однако, когда данные связанные с внешним объектом, управляются каким-либо другим способом - например, память выделяемая `malloc` или файловые дискрипторы— для них можно указать функцию *finalizer* для освобождения этих ресурсов, когда внешний объект будет утилизироваться.

Как описано в разделе см. Раздел 5.4.2 [Garbage Collection], страница 72, сборщик мусора Guile при необходимости восстановит недоступную память. Этот процесс утилизации выполняется одновременно с основной программой. Когда Guile анализирует кучу и определяет, что память объекта может быть восстановлена, эта память помещается в "список свободной памяти(free list)" объектов, которые могут быть утилизированы. Обычно это его конец —объект доступен для немедленного повторного использования. Однако некоторые объекты могут иметь связанные с ними "финализаторы(finalizers)" — функции, которые вызываются для утилизируемых объектов для выполнения любых действий по внешней очистки.

Финализаторы это сложная работа и лучше их избегать. Они могут быть вызваны в неожиданное время, или вообще не вызваны — например, они не вызываются при завершении процесса. Они не помогают сборщику мусора в его работе; на самом деле, они являются препятствием. Кроме того, они нарушают внутренний учет сборщика мусора. Сборщик мусора(GC) решает сканировать кучу, когда он думает, что это необходимо, после некоторого выделения. Финализируемые объекты всегда представляют собой некоторый выделенный объем, который невидим для сборщика мусора. Эффект может заключаться в том, что фактическое использование ресурсов системы с финализируемыми объектами выше чем предполагает о них сборщик мусора(GC).

Все это оговорки в сторону, некоторые типы внешних объектов требуют финализаторов. Например, если у нас тип внешнего объекта, который являлся оберткой для файлового дескриптора— и мы не предлагаем это, поскольку у Guile уже есть порты —тогда вы можете определить этот тип следующим образом:

```
static SCM file_type;

static void
finalize_file (SCM file)
{
  int fd = scm_foreign_object_signed_ref (file, 0);
  if (fd >= 0)
    {
      scm_foreign_object_signed_set_x (file, 0, -1);
      close (fd);
    }
}

static void
init_file_type (void)
{
  SCM name, slots;
  scm_t_struct_finalize finalizer;
```

```
  name = scm_from_utf8_symbol ("file");
  slots = scm_list_1 (scm_from_utf8_symbol ("fd"));
  finalizer = finalize_file;

  image_type =
    scm_make_foreign_object_type (name, slots, finalizer);
}

static SCM
make_file (int fd)
{
  return scm_make_foreign_object_1 (file_type, (void *) fd);
}
```

Обратите внимание, что финализатор может быть вызван способами и во время, которое вы не ожидаете. В частности, если пользовательский Guile построен с поддержкой потоков, финализатор может быть вызван из любого потока, в котором работает Guile. В Guile 2.0, финализаторы вызываются через “asyncs”, который перемежает их с помощью кода Scheme; см. Раздел 6.22.3 [Asyncs], страница 466. В Guile 2.2 будет выделенный поток финализации, чтобы быть уверенными что поток финализации не работает в критической секции другого потока не известного Guile.

В любом случае, финализаторы запускаются одновременно с основной программой, и поэтому им необходимо быть асинхронными и потоко-безопасными(thread-safe). Если по какой-то причине это невозможно, например потому что вы встраиваете Guile в какое-то приложение, которое не является потоко-безопасным, у вас есть несколько способов действий. Один заключается в том, чтобы использовать опекунов(guardians) вместо финализаторов и организовать передачу объектов опекунам для финализации. См. Раздел 6.19.4 [Guardians], страница 430, за дальнейшей инофрмацией. Другой способ - полностью отключить автоматическую финализацию и организовать вызов `scm_run_finalizers ()` в соответствующих точках. См. Раздел 6.7 [Foreign Objects], страница 257, для получения дополнительной информации об этих интерфейсах.

Финализаторам разрешено выделять память, иметь доступ к памяти управляемой сборщиком мусора(GC), и вообще делать все, что может сделать любой код пользователя Guile. Это было не так в 1.8, где финализаторы были гораздо более ограничены. В частности, в Guile 2.0, финализаторы могут реанимировать объекты. Мы не рекомендуем, чтобы пользователи использовали эту возможность, однако как реанимированный объект может повторно открыть другие финализированные объекты которые повторно финализируются при возвращении обратно в Scheme. Эти объекты не будут финилизироваться повторно, но они могут привести проблеме использования памяти после освобождения(use-after-free) для кода, который обрабатывает объекты этого конкретного типа внешних объектов. Защищаясь от этой возможности, надежные процедуры финализации должны очищать состояние от внешнего объекта, как в приведенном выше примере `free_file`.

Одно, последнее предостережение. Финализаторы внешних объектов связвны со сроком жизни внешнего бъекта, но не его полей. Если вы получаете доступ к полю финализируемого внешнего объекта, и не организуете сохранение ссылки на сам внеш-

ний объект, может случиться так, что внешний объект будет финализирован, пока вы работаете с его полем.

Например, рассмотрим процедуру чтения некоторых данных из файла, из нашего примера выше.

```
SCM
read_bytes (SCM file, SCM n)
{
  int fd;
  SCM buf;
  size_t len, pos;

  scm_assert_foreign_object_type (file_type, file);

  fd = scm_foreign_object_signed_ref (file, 0);
  if (fd < 0)
    scm_wrong_type_arg_msg ("read-bytes", SCM_ARG1,
                            file, "open file");

  len = scm_to_size_t (n);
  SCM buf = scm_c_make_bytevector (scm_to_size_t (n));

  pos = 0;
  while (pos < len)
    {
      char *bytes = SCM_BYTEVECTOR_CONTENTS (buf);
      ssize_t count = read (fd, bytes + pos, len - pos);
      if (count < 0)
        scm_syserror ("read-bytes");
      if (count == 0)
        break;
      pos += count;
    }

  scm_remember_upto_here_1 (file);

  return scm_values (scm_list_2 (buf, scm_from_size_t (pos)));
}
```

После прелюдии, используется только значение `fd` и компилятору C нет необходимости сохранять объект `file`(т.е оптимизирующий код компилятора может уничтожить ссылку на этот объект и выделить ее для использования другим объектом). Если вызов `scm_c_make_bytevector` приведет к сборке мусора, ссылки на `file` может не оказаться в стеке или где нибудь еще и он может быть финализирован(закрыт), оставив вызов `read` для чтения закрытого (или, в многопоточной программе, возможно повторно используемый) файлового дескриптора. Использование вызова `scm_remember_upto_here_1` предотвращает это, создавая ссылку на `file` полсе доступа ко всем данным. См. Раздел 6.19.1 [Garbage Collection Functions], страница 425.

scm_remember_upto_here_1 требуется только для финализируемых объектов, поскольку сбор мусора других значений невидим для программы – он происходит когда это необходимо, и не является наблюдаемым. Но если вы сможете, уберечь себя от головной боли, создайте свою программу таким образом, чтобы финализация в ней была бы не нужна.

5.5.5 Внешние Объекты и Scheme

Также возможно создавать внешние объекты и типы объектов из Scheme и получать доступ к полям внешних объектов из Scheme. Например, пример файла(file) из последнего раздела может быть эквивалентно выражен следующим образом:

```
(define-module (my-file)
  #:use-module (system foreign-object)
  #:use-module ((oop goops) #:select (make))
  #:export (make-file))

(define (finalize-file file)
  (let ((fd (struct-ref file 0)))
    (unless (< fd 0)
      (struct-set! file 0 -1)
      (close-fdes fd))))

(define <file>
  (make-foreign-object-type '<file> '(fd)
                            #:finalizer finalize-file))

(define (make-file fd)
  (make <file> #:fd fd))
```

Здесь мы видим, что результат make-foreign-object-type, который эквивалентен вызовуscm_make_foreign_object_type, является структурой vtable. См. Раздел 6.6.18.1 [Vtables], страница 236, для получения дополнительной информациин. Чтобы создать экземпляр внешнего объекта, который действительно является структурой Guile, мы использкем make. (Мы моглибы использовать make-struct/no-tail, но для детализации, добавляем финализатор в метод инициализации initialize вызываемый make). Для доспупа к полям, мы используем struct-ref и struct-set!. См. Раздел 6.6.18.2 [Structure Basics], страница 236.

Существует удобный синтаксис define-foreign-object-type, который определяет тип, наряду с констурктором и процедурами доступа(getters) для полей. Сооответствующий вызов define-foreign-object-type для объекта типа файла(file) может выглядеть так:

```
(use-modules (system foreign-object))

(define-foreign-object-type <file>
  make-file
  (fd)
  #:finalizer finalize-file)
```

Здесь определяется тип `<file>` с одним полем, конструктором `make-file`, и процедурой доступа для поля `fd`, связанная с `fd`.

Типы внешних объектов это не только vtable, но и фактически классы GOOPS, как это отмечено выше. см. См. Глава 8 [GOOPS], страница 787, для получения дополнительной информации о системе объектно-ориентированного программирования Guile. Таким образом, можно определить методы печать(print) и равно(equality) используя GOOPS:

```
(use-modules (oop goops))

(define-method (write (file <file>) port)
  ;; Assuming existence of the `fd' getter
  (format port "#<<file> ~a>" (fd file)))

(define-method (equal? (a <file>) (b <file>))
  (eqv? (fd a) (fd b)))
```

Можно даже создавать подклассы внешних объектов.

```
(define-class <named-file> (<file>)
  (name #:init-keyword #:name #:init-value #f #:accessor name))
```

Возникает вопрос, как строяться эти значения, учитвывая что, `make-file` возвращает просто старый объект `<file>`. Оказывается, вы можете использовать интерфейс постороения GOOPS, где каждое поле внешнего объекта имеет связанный аргумент инициализации - ключевое слово.

```
(define* (my-open-file name #:optional (flags O_RDONLY))
  (make <named-file> #:fd (open-fdes name flags) #:name name))

(define-method (write (file <named-file>) port)
  (format port "#<<file> ~s ~a>" (name file) (fd file)))
```

См. См. Раздел 6.7 [Foreign Objects], страница 257, для полной документации по интерфейсам Scheme к внешним объектам. См. См. Глава 8 [GOOPS], страница 787, для получения дополнительных сведений о GOOPS.

В заключении, вы можете удивиться, как эта система поддерживает инкапсуляцию чувствительных(sensitive) значений. Первое, мы должны признать, что некоторые объекты по существу не безопасны и имеют глобальных охват(область действия). Например, в Си, целостность и конфеденциальность части программы находиться в милости от каждой другой части этой программы, поскольку любая часть программы может читать или писать что нибудь в это адресное пространство. В то же время, принципиальный доступ к струрированным данным организован в Си в лексических границах(области видимости); если вы не не открываете процедуры доступа для своего объекта, вы доверяете другим частям программы, что они не обошли этот барьер.

Ситуация не отличается в Scheme. Хотя небезопасных конструкций в Scheme значительно меньше чем в Си, они существуют. Модуль `(system foreign)` может использоваться для нарушения конфиденциальности и целостности, и не должен размещаться ненадежным кодом. Хотя `struct-ref` и `struct-set!` менее опасны, они по прежнему имеют способность насквозь проходить сквозь абстракции. Выполнение `struct-set!` на слоте с внешним объектом может вызвать сбой небезопасного

внешнего кода. В конечном счете, структуры в Scheme являются возможностью для абстрагирования, а не самими абстракциями.

Это оставляет нам лексические возможности, такие как конструкторы и процедуры доступа. Здесь, где заключается инкапсуляция: практическая степень, в которой внутренности ваших внешних объектов раскрывается степень, которой их процедуры доступа являются лексически доступными в коде пользователя. Если вы хотите разрешить пользователям ссылаться на поля вашего внешнего объекта, предоставьте им процедуру получения доступа к полю(getter). В противном случае вы должны предположить, что доступ к вашему объекту возможен только из вашего кода, который имеет соответствующие полномочия, или через код со сквозным доступом `struct-ref` и такой, который имеет сквозные полномочия.

5.6 Функция Snarfing

При написании кода Си для использования с Guile, вы обычно определяете набор Си функций, а затем делаете некоторые из них видимыми для мира Scheme вызывая `scm_c_define_gsubr` или связанные функции. Если у вас много функций для публикации, иногда бывает достадно держать список вызовов в `scm_c_define_gsubr` в синхронизации со списком определений функций.

Guile предоставляет программу `guile-snarf` для решения этой проблемы. Используя этот инструмент, вы можете хранить всю информацию, необходимую для определения функций наряду с определением самой функции; `guile-snarf` будет извлекать эту инфромацию из вашего исходного кода и автоматически сгенерирует файл вызовов `scm_c_define_gsubr` который вы можете включить `#include` в функцию инициализации.

Механизм snarfing"а работает для многих видов действий по инициализации, а не только для сбора вызовов `scm_c_define_gsubr`. Полный список того, что можно сделать см., См. Раздел 6.5 [Snarfing Macros], страница 111.

Программа `guile-snarf` вызывается следующим образом:

```
guile-snarf [-o outfile] [cpp-args ...]
```

Эта команда будет извлекать операции инициализации в *outfile*. Когда никакой внешний файл *outfile* не указан или когда *outfile* является `-`, будет использоваться стандартный вывод. Препроцессор Си вызывается с *cpp-args* (который обычно включает входной файл) и выход фильтруется для извлечения операций по инициализации.

Если во время обработки есть ошибки, *outfile* удаляется и программа выходит с ненулевым статусом.

Во время snarfing"а, определяется препроцессорный макрос `SCM_MAGIC_SNARFER`. Вы можете использовать его, чтобы избежать включения выходных файлов snarfer, которые еще не существуют, написав код следующим образом:

```
#ifndef SCM_MAGIC_SNARFER
#include "foo.x"
#endif
```

Вот как вы можете определить функцию Scheme `clear-image`, реализованную Си функцией `clear_image`:

```
#include <libguile.h>

SCM_DEFINE (clear_image, "clear-image", 1, 0, 0,
            (SCM image),
            "Clear the image.")
{
  /* C code to clear the image in image... */
}

void
init_image_type ()
{
#include "image-type.x"
}
```

Объявление `SCM_DEFINE` говорит, что функция Си `clear_image` реализует функцию Scheme называемую `clear-image`, которая принимает один требуемый аргумент (типа `SCM` и именуемый `image`), необязательный аргумент и завершающий аргумент. Строка `"Clear the image."` предоставляет короткий текст справки для функции, он называется *docstring*.

Макрос `SCM_DEFINE` также определяет статический массив символов, инициализированной на Scheme имени функции. В этом случае, `s_clear_image` устанавливается в Си строку, "clear-image". Вы можете использовать этот символ при создании сообщений об ошибках.

Предполагая, что текст выше живет в файле с именем `image-type.c`, вам нужно будет выполнить следующую команду для подготовки этого файла для компиляции:

```
guile-snarf -o image-type.x image-type.c
```

Здесь сканируется файл `image-type.c` для объявления `SCM_DEFINE` и записывается в выходной файл `image-type.x`:

```
scm_c_define_gsubr ("clear-image", 1, 0, 0, (SCM (*)() ) clear_image);
```

При компиляции обычно, `SCM_DEFINE` представляет собой макрос, который расширяется до заголовка функции `clear_image`.

Обратите внимание, что имя выходного файла соответствует `#include` из входного файла. Кроме того, вам все еще необходимо предоставить всю ту же информацию, которую вы бы использовали для `scm_c_define_gsubr`, но вы можете разместить эту информацию рядом с определением функции, поэтому она менее вероятно станет неправильной или устаревшей.

Если у вас есть много файлов, которые должна обрабаывать `guile-snarf`, вам следует рассмотреть возможность использования фрагмента в вашем Makefile:

```
snarfcppopts = $(DEFS) $(INCLUDES) $(CPPFLAGS) $(CFLAGS)
.SUFFIXES: .x
.c.x:
guile-snarf -o $@ $< $(snarfcppopts)
```

Здесь говориться make запускать `guile-snarf` для создания каждого необходимого файла `.x` из соответствующего файла `.c`.

Программа `guile-snarf` передает свои аргументы командной строки непосредственно в препроцессор Си, который использует их для извлечения необходимой ему информации из исходного кода. это означает что вы можете передать нормальные флаги компиляции для `guile-snarf`, чтобы определить символы препроцессора, добавить заголовочные файлы и т.д.

5.7 Обзор программирования на Guile

Guile разработан как интерпретируемый язык расширения, который легко интегрируется с приложениями написанными на Си(и Си++). Большая победа здесь для разработчика приложений заключается в том, что интеграция Guile, как говорит веб страница, "снижает энергию хактивации (взлома) вашего проекта." Снижение энергии взлома означет, что вы как разработчик приложения и ваши пользователи, воспользуются преимуществами которые вытекают из возможности расширить приложение в языке расширения высокого уровня, а не просто старым Си.

В абстрактных терминах трудно объяснить, что на самом деле означает и что включает в себя этот процесс интеграции, поэтому вместо этого давайте начнем с того что прыгнем прямо в пример того как вы можете интегрировать Guile в существующую программу и то, что вы могли бы ожидать от этого. В этом примере под нашими поясами, мы вернемся к более общему анализу аргументов и диапазону доступных вариантов программирования.

5.7.1 Как можно расширить Dia используя Guile

Dia это свободная программа для создания схемных диаграмм, таких как блок-схемы и планы комнат (`http://www.gnome.org/projects/dia/`). Этот раздел ведет мысленный эксперимент добавления Guile в Dia. При этом он призван проилюстрировать некотоыре из этапов и соображения, связанные с добавлением Guile к приложениям в целом.

5.7.1.1 Решите, почему вы хотите добавить Guile

Прежде всего, вы должны понять, почему вы хотите добавить Guile в Dia вообще, и это означает, сформировать картину того, что делает Dia и как она это делает. Итак, каковы составляющие приложения Dia?

- Самое главное, объекты домена приложения(*application domain objects*) — другими словами, концепции, которые отличают Dia от других приложений, например текстовых процессоров или таблиц: формы, шаблоны, коннекторы, страницы, а также свойства всех этих вещей.
- Код, управляющий графическим интерфейсом приложения, включающий размещение и отображение указанных выше объектов.
- Код, который обрабатывает входные события, которые указывают, что пользователь приложения хочет что-то сделать.

(Другими словами, учебник примера парадигмы модель-вид-контроллер (*model - view - controller*).)

Следующий вопрос: как Dia будет полезен после завершения интеграции с Guile? Возможны несколько (пложительных!) ответов и выбор, очевидно, зависит от приложения разработчиков. Тем не менее, один ответ заключается в том, что основным

преимуществом будет способность манипулировать объектами домена Dia приложения из Scheme.

Предположим, что Dia сделал набор процедур доступных в Scheme, представляющих наиболее основные операции над объектами, такими как фигуры, коннекторы и т.д. Используя Scheme, пользователь приложения может затем написать код, который основывается на этих операциях для создания более сложных процедур. Например, при базовых процедурах перечисления объектов на странице, чтобы определить, является ли объект квадратом, и изменить шаблон заполнения единичной фигуры, пользователь может написать процедуру Scheme, чтобы изменить шаблон заполнения всех квадратов на текущей странице.

```
(define (change-squares'-fill-pattern new-pattern)
  (for-each-shape current-page
    (lambda (shape)
      (if (square? shape)
          (change-fill-pattern shape new-pattern)))))
```

5.7.1.2 Четыре шага необходимых для добавления Guile

Предполагая эту цель, для ее достижения необходимы четыре шага.

Во-первых. вам нужен способ предоставления ваших объектов, специфичных для приложения, таких как `shape` в предыдущем примере, когда они передаются в мир Scheme. Если объекты настолько просты, что они естественным образом отображают встроенные типы данных Scheme, такие как числа и строки, вы вероятно захотите использовать интерфейс внешних объектов(*foreign object*) Guile для создания нового типа данных Scheme для ваших объектов.

Во-вторых, вам нужно написать код для основных операций, таких как `for-each-shape` и `square?` чтобы они имели доступ и управляли вашими существующими структурами данных. Затем сделайте эти операции доступными как примитивы(*primitives*) на уровне Scheme.

В-третьих, вам необходимо предоставить некоторый механизм в приложении Dia, который пользователь может подключить, чтобы вызвать произвольный код Scheme для вычисления.

Наконец, вам нужно немного перестроить свой Си код верхнего уровня приложения, чтобы он правильно инициализировал интерпретатор Guile и объявлял ваши внешние объекты(*foreign objects*) и примитивы(*primitives*) в мире Scheme.

Следующие подразделы в свою очередь расширяются по этим четырем пунктам.

5.7.1.3 Как представлять данные Dia в Scheme

Для всех, кроме самых тривиальных приложений, вы, вероятно, захотите разрешить представление объектов домена вашего приложения на уровне Scheme. Здесь появляются внешние объекты(foreign objects) и связанные с ними вопросы управления их жизненным циклом и сборкой мусора.

Чтобы получить более конкретную информацию об этом, давайте еще раз взглянем на пример, который мы дали ранее о том, как пользователи приложения могут использовать Guile для создания функций более высокого уровня из примитивов, которые обеспечивает сам Dia.

```
(define (change-squares'-fill-pattern new-pattern)
  (for-each-shape current-page
    (lambda (shape)
      (if (square? shape)
          (change-fill-pattern shape new-pattern)))))
```

Рассмотрим, что здесь храниться в переменной `shape`. Для каждой фигуры(shape) на текущей странице примитив `for-each-shape` вызывает лямбда-функцию `(lambda (shape) ...)` с аргументом, представляющим фигуру(shape). Вопрос: как этот аргумент представлен на уровне Scheme? Проблемы являются следующими.

- Каким бы ни было представление, оно должно быть снова декодировано кодом Си для примитивов `square?` и `change-fill-pattern`. Другими словами, примитивы такие как `square?` имеет возможность, так или иначе, превратить значение, которое оно получает обратно во что-то, что указывает путь к базовой Си структуре, описывающей фигуру.
- Представление также должно справляться с тем, что код Scheme удерживает значения для дальнейшего использования. Что произойдет, если код Scheme сохранит `shape` в глобальной переменной, но когда эта фигура(shape) удаляется(таким образом, что код Scheme не знает об этом) и затем некоторый другой код Scheme снова использует эту глобальную переменную в вызове, скажем в `square?`?
- Время жизни и распределение памяти объектов, которые существуют только в мире Scheme управляются автоматически сборщиком мусора Guile, используя только одно простое правило: когда нет никаких ссылок на объект, объект считается мертвым и поэтому его память освобождается. Но для объектов, которые существуют как в Си так и в Scheme, расклад более сложный; в случае с Dia, где аргумент `shape` передается туда и и обратно в мире Scheme, было бы совершенно не правильно удалять Си фигуры только потому, что код Scheme завершил выполнение. Как избежать этого?

Одно из решений этих вопросов заключается в представлении фигуры(shape) на уровне Scheme новой, Scheme-специфичной Си структурой обернутой в качестве внешнего объекта. Внешний объект, что передается в код и выводиться из кода Scheme и Scheme-специфичная Си структура внутри внешнего объекта указывает на лежащую в их основе Си структуру Dia, так что код для примитивов, таких как `square?` может получить ее.

Чтобы справиться с удалением фигуры, в то время как код Scheme все еще хранит значение фигуры, базовая Си структура должна иметь новое поле, которое указывает на Scheme-специфичный внешний объект. Когда фигура удаляется, соответствую код по цепочке к Scheme-специфичномой структуре и устанавливает этот обратный указатель на базовую структуру в NULL. Таким образом, значение внешнего объекта для фигуры продолжает существовать, но любой код примитивов, которые попытается ее использовать, обнаружит, что базовая фигура была удалена, поскольку указатель базовой стурктуры - NULL.

Итак, суммируем шаги, связанные с этим решением проблемы (и предполагая, что базовой структурой Си для фигуры является `struct dia_shape`):

- Определим новую Scheme-специфичную структуру которая указывает(*points*) на базовую Си структуру:

```
struct dia_guile_shape
{
  struct dia_shape * c_shape;   /* NULL => deleted */
}
```

- Добавим поле в `struct dia_shape`, которое указывает на Scheme-специфичную структуру `struct dia_guile_shape`, если оно одно —

```
struct dia_shape
{
  ...
  struct dia_guile_shape * guile_shape;
}
```

  — так что Си код может установить `guile_shape->c_shape` в NULL когда базовая фигура удаляется.
- Обернем `struct dia_guile_shape` как тип внешнего объекта(foreign object).
- Всякий раз, когда вам нужно представить Си фигуру на Scheme уровне, создавайте экземпляр внешнего объекта для нее и передавайте его.
- В коде примитивов, который получает экземпляр внешнего объекта фигуры, проверяйте поле `c_shape` при его расшифровке, чтобы выяснить, продолжает ли сущетсвовать лежащая в его основе базовая Си фигура.

Что касается управления памятью, значения внешний объектов и их Scheme-специфичные структуры находятся под управлением сборщика мусора, тогда как базовые Си структуры явно управляются точно также, как Dia управляла бы ими до того как мы подумали о добавлении Guile.

Когда сборщик мусора решает освободить значение внешнего объкта фигуры, он вызывает функцию *finalizer* которая была указана при определении типа внешнего объекта фигуры. Поддерживать правильность поля `guile_shape` в базоваой Си структуре, эта функция должна перейти к базовой сруктуре Си(если она еще существует) и установить ее поле `guile_shape` в NULL.

Полную документацию по определению и использованию типов внешних объектов см. Раздел 5.5 [Defining New Foreign Object Types], страница 79.

5.7.1.4 Написание примитивов Guile для Dia

Как только будут определены детали представления объекта, написание кода примитивной функции которая вам нужна, как правило, очень просто.

Примитив это просто Си функция, аргументы и возвращаемые значение которой имеют тип `SCM`, и чье тело делает то что вы хотите. В качестве примера можно привести реализацию примитива `square?`:

```
static SCM square_p (SCM shape)
{
  struct dia_guile_shape * guile_shape;

  /* Check that arg is really a shape object. */
  scm_assert_foreign_object_type (shape_type, shape);
```

```
  /* Access Scheme-specific shape structure. */
  guile_shape = scm_foreign_object_ref (shape, 0);

  /* Find out if underlying shape exists and is a
     square; return answer as a Scheme boolean. */
  return scm_from_bool (guile_shape->c_shape &&
                        (guile_shape->c_shape->type == DIA_SQUARE));
}
```

Обратите внимание на то, как легко перейти из параметра `SCM shape` который принимает `square_p` — который является внешним объектом — к Scheme-специфичной структуре внутри внешнего объекта, а оттуда и к базовой Си структуре фигуры.

В этом коде `scm_assert_foreign_object_type`, `scm_foreign_object_ref` и `scm_from_bool` из стандартного Guile API. Мы преполагаем, что `shape_type` был доступен нам когда мы создали форму внешнего объекта используя `scm_make_foreign_object_type`. Вызов `scm_assert_foreign_object_type` гарантирует, что *shape* действительноis является фигурой. Это необходимо для защиты кода Scheme, использование процедуры `square?` не корретно, как в форме `(square? "hello")`; Скрытая типизация в Scheme означает, что такие ошибки использования должнф быть пойманы во время выполнения.

Написав код Си для ваших примитивов, вы должны сделать их доступными как процедуры Scheme вызывая функцию `scm_c_define_gsubr`. `scm_c_define_gsubr` (см. Раздел 6.9.2 [Primitive Procedures], страница 264) принимает аргументы, которые определяются на уровне Scheme по имени примитива и количеству требуемых аргументов, необязательных и остальных аргументов которые функция может принять. Примитив `square?` всегда требует только один аргумент, пэтому вызов делающий его доступным в Scheme, читается следующим образом:

```
scm_c_define_gsubr ("square?", 1, 0, 0, square_p);
```

Где поставить этот вызов, смотри подраздел ниже, следующий за структурой доступа к Guile коду (см. Раздел 5.7.1.6 [Dia Structure], страница 95).

5.7.1.5 Предоставление хука для выполнения кода Scheme

Чтобы сделать интеграцию Guile полезной, вы должны создать какой-то хук в своем приложении, который пользователи приложения могли бы использовать для выполнения кода Scheme.

Технически это просто; вам просто нужно принять решение о механизме, который подходит для вашего приложения. Подумайте о Emacs, например, когда вы вводите *`ESC :`*, вы получаете подсказку, в которой вы можете ввести любой код Elisp, который затем выполнит Emacs. Или, снова, как в Emacs, вы можете предоставить механизм(такой как файл инициализации), чтобы позволить коду Scheme, который должен быть связан с определенной последовательностью клавиш и выполнять этот код, когда вводиться эта последовательность клавиш.

В любом случае, если у вас есть код Scheme, который вы хотите выполнять, как строку оканчивающуюся нулем, вы можете сказать Guile выполнить ее, вызывая функцию `scm_c_eval_string`.

5.7.1.6 Структура верхнего уровня доступа Guile в Dia

Предположим, что код пред - Guile Dia выглядит структурно следующим образом:

- `main ()`
  - делает много операций инициализации и настройки
  - вход в основной цикл обработки сообщений Gtk

Когда вы добавляете Guile в программу, одно (если точно, техническое) требование состоит в том, чтобы сборщим мусора Guile знал, где находиться нижняя часть стека Си. Самый простой способ убедиться в этом использовать `scm_boot_guile` следующим образом:

- `main ()`
  - делаем много операций инициализации и настройки
  - `scm_boot_guile (argc, argv, inner_main, NULL)`
- `inner_main ()`
  - определяем все типы внешних объектов
  - экспортируем примитивы в Scheme с помощью `scm_c_define_gsubr`
  - входим в основной цикл обработки Gtk

Другими словами, вы перемещаете внутренности того, что было ранее в вашей основной функции `main` в новую функцию, называемую `inner_main`, а затем добавляете вызов `scm_boot_guile`, с параметром `inner_main`, в конец функции `main`.

Предполагая, что вы используете внешние объекты и написали код примитивов, как описано в передыдущих подразделах вам также необходимо вставить вызовы для объявления ваших новых внешних объектов и экспортировать примитивы в Scheme. Эти декларации должны происходить *внутри* динамического вызова `scm_boot_guile`, на также *перед* любым кодом которы бы мог использовать их — начало функции `inner_main` идеальное место для их размещения.

5.7.1.7 Далее с Dia и Guile

Шаги, описанные до сих пор, реализуют первоначальную интеграцию Guile, которая уже дает большую дополнительную мощность для пользователей приложений Dia. Но есть дальнейшие шаги, которые вы могли бы предпринять, и интересно рассмотреть некоторые из них.

В общем, вы можете постепенно продвигать больше исходного кода Dia из Си в Scheme. Это может сделать код более удобным для обслуживания и расширяемым, и он может открыть дверь для новых парадигм программирования, которые сложно реализовать на Си, но легко на Scheme.

Конкретным примером этого является то, что вы можете использовать пакет guile-gtk, который предоставляет процедуры уровня Scheme для большей части библиотеки Gtk+, для перемещения кода, который размещает и отображает объекты Dia, с Си на Scheme.

Поскольку вы следуете этому пути, естественно становиться менее полезным поддерживать различие между оригинальным исходным кодом Dia не связанным с Guile, и его более поздним кодом, реализующим внешние объекты и примитивы для мира Scheme.

Например предположим, что оригинальный исходный код имел функцию `dia_change_fill_pattern`:

```
void dia_change_fill_pattern (struct dia_shape * shape,
                              struct dia_pattern * pattern)
{
  /* real pattern change work */
}
```

Во время первоначальной интеграции Guile, вы добавляете примитив `change_fill_pattern` для Scheme целей, которые обращаются к базовым структурам из своих значений внешних объектов и используют `dia_change_fill_pattern`, чтобы выполнить настоящую работу.

```
SCM change_fill_pattern (SCM shape, SCM pattern)
{
  struct dia_shape * d_shape;
  struct dia_pattern * d_pattern;

  ...

  dia_change_fill_pattern (d_shape, d_pattern);

  return SCM_UNSPECIFIED;
}
```

На этом этапе имеет смысл сохранить `dia_change_fill_pattern` и `change_fill_pattern` отдельно, поскольку `dia_change_fill_pattern` можно так же вызвать вообще без перехода через Scheme, скажем, потому что пользователь нажимает кнопку, которая вызывает Си зарегистрированный в Gtk+ обрабочик.

Но, если код для создания кнопок и регистрации их обратных вызовов перемещается в Scheme (используя guile-gtk), может оказаться, что `dia_change_fill_pattern` больше не может быть вызван иначе, чем через Scheme. В этом случае, имеет смысл отменить его и переместить его содержимое непосредственно в `change_fill_pattern`, например:

```
SCM change_fill_pattern (SCM shape, SCM pattern)
{
  struct dia_shape * d_shape;
  struct dia_pattern * d_pattern;

  ...

  /* real pattern change work */

  return SCM_UNSPECIFIED;
}
```

Таким образом, дальейшая интеграция Guile постепенно уменьшает количество функционального кода Си, который вы должны поддерживать в долгосрочной перспективе.

Аналогичный аргумент применим и к представлению данных. При обсуждении внешних объектов ранее возникали проблемы из-за разных моделей управления памятью и временем жизни, которые обычно применяются к структурам данных в Си и в Scheme. Однако, с дальнейшей интеграцией Guile, вы можете решить эту проблему более радикально, разрешив всем вашим структурам данных быть под контролем сборщика мусора и сохранять живые ссылки из мира Scheme. Вместо того, чтобы поддерживать массив или связный список фигур в Си, вы бы вместо этого хранили список в мире Scheme.

Скорее, подобно объединению `dia_change_fill_pattern` и `change_fill_pattern`, практический результат такого изменения заключается в том, что вам больше не придется поддерживать отдпельно структуры `dia_shape` и `dia_guile_shape`, и поэтому больше не нужно беспокоиться о связи между ними. Вместо этого вы сможете изменить определение внешнего объекта, чтобы обернуть структуру `dia_shape` напрямую и отправить `dia_guile_shape` на помойку. Убрать промежуточное звено!

Наконец, мы приходим к святому граалю свободного программного обеспечния/языка расширения Guile. Когда у вас есть представление Scheme для интересующих типов данных Dia таких как фигуры(shapes), и удобная группа примитивов для манипулирования ими, внезапно становиться ясно, что у вас есть набор функциональных возможностей, котоырй может иметь далеко идущее применение за пределами самой Dia. Другими словами, типы данных и примитивы теперь могут стать библиотекой, а Dia становиться только одним из возможных приложений, использующих эту библиотеку, хотя и важным на этой раннее стадии.

В этой модели, Guile становиться только клеем, который связывает все вместе. Представьте себе приложение, которое сочетает в себе функциональность от Dia, Gnumeric и GnuCash — это сейчас сложно, потому что такого приложения пока не существует, но это однажды произойдет . . .

5.7.2 Почему Scheme более доступная чем Си

Основополагающим предложением Guile является предположение, что программирование на языке высокого уровня, в частности Guile реализации Scheme, обязательно лучше каки-то образом чем программирование на Си. Что мы подразумеваем под этим утверждением, и как мы можем быть так уверены?

Один класс преимуществ применяется не только к Scheme, но и в целом к любому интерпретируемому языку высокого уровня, языку сценариев, таких как Emacs Lisp, Python, Ruby, или язык макросов TEX. Общими особенностями всех этих языков по сравнению с Си являются:

- Они поддаются быстрым экспериментальным циклам разработки, сочетание их интерпретируемости и интегрированной среды разработки в которой их используют.
- Они освобождают разработчиков от некоторых низкоуровневых "бухгалтерских" задач связанных с программированием на Си, в частности по управлению памятью.
- Они предоставляют функции высокого уровня, такие как контейнерные объекты и обработка исключений, которые упрощают общие задачи программирования.

В случае Scheme, особенности, которые делают программирование проще — и веслее! — являются его мощными механизмами для абстаргирования частей программ(замыкания — см. Раздел 3.4 [About Closure], страница 27) и для итерации (см. Раздел 6.13.4 [while do], страница 321).

Свидетельства в поддержку этого аргумента являются эмпирическими: огромное количество кода, который был написан на языках расширения для приложений, поддерживающих этот механизм. Наиболее примечательны расширения, написанные в Emacs Lisp для GNU Emacs, на языке макросов TEX для TEX, и на Script-Fu для Gimp, но теперь все чаще появляется значительный код эко системы для основанных на Guile приложений, таких как Lilypond и GnuCash. Это близкое к немыслимому, что подобное количество функциональности могло быть добавлено к этим приложениям, просто написав новый код на своих базовых языках реализации.

5.7.3 Пример: Использование Guile для тестового стенда Приложения

В качестве примера того, что это означает на практике, представьте себе, как написать тестовый стенд для приложения которое тестируется путем отправки различных запросов(через интерфейс Си) и получения определенных результатов. Предположим далее, что приложение сохраняет представление о своем текущем состоянии, и что "правильный" вывод для данного запроса можт зависеть от текущего состояния приложения. Полный план "белого ящика"[4] для этого приложения будет направлен на то, что бы представить все возможные запросы в каждом различимом состоянии и проверки вывода для комбинации всех запросов/состояний.

Написание всего тестового кода на Си было бы очень утомительным. Предположим, что тестовый стенд добавляет одну новую Си функцю, чтобы отправить произвольный запрос и вернуть ответ а зетем использует Guile для экспорта этой функции в виде процедуры Scheme. Остальна часть тестового стенда может быть написана на Scheme, и поэтому выгоды от всех преимуществ программирования на Scheme, описаны в предыдущем разделе.

(В этом конкретном примере есть дополнительное преимущество написания большей части тестового стенда на Scheme. Общей проблемой тестирования белого ящика является то, что ошибки и ошибочные предположения в тестируемом приложении можно легко воспроизвести в тестовом коде. Сложнее копировать ошибки, подобные этому, когда тестовый стенд написан на языке отличном от языка приложения.)

5.7.4 Выбор параметров программирования

Предыдущие аргументы и примеры указывают на модель программирования Guile, которая применима во многих случаях. Согласно этой модели, программирование Guile включает в себя баланс между программированием на Си и Scheme, с целью извлечь максимально возможный выигрыш от уровня Scheme и наименьшего количества работы на уровне Си.

Работа на уровне Си, требуемая в этой модели, обычно состоит из упаковки и экспорта функций и объектов приложения, чтобы их можно было увидеть и обработать на уровне Scheme. Чтобы помочь в этом, интерфейс языка Guile Си включает

[4] План тестирования белого ящика(*white box*) - это тот, который включает знание внутреннего дизайна приложения в соответствии с заданными тестами.

в себя служебные функции, которые делают эту интеграцию очень простой для разработчика приложений. Эти функции описанны далее в этой части руководства: см. REFFIXME.

Однако эта модель в действительности является одной из множества вариантов программирования. Если все функции, которые вам нужны доступны из Scheme, вы можете выбрать писать свое приложение на Scheme (или одном из других языков высокого уровня, трансляцию которого поддерживает Guile), и просто использовать Guile в качестве интерпретатора для Scheme. (В будущем мы надеемся, что Guile также сможет скомпилировать код Scheme, сократив разрыв производительности между кодом Си и Scheme.) Или, с другой стороны шкалы Си–Scheme, вы можете написать большую часть своего приложения на Си и только иногда вызывать Guile для конкретных действий, таких как чтение конфигурационного файла или выполнения определенного пользователем расширения. Выбор сводиться к двум основным вопросам:

- Какие части приложения вы пишете на Си, а какие на Scheme (или другом транслируемом языке высокого уровня)?
- Как вы разрабатываете интерфейс между частями Си и Scheme вашего приложения?

Это конечно вопросы дизайна, и правильный дизайн для любого конкретного приложения всегда будет зависеть от конкретных требований, которые вы пытаетесь выполнить в контексте Guile, однако сущетствуют некоторые общепринятые соображения, которые могут помочь вам при выработке своих ответов.

5.7.4.1 Какая функциональность уже доступна?

Предположим, ради аргумента, что вы предпочитаете писать ваше приложение на Scheme. Тогда доступный API состоит из:

- стандартной Scheme
- плюс расширения стандартной Scheme предоставляемые Guile в ее базовом дистрибутиве
- плюс любая дополнительная функциональность, которую вы или другие собрали в пакет, чтобы его можно было загрузить как модуль Guile Scheme.

Модуль в последней категории может быть либо чистым модулем Scheme — другими словами набором служебных процедур, закодированных на Scheme — или модулем который прдоставляет Scheme интерфейс к библиотеке расширений, закодированной на Си — или другими словами, пакетом в котором некто выполнил хорошую работу по завершению некоторого полезного для вас кода на Си. Набор доступных модулей быстро растет и уже включает такие полезные примеры как `(gtk gtk)`, который делает доступными функции рисования Gtk+ на Scheme, и `(database postgres)`, который обеспечивает SQL доступ к базе данных Postgres.

Учитывая растущий набор разработанных модулей, вполне возможно, что ваше приложение может быть реализовано путем комбинации объединения этих модулей с новым кодом приложений, написанным на Scheme.

Если этого подхода недостаточно, поскольку функциональность, которая требуется вашему приложению уже недоступна в этой форме, и невозможно написать новую функциональность на Scheme, вам нужно будет написать код Си. Ели требуемая

функция уже доступна на Си (например: в библиотеке), все что вам нужно, это немного клея, чтоебы связать его с миром Guile. Если вам не нужно опять писать базовый код, так встройте его в Guile.

В любом случае важны два общих соображения. Во первых, что такое интерфейс по которому функциональность представляется миру Scheme? Состоит ли интерфейс только из вызова функций(например, простой интерфейс рисования), или он должен включать объекты(*objects*), которые могут быть переданы между Си и Scheme и управляться обоими мирами. Во-вторых, как управление жизненным циклом и памятью объектов в коде Си соотносится к управляемым сборщиком мусора объектам Scheme? В случае когда базовый код Си еще не написан, большинство трудностей управления памятью можно избежать использованием функций языка Си интерфейса Guile с самого начала.

Для полной документации по написанию кода Си для Guile и подключения существующего Си кода к миру Guile, см. REFFIXME.

5.7.4.2 Функциональные и скоростные ограничения

5.7.4.3 Ваш предпочтительный стиль программирования

5.7.4.4 Какие управляющие программы выполнять?

5.7.5 Как насчет Пользователей Приложения?

До сих пор мы рассматривали, какие средства программирования Guile предоставляет для разработчика приложения. Но что, если вместо этого хотите использовать существующее приложение на основе Guile и хотите знать какие у вас есть варианты для программирования и расширения этого приложения?

Ответ на этот вопрос варьируется от одного приложения к другому, поскольку параметры доступные, неизбежно зависят от того предоставил ли разработчик приложений какие-либо перехватчики(hooks) что бы вы могли присоединить к ним собственный код, и если есть такие перехватчики, что они позволяют вам делать.[5] Например ...

- Если приложение позволяет загружать и выполнять любой Guile, мир становиться вашим. Вы можете расширять приложение любым способом, который вы выберете.
- Более острожное приложение может позволить вам загружать и выполнять код Guile, но только в безопасной(*safe*) средее, где доступный интерфейс ограничен приложением из стандатного Guile API.
- Или действительно пугливое приложение может НЕ обеспечить крюкчек(hook), чтобы выполнять пользовательский код, а вместо этого просто использовать синтаксис Scheme как удобный способ для пользователя указать приложению данные или параметры конфигурации.

[5] Конечно, в мире свободного программного обеспечения у вас всегда есть свобода изменять исходный код в соответствии с вашими требованиями. Здесь мы имеем дело с вариантами расширения, которые предоставляет приложение без необходимости изменения исходного кода.

В последних двух случаях, что выможете сделать, по определению ограничено приложением, и вы должны обратиться к документации по приложению чтобы узнать ваши опции.

Наиболее известным примером первого случая является Emacs, с его языком расширения Emacs Lisp: как и текстовый редактор, Emacs поддерживает загрузку и выполнение произвольного кода Emacs Lisp. Результат такой открытости был впечатляющим: Emacs сейчас имеет преимущества библиотек Emacs Lisp от пользователей, котоыре расширяют базовые функции редактирования, делая все от чтения новостей до психоанализа и игры в приключенческие игры. Только ограничение заключается в том, что расширения ограничены функциональностью предоставляемой встроенным набором примитивных операций Emacs. Например, вы можете взаимодействовать и отображать данные по манипулированию содержимым буфера Emacs, но вы не можете размещать всплывающие окна рисовать окна с размещением, который полностью отличается от стандарта Emacs.

Эта ситуация с приложением Guile, которое поддерживает загрузку произольного кода пользователя аналогична, за исключением, возможно, даже более того, поскольку Guile также поддерживает загрузку библиотек расширения, написанных на Си. Этот последний пункт позволяет коду пользователя добавлять новые примитивные операции в Guile, и поэтому обойти ограничение, существующее в Emacs Lisp.

На этом этапе различие между разработчиками приложения и пользователем приложения становятся размытыми. Вместо того, чтобы видеть себя пользователем расширяющим приложение, вы могли бы сказать, что вы разрабатываете новое приложение, используя некотурую примитивную функциональность, предоставляемую оригинальным приложением. Таким образом, все обсуждения предыдущих разделов этой главы имеют отношение к тому, как вы можете продолжить разработку ваших расширений.

5.8 Поддержка Autoconf

Autoconf, часть системы сборки GNU, облегчает пользователям сборку вашего пакета. В данном разделе описана поддержка Guile Autoconf.

5.8.1 Autoconf Background

Как объясняется в руководстве *GNU Autoconf Manual*, любой пакет требует настроки во время сборки (см. Раздел "Introduction" в *The GNU Autoconf Manual*). Если ваш пакет использует Guile (или использует пакет, который в свою очередь использует Guile), вам вероятно, нужно знать, какие конкретно Guile функции доступны и подробности о них.

Способ сделать это - написать функциональные тесты и организовать их выполнение в конфигурационном скрипте `configure`, как правило надо добавить тесты в `configure.ac`, и запустить `autoconf` для создания `configure`. Затем пользователи вашего пакета запускают `configure` обычным способом.

Макросы это способ сделать тестирование общих функций простым выражением. Autoconf предоставляет широкий спектр макросов (см. Раздел "Existing Tests" в *The GNU Autoconf Manual*), и установка Guile обеспечивает специфичные для Guile тесты в следующих областях: обнаружение программ, сообщение о флагах компиляции, и проверку модулей Scheme.

5.8.2 Макросы Autoconf

Как упоминалось ранее в этой главе, Guile поддерживает параллельную установку и использует `pkg-config` позволяющий пользователю выбирать, какая версия Guile ему интересна. `pkg-config` имеет свой собственный набор макросов Autoconf, которые, вероятно, установлены на большинстве систем разработчиков. Наиболее полезным из этих макросов является `PKG_CHECK_MODULES`.

```
PKG_CHECK_MODULES([GUILE], [guile-2.2])
```

Этот пример ищет Guile и устанавливает переменные `GUILE_CFLAGS` и `GUILE_LIBS` соответственно, или печатает ошибку и выходит, если Guile не был найден.

Guile поставляется с дополнительными макросами Autoconf, предоставляющими дополнительную информацию, установленными в *prefix*`/share/aclocal/guile.m4`. Их имена начинаются с `GUILE_`.

`GUILE_PKG` [*VERSIONS*] [Autoconf Macro]

Этот макрос запускает утилиту `pkg-config` для поиска файлов разработки доступной версии Guile.

По умолчанию, этот макрос будет искать последнюю стабильную версию Guile (например 2.2), возвращаясь назад к предыдущей стабильной версии (например 2.0) если она доступна. Если не найден файла guile-*VERSION*.pc сообщается об ошибке. Найденая версия сохраняется в *GUILE_EFFECTIVE_VERSION*.

Если `GUILE_PROGS` уже был вызван, этот макрос гарантирует, что файлы разработки будут иметь ту же версию, что и программа Guile.

GUILE_EFFECTIVE_VERSION помечается для подстановки , как `AC_SUBST`.

`GUILE_FLAGS` [Autoconf Macro]

Этот макрос запускает утилиту `pkg-config` чтобы найти как компилировать и связвать программы с Guile. Он уставнавливает четыре переменных: *GUILE_CFLAGS*, *GUILE_LDFLAGS*, *GUILE_LIBS*, и *GUILE_LTLIBS*.

GUILE_CFLAGS: флаги для передачи в C или C++ компилятор для создания кода использующего заголовочные файлы Guile. Это почти всегда только один или несколько флагов `-I`.

GUILE_LDFLAGS: флаги для передачи компилятору, чтобы связать программу с Guile. Они включают в себя `-lguile-`*`VERSION`* для библиотеки Guile, а также может включать один или более флагов `-L`, чтобы сообщить компилятору где искать библиотеки. Но это не включает флаги, которые влияют на путь поиска библиотек во время выполнения программы и это может привести к построению программы которая не запускается, если все необходимые библиотеки не установлены в стандартное расположение, такое как `/usr/lib`.

GUILE_LIBS и *GUILE_LTLIBS*: флаги для передачи в компилятор или libtool, соответственно, чтобы связать программу с Guile. Они включают флаги, которые расширяют пусть поиска для библиотек во время выполнения, чтобы разделяемые библиотеки можно было найти в местах где они были во время связывания программы, даже в нестандартныхъ местах. *GUILE_LIBS* должен быть использован при связывании программы непосредственно компилятором, тогда как *GUILE_LTLIBS* должен быть использован, когда связывание программы осуществляется через libtool.

Переменные помечаемые для подстановки, как `AC_SUBST`.

`GUILE_SITE_DIR` [Autoconf Macro]

Этот макрос ищет Guile директорию "site". Переменная *GUILE_SITE* устанавливает Guile директорию "site" для исходных файлов Scheme (обычно это что-то похожее на PREFIX/share/guile/site). *GUILE_SITE_CCACHE* будет указывать на каталог для скомпилированных файлов Scheme также исвестных как файлы `.go` (обычно это что то похожее на PREFIX/lib/guile/*GUILE_EFFECTIVE_VERSION*/site-ccache). *GUILE_EXTENSION* будет указывать на каталог для скомпилированных Си расширений (обычно это что то похожее на PREFIX/lib/guile/*GUILE_EFFECTIVE_VERSION*/extensions). Последние два устанавливаются пустыми, если конкретная версия Guile не поддерживает их. Обратите внимание, что этот макрос будет запускать макросы `GUILE_PKG` и `GUILE_PROGS` если они небыли запущены ранее.

Переменные помечаются для подстановки как `AC_SUBST`.

`GUILE_PROGS` [*VERSION*] [Autoconf Macro]

Этот макрос ищет программы `guile` и `guild`, устанавливая перменные *GUILE* и *GUILD* путями к ним, соответственно. Макрос попытается найти `guile` с суффиксом `-X.Y`, а затем ищет его с суффиксом `X.Y`, и затем возвращается к поиску для `guile` без суффикса. Если `guile` все еще не найден, выдает сигнал ошибки. Суффикс, если есть, то что требовалось, чтобы найти `guile` будет использован и для `guild`.

По умолчанию, этот макрос будет искать последнюю стабильную версию Guile (например 2.2). x.y или x.y.z версии могут быть указаны. Если найдена более старая версия, макрос сигнализирует об ошибке.

Эффективная версия найденого `guile` устанавливает *GUILE_EFFECTIVE_VERSION*. Этот макрос гарантирует, что эффективная версия совместима с результатом предыдущего вызова `GUILE_FLAGS`, если он был.

Как устаревший интерфейс, он так же ищет `guile-config` и `guile-tools`, устанавливая *GUILE_CONFIG* и *GUILE_TOOLS*.

Переменные помечаются для подстановки как `AC_SUBST`.

`GUILE_CHECK_RETVAL` *var check* [Autoconf Macro]
var это имя переменной оболочки(shell), для которой необходимо установить возвращаемое значение. *check* это выражение Guile Scheme, вычисляемое с "$GUILE -c", и возвращающее либо 0 или non-#f чтобы указать что успешность проверки. Не-0 число или #f указывают на ошибку. Избегайте использования символьного знака "#" поскольку это сбивает с толку autoconf.

`GUILE_MODULE_CHECK` *var module featuretest description* [Autoconf Macro]
var это имя переменной оболочки(shell), для которой нужно установить "yes" или "no". *module* это список символов, подобных: (ice-9 common-list). *featuretest* это выражение приемлемое для GUILE_CHECK, q.v. *description* описание, это фраза глагола в настоящем времени (передаваемая в AC_MSG_CHECKING).

`GUILE_MODULE_AVAILABLE` *var module* [Autoconf Macro]
var это имя перемнной оболочки, для которой надо установить "yes" или "no". *module* это список символов, таких как: (ice-9 common-list).

`GUILE_MODULE_REQUIRED` *symlist* [Autoconf Macro]
symlist это список символов, БЕЗ окружающих скобок, например: ice-9 common-list.

`GUILE_MODULE_EXPORTS` *var module modvar* [Autoconf Macro]
var это переменная оболочки которую нужно установить в "yes" или "no". *module* это список симоволов, например: (ice-9 common-list). *modvar* это переменная Guile Scheme для проверки.

`GUILE_MODULE_REQUIRED_EXPORT` *module modvar* [Autoconf Macro]
module это список символов, таких как: (ice-9 common-list). *modvar* это переменная Guile Scheme для проверки.

5.8.3 Использоание Макросов Autoconf

Использовать макросы autoconf очень просто: Добавьте "вызовы" макросов (на самом деле экземпляров) в `configure.ac`, запустите `aclocal`, и наконец, запустите `autoconf`. Если в вашей системе не установлен guile.m4, поместите нужные определения макросов (формы `AC_DEFUN`) в `acinclude.m4`, и `aclocal` будет делать правильные вещи.

Некоторые макросы могут использоваться внутри обычной командной оболочки: `if foo ; then GUILE_BAZ ; fi`, но это не гарантировано. Вероятно хорошей идеей будет создание экземпляра макросов на верхнем уровне.

Теперь мы включим два примера, один простой и один сложный.

Первый пример для пакета, который использует libguile, и следовательно должен знать, как компилировать и связывать с ней. Поэтому мы используем `PKG_CHECK_MODULES` чтобы установить переменные `GUILE_CFLAGS` и `GUILE_LIBS`, которые автоматически подставляются в Makefile.

In configure.ac:

```
PKG_CHECK_MODULES([GUILE], [guile-2.2])
```

In Makefile.in:

```
GUILE_CFLAGS  = @GUILE_CFLAGS@
GUILE_LIBS = @GUILE_LIBS@

myprog.o: myprog.c
        $(CC) -o $ $(GUILE_CFLAGS) $<
myprog: myprog.o
        $(CC) -o $ $< $(GUILE_LIBS)
```

Второй пример для пакета модулей Guile Scheme, который использует внешюю программу и другие модули Guile Scheme (некоторые могут назвать это пактом "чистой scheme"). Поэтому мы используем макрос the `GUILE_SITE_DIR`, обычный макрос `AC_PATH_PROG`, и макрос `GUILE_MODULE_AVAILABLE`.

In configure.ac:

```
GUILE_SITE_DIR

probably_wont_work=""

# pgtype pgtable
GUILE_MODULE_AVAILABLE(have_guile_pg, (database postgres))
test $have_guile_pg = no &&
    probably_wont_work="(my pgtype) (my pgtable) $probably_wont_work"

# gpgutils
AC_PATH_PROG(GNUPG,gpg)
test x"$GNUPG" = x &&
    probably_wont_work="(my gpgutils) $probably_wont_work"

if test ! "$probably_wont_work" = "" ; then
    p="         ***"
    echo
    echo "$p"
    echo "$p NOTE:"
    echo "$p The following modules probably won't work:"
    echo "$p   $probably_wont_work"
    echo "$p They can be installed anyway, and will work if their"
```

```
        echo "$p dependencies are installed later.  Please see README."
        echo "$p"
        echo
    fi
```

In Makefile.in:

```
    instdir = @GUILE_SITE@/my

    install:
          $(INSTALL) my/*.scm $(instdir)
```

6 API Reference

Guile предоставляет интерфейс прикладного программирования (*API*) для разработчиков на двух языках: Scheme и Си. В данной части руководства содержиться справочная документация для всех функций, доступных через интерфейсы обоих языков Scheme и Си.

6.1 Обзор Guile API

Интерфейс прикладного программирования(*API*) Guile обеспечивает доступную функциональность которую разработчик приложения может использовать в программировании на Си или Scheme. Интерфейс состоит из элементов *elements* которые могут быть маросами, функциями или переменными в Си, а также процедур, переменных, синтаксиса или других типов объектов в Scheme.

Многие элементы доступны как в Scheme так и для Си, в соответствующей форме. Например, процедура Scheme `assq` также доступна как `scm_assq` для языка Си. Эти элементы документируются только один раз, описывая их аспекты как в Scheme так и в Си.

Имя элемента Scheme регулярно связано с именем в Си. Кроме того, Си функция принимает свои параметры систематическим образом.

Как правило, имя Си функции может быть получено по ее имени в Scheme, использованием некоторых простых текстовых преобразований:

- Replace `-` (hyphen) with `_` (underscore).
- Replace `?` (question mark) with `_p`.
- Replace `!` (exclamation point) with `_x`.
- Replace internal `->` with `_to_`.
- Replace `<=` (less than or equal) with `_leq`.
- Replace `>=` (greater than or equal) with `_geq`.
- Replace `<` (less than) with `_less`.
- Replace `>` (greater than) with `_gr`.
- Prefix with `scm_`.

Функция Си всегда принимает фиксированное количество аргументов типа `SCM`, даже если соответствующая функция Scheme принимает переменное количество.

Для некоторых функций Scheme некоторые последние аргументы являются необязательными; соответствующая Си функция должна вызываться с указанием всех необязательных аргументов. Для получения эффекта вызова, при котором аргумент не был бы укзана, передавайте `SCM_UNDEFINED` как его значение. Вы не можете использовать этот аргумент в середине; когда один аргумент является `SCM_UNDEFINED` все последующие аргументы должны быть `SCM_UNDEFINED`.

Некоторые функции Scheme принимают произвольное количество *rest*(остальных) аргументов; соответстувющая Си функция должна вызываться со списком всех этих аргуметов. Этот список всегда последний аргумент Си функции.

Эти два вариантаа также могут быть объединены.

Тип возвращаемого значения функции Си, которая соответствует функции Scheme всегда `SCM`. Поэтому в описанниях ниже типы значений часто опускаются, как для возвращаемого значения так и для аргументов.

6.2 Deprecation

Время от времени функции и другие элементы Guile устаревают. Мехнизм Неодобрения(*deprecation*) Guile может вам помочь справиться с этим.

Когда вы используете функцию, которая устарела вы скорее всего, получите предупреждение во время выполнения. Кроме того, если у вас достаточно новый набор инструментов, использование устаревшей функции из `libguile` вызовет предупрежение во время ликовки(link-time).

Основной источник информации о том, какие интерфейсы устарели в данном релизе это файл `NEWS`. Этот файл также документирует, что вы должны использовать вместо устаревшей вещи.

Файл `README` содержит инструкуии о том, как контролировать включение или удаление устаревшей функции из публичного API Guile, и способы управления устаревшими функциями с помощью предупреждающих сообщений.

Идея этого механизма заключается в том, что обычно все устаревшие интерфейсы доступны, но вы получаете обратную связь при компиляции и запуске кода, который их использует. так что вы можете перейти на новый APIs когда появиться свободное время переписать старые вызовы.

6.3 Тип SCM

Guile представляет все значения Scheme одним Си типом `SCM`. Для введения в эту тему, См. Раздел 5.4.1 [Dynamic Types], страница 69.

`SCM` [C Type]

`SCM` это абстрактный Си тип пользовательского уровня, который используется для представления всех объектов Scheme в Guile, независимо от типа объекта Scheme. Нет операций Си, кроме присвоения гарантированно работающих с переменными типа `SCM`, поэтому вы должны использовать только макросы и функции для работы со значениями `SCM`. Значения конвертируются между типами данных Си и типом `SCM` с помощью функций утилит и макросов.

`scm_t_bits` [C Type]

`scm_t_bits` это беззнаковый обобщенный тип данных который гарантированно будет достаточно большим чтобы хранить всю информацию. которая потребуется для представления любого объекта Scheme. Хотя этот тип данных в основном используется для реализации внутренних функций Guile, испольование этого типа также необходимо для написания определенных видов расширений для Guile.

`scm_t_signed_bits` [C Type]

Это обобщенный тип со знаком, того же размера что и `scm_t_bits`.

`scm_t_bits SCM_UNPACK` (*SCM x*) [C Macro]

Преобразует `SCM` значение *x* в его представление в виде обобщенного типа. Только после применения `SCM_UNPACK` можно получить доступ к битам и содержимому значения `SCM`.

`SCM SCM_PACK` (*scm_t_bits x*) [C Macro]

Принимает правильное обобщенное представление объекта Scheme и преобразует его в представление в качестве значения `SCM`.

6.4 Инициализация Guile

Каждый поток, который хочет использовать функции из API Guile, должен поместить себя в guile режим с помощью вызова функции `scm_with_guile` или `scm_init_guile`. Глобальное состояние Guile инициализируется автоматически, когда первый поток входит в режим guile.

Когда поток хочет заблокировать все функции API Guile, он должен выйти временно из режима guile вызвав `scm_without_guile`, См. Раздел 6.22.6 [Blocking], страница 473.

Потоки созданные с помощью `call-with-new-thread` или `scm_spawn_thread` начинаются в режиме guile, поэтому вам не нужно его инициализировать.

`void * scm_with_guile` (*void *(*func)(void *), void *data*) [C Function]

Вызывает *func*, передавая ей данные *data* и возвращая то что возвращает функция *func*. Пока *func* выполняется, текущий поток находиться в режиме guilee и может использовать Guile API.

Когда `scm_with_guile` вызывается из режима guile, поток остается в режиме guile, когда `scm_with_guile` возвращается(завершается).

В противном случае, он переводит текущий поток в режим guile и, если необходимо, выдает ему представление Scheme которое содержит в списке возврат функции, например, `all-threads`. Это представление Scheme не удаляется, когда происходит возврат из `scm_with_guile`, так что данный поток вседа представлен одним и тем же значением Scheme в течении всего времени жизни(, если вообще).

Когда этот первый поток входит в режим guile, глобальное состояние Guile инициализируется перед вызовом `func`.

Функция *func* вызывается через `scm_with_continuation_barrier`; таким образом, `scm_with_guile` возвращается ровно один раз.

Когда `scm_with_guile` возвращается, поток больше не находиться в режиме guile (кроме случаев, когда `scm_with_guile` был вызван из режима guile, см выше). Таким образом, только `func` может хранить `SCM` переменные в стеке и быть уверенным что они защищены от сборщика мусора. См. `scm_init_guile` для другого подхода к инициализации Guile, который не имеет данных ограничений.

Это хорошо вызвать `scm_with_guile`, когда поток временно вышел из режима guile использовав `scm_without_guile`. Затем он просто опять временно войдет в режим guile.

`void scm_init_guile ()` [C Function]

Организует вещи так, чтобы весь код в текущем потоке выполнялся как будто внутри вызова `scm_with_guile`. То есть все функции вызываемые текущим потоком, могут полагать что значения `SCM` в их кадрах стека защищены от сборщика мусора (за исключением случаев, когда поток явно вышел из режима guile, коченчно).

Когда `scm_init_guile` вызывается из потока, который уже находиться в одном режиме guile, ничего не происходит. Такое поведение имеет значение, когда вы вызываете `scm_init_guile` в то время когда поток только временно вышел из режима guile: в этом случае поток не будет в режиме guile после возврата `scm_init_guile`. Таким образом, вы не должны использовать `scm_init_guile` по такому сценарию.

Когда в потоке, переведенном в режим guile `scm_init_guile`, происходит непрехваченное исключение, печатается короткое сообщение в текущий порт ошибки и поток завершается с помощью `scm_pthread_exit (NULL)`. На продолжения не накладывается никаких ограничений.

Функция `scm_init_guile` может быть недоступной на всех платформах, так как это требует некоторой магии поиска границ стека, которая могла бы быть перенесена на все платформы, на которых работает Guile. Таким образом, если вы можете, лучше использовать `scm_with_guile` или его вариант `scm_boot_guile` вместо этой функции.

`void scm_boot_guile (`*int* ***argc***, *char* ****argv***, *void* (****main_func***) (*void* ****data***, *int* ***argc***, *char* ****argv***), *void* ****data***`)` [C Function]

Входит в режим guile как с `scm_with_guile` и вызывает *main_func*, передавая ей *data*, *argc*, и *argv* как указано. Когда *main_func* возвращается, `scm_boot_guile` вызывает `exit (0)`; `scm_boot_guile` никогда не возвращается. Если вам нужно другое значение выхода, используйте в *main_func* вызов `exit` сами. Если вы не хотите выходить вообще, используйте `scm_with_guile` вместо `scm_boot_guile`.

Функция `scm_boot_guile` организует для функции `command-line` Scheme возврат строк переданных с помощью *argc* и *argv*. Если *main_func* изменяет *argc* или *argv*, она должна вызвать `scm_set_program_arguments` с окончательным списком, чтобы код Scheme узнал, какие аргументы необходимо обработать. (см. Раздел 7.2.6 [Runtime Environment], страница 540).

`void scm_shell (`*int* ***argc***, *char* ****argv***`)` [C Function]

Обрабатывает аргументы командной строки в манере исполняемого файла `guile`. Это включет загрузку обычных файлов инициализации Guile, взаимодействие с пользователем или запуск любых сценариев или выражений, заданных параметрами `-s` или `-e`, и затем выход. См. Раздел 4.2 [Invoking Guile], страница 37, для более подробной информации.

Поскольку эта фунция не возвращается, вы должны выполнить всю инициализацию для конкретного приложения перед вызовом этой функции.

6.5 Snarfing Macros

Следующие макросы выполняют две разные функции: при обычной компиляции, они расширяются одним способом; при обработке во время snarfing'a, они приводят к тому что программа `guile-snarf` подхватывает(присоединяет) немного инициализационного кода, См. Раздел 5.6 [Function Snarfing], страница 88.

В описаниях ниже используется термин 'обычно' для обозначения случая, когда код компилируется нормально, и 'во время snarfing'a', когда код обрабатывается программой `guile-snarf`.

`SCM_SNARF_INIT` (*code*) [C Macro]

Обычно, `SCM_SNARF_INIT` расширяется в ноль(т.е ничего не дает на выходе); во время snarfing, он заставляет включить *code* в файл действий по инициализации, после которого ставиться точка с запятой.

Это фундаментальный макрос для упрощения(snarfing) действий инициализации. Более специализированный макросы ниже используют его внутри себя.

`SCM_DEFINE` (*c_name, scheme_name, req, opt, var, arglist, docstring*) [C Macro]

Обычно, макрос расширяется в

```
static const char s_c_name[] = scheme_name;
SCM
c_name arglist
```

во время snarfing'a, он вызывает

```
scm_c_define_gsubr (s_c_name, req, opt, var,
                    c_name);
```

добавление к действиям инициализации. Таким образом, вы можете использовать его для объявления Си функции с именем *c_name*, которое будет доступно Scheme с именем *scheme_name*.

Обратите внимание, что аргумент *arglist* должен иметь круглые скобки вокруг себя.

`SCM_SYMBOL` (*c_name, scheme_name*) [C Macro]
`SCM_GLOBAL_SYMBOL` (*c_name, scheme_name*) [C Macro]

Обычно, эти макросы расширяются в

```
static SCM c_name
```

или

```
SCM c_name
```

соответственно. Во время snarfing'a, они оба расширяются в код инициализации

```
c_name = scm_permanent_object (scm_from_locale_symbol (scheme_name));
```

Таким образом, вы можете использовать их для объявления статической или глобальной переменной типа `SCM`, которая будет инициализироваться символом с именем *scheme_name*.

`SCM_KEYWORD` (*c_name, scheme_name*) [C Macro]
`SCM_GLOBAL_KEYWORD` (*c_name, scheme_name*) [C Macro]

Обычно, эти макросы расширяются в

```
static SCM c_name
```

или

```
SCM c_name
```

соответственно. Во время snarfing'а, они оба расширяются в код инициализации

```
c_name = scm_permanent_object (scm_c_make_keyword (scheme_name));
```

Таким образом, вы можете использовать их для объявления статической или глобальной переменной `SCM` типа, которая будет инициализироваться ключевым словом с именем *scheme_name*.

`SCM_VARIABLE` (*c_name*, *scheme_name*) [C Macro]
`SCM_GLOBAL_VARIABLE` (*c_name*, *scheme_name*) [C Macro]

Эти макросы эквивалентны макросам `SCM_VARIABLE_INIT` и `SCM_GLOBAL_VARIABLE_INIT`, соответственно, со значачением *value* равной `SCM_BOOL_F`.

`SCM_VARIABLE_INIT` (*c_name*, *scheme_name*, *value*) [C Macro]
`SCM_GLOBAL_VARIABLE_INIT` (*c_name*, *scheme_name*, *value*) [C Macro]

Обычно эти макросы расширяются в

```
static SCM c_name
```

или

```
SCM c_name
```

соответственно. А во время snarfing, они оба расширяются в код инициализации

```
c_name = scm_permanent_object (scm_c_define (scheme_name, value));
```

Таким образом, вы можете использовать их для объявления статической или глобальной Си переменной типа `SCM`, которая будет инициализирована для объекта представляющего переменную Scheme с именем *scheme_name* в текущем модуле. Переменная будет определена когда она еще не существует. Она всегда установлена в значение *value*.

6.6 Data Types

Типы данных Guile образуют мощную встроенную библиотеку представлений и функциональных возможностей, которые вы можете использовать в вашей проблемной области. В этой главе рассматриваются типы данных, встроенные в Guile, от простых до сложных.

6.6.1 Booleans

Два логических значения `#t` истина и `#f` ложь. Они также могут быть записаны как `#true` и `#false`, согласно R7RS.

Boolean values are returned by predicate procedures, such as the general equality predicates `eq?`, `eqv?` and `equal?` (см. Раздел 6.11.1 [Equality], страница 301) and numerical and string comparison operators like `string=?` (см. Раздел 6.6.5.7 [String Comparison], страница 158) and `<=` (см. Раздел 6.6.2.8 [Comparison], страница 126).

```
(<= 3 8)
⇒ #t

(<= 3 -3)
⇒ #f
```

```
(equal? "house" "houses")
⇒ #f

(eq? #f #f)
⇒
#t
```

In test condition contexts like `if` and `cond` (см. Раздел 6.13.2 [Conditionals], страница 318), where a group of subexpressions will be evaluated only if a *condition* expression evaluates to "true", "true" means any value at all except `#f`.

```
(if #t "yes" "no")
⇒ "yes"

(if 0 "yes" "no")
⇒ "yes"

(if #f "yes" "no")
⇒ "no"
```

Результатом этой асимметрии является то, что типичный исходный код Scheme чаще всего испльзует `#f` чем `#t`: `#f` необходима для представления ложного значения `if` или `cond`, тогда как `#t` не является необходимым для того чтобы представлять истинное значение в `if` или `cond`.

Важно отметить, что `#f` **НЕ** эквивалентен никакому другому значению Scheme. В частности, `#f` это не тоже самое, что число 0 (как в C и C++), и не тоже самое, что "пустой список" (как в некоторых диалектах Лиспа).

В Си, два логических значения Scheme доступны как две константы `SCM_BOOL_T` для `#t` и `SCM_BOOL_F` для `#f`. Следует соблюдать осторожность с ложным значением `SCM_BOOL_F`: это не false при использовании в условных выражениях Си. Чтобы проверить значение используйте `scm_is_false` или `scm_is_true`.

`not` *x* [Scheme Procedure]
`scm_not` (*x*) [C Function]
Return `#t` if *x* is `#f`, else return `#f`.

`boolean?` *obj* [Scheme Procedure]
`scm_boolean_p` (*obj*) [C Function]
Return `#t` if *obj* is either `#t` or `#f`, else return `#f`.

`SCM SCM_BOOL_T` [C Macro]
The `SCM` representation of the Scheme object `#t`.

`SCM SCM_BOOL_F` [C Macro]
The `SCM` representation of the Scheme object `#f`.

`int scm_is_true` (*SCM obj*) [C Function]
Return `0` if *obj* is `#f`, else return `1`.

`int scm_is_false` (*SCM obj*) [C Function]
Return `1` if *obj* is `#f`, else return `0`.

`int scm_is_bool` (*SCM obj*) [C Function]
Return `1` if *obj* is either `#t` or `#f`, else return `0`.

`SCM scm_from_bool` (*int val*) [C Function]
Return `#f` if *val* is `0`, else return `#t`.

`int scm_to_bool` (*SCM val*) [C Function]
Возвращает `1` если значение *val* равно `SCM_BOOL_T`, возвращает `0` если значение *val* равно `SCM_BOOL_F`, иначе сигнал ошибки 'wrong type'(неправильный тип).

Вы должны использовать `scm_is_true` вместо этой функции, когда хотите просто проверить истинность значения `SCM`.

6.6.2 Numerical data types

Guile поддерживает богатую "башню" из числовых типов - целые числа, рациональные, реальные и комплексные — и предоставляет обширный набор математических и научных функций для работы с числовыми данными. В этом разделе руководства указаны эти типы и функции.

Вы также можете обнаружить что раздел освещает R5RS's - представление чисел в Scheme, для большей ясности: см Раздел "Numbers" в *R5RS*.

6.6.2.1 Scheme's Numerical "Tower"

Числовая "башня" Scheme состоит из следующих числовых категорий:

integers Целые числа, положительные или отритцательные; например –5, 0, 18.

rationals Рациональные числа - набор чисел, который может быть выражен как p/q где p и q целые; например 9/16, но pi (иррациональное число) нельзя так представить. К ним относятся и целые числа, т.к ($n/1$).

real numbers
Реальные(Вещественные) числа - набор чисел, который описывает все возможные положения вдоль одномерной линии. Он включает в себя и рациональные и иррациональные числа.

complex numbers
Комплексные числа, набор чисел, который описывает все возможные положения в двумерном пространстве. Он включает как реальные так и мнимые числа ($a + bi$, где a это *real part(реальная часть)*, b это *imaginary part(мнимая часть)*, и i это квадратный корень из -1.)

Это называется башней, потому что каждая категория "сидит" над той, которая следует за ней, в том смысле, что целое число, также является рациональным, а все рациональные числа, также вещественны и каждое вещественное число одновременно и комплексное(но с нулевой мнимой частью)

В добавление к классификации в целые числа, рациональные, вещественные и комплексные числа, Scheme также различает, представлено ли число точно или нет. Например, результат $2\sin(\pi/4)$ точно $\sqrt{2}$, но Guile неможет представлять ни $\pi/4$ ни $\sqrt{2}$ точно. Вместо этого он хранит неточное приближение, используя Си тип `double`.

Guile может представлять точные рациональности любой величины, неточные рациональности которые описываеются в Си `double`, и неточные комплексные числа с типом Си `double` реальной и мнимой частью.

Предикат `number?` может быть применен к любому значению Scheme, чтобы выяснить является ли значение числом для любого из поддерживаемых числовых типов.

`number?` *obj* [Scheme Procedure]
`scm_number_p` (*obj*) [C Function]
Return `#t` if *obj* is any kind of number, else `#f`.

For example:

```
(number? 3)
⇒ #t

(number? "hello there!")
⇒ #f

(define pi 3.141592654)
(number? pi)
⇒ #t
```

`int scm_is_number` (*SCM obj*) [C Function]
This is equivalent to `scm_is_true (scm_number_p (obj))`.

Следующие несколько подразделов подробно описывают каждый из численных типов данных Guile.

6.6.2.2 Integers

Integers are whole numbers, that is numbers with no fractional part, such as 2, 83, and −3789.

Целые числа в Guile могут быть сколь угодно большими, как показано в следующем примере.

```
(define (factorial n)
  (let loop ((n n) (product 1))
    (if (= n 0)
        product
        (loop (- n 1) (* product n)))))

(factorial 3)
⇒ 6

(factorial 20)
⇒ 2432902008176640000

(- (factorial 45))
⇒ -119622220865480194561963161495657715064383733760000000000
```

Читатели, чьи имеющие знания в языках программирования низкого уровня (типа Си), где целые числа ограничены необходимостью умещаться в 4 или 8 байт памяти, могут найти это удивительным и подозревать, что представление целых чисел Guile неэффективно. Фактически, Guile достигает почти оптимального баланса удобства и эффективности, используя представление целых чисел, где это возможно, в виде удобном для работы компьютера(т.е в Си типах), и более общее представление, где требуемое число не соответсвует ограничениям используемого компьютера. Преобразованием между этими двумя представлениями происходит автоматически и полностью невидимо для программиста Scheme.

Си имеет множество разных целых типов, а Guile предлагает множество функций для преобразования между ними и представлением `SCM`. Например, Си `int` можно обрабатывать с помощью `scm_to_int` и `scm_from_int`. Guile также определяет несколько собственных целочисленных Си типов, чтобы помочь преодолевать различия между различными компьютерными системами.

C integer types that are not covered can be handled with the generic `scm_to_signed_integer` and `scm_from_signed_integer` for signed types, or with `scm_to_unsigned_integer` and `scm_from_unsigned_integer` for unsigned types.

Целые числа в Scheme могут быть точными и не точными. Например, число записываемое как `3.0` с явной десятичной точкой является неточным, но оно также является целым числом. Функции `integer?` и `scm_is_integer` сообщают истина для такого чилса, но функции `exact-integer?`, `scm_is_exact_integer`, `scm_is_signed_integer`, и `scm_is_unsigned_integer` разрешают работу только с точными числами и сообщают об ошибке. Аналогично, функции преобразования, такие как `scm_to_signed_integer` принимают только точные целые числа.

Обоснование такого поведения заключается в том. что неточность числа не должна быть устранена молча. Если вы хотите работать с неточным числом, вы можете явно вставить вызов `inexact->exact` или его эквилвалент в Си `scm_inexact_to_exact`. (Только неточные целые числа будут преобразованы этим вызовом в точные; нецелые числа надо преобразовывать почастям.)

`integer?` *x* [Scheme Procedure]
`scm_integer_p` (*x*) [C Function]

Return `#t` if *x* is an exact or inexact integer number, else return `#f`.

```
(integer? 487)
⇒ #t

(integer? 3.0)
⇒ #t

(integer? -3.4)
⇒ #f

(integer? +inf.0)
⇒ #f
```

`int scm_is_integer` (*SCM x*) [C Function]

This is equivalent to `scm_is_true (scm_integer_p (x))`.

`exact-integer?` *x* [Scheme Procedure]
`scm_exact_integer_p` (*x*) [C Function]

Return `#t` if *x* is an exact integer number, else return `#f`.

```
(exact-integer? 37)
⇒ #t

(exact-integer? 3.0)
⇒ #f
```

`int scm_is_exact_integer` (*SCM x*) [C Function]

This is equivalent to `scm_is_true (scm_exact_integer_p (x))`.

`scm_t_int8` [C Type]
`scm_t_uint8` [C Type]
`scm_t_int16` [C Type]
`scm_t_uint16` [C Type]
`scm_t_int32` [C Type]
`scm_t_uint32` [C Type]
`scm_t_int64` [C Type]
`scm_t_uint64` [C Type]
`scm_t_intmax` [C Type]
`scm_t_uintmax` [C Type]

The C types are equivalent to the corresponding ISO C types but are defined on all platforms, with the exception of `scm_t_int64` and `scm_t_uint64`, which are only defined when a 64-bit type is available. For example, `scm_t_int8` is equivalent to `int8_t`.

You can regard these definitions as a stop-gap measure until all platforms provide these types. If you know that all the platforms that you are interested in already provide these types, it is better to use them directly instead of the types provided by Guile.

`int scm_is_signed_integer` (*SCM x, scm_t_intmax min, scm_t_intmax max*) [C Function]
`int scm_is_unsigned_integer` (*SCM x, scm_t_uintmax min, scm_t_uintmax max*) [C Function]

Return `1` when *x* represents an exact integer that is between *min* and *max*, inclusive.

These functions can be used to check whether a `SCM` value will fit into a given range, such as the range of a given C integer type. If you just want to convert a `SCM` value to a given C integer type, use one of the conversion functions directly.

`scm_t_intmax scm_to_signed_integer` (*SCM x, scm_t_intmax min, scm_t_intmax max*) [C Function]
`scm_t_uintmax scm_to_unsigned_integer` (*SCM x, scm_t_uintmax min, scm_t_uintmax max*) [C Function]

When *x* represents an exact integer that is between *min* and *max* inclusive, return that integer. Else signal an error, either a 'wrong-type' error when *x* is not an exact integer, or an 'out-of-range' error when it doesn't fit the given range.

`SCM scm_from_signed_integer` (*scm_t_intmax x*) [C Function]
`SCM scm_from_unsigned_integer` (*scm_t_uintmax x*) [C Function]
Return the `SCM` value that represents the integer *x*. This function will always succeed and will always return an exact number.

`char scm_to_char` (*SCM x*) [C Function]
`signed char scm_to_schar` (*SCM x*) [C Function]
`unsigned char scm_to_uchar` (*SCM x*) [C Function]
`short scm_to_short` (*SCM x*) [C Function]
`unsigned short scm_to_ushort` (*SCM x*) [C Function]
`int scm_to_int` (*SCM x*) [C Function]
`unsigned int scm_to_uint` (*SCM x*) [C Function]
`long scm_to_long` (*SCM x*) [C Function]
`unsigned long scm_to_ulong` (*SCM x*) [C Function]
`long long scm_to_long_long` (*SCM x*) [C Function]
`unsigned long long scm_to_ulong_long` (*SCM x*) [C Function]
`size_t scm_to_size_t` (*SCM x*) [C Function]
`ssize_t scm_to_ssize_t` (*SCM x*) [C Function]
`scm_t_uintptr scm_to_uintptr_t` (*SCM x*) [C Function]
`scm_t_ptrdiff scm_to_ptrdiff_t` (*SCM x*) [C Function]
`scm_t_int8 scm_to_int8` (*SCM x*) [C Function]
`scm_t_uint8 scm_to_uint8` (*SCM x*) [C Function]
`scm_t_int16 scm_to_int16` (*SCM x*) [C Function]
`scm_t_uint16 scm_to_uint16` (*SCM x*) [C Function]
`scm_t_int32 scm_to_int32` (*SCM x*) [C Function]
`scm_t_uint32 scm_to_uint32` (*SCM x*) [C Function]
`scm_t_int64 scm_to_int64` (*SCM x*) [C Function]
`scm_t_uint64 scm_to_uint64` (*SCM x*) [C Function]
`scm_t_intmax scm_to_intmax` (*SCM x*) [C Function]
`scm_t_uintmax scm_to_uintmax` (*SCM x*) [C Function]
`scm_t_intptr scm_to_intptr_t` (*SCM x*) [C Function]
`scm_t_uintptr scm_to_uintptr_t` (*SCM x*) [C Function]
When *x* represents an exact integer that fits into the indicated C type, return that integer. Else signal an error, either a 'wrong-type' error when *x* is not an exact integer, or an 'out-of-range' error when it doesn't fit the given range.

The functions `scm_to_long_long`, `scm_to_ulong_long`, `scm_to_int64`, and `scm_to_uint64` are only available when the corresponding types are.

`SCM scm_from_char` (*char x*) [C Function]
`SCM scm_from_schar` (*signed char x*) [C Function]
`SCM scm_from_uchar` (*unsigned char x*) [C Function]
`SCM scm_from_short` (*short x*) [C Function]
`SCM scm_from_ushort` (*unsigned short x*) [C Function]
`SCM scm_from_int` (*int x*) [C Function]
`SCM scm_from_uint` (*unsigned int x*) [C Function]
`SCM scm_from_long` (*long x*) [C Function]
`SCM scm_from_ulong` (*unsigned long x*) [C Function]

`SCM scm_from_long_long` (*long long x*) [C Function]
`SCM scm_from_ulong_long` (*unsigned long long x*) [C Function]
`SCM scm_from_size_t` (*size_t x*) [C Function]
`SCM scm_from_ssize_t` (*ssize_t x*) [C Function]
`SCM scm_from_uintptr_t` (*uintptr_t x*) [C Function]
`SCM scm_from_ptrdiff_t` (*scm_t_ptrdiff x*) [C Function]
`SCM scm_from_int8` (*scm_t_int8 x*) [C Function]
`SCM scm_from_uint8` (*scm_t_uint8 x*) [C Function]
`SCM scm_from_int16` (*scm_t_int16 x*) [C Function]
`SCM scm_from_uint16` (*scm_t_uint16 x*) [C Function]
`SCM scm_from_int32` (*scm_t_int32 x*) [C Function]
`SCM scm_from_uint32` (*scm_t_uint32 x*) [C Function]
`SCM scm_from_int64` (*scm_t_int64 x*) [C Function]
`SCM scm_from_uint64` (*scm_t_uint64 x*) [C Function]
`SCM scm_from_intmax` (*scm_t_intmax x*) [C Function]
`SCM scm_from_uintmax` (*scm_t_uintmax x*) [C Function]
`SCM scm_from_intptr_t` (*scm_t_intptr x*) [C Function]
`SCM scm_from_uintptr_t` (*scm_t_uintptr x*) [C Function]

Return the `SCM` value that represents the integer *x*. These functions will always succeed and will always return an exact number.

`void scm_to_mpz` (*SCM val, mpz_t rop*) [C Function]

Assign *val* to the multiple precision integer *rop*. *val* must be an exact integer, otherwise an error will be signalled. *rop* must have been initialized with `mpz_init` before this function is called. When *rop* is no longer needed the occupied space must be freed with `mpz_clear`. См. Раздел “Initializing Integers” в *GNU MP Manual*, for details.

`SCM scm_from_mpz` (*mpz_t val*) [C Function]

Return the `SCM` value that represents *val*.

6.6.2.3 Real and Rational Numbers

Математически вещественные числа представляют собой набор чисел, описывающих все возможные точки вдоль непрерывной бесконечной одномерной линии. Рациональные числа - это совокупность всех чисел которые могут быть записаны как дроби p/q, где p и q целые числа. Все рациональные числа являются также и вещественными, но существуют вещественные числа, которые не являются рациональными, например: $\sqrt{2}$, и π.

Guile может представлять как точные, так и не точные рациональные числа, не не может представлять тоными конечные иррациональные числа. Точные рациональности представлены путем хранения числителя и знаменателя в виде двух точных целых чисел. Неточные рациональности храняться как числа с плавающей запятой, использущие тип Си `double`.

Точные рациональные числа записываются как часть целых чисел. Не должно быть пробелов вокруг косой черты.

```
1/2
-22/7
```

Несмотря на то, что фактическое кодирование неточных рациональных чисел находиться в двоичном представлении, может оказаться полезным думать о нем как о десятичном числе с ограниченным числом значимых цифр и десятичной точкой стоящей гдето, так как это соответствует стандартной нотации для нецелых чисел. Например:

```
0.34
-0.00000142857931198
-5648394822220000000000.0
4.0
```

Ограниченная точность кодирования Guile's означает, что любое конечное "реальное" число в Guile может быть записано в рациональной форме, умножая и деля на достаточные степени 10 (или фактически, 2). Например, '`-0.00000142857931198`' это тоже, что и −142857931198 деленное на 100000000000000000. В текущей реализации Guile's , следовательно, предикаты `rational?` и `real?` эквивалентны для конечных чисел.

Dividing by an exact zero leads to a error message, as one might expect. However, dividing by an inexact zero does not produce an error. Instead, the result of the division is either plus or minus infinity, depending on the sign of the divided number and the sign of the zero divisor (some platforms support signed zeroes '`-0.0`' and '`+0.0`'; '`0.0`' is the same as '`+0.0`').

Dividing zero by an inexact zero yields a NaN ('not a number') value, although they are actually considered numbers by Scheme. Attempts to compare a NaN value with any number (including itself) using `=`, `<`, `>`, `<=` or `>=` always returns `#f`. Although a NaN value is not `=` to itself, it is both `eqv?` and `equal?` to itself and other NaN values. However, the preferred way to test for them is by using `nan?`.

The real NaN values and infinities are written '`+nan.0`', '`+inf.0`' and '`-inf.0`'. This syntax is also recognized by `read` as an extension to the usual Scheme syntax. These special values are considered by Scheme to be inexact real numbers but not rational. Note that non-real complex numbers may also contain infinities or NaN values in their real or imaginary parts. To test a real number to see if it is infinite, a NaN value, or neither, use `inf?`, `nan?`, or `finite?`, respectively. Every real number in Scheme belongs to precisely one of those three classes.

On platforms that follow IEEE 754 for their floating point arithmetic, the '`+inf.0`', '`-inf.0`', and '`+nan.0`' values are implemented using the corresponding IEEE 754 values. They behave in arithmetic operations like IEEE 754 describes it, i.e., `(= +nan.0 +nan.0)` ⇒ `#f`.

`real?` *obj* [Scheme Procedure]
`scm_real_p` (*obj*) [C Function]

Return `#t` if *obj* is a real number, else `#f`. Note that the sets of integer and rational values form subsets of the set of real numbers, so the predicate will also be fulfilled if *obj* is an integer number or a rational number.

`rational?` *x* [Scheme Procedure]
`scm_rational_p` (*x*) [C Function]
Return `#t` if *x* is a rational number, `#f` otherwise. Note that the set of integer values forms a subset of the set of rational numbers, i.e. the predicate will also be fulfilled if *x* is an integer number.

`rationalize` *x eps* [Scheme Procedure]
`scm_rationalize` (*x*, *eps*) [C Function]
Returns the *simplest* rational number differing from *x* by no more than *eps*.

As required by R5RS, `rationalize` only returns an exact result when both its arguments are exact. Thus, you might need to use `inexact->exact` on the arguments.

```
(rationalize (inexact->exact 1.2) 1/100)
⇒ 6/5
```

`inf?` *x* [Scheme Procedure]
`scm_inf_p` (*x*) [C Function]
Return `#t` if the real number *x* is '`+inf.0`' or '`-inf.0`'. Otherwise return `#f`.

`nan?` *x* [Scheme Procedure]
`scm_nan_p` (*x*) [C Function]
Return `#t` if the real number *x* is '`+nan.0`', or `#f` otherwise.

`finite?` *x* [Scheme Procedure]
`scm_finite_p` (*x*) [C Function]
Return `#t` if the real number *x* is neither infinite nor a NaN, `#f` otherwise.

`nan` [Scheme Procedure]
`scm_nan` () [C Function]
Return '`+nan.0`', a NaN value.

`inf` [Scheme Procedure]
`scm_inf` () [C Function]
Return '`+inf.0`', positive infinity.

`numerator` *x* [Scheme Procedure]
`scm_numerator` (*x*) [C Function]
Return the numerator of the rational number *x*.

`denominator` *x* [Scheme Procedure]
`scm_denominator` (*x*) [C Function]
Return the denominator of the rational number *x*.

`int scm_is_real` (*SCM val*) [C Function]
`int scm_is_rational` (*SCM val*) [C Function]
Equivalent to `scm_is_true (scm_real_p (val))` and `scm_is_true (scm_rational_p (val))`, respectively.

`double scm_to_double` (*SCM val*) [C Function]
Returns the number closest to *val* that is representable as a `double`. Returns infinity for a *val* that is too large in magnitude. The argument *val* must be a real number.

`SCM scm_from_double` (*double val*) [C Function]
Return the `SCM` value that represents *val*. The returned value is inexact according to the predicate `inexact?`, but it will be exactly equal to *val*.

6.6.2.4 Complex Numbers

Комплексные числа - это набор чисел, описывающих всевозможные точки в двумерном пространстве. Известные две координаты конкретной точки в этом пространстве называются как *real(реальная)* и *imaginary(мнимая)* части комплексного числа, которое описывает эту точку.

В Guile, комплексные числа записываются в прямоугольной форме как сумма их реальной и мнимой частей, используя символ `i` для обозначения мнимой части.

```
3+4i
⇒
3.0+4.0i

(* 3-8i 2.3+0.3i)
⇒
9.3-17.5i
```

Также может использоваться Полярная форма с симовлом '`@`' между величиной и углом.

```
1@3.141592 ⇒ -1.0      (approx)
-1@1.57079 ⇒ 0.0-1.0i  (approx)
```

Guile представляет комплексное число как пару неточных действительных чисел, поэтому реальная и мнимая части комплексного числа имеют одинаковые свойства неточности и ограниченной точности как единичные неточные действительные числа.

Обратите внимание, что каждая часть комплексного числа может содержать любое неточное реальное значение, включая специальные значения '`+nan.0`', '`+inf.0`' и '`-inf.0`', а также любой из нулей со знаком '`0.0`' или '`-0.0`'.

`complex?` *z* [Scheme Procedure]
`scm_complex_p` (*z*) [C Function]
Return `#t` if *z* is a complex number, `#f` otherwise. Note that the sets of real, rational and integer values form subsets of the set of complex numbers, i.e. the predicate will also be fulfilled if *z* is a real, rational or integer number.

`int scm_is_complex` (*SCM val*) [C Function]
Equivalent to `scm_is_true (scm_complex_p (val))`.

6.6.2.5 Exact and Inexact Numbers

R5RS требует чтобы, за небольшим исключением, вычисления с использоанием неточных чисел всегда давали неточный результат. Чтобы соответствовать этому требованию, Guile различает точное целочисленное значение, такое как '`5`' и соотвественно, неточное целочисленное значение, ограниченной точности, не имеющее дробной части и печатающееся как '`5.0`'. Guile конвертирует последнее значение в первое только, когда оно вынуждено делать это путем вызова процедуры `inexact->exact`.

Единственное исключение из вышеуказанного требования - когда значения неточных чисел не влияют на результат. Например `(expt n 0)` равно '1' для любого значения `n`, поэтому `(expt 5.0 0)` разрешено возвращать точное '1'.

`exact?` *z* [Scheme Procedure]
`scm_exact_p` (*z*) [C Function]
Return `#t` if the number *z* is exact, `#f` otherwise.

```
(exact? 2)
⇒ #t

(exact? 0.5)
⇒ #f

(exact? (/ 2))
⇒ #t
```

`int scm_is_exact` (*SCM z*) [C Function]
Return a `1` if the number *z* is exact, and `0` otherwise. This is equivalent to `scm_is_true (scm_exact_p (z))`.

An alternate approch to testing the exactness of a number is to use `scm_is_signed_integer` or `scm_is_unsigned_integer`.

`inexact?` *z* [Scheme Procedure]
`scm_inexact_p` (*z*) [C Function]
Return `#t` if the number *z* is inexact, `#f` else.

`int scm_is_inexact` (*SCM z*) [C Function]
Return a `1` if the number *z* is inexact, and `0` otherwise. This is equivalent to `scm_is_true (scm_inexact_p (z))`.

`inexact->exact` *z* [Scheme Procedure]
`scm_inexact_to_exact` (*z*) [C Function]
Return an exact number that is numerically closest to *z*, when there is one. For inexact rationals, Guile returns the exact rational that is numerically equal to the inexact rational. Inexact complex numbers with a non-zero imaginary part can not be made exact.

```
(inexact->exact 0.5)
⇒ 1/2
```

The following happens because 12/10 is not exactly representable as a `double` (on most platforms). However, when reading a decimal number that has been marked exact with the "#e" prefix, Guile is able to represent it correctly.

```
(inexact->exact 1.2)
⇒ 5404319552844595/4503599627370496

#e1.2
⇒ 6/5
```

`exact->inexact` *z* [Scheme Procedure]
`scm_exact_to_inexact` (*z*) [C Function]

Convert the number *z* to its inexact representation.

6.6.2.6 Read Syntax for Numerical Data

Синтаксис чтения целых чисел представляет собой строку цифр, необязательно предшествует которой символ плюс или минус, код указывающий базу системы счисления, в которой закодировано число и код указывающий является ли число точным или неточным. Поддерживаемые базовые коды:

`#b`
`#B` целое записанное в бинарном формате (база 2)

`#o`
`#O` целое записанное в восмеричном формате (base 8)

`#d`
`#D` целое записанное в десятичном формате (base 10)

`#x`
`#X` целое записанное в шестнадцатеричном формате (base 16)

Если код базы опущен, целое число считается десятичным. Следующие примеры покажут, как использовать эти базовые коды.

```
-13
⇒ -13

#d-13
⇒ -13

#x-13
⇒ -19

#b+1101
⇒ 13

#o377
⇒ 255
```

Коды для указания точности (которые, кстати, могут применяться ко всем числовым значениям):

`#e`
`#E` the number is exact

`#i`
`#I` the number is inexact.

Если индикатор точности опущен, это число точно, если оно не содержит точку. Поскольку Guile не может представлять точные комплексные числа, при запросе о точности комплексного числа возникает ошибка.

```
(exact? 1.2)
```

```
⇒ #f

(exact? #e1.2)
⇒ #t

(exact? #e+1i)
ERROR: Wrong type argument
```

Guile also understands the syntax '`+inf.0`' and '`-inf.0`' for plus and minus infinity, respectively. The value must be written exactly as shown, that is, they always must have a sign and exactly one zero digit after the decimal point. It also understands '`+nan.0`' and '`-nan.0`' for the special 'not-a-number' value. The sign is ignored for 'not-a-number' and the value is always printed as '`+nan.0`'.

6.6.2.7 Operations on Integer Values

`odd?` *n* [Scheme Procedure]
`scm_odd_p` (*n*) [C Function]

Return `#t` if *n* is an odd number, `#f` otherwise.

`even?` *n* [Scheme Procedure]
`scm_even_p` (*n*) [C Function]

Return `#t` if *n* is an even number, `#f` otherwise.

`quotient` *n d* [Scheme Procedure]
`remainder` *n d* [Scheme Procedure]
`scm_quotient` (*n*, *d*) [C Function]
`scm_remainder` (*n*, *d*) [C Function]

Return the quotient or remainder from *n* divided by *d*. The quotient is rounded towards zero, and the remainder will have the same sign as *n*. In all cases quotient and remainder satisfy $n = q * d + r$.

```
(remainder 13 4) ⇒ 1
(remainder -13 4) ⇒ -1
```

See also `truncate-quotient`, `truncate-remainder` and related operations in Раздел 6.6.2.11 [Arithmetic], страница 128.

`modulo` *n d* [Scheme Procedure]
`scm_modulo` (*n*, *d*) [C Function]

Return the remainder from *n* divided by *d*, with the same sign as *d*.

```
(modulo 13 4) ⇒ 1
(modulo -13 4) ⇒ 3
(modulo 13 -4) ⇒ -3
(modulo -13 -4) ⇒ -1
```

See also `floor-quotient`, `floor-remainder` and related operations in Раздел 6.6.2.11 [Arithmetic], страница 128.

`gcd` *x...* [Scheme Procedure]
`scm_gcd` (*x*, *y*) [C Function]

Return the greatest common divisor of all arguments. If called without arguments, 0 is returned.

The C function `scm_gcd` always takes two arguments, while the Scheme function can take an arbitrary number.

`lcm` *x...* [Scheme Procedure]
`scm_lcm` (*x*, *y*) [C Function]
Return the least common multiple of the arguments. If called without arguments, 1 is returned.

The C function `scm_lcm` always takes two arguments, while the Scheme function can take an arbitrary number.

`modulo-expt` *n k m* [Scheme Procedure]
`scm_modulo_expt` (*n*, *k*, *m*) [C Function]
Return *n* raised to the integer exponent *k*, modulo *m*.

```
(modulo-expt 2 3 5)
   ⇒ 3
```

`exact-integer-sqrt` ***k*** [Scheme Procedure]
`void scm_exact_integer_sqrt` (*SCM **k**, SCM *__s__, SCM *__r__*) [C Function]
Return two exact non-negative integers *s* and *r* such that $k = s^2 + r$ and $s^2 <= k < (s+1)^2$. An error is raised if *k* is not an exact non-negative integer.

```
(exact-integer-sqrt 10) ⇒ 3 and 1
```

6.6.2.8 Comparison Predicates

Фукнции сравнения Си ниже всегда принимают два аргумента, а функции Scheme могут принимать произвольное число аргументов. Также имейте в виду, что функции Си возвращают один из буферных значений Scheme `SCM_BOOL_T` или `SCM_BOOL_F`, которые являются для Си истинными(что не верно). Таким образом, всегда пишите `scm_is_true (scm_num_eq_p (x, y))` при сравнении двух величин Scheme, например для выяснения равенства `x` и `y`.

`=` [Scheme Procedure]
`scm_num_eq_p` (*x*, *y*) [C Function]
Return `#t` if all parameters are numerically equal.

`<` [Scheme Procedure]
`scm_less_p` (*x*, *y*) [C Function]
Return `#t` if the list of parameters is monotonically increasing.

`>` [Scheme Procedure]
`scm_gr_p` (*x*, *y*) [C Function]
Return `#t` if the list of parameters is monotonically decreasing.

`<=` [Scheme Procedure]
`scm_leq_p` (*x*, *y*) [C Function]
Return `#t` if the list of parameters is monotonically non-decreasing.

`>=` [Scheme Procedure]
`scm_geq_p` (*x*, *y*) [C Function]
Return `#t` if the list of parameters is monotonically non-increasing.

`zero?` *z* [Scheme Procedure]
`scm_zero_p` (*z*) [C Function]
Return `#t` if *z* is an exact or inexact number equal to zero.

`positive?` *x* [Scheme Procedure]
`scm_positive_p` (*x*) [C Function]
Return `#t` if *x* is an exact or inexact number greater than zero.

`negative?` *x* [Scheme Procedure]
`scm_negative_p` (*x*) [C Function]
Return `#t` if *x* is an exact or inexact number less than zero.

6.6.2.9 Converting Numbers To and From Strings

Следующие процедуры считывают и записывают числа в соответствии с их внешним представлением как определено в R5RS (см. Раздел "Lexical structure" в *The Revised^5 Report on the Algorithmic Language Scheme*). См. Раздел 6.25.4 [Number Input and Output], страница 489, для разбора числа, зависящего от текущей локали.

`number->string` *n* [*radix*] [Scheme Procedure]
`scm_number_to_string` (*n, radix*) [C Function]
Return a string holding the external representation of the number *n* in the given *radix*. If *n* is inexact, a radix of 10 will be used.

`string->number` *string* [*radix*] [Scheme Procedure]
`scm_string_to_number` (*string, radix*) [C Function]
Return a number of the maximally precise representation expressed by the given *string*. *radix* must be an exact integer, either 2, 8, 10, or 16. If supplied, *radix* is a default radix that may be overridden by an explicit radix prefix in *string* (e.g. `"#o177"`). If *radix* is not supplied, then the default radix is 10. If string is not a syntactically valid notation for a number, then `string->number` returns `#f`.

`SCM scm_c_locale_stringn_to_number` (*const char *string, size_t len, unsigned radix*) [C Function]
As per `string->number` above, but taking a C string, as pointer and length. The string characters should be in the current locale encoding (`locale` in the name refers only to that, there's no locale-dependent parsing).

6.6.2.10 Complex Number Operations

`make-rectangular` *real_part imaginary_part* [Scheme Procedure]
`scm_make_rectangular` (*real_part, imaginary_part*) [C Function]
Return a complex number constructed of the given *real-part* and *imaginary-part* parts.

`make-polar` *mag ang* [Scheme Procedure]
`scm_make_polar` (*mag, ang*) [C Function]
Return the complex number *mag* * e^(i * *ang*).

`real-part` *z* [Scheme Procedure]
`scm_real_part` (*z*) [C Function]
Return the real part of the number *z*.

`imag-part` *z* [Scheme Procedure]
`scm_imag_part` (*z*) [C Function]
Return the imaginary part of the number *z*.

`magnitude` *z* [Scheme Procedure]
`scm_magnitude` (*z*) [C Function]
Return the magnitude of the number *z*. This is the same as `abs` for real arguments, but also allows complex numbers.

`angle` *z* [Scheme Procedure]
`scm_angle` (*z*) [C Function]
Return the angle of the complex number *z*.

`SCM scm_c_make_rectangular` (*double re, double im*) [C Function]
`SCM scm_c_make_polar` (*double x, double y*) [C Function]
Like `scm_make_rectangular` or `scm_make_polar`, respectively, but these functions take `double`s as their arguments.

`double scm_c_real_part` (*z*) [C Function]
`double scm_c_imag_part` (*z*) [C Function]
Returns the real or imaginary part of *z* as a `double`.

`double scm_c_magnitude` (*z*) [C Function]
`double scm_c_angle` (*z*) [C Function]
Returns the magnitude or angle of *z* as a `double`.

6.6.2.11 Arithmetic Functions

Арфиметические функции Си ниже всегда принимают два аргумента, а функции Scheme могут принимать произвольное число аргументов. Когда вам нужно вызвать их только с одним аргуметом, например для вычисления эквивалента `(- x)`, передайте `SCM_UNDEFINED` вторым аргументом, вот так: `scm_difference (x, SCM_UNDEFINED)`.

`+` *z1 ...* [Scheme Procedure]
`scm_sum` (*z1, z2*) [C Function]
Return the sum of all parameter values. Return 0 if called without any parameters.

`-` *z1 z2 ...* [Scheme Procedure]
`scm_difference` (*z1, z2*) [C Function]
If called with one argument *z1*, *-z1* is returned. Otherwise the sum of all but the first argument are subtracted from the first argument.

`*` *z1 ...* [Scheme Procedure]
`scm_product` (*z1, z2*) [C Function]
Return the product of all arguments. If called without arguments, 1 is returned.

`/` *z1 z2 ...* [Scheme Procedure]
`scm_divide` (*z1, z2*) [C Function]
Divide the first argument by the product of the remaining arguments. If called with one argument *z1*, $1/z1$ is returned.

`1+` *z* [Scheme Procedure]
`scm_oneplus` (*z*) [C Function]
Return $z + 1$.

`1-` *z* [Scheme Procedure]
`scm_oneminus` (*z*) [C function]
Return $z - 1$.

`abs` *x* [Scheme Procedure]
`scm_abs` (*x*) [C Function]
Return the absolute value of *x*.

x must be a number with zero imaginary part. To calculate the magnitude of a complex number, use `magnitude` instead.

`max` *x1 x2 ...* [Scheme Procedure]
`scm_max` (*x1, x2*) [C Function]
Return the maximum of all parameter values.

`min` *x1 x2 ...* [Scheme Procedure]
`scm_min` (*x1, x2*) [C Function]
Return the minimum of all parameter values.

`truncate` *x* [Scheme Procedure]
`scm_truncate_number` (*x*) [C Function]
Round the inexact number *x* towards zero.

`round` *x* [Scheme Procedure]
`scm_round_number` (*x*) [C Function]
Round the inexact number *x* to the nearest integer. When exactly halfway between two integers, round to the even one.

`floor` *x* [Scheme Procedure]
`scm_floor` (*x*) [C Function]
Round the number *x* towards minus infinity.

`ceiling` *x* [Scheme Procedure]
`scm_ceiling` (*x*) [C Function]
Round the number *x* towards infinity.

`double scm_c_truncate` (*double x*) [C Function]
`double scm_c_round` (*double x*) [C Function]
Like `scm_truncate_number` or `scm_round_number`, respectively, but these functions take and return `double` values.

`euclidean/` ***x y*** [Scheme Procedure]
`euclidean-quotient` ***x y*** [Scheme Procedure]
`euclidean-remainder` ***x y*** [Scheme Procedure]
`void scm_euclidean_divide` (*SCM x, SCM y, SCM *q, SCM *r*) [C Function]
`SCM scm_euclidean_quotient` (*SCM x, SCM y*) [C Function]

`SCM scm_euclidean_remainder (SCM x, SCM y)` [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `euclidean-quotient` returns the integer q and `euclidean-remainder` returns the real number r such that $x = q * y + r$ and $0 <= r < |y|$. `euclidean/` returns both q and r, and is more efficient than computing each separately. Note that when $y > 0$, `euclidean-quotient` returns $floor(x/y)$, otherwise it returns $ceiling(x/y)$.

Note that these operators are equivalent to the R6RS operators `div`, `mod`, and `div-and-mod`.

```
(euclidean-quotient 123 10) ⇒ 12
(euclidean-remainder 123 10) ⇒ 3
(euclidean/ 123 10) ⇒ 12 and 3
(euclidean/ 123 -10) ⇒ -12 and 3
(euclidean/ -123 10) ⇒ -13 and 7
(euclidean/ -123 -10) ⇒ 13 and 7
(euclidean/ -123.2 -63.5) ⇒ 2.0 and 3.8
(euclidean/ 16/3 -10/7) ⇒ -3 and 22/21
```

`floor/ x y` [Scheme Procedure]
`floor-quotient x y` [Scheme Procedure]
`floor-remainder x y` [Scheme Procedure]
`void scm_floor_divide (SCM x, SCM y, SCM *q, SCM *r)` [C Function]
`SCM scm_floor_quotient (x, y)` [C Function]
`SCM scm_floor_remainder (x, y)` [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `floor-quotient` returns the integer q and `floor-remainder` returns the real number r such that $q = floor(x/y)$ and $x = q * y + r$. `floor/` returns both q and r, and is more efficient than computing each separately. Note that r, if non-zero, will have the same sign as y.

When x and y are integers, `floor-remainder` is equivalent to the R5RS integer-only operator `modulo`.

```
(floor-quotient 123 10) ⇒ 12
(floor-remainder 123 10) ⇒ 3
(floor/ 123 10) ⇒ 12 and 3
(floor/ 123 -10) ⇒ -13 and -7
(floor/ -123 10) ⇒ -13 and 7
(floor/ -123 -10) ⇒ 12 and -3
(floor/ -123.2 -63.5) ⇒ 1.0 and -59.7
(floor/ 16/3 -10/7) ⇒ -4 and -8/21
```

`ceiling/ x y` [Scheme Procedure]
`ceiling-quotient x y` [Scheme Procedure]
`ceiling-remainder x y` [Scheme Procedure]
`void scm_ceiling_divide (SCM x, SCM y, SCM *q, SCM *r)` [C Function]
`SCM scm_ceiling_quotient (x, y)` [C Function]
`SCM scm_ceiling_remainder (x, y)` [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `ceiling-quotient` returns the integer q and `ceiling-remainder` returns

the real number r such that $q = ceiling(x/y)$ and $x = q * y + r$. `ceiling/` returns both q and r, and is more efficient than computing each separately. Note that r, if non-zero, will have the opposite sign of y.

```
(ceiling-quotient 123 10) ⇒ 13
(ceiling-remainder 123 10) ⇒ -7
(ceiling/ 123 10) ⇒ 13 and -7
(ceiling/ 123 -10) ⇒ -12 and 3
(ceiling/ -123 10) ⇒ -12 and -3
(ceiling/ -123 -10) ⇒ 13 and 7
(ceiling/ -123.2 -63.5) ⇒ 2.0 and 3.8
(ceiling/ 16/3 -10/7) ⇒ -3 and 22/21
```

`truncate/` *x y* [Scheme Procedure]
`truncate-quotient` *x y* [Scheme Procedure]
`truncate-remainder` *x y* [Scheme Procedure]
`void scm_truncate_divide` (*SCM x*, *SCM y*, *SCM *q*, *SCM *r*) [C Function]
`SCM scm_truncate_quotient` (*x*, *y*) [C Function]
`SCM scm_truncate_remainder` (*x*, *y*) [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `truncate-quotient` returns the integer q and `truncate-remainder` returns the real number r such that q is x/y rounded toward zero, and $x = q * y + r$. `truncate/` returns both q and r, and is more efficient than computing each separately. Note that r, if non-zero, will have the same sign as x.

When x and y are integers, these operators are equivalent to the R5RS integer-only operators `quotient` and `remainder`.

```
(truncate-quotient 123 10) ⇒ 12
(truncate-remainder 123 10) ⇒ 3
(truncate/ 123 10) ⇒ 12 and 3
(truncate/ 123 -10) ⇒ -12 and 3
(truncate/ -123 10) ⇒ -12 and -3
(truncate/ -123 -10) ⇒ 12 and -3
(truncate/ -123.2 -63.5) ⇒ 1.0 and -59.7
(truncate/ 16/3 -10/7) ⇒ -3 and 22/21
```

`centered/` *x y* [Scheme Procedure]
`centered-quotient` *x y* [Scheme Procedure]
`centered-remainder` *x y* [Scheme Procedure]
`void scm_centered_divide` (*SCM x*, *SCM y*, *SCM *q*, *SCM *r*) [C Function]
`SCM scm_centered_quotient` (*SCM x*, *SCM y*) [C Function]
`SCM scm_centered_remainder` (*SCM x*, *SCM y*) [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `centered-quotient` returns the integer q and `centered-remainder` returns the real number r such that $x = q * y + r$ and $-|y/2| <= r < |y/2|$. `centered/` returns both q and r, and is more efficient than computing each separately.

Note that `centered-quotient` returns x/y rounded to the nearest integer. When x/y lies exactly half-way between two integers, the tie is broken according to the sign of y. If $y > 0$, ties are rounded toward positive infinity, otherwise they are

rounded toward negative infinity. This is a consequence of the requirement that $-|y/2| <= r < |y/2|$.

Note that these operators are equivalent to the R6RS operators `div0`, `mod0`, and `div0-and-mod0`.

```
(centered-quotient 123 10) ⇒ 12
(centered-remainder 123 10) ⇒ 3
(centered/ 123 10) ⇒ 12 and 3
(centered/ 123 -10) ⇒ -12 and 3
(centered/ -123 10) ⇒ -12 and -3
(centered/ -123 -10) ⇒ 12 and -3
(centered/ 125 10) ⇒ 13 and -5
(centered/ 127 10) ⇒ 13 and -3
(centered/ 135 10) ⇒ 14 and -5
(centered/ -123.2 -63.5) ⇒ 2.0 and 3.8
(centered/ 16/3 -10/7) ⇒ -4 and -8/21
```

`round/` *x y* [Scheme Procedure]
`round-quotient` *x y* [Scheme Procedure]
`round-remainder` *x y* [Scheme Procedure]
`void scm_round_divide` (*SCM x, SCM y, SCM *q, SCM *r*) [C Function]
`SCM scm_round_quotient` (*x*, *y*) [C Function]
`SCM scm_round_remainder` (*x*, *y*) [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `round-quotient` returns the integer q and `round-remainder` returns the real number r such that $x = q * y + r$ and q is x/y rounded to the nearest integer, with ties going to the nearest even integer. `round/` returns both q and r, and is more efficient than computing each separately.

Note that `round/` and `centered/` are almost equivalent, but their behavior differs when x/y lies exactly half-way between two integers. In this case, `round/` chooses the nearest even integer, whereas `centered/` chooses in such a way to satisfy the constraint $-|y/2| <= r < |y/2|$, which is stronger than the corresponding constraint for `round/`, $-|y/2| <= r <= |y/2|$. In particular, when x and y are integers, the number of possible remainders returned by `centered/` is $|y|$, whereas the number of possible remainders returned by `round/` is $|y| + 1$ when y is even.

```
(round-quotient 123 10) ⇒ 12
(round-remainder 123 10) ⇒ 3
(round/ 123 10) ⇒ 12 and 3
(round/ 123 -10) ⇒ -12 and 3
(round/ -123 10) ⇒ -12 and -3
(round/ -123 -10) ⇒ 12 and -3
(round/ 125 10) ⇒ 12 and 5
(round/ 127 10) ⇒ 13 and -3
(round/ 135 10) ⇒ 14 and -5
(round/ -123.2 -63.5) ⇒ 2.0 and 3.8
(round/ 16/3 -10/7) ⇒ -4 and -8/21
```

6.6.2.12 Scientific Functions

Следующие процедуры принимают любое число в качестве аргументов, включая комплексные числа.

`sqrt` *z* [Scheme Procedure]
Return the square root of *z*. Of the two possible roots (positive and negative), the one with a positive real part is returned, or if that's zero then a positive imaginary part. Thus,

```
(sqrt 9.0)       ⇒ 3.0
(sqrt -9.0)      ⇒ 0.0+3.0i
(sqrt 1.0+1.0i)  ⇒ 1.09868411346781+0.455089860562227i
(sqrt -1.0-1.0i) ⇒ 0.455089860562227-1.09868411346781i
```

`expt` *z1 z2* [Scheme Procedure]
Return *z1* raised to the power of *z2*.

`sin` *z* [Scheme Procedure]
Return the sine of *z*.

`cos` *z* [Scheme Procedure]
Return the cosine of *z*.

`tan` *z* [Scheme Procedure]
Return the tangent of *z*.

`asin` *z* [Scheme Procedure]
Return the arcsine of *z*.

`acos` *z* [Scheme Procedure]
Return the arccosine of *z*.

`atan` *z* [Scheme Procedure]
`atan` *y x* [Scheme Procedure]
Return the arctangent of *z*, or of y/x.

`exp` *z* [Scheme Procedure]
Return e to the power of *z*, where e is the base of natural logarithms (2.71828...).

`log` *z* [Scheme Procedure]
Return the natural logarithm of *z*.

`log10` *z* [Scheme Procedure]
Return the base 10 logarithm of *z*.

`sinh` *z* [Scheme Procedure]
Return the hyperbolic sine of *z*.

`cosh` *z* [Scheme Procedure]
Return the hyperbolic cosine of *z*.

`tanh` *z* [Scheme Procedure]
Return the hyperbolic tangent of *z*.

`asinh` *z* [Scheme Procedure]
Return the hyperbolic arcsine of *z*.

`acosh` *z* [Scheme Procedure]
Return the hyperbolic arccosine of *z*.

`atanh` *z* [Scheme Procedure]
Return the hyperbolic arctangent of *z*.

6.6.2.13 Bitwise Operations

Для следующих побитовых функций отритцательные числа рассматриваются как двоичные дополнения бесконечной точности. Например −6 в битовом представлении . . . 111010, как число бесконечным числом единиц слева. Это видно, что добавление 6 (binary 110) к такой битовой схеме дает все нули.

`logand` *n1 n2 . . .* [Scheme Procedure]
`scm_logand` (*n1*, *n2*) [C Function]
Return the bitwise and of the integer arguments.

```
(logand) ⇒ -1
(logand 7) ⇒ 7
(logand #b111 #b011 #b001) ⇒ 1
```

`logior` *n1 n2 . . .* [Scheme Procedure]
`scm_logior` (*n1*, *n2*) [C Function]
Return the bitwise or of the integer arguments.

```
(logior) ⇒ 0
(logior 7) ⇒ 7
(logior #b000 #b001 #b011) ⇒ 3
```

`logxor` *n1 n2 . . .* [Scheme Procedure]
`scm_loxor` (*n1*, *n2*) [C Function]
Return the bitwise xor of the integer arguments. A bit is set in the result if it is set in an odd number of arguments.

```
(logxor) ⇒ 0
(logxor 7) ⇒ 7
(logxor #b000 #b001 #b011) ⇒ 2
(logxor #b000 #b001 #b011 #b011) ⇒ 1
```

`lognot` *n* [Scheme Procedure]
`scm_lognot` (*n*) [C Function]
Return the integer which is the ones-complement of the integer argument, ie. each 0 bit is changed to 1 and each 1 bit to 0.

```
(number->string (lognot #b10000000) 2)
   ⇒ "-10000001"
(number->string (lognot #b0) 2)
   ⇒ "-1"
```

`logtest` *j k* [Scheme Procedure]
`scm_logtest` (*j*, *k*) [C Function]

Test whether *j* and *k* have any 1 bits in common. This is equivalent to `(not (zero? (logand j k)))`, but without actually calculating the `logand`, just testing for non-zero.

```
(logtest #b0100 #b1011) ⇒ #f
(logtest #b0100 #b0111) ⇒ #t
```

`logbit?` *index j* [Scheme Procedure]
`scm_logbit_p` (*index*, *j*) [C Function]

Test whether bit number *index* in *j* is set. *index* starts from 0 for the least significant bit.

```
(logbit? 0 #b1101) ⇒ #t
(logbit? 1 #b1101) ⇒ #f
(logbit? 2 #b1101) ⇒ #t
(logbit? 3 #b1101) ⇒ #t
(logbit? 4 #b1101) ⇒ #f
```

`ash` *n count* [Scheme Procedure]
`scm_ash` (*n*, *count*) [C Function]

Return $floor(n * 2^{count})$. *n* and *count* must be exact integers.

With *n* viewed as an infinite-precision twos-complement integer, `ash` means a left shift introducing zero bits when *count* is positive, or a right shift dropping bits when *count* is negative. This is an "arithmetic" shift.

```
(number->string (ash #b1 3) 2)     ⇒ "1000"
(number->string (ash #b1010 -1) 2) ⇒ "101"

;; -23 is bits ...11101001, -6 is bits ...111010
(ash -23 -2) ⇒ -6
```

`round-ash` *n count* [Scheme Procedure]
`scm_round_ash` (*n*, *count*) [C Function]

Return $round(n * 2^{c}ount)$. *n* and *count* must be exact integers.

With *n* viewed as an infinite-precision twos-complement integer, `round-ash` means a left shift introducing zero bits when *count* is positive, or a right shift rounding to the nearest integer (with ties going to the nearest even integer) when *count* is negative. This is a rounded "arithmetic" shift.

```
(number->string (round-ash #b1 3) 2)     ⇒ \"1000\"
(number->string (round-ash #b1010 -1) 2) ⇒ \"101\"
(number->string (round-ash #b1010 -2) 2) ⇒ \"10\"
(number->string (round-ash #b1011 -2) 2) ⇒ \"11\"
(number->string (round-ash #b1101 -2) 2) ⇒ \"11\"
(number->string (round-ash #b1110 -2) 2) ⇒ \"100\"
```

`logcount` *n* [Scheme Procedure]
`scm_logcount` (*n*) [C Function]

Return the number of bits in integer *n*. If *n* is positive, the 1-bits in its binary representation are counted. If negative, the 0-bits in its two's-complement binary representation are counted. If zero, 0 is returned.

```
(logcount #b10101010)
   ⇒ 4
(logcount 0)
   ⇒ 0
(logcount -2)
   ⇒ 1
```

`integer-length` *n* [Scheme Procedure]
`scm_integer_length` (*n*) [C Function]

Return the number of bits necessary to represent *n*.

For positive *n* this is how many bits to the most significant one bit. For negative *n* it's how many bits to the most significant zero bit in twos complement form.

```
(integer-length #b10101010) ⇒ 8
(integer-length #b1111)     ⇒ 4
(integer-length 0)          ⇒ 0
(integer-length -1)         ⇒ 0
(integer-length -256)       ⇒ 8
(integer-length -257)       ⇒ 9
```

`integer-expt` *n k* [Scheme Procedure]
`scm_integer_expt` (*n*, *k*) [C Function]

Return *n* raised to the power *k*. *k* must be an exact integer, *n* can be any number.

Negative *k* is supported, and results in $1/n^{|k|}$ in the usual way. n^0 is 1, as usual, and that includes 0^0 is 1.

```
(integer-expt 2 5)   ⇒ 32
(integer-expt -3 3)  ⇒ -27
(integer-expt 5 -3)  ⇒ 1/125
(integer-expt 0 0)   ⇒ 1
```

`bit-extract` *n start end* [Scheme Procedure]
`scm_bit_extract` (*n*, *start*, *end*) [C Function]

Return the integer composed of the *start* (inclusive) through *end* (exclusive) bits of *n*. The *start*th bit becomes the 0-th bit in the result.

```
(number->string (bit-extract #b1101101010 0 4) 2)
   ⇒ "1010"
(number->string (bit-extract #b1101101010 4 9) 2)
   ⇒ "10110"
```

6.6.2.14 Random Number Generation

Псевдослучайные чила генерируются из объекта случаного состояния, который может быть создан функциями `seed->random-state` или `datum->random-state`. Внешнее представление (т.е.. которое может быть записано `write` и считано `read`) объекта

случайного состояния) может быть получено вызовом `random-state->datum`. Параметр *state* для различных функци ниже, не обязателен, по умолчанию используется объект состояния в переменной `*random-state*`.

`copy-random-state` [*state*] [Scheme Procedure]
`scm_copy_random_state` (*state*) [C Function]
Возвращает копию случайного состояния *state*.

`random` *n* [*state*] [Scheme Procedure]
`scm_random` (*n*, *state*) [C Function]
Возвращает номер в рамках [0, *n*).

Принимает полжительное целое или вещественное n и возвращает число того же типа между нулем(включительно) и *n* (исключая). Возвращаемые значения имеют равномерное распределение.

`random:exp` [*state*] [Scheme Procedure]
`scm_random_exp` (*state*) [C Function]
Возвращает не точное вещественное значение в экспотенциальном распределении со средним 1. Для экспотенциального распределения со средним равным *u* используйте `(* u (random:exp))`.

`random:hollow-sphere!` *vect* [*state*] [Scheme Procedure]
`scm_random_hollow_sphere_x` (*vect*, *state*) [C Function]
Заполняет вектор *vect* неточными вещественными случайными числами сумма квадратов которых равна 1.0. Думая о векторе *vect* как о пространстве размерности *n* = `(vector-length vect)`, координаты равномерно распределены по поверхности единичной n-сферы.

`random:normal` [*state*] [Scheme Procedure]
`scm_random_normal` (*state*) [C Function]
Возвращает не точное нормальное распределение. Это распределение имеет среднее 0 и стандартное отклонение 1. Для номрмального распределения со средним *m* и стандартным отклонением *d* используйте `(+ m (* d (random:normal)))`.

`random:normal-vector!` *vect* [*state*] [Scheme Procedure]
`scm_random_normal_vector_x` (*vect*, *state*) [C Function]
Заполняет вектор *vect* неточными вещественными случайными числами, которые являются независимыми и стандартно распределенными (т.е со средним 0 и дисперсией 1).

`random:solid-sphere!` *vect* [*state*] [Scheme Procedure]
`scm_random_solid_sphere_x` (*vect*, *state*) [C Function]
Заполняет вектор *vect* неточными вещественными случайными числами, сумма квадратов которых меньше 1.0. Думая о векторе *vect* как о координатах в пространстве размерности *n* = `(vector-length vect)`, координаты равномерно распределены внутри единично *n*-сферы.

`random:uniform` [*state*] [Scheme Procedure]
`scm_random_uniform` (*state*) [C Function]
Возвращает равномернораспределенное неточное вещественное число в границах [0,1).

`seed->random-state` *seed* [Scheme Procedure]
`scm_seed_to_random_state` (*seed*) [C Function]
Возвращает новое случайное состояние с использванием *seed*.

`datum->random-state` *datum* [Scheme Procedure]
`scm_datum_to_random_state` (*datum*) [C Function]
Возвращает новое случайное состояние из *datum*, которое должно быть получено вызовом `random-state->datum`.

`random-state->datum` *state* [Scheme Procedure]
`scm_random_state_to_datum` (*state*) [C Function]
Возвращает представление случайного состояния *state* которое может быть записано и прочитано с помощью функций чтения Scheme.

`random-state-from-platform` [Scheme Procedure]
`scm_random_state_from_platform` () [C Function]
Создание нового случайного состояния, сгенерированное из энтропии, специфичной для платформы, подходящее для использования в не критичных для безопасности приложениях. В настоящее время `/dev/urandom` сначала, основывается на времени и дате, идентификаторе процесса, адресе недавно выделенной ячейки кучи, адресе из локального фрейма стека и таймере высокого разрешения, если он доступен.

`*random-state*` [Переменная]
Глобальное случайное состояние, используемое вышеуказанными функциями, когда параметр *state* не задан.

Обратите внимание, что начальное значение `*random-state*` одинаково при каждом запуске Guile. Поэтому, если вы не передадите параметр состояния указанным выше процедурам, и вы не установите `*random-state*` вызвом `(seed->random-state your-seed)`, где `your-seed` это не одно и тоже значение каждый раз, вы получите одну и туже последовательность "случайных" чисел при каждом запуске программы.

Например, если соответствующий код не был изменен, `(map random (cdr (iota 30)))`, если первое использование случайных чисел с момента запуска Guile всегда будет давать:

```
(map random (cdr (iota 19)))
⇒
(0 1 1 2 2 2 1 2 6 7 10 0 5 3 12 5 5 12)
```

Чтобы разумно распределять случайное состояние для приложений, не относящихся к супер безопасным, сделайте это во время инициализации вашей программы.

```
(set! *random-state* (random-state-from-platform))
```

6.6.3 Characters

В схеме существует тип данных для описания одиночных символов

Определение того, что такое символ может быть сложнее, чем кажется. Guile следует рекомендациям R6RS и использует стандарт Unicode, чтобы определить, что является символом. Итак, для Guile символ - это что-то из базы данных символов Юникода.

База данных символов Юникода представляет собой таблицу символов, индексированную с использованием целых чисел называемых «кодовыми точками». Действительные кодовые точки находятся в диапазоне от 0 до `#xD7FF` включительно или `#xE000` до `#x10FFFF` включительно, что составляет около 1,1 миллиона кодовых точек.

Любая кодовая точка, которая была назначена символу или которая имеет значение в Unicode называется «обозначенной кодовой точкой». Большинство обозначенных кодовых точек, около 200 000 указывают на символы, акценты или сочетания меток, которые изменяют другие символы, символы, пробелы и управляющие символы. Некоторые из них не являются символами, но являются индикаторами, которые указывают, как форматировать или отображать соседние символы.

Если кодовая точка не является назначенной кодовой точкой - т.е если она не назначена символу в стандарте Unicode - это 'зарезервированная кодовая точка', что означает, что она зарезервирована для будущего использования. Большинство кодовых точек, около 800 000, являются «зарезервированными кодовыми точками».

По соглашению, кодовая точка Unicode записывается как "U+XXXX", где "XXXX" является шестнадцатеричным числом. Обратите внимание, что это удобное обозначение является недопустимым кодом. Guile не интерпретирует "U+XXXX" как символ.

В Scheme символьный литерал записывается как `#\`*`name`*, где *name* - это имя символа, который вам нужен. Печатные символы имеют свое обычное односимвольное имя; Например, `#\a` - это нижний регистр `a`.

Некоторые из кодовых точек - это 'комбинирующие символы', которые не предназначены для печати а вместо этого предназначены для изменения внешнего вида предыдущего символа. Для комбинирующих символов, альтернативная форма символьного литерала - `#\`, затем U+25CC (маленький, пунктирный круг), за которым следует комбинирующий символ. Это позволяет создать комбинированный символ, который будет нарисован на круге, а не обычную обратную косую черту `#\`.

Многие из непечатаемых символов, таких как символы пробелов и управляющие символы, также имеют имена.

Наиболее часто используемые непечатаемые символы имеют длинные имена символов, описанны в таблице ниже

Character Name	Codepoint
`#\nul`	U+0000
`#\alarm`	U+0007
`#\backspace`	U+0008
`#\tab`	U+0009
`#\linefeed`	U+000A
`#\newline`	U+000A

`#\vtab`	U+000B
`#\page`	U+000C
`#\return`	U+000D
`#\esc`	U+001B
`#\space`	U+0020
`#\delete`	U+007F

Также есть короткие имена для всех “C0 управляющих символов” (те, у кого есть кодовые точки ниже 32). В следующей таблице указаны краткие имена для каждого символа.

0 = `#\nul`	1 = `#\soh`	2 = `#\stx`	3 = `#\etx`
4 = `#\eot`	5 = `#\enq`	6 = `#\ack`	7 = `#\bel`
8 = `#\bs`	9 = `#\ht`	10 = `#\lf`	11 = `#\vt`
12 = `#\ff`	13 = `#\cr`	14 = `#\so`	15 = `#\si`
16 = `#\dle`	17 = `#\dc1`	18 = `#\dc2`	19 = `#\dc3`
20 = `#\dc4`	21 = `#\nak`	22 = `#\syn`	23 = `#\etb`
24 = `#\can`	25 = `#\em`	26 = `#\sub`	27 = `#\esc`
28 = `#\fs`	29 = `#\gs`	30 = `#\rs`	31 = `#\us`
32 = `#\sp`			

Короткое имя для символа удалить “delete” (кодовая точка U+007F) `#\del`.

The R7RS name for the “escape” character (code point U+001B) is `#\escape`.

Есть также несколько альтернативных имен, оставшихся для совместимости с предыдущими версиями из Guile.

Alternate	Standard
`#\nl`	`#\newline`
`#\np`	`#\page`
`#\null`	`#\nul`

Characters may also be written using their code point values. They can be written with as an octal number, such as `#\10` for `#\bs` or `#\177` for `#\del`.

Если кто-то предпочитает использовать шестнадцатеричный код, сущетствует дополнительный синткасис с обратной косой чертой: `#\xHHHH` – за символом ’x’ следует шестнадцатеричное число от одной до восми цифр.

`char?` *x* [Scheme Procedure]
`scm_char_p` (*x*) [C Function]
Return `#t` if *x* is a character, else `#f`.

Fundamentally, the character comparison operations below are numeric comparisons of the character’s code points.

`char=?` *x y* [Scheme Procedure]
Return `#t` if code point of *x* is equal to the code point of *y*, else `#f`.

`char<?` *x y* [Scheme Procedure]
Return `#t` if the code point of *x* is less than the code point of *y*, else `#f`.

`char<=?` *x y* [Scheme Procedure]
Return `#t` if the code point of *x* is less than or equal to the code point of *y*, else `#f`.

`char>?` *x y* [Scheme Procedure]
Return `#t` if the code point of *x* is greater than the code point of *y*, else `#f`.

`char>=?` *x y* [Scheme Procedure]
Return `#t` if the code point of *x* is greater than or equal to the code point of *y*, else `#f`.

Case-insensitive character comparisons use *Unicode case folding*. In case folding comparisons, if a character is lowercase and has an uppercase form that can be expressed as a single character, it is converted to uppercase before comparison. All other characters undergo no conversion before the comparison occurs. This includes the German sharp S (Eszett) which is not uppercased before conversion because its uppercase form has two characters. Unicode case folding is language independent: it uses rules that are generally true, but, it cannot cover all cases for all languages.

`char-ci=?` *x y* [Scheme Procedure]
Return `#t` if the case-folded code point of *x* is the same as the case-folded code point of *y*, else `#f`.

`char-ci<?` *x y* [Scheme Procedure]
Return `#t` if the case-folded code point of *x* is less than the case-folded code point of *y*, else `#f`.

`char-ci<=?` *x y* [Scheme Procedure]
Return `#t` if the case-folded code point of *x* is less than or equal to the case-folded code point of *y*, else `#f`.

`char-ci>?` *x y* [Scheme Procedure]
Return `#t` if the case-folded code point of *x* is greater than the case-folded code point of *y*, else `#f`.

`char-ci>=?` *x y* [Scheme Procedure]
Return `#t` if the case-folded code point of *x* is greater than or equal to the case-folded code point of *y*, else `#f`.

`char-alphabetic?` *chr* [Scheme Procedure]
`scm_char_alphabetic_p` (*chr*) [C Function]
Return `#t` if *chr* is alphabetic, else `#f`.

`char-numeric?` *chr* [Scheme Procedure]
`scm_char_numeric_p` (*chr*) [C Function]
Return `#t` if *chr* is numeric, else `#f`.

`char-whitespace?` *chr* [Scheme Procedure]
`scm_char_whitespace_p` (*chr*) [C Function]
Return `#t` if *chr* is whitespace, else `#f`.

`char-upper-case?` *chr* [Scheme Procedure]
`scm_char_upper_case_p` (*chr*) [C Function]
Return `#t` if *chr* is uppercase, else `#f`.

`char-lower-case?` *chr* [Scheme Procedure]
`scm_char_lower_case_p` (*chr*) [C Function]
Return `#t` if *chr* is lowercase, else `#f`.

`char-is-both?` *chr* [Scheme Procedure]
`scm_char_is_both_p` (*chr*) [C Function]
Return `#t` if *chr* is either uppercase or lowercase, else `#f`.

`char-general-category` *chr* [Scheme Procedure]
`scm_char_general_category` (*chr*) [C Function]
Return a symbol giving the two-letter name of the Unicode general category assigned to *chr* or `#f` if no named category is assigned. The following table provides a list of category names along with their meanings.

Lu	Uppercase letter	Pf	Final quote punctuation
Ll	Lowercase letter	Po	Other punctuation
Lt	Titlecase letter	Sm	Math symbol
Lm	Modifier letter	Sc	Currency symbol
Lo	Other letter	Sk	Modifier symbol
Mn	Non-spacing mark	So	Other symbol
Mc	Combining spacing mark	Zs	Space separator
Me	Enclosing mark	Zl	Line separator
Nd	Decimal digit number	Zp	Paragraph separator
Nl	Letter number	Cc	Control
No	Other number	Cf	Format
Pc	Connector punctuation	Cs	Surrogate
Pd	Dash punctuation	Co	Private use
Ps	Open punctuation	Cn	Unassigned
Pe	Close punctuation		
Pi	Initial quote punctuation		

`char->integer` *chr* [Scheme Procedure]
`scm_char_to_integer` (*chr*) [C Function]
Return the code point of *chr*.

`integer->char` *n* [Scheme Procedure]
`scm_integer_to_char` (*n*) [C Function]
Return the character that has code point *n*. The integer *n* must be a valid code point. Valid code points are in the ranges 0 to `#xD7FF` inclusive or `#xE000` to `#x10FFFF` inclusive.

`char-upcase` *chr* [Scheme Procedure]
`scm_char_upcase` (*chr*) [C Function]
Return the uppercase character version of *chr*.

`char-downcase` *chr* [Scheme Procedure]
`scm_char_downcase` (*chr*) [C Function]
Return the lowercase character version of *chr*.

`char-titlecase` *chr* [Scheme Procedure]
`scm_char_titlecase` (*chr*) [C Function]
Return the titlecase character version of *chr* if one exists; otherwise return the uppercase version.

For most characters these will be the same, but the Unicode Standard includes certain digraph compatibility characters, such as `U+01F3` "dz", for which the uppercase and titlecase characters are different (`U+01F1` "DZ" and `U+01F2` "Dz" in this case, respectively).

`scm_t_wchar scm_c_upcase` (*scm_t_wchar c*) [C Function]
`scm_t_wchar scm_c_downcase` (*scm_t_wchar c*) [C Function]
`scm_t_wchar scm_c_titlecase` (*scm_t_wchar c*) [C Function]
These C functions take an integer representation of a Unicode codepoint and return the codepoint corresponding to its uppercase, lowercase, and titlecase forms respectively. The type `scm_t_wchar` is a signed, 32-bit integer.

Символы также имеют "формальные имена", которые определены Unicode. Этим имена могут быть доступны в Guile из модуля `(ice-9 unicode)`:

```
(use-modules (ice-9 unicode))
```

`char->formal-name` *chr* [Scheme Procedure]
Return the formal all-upper-case Unicode name of *ch*, as a string, or `#f` if the character has no name.

`formal-name->char` *name* [Scheme Procedure]
Return the character whose formal all-upper-case Unicode name is *name*, or `#f` if no such character is known.

6.6.4 Character Sets

Функции, описанные в данном разделе, соответствуют SRFI-14.

Тип данных *(Набор символов)charset* реализует множество символов. (см. Раздел 6.6.3 [Characters], страница 139). Поскольку внутреннее представление множества символов не видно пользователю, предоставляется множество процедур для их обработки.

Наборы символов могут быть созданы, расширены, проверены на принадлежность символов и могут быть сравнены с другими наборами символов.

6.6.4.1 Character Set Predicates/Comparison

Используйте эти процедуры для проверки того, является ли объект набором символов, или проверки равны ли между собой несколько наборов символов или является ли один набор символов, подмножеством другого. Функция `char-set-hash` вычисляет хеш-значение набора символов и может быть использована в быстрых процедурах поиска.

`char-set?` *obj* [Scheme Procedure]
`scm_char_set_p` (*obj*) [C Function]
Return `#t` if *obj* is a character set, `#f` otherwise.

`char-set=` *char_set ...* [Scheme Procedure]
`scm_char_set_eq` (*char_sets*) [C Function]
Return `#t` if all given character sets are equal.

`char-set<=` *char_set ...* [Scheme Procedure]
`scm_char_set_leq` (*char_sets*) [C Function]
Return `#t` if every character set *char_set*i is a subset of character set *char_set*i+1.

`char-set-hash` *cs* [*bound*] [Scheme Procedure]
`scm_char_set_hash` (*cs, bound*) [C Function]
Compute a hash value for the character set *cs*. If *bound* is given and non-zero, it restricts the returned value to the range 0 ... *bound* - 1.

6.6.4.2 Iterating Over Character Sets

Указатели в наборе символов - это средство для итерации над элементами набора символов. После создания указателя(курсора) набора символов с помощью функции `char-set-cursor`, указатель может быть разименован(получен элемент набора символов) `char-set-ref`, перемещен к следующему элементу `char-set-cursor-next`. Проверить когда указатель прошел последний элемент можно функцией `end-of-char-set?`.

Кроме того, предоставляются функции сопоставления(mapping) и функции складывания(fold)/создания(unfold) набора символов.

`char-set-cursor` *cs* [Scheme Procedure]
`scm_char_set_cursor` (*cs*) [C Function]
Return a cursor into the character set *cs*.

`char-set-ref` *cs cursor* [Scheme Procedure]
`scm_char_set_ref` (*cs, cursor*) [C Function]
Return the character at the current cursor position *cursor* in the character set *cs*. It is an error to pass a cursor for which `end-of-char-set?` returns true.

`char-set-cursor-next` *cs cursor* [Scheme Procedure]
`scm_char_set_cursor_next` (*cs, cursor*) [C Function]
Advance the character set cursor *cursor* to the next character in the character set *cs*. It is an error if the cursor given satisfies `end-of-char-set?`.

`end-of-char-set?` *cursor* [Scheme Procedure]
`scm_end_of_char_set_p` (*cursor*) [C Function]
Return `#t` if *cursor* has reached the end of a character set, `#f` otherwise.

`char-set-fold` *kons knil cs* [Scheme Procedure]
`scm_char_set_fold` (*kons, knil, cs*) [C Function]
Fold the procedure *kons* over the character set *cs*, initializing it with *knil*.

`char-set-unfold` *p f g seed* [*base_cs*] [Scheme Procedure]
`scm_char_set_unfold` (*p, f, g, seed, base_cs*) [C Function]
This is a fundamental constructor for character sets.

- *g* is used to generate a series of "seed" values from the initial seed: *seed*, (*g seed*), (*g*^2 *seed*), (*g*^3 *seed*), ...

- *p* tells us when to stop – when it returns true when applied to one of the seed values.
- *f* maps each seed value to a character. These characters are added to the base character set *base_cs* to form the result; *base_cs* defaults to the empty set.

`char-set-unfold!` *p f g seed base_cs* [Scheme Procedure]
`scm_char_set_unfold_x` (*p, f, g, seed, base_cs*) [C Function]

This is a fundamental constructor for character sets.

- *g* is used to generate a series of "seed" values from the initial seed: *seed*, (*g seed*), (*g*^2 *seed*), (*g*^3 *seed*), ...
- *p* tells us when to stop – when it returns true when applied to one of the seed values.
- *f* maps each seed value to a character. These characters are added to the base character set *base_cs* to form the result; *base_cs* defaults to the empty set.

`char-set-for-each` *proc cs* [Scheme Procedure]
`scm_char_set_for_each` (*proc, cs*) [C Function]

Apply *proc* to every character in the character set *cs*. The return value is not specified.

`char-set-map` *proc cs* [Scheme Procedure]
`scm_char_set_map` (*proc, cs*) [C Function]

Map the procedure *proc* over every character in *cs*. *proc* must be a character -> character procedure.

6.6.4.3 Creating Character Sets

С помощью этих процедур создаются новые наборы символов.

`char-set-copy` *cs* [Scheme Procedure]
`scm_char_set_copy` (*cs*) [C Function]

Return a newly allocated character set containing all characters in *cs*.

`char-set` *chr ...* [Scheme Procedure]
`scm_char_set` (*chrs*) [C Function]

Return a character set containing all given characters.

`list->char-set` *list* [*base_cs*] [Scheme Procedure]
`scm_list_to_char_set` (*list, base_cs*) [C Function]

Convert the character list *list* to a character set. If the character set *base_cs* is given, the character in this set are also included in the result.

`list->char-set!` *list base_cs* [Scheme Procedure]
`scm_list_to_char_set_x` (*list, base_cs*) [C Function]

Convert the character list *list* to a character set. The characters are added to *base_cs* and *base_cs* is returned.

`string->char-set` *str* [*base_cs*] [Scheme Procedure]
`scm_string_to_char_set` (*str, base_cs*) [C Function]

Convert the string *str* to a character set. If the character set *base_cs* is given, the characters in this set are also included in the result.

`string->char-set!` *str base_cs* [Scheme Procedure]
`scm_string_to_char_set_x` (*str, base_cs*) [C Function]
Convert the string *str* to a character set. The characters from the string are added to *base_cs*, and *base_cs* is returned.

`char-set-filter` *pred cs* [*base_cs*] [Scheme Procedure]
`scm_char_set_filter` (*pred, cs, base_cs*) [C Function]
Return a character set containing every character from *cs* so that it satisfies *pred*. If provided, the characters from *base_cs* are added to the result.

`char-set-filter!` *pred cs base_cs* [Scheme Procedure]
`scm_char_set_filter_x` (*pred, cs, base_cs*) [C Function]
Return a character set containing every character from *cs* so that it satisfies *pred*. The characters are added to *base_cs* and *base_cs* is returned.

`ucs-range->char-set` *lower upper* [*error* [*base_cs*]] [Scheme Procedure]
`scm_ucs_range_to_char_set` (*lower, upper, error, base_cs*) [C Function]
Return a character set containing all characters whose character codes lie in the half-open range [*lower*,*upper*).

If *error* is a true value, an error is signalled if the specified range contains characters which are not contained in the implemented character range. If *error* is `#f`, these characters are silently left out of the resulting character set.

The characters in *base_cs* are added to the result, if given.

`ucs-range->char-set!` *lower upper error base_cs* [Scheme Procedure]
`scm_ucs_range_to_char_set_x` (*lower, upper, error, base_cs*) [C Function]
Return a character set containing all characters whose character codes lie in the half-open range [*lower*,*upper*).

If *error* is a true value, an error is signalled if the specified range contains characters which are not contained in the implemented character range. If *error* is `#f`, these characters are silently left out of the resulting character set.

The characters are added to *base_cs* and *base_cs* is returned.

`->char-set` *x* [Scheme Procedure]
`scm_to_char_set` (*x*) [C Function]
Coerces x into a char-set. *x* may be a string, character or char-set. A string is converted to the set of its constituent characters; a character is converted to a singleton set; a char-set is returned as-is.

6.6.4.4 Querying Character Sets

С помощью этих процедур осущетсвляется получение доступа к элементам и получение другой информации о наборе символов.

`%char-set-dump` *cs* [Scheme Procedure]
Returns an association list containing debugging information for *cs*. The association list has the following entries.

`char-set` The char-set itself

`len` The number of groups of contiguous code points the char-set contains

`ranges` A list of lists where each sublist is a range of code points and their associated characters

The return value of this function cannot be relied upon to be consistent between versions of Guile and should not be used in code.

`char-set-size` *cs* [Scheme Procedure]
`scm_char_set_size` (*cs*) [C Function]
Return the number of elements in character set *cs*.

`char-set-count` *pred cs* [Scheme Procedure]
`scm_char_set_count` (*pred, cs*) [C Function]
Return the number of the elements int the character set *cs* which satisfy the predicate *pred*.

`char-set->list` *cs* [Scheme Procedure]
`scm_char_set_to_list` (*cs*) [C Function]
Return a list containing the elements of the character set *cs*.

`char-set->string` *cs* [Scheme Procedure]
`scm_char_set_to_string` (*cs*) [C Function]
Return a string containing the elements of the character set *cs*. The order in which the characters are placed in the string is not defined.

`char-set-contains?` *cs ch* [Scheme Procedure]
`scm_char_set_contains_p` (*cs, ch*) [C Function]
Return `#t` if the character *ch* is contained in the character set *cs*, or `#f` otherwise.

`char-set-every` *pred cs* [Scheme Procedure]
`scm_char_set_every` (*pred, cs*) [C Function]
Return a true value if every character in the character set *cs* satisfies the predicate *pred*.

`char-set-any` *pred cs* [Scheme Procedure]
`scm_char_set_any` (*pred, cs*) [C Function]
Return a true value if any character in the character set *cs* satisfies the predicate *pred*.

6.6.4.5 Character-Set Algebra

Наборами символов можно манипулировать с помощью операций общей алгебры множеств, таких как объединение, дополнения, пересечения и т.д. Все эти процедуры имеют побочные эффекты, которые изменяют их аргумент(ы) - наборы символов.

`char-set-adjoin` *cs chr ...* [Scheme Procedure]
`scm_char_set_adjoin` (*cs, chrs*) [C Function]
Add all character arguments to the first argument, which must be a character set.

`char-set-delete` *cs chr ...* [Scheme Procedure]
`scm_char_set_delete` (*cs, chrs*) [C Function]
Delete all character arguments from the first argument, which must be a character set.

`char-set-adjoin!` *cs chr ...* [Scheme Procedure]
`scm_char_set_adjoin_x` (*cs, chrs*) [C Function]
Add all character arguments to the first argument, which must be a character set.

`char-set-delete!` *cs chr ...* [Scheme Procedure]
`scm_char_set_delete_x` (*cs, chrs*) [C Function]
Delete all character arguments from the first argument, which must be a character set.

`char-set-complement` *cs* [Scheme Procedure]
`scm_char_set_complement` (*cs*) [C Function]
Return the complement of the character set *cs*.

Note that the complement of a character set is likely to contain many reserved code points (code points that are not associated with characters). It may be helpful to modify the output of `char-set-complement` by computing its intersection with the set of designated code points, `char-set:designated`.

`char-set-union` *cs ...* [Scheme Procedure]
`scm_char_set_union` (*char_sets*) [C Function]
Return the union of all argument character sets.

`char-set-intersection` *cs ...* [Scheme Procedure]
`scm_char_set_intersection` (*char_sets*) [C Function]
Return the intersection of all argument character sets.

`char-set-difference` *cs1 cs ...* [Scheme Procedure]
`scm_char_set_difference` (*cs1, char_sets*) [C Function]
Return the difference of all argument character sets.

`char-set-xor` *cs ...* [Scheme Procedure]
`scm_char_set_xor` (*char_sets*) [C Function]
Return the exclusive-or of all argument character sets.

`char-set-diff+intersection` *cs1 cs ...* [Scheme Procedure]
`scm_char_set_diff_plus_intersection` (*cs1, char_sets*) [C Function]
Return the difference and the intersection of all argument character sets.

`char-set-complement!` *cs* [Scheme Procedure]
`scm_char_set_complement_x` (*cs*) [C Function]
Return the complement of the character set *cs*.

`char-set-union!` *cs1 cs ...* [Scheme Procedure]
`scm_char_set_union_x` (*cs1, char_sets*) [C Function]
Return the union of all argument character sets.

`char-set-intersection!` *cs1 cs ...* [Scheme Procedure]
`scm_char_set_intersection_x` (*cs1*, *char_sets*) [C Function]
Return the intersection of all argument character sets.

`char-set-difference!` *cs1 cs ...* [Scheme Procedure]
`scm_char_set_difference_x` (*cs1*, *char_sets*) [C Function]
Return the difference of all argument character sets.

`char-set-xor!` *cs1 cs ...* [Scheme Procedure]
`scm_char_set_xor_x` (*cs1*, *char_sets*) [C Function]
Return the exclusive-or of all argument character sets.

`char-set-diff+intersection!` *cs1 cs2 cs ...* [Scheme Procedure]
`scm_char_set_diff_plus_intersection_x` (*cs1*, *cs2*, *char_sets*) [C Function]
Return the difference and the intersection of all argument character sets.

6.6.4.6 Standard Character Sets

Чтобы использовать тип данных и процедуры набора символов, существуют предопределенные наборы символов.

Эти наборы символов независимы от локали и не пересчитываются при вызове `setlocale`. Они содержат символы из всего диапазона кодовых точек Unicode. Например, `char-set:letter` содержит около 100,000 символов.

`char-set:lower-case` [Scheme Variable]
`scm_char_set_lower_case` [C Variable]
All lower-case characters.

`char-set:upper-case` [Scheme Variable]
`scm_char_set_upper_case` [C Variable]
All upper-case characters.

`char-set:title-case` [Scheme Variable]
`scm_char_set_title_case` [C Variable]
All single characters that function as if they were an upper-case letter followed by a lower-case letter.

`char-set:letter` [Scheme Variable]
`scm_char_set_letter` [C Variable]
All letters. This includes `char-set:lower-case`, `char-set:upper-case`, `char-set:title-case`, and many letters that have no case at all. For example, Chinese and Japanese characters typically have no concept of case.

`char-set:digit` [Scheme Variable]
`scm_char_set_digit` [C Variable]
All digits.

`char-set:letter+digit` [Scheme Variable]
`scm_char_set_letter_and_digit` [C Variable]
The union of `char-set:letter` and `char-set:digit`.

`char-set:graphic` [Scheme Variable]
`scm_char_set_graphic` [C Variable]
All characters which would put ink on the paper.

`char-set:printing` [Scheme Variable]
`scm_char_set_printing` [C Variable]
The union of `char-set:graphic` and `char-set:whitespace`.

`char-set:whitespace` [Scheme Variable]
`scm_char_set_whitespace` [C Variable]
All whitespace characters.

`char-set:blank` [Scheme Variable]
`scm_char_set_blank` [C Variable]
All horizontal whitespace characters, which notably includes `#\space` and `#\tab`.

`char-set:iso-control` [Scheme Variable]
`scm_char_set_iso_control` [C Variable]
The ISO control characters are the C0 control characters (U+0000 to U+001F), delete (U+007F), and the C1 control characters (U+0080 to U+009F).

`char-set:punctuation` [Scheme Variable]
`scm_char_set_punctuation` [C Variable]
All punctuation characters, such as the characters `!"#%&'()*,-./:;?@[\\]_{}`

`char-set:symbol` [Scheme Variable]
`scm_char_set_symbol` [C Variable]
All symbol characters, such as the characters `$+<=>^`|~`.

`char-set:hex-digit` [Scheme Variable]
`scm_char_set_hex_digit` [C Variable]
The hexadecimal digits `0123456789abcdefABCDEF`.

`char-set:ascii` [Scheme Variable]
`scm_char_set_ascii` [C Variable]
All ASCII characters.

`char-set:empty` [Scheme Variable]
`scm_char_set_empty` [C Variable]
The empty character set.

`char-set:designated` [Scheme Variable]
`scm_char_set_designated` [C Variable]
This character set contains all designated code points. This includes all the code points to which Unicode has assigned a character or other meaning.

`char-set:full` [Scheme Variable]
`scm_char_set_full` [C Variable]
This character set contains all possible code points. This includes both designated and reserved code points.

6.6.5 Strings

Строки представляют собой последовательности символов фиксированной длины. Они могут быть созданы путем вызова процедуры конструктора, но они так же могут непосредственно вводиться в REPL или размещаться в исходных файлах Scheme.

Строки всегда содержат информацию о том, сколько символов они содержат, поэтому нет специального симовола конца строки, как например в Си. Это означает, что Scheme строки могут содержать любой символ, даже '`#\nul`' '`\0`'.

Чтобы эффективно использовать строки, вам нужно немного узнать о том, как Guile их реализует. В Guile, строка состоит из двух частей, головы и выделенной памяти, где фактически хранятся символы. Когда копируется строка(или ее подстрока), создается только новый заголовок, память обычно не копируется. Два заголовка начинают указывать на одну и туже память.

Когда одна из этих двух строк изменяется, например вызовом `string-set!` их общая память копируется так, что каждая строка получает свою копию собственной памяти и модификация не может случайно изменить другие строки. Таким образом, строки Guile являются "копируемыми при записи"; Настоящее копирование их памяти задерживается до тех пор, пока не будет изменена какая либо из строк.

Эта реализация делает такие функции, как `substring` очень эффективными в общем случаее когда никаких изменений используемых строк не производиться.

Если вы знаете, что ваши строки сразу изменяются, вы можете использовать `substring/copy` вместо `substring`. Эта функция немедленно выполняет копию во время создания подстроки. Это более эффективно, особенно в многопоточной программе. Также `substring/copy` поможет избежать проблемы, заключающейся в том, что короткая подстрока сохраняется в памяти очень большой оригинальной строки, которую в противном случае можно было бы повторно использовать.

Если вы хотите полностью отказаться от копирования, что бы изменения одной строки отображались в другой, вы можете использовать функцию `substring/shared`. Строки созданные процедурой, называются изменяющимися разделяемыми строками, которые меняются при модификации каждой из строк.

Если вы хотите предотвратить изменения, используйте вызов `substring/read-only`.

Guile предоставляет все процедуры SRFI-13 и еще несколько.

6.6.5.1 String Read Syntax

Синтаксис чтения строк - это произвольно длинная последовательность символов, заключенных в двойные кавычки(").

Обратная косая черта это символ экранирования и может использоваться для вставки следующих специальных символов. `\"` и `\\` в соответсвии со стандартом R5RS, `\|` по стандарту R7RS, следующие за ними 7 символов по стандарту R6RS — обратите внимание они следуют Си синтаксису, а остальные 4 — Guile расширения.

`\\` Backslash character.

`\"` Double quote character (an unescaped " is otherwise the end of the string).

`\|` Vertical bar character.

`\a` Bell character (ASCII 7).

`\f` Formfeed character (ASCII 12).

`\n` Newline character (ASCII 10).

`\r` Carriage return character (ASCII 13).

`\t` Tab character (ASCII 9).

`\v` Vertical tab character (ASCII 11).

`\b` Backspace character (ASCII 8).

`\0` NUL character (ASCII 0).

`\(` Открывающая скобка. Этот символ предназначен для использования в начале строк в многострочной строке, чтобы избежать путаницы режимов в Emacs lisp.

`\` следует новая линия (ASCII 10)
Ничего. Таким образом, если `\` это последний символ в строке, строка будет продолжена с первым символом из следующей строки без разрыва строки.

Если включен режим чтения `hungry-eol-escapes`, что не соотвествует умолчанию, начальные пробелы на следующей строке отбрасываются.

```
"foo\
  bar"
⇒ "foo  bar"
(read-enable 'hungry-eol-escapes)
"foo\
  bar"
⇒ "foobar"
```

`\xHH` Character code given by two hexadecimal digits. For example `\x7f` for an ASCII DEL (127).

`\uHHHH` Character code given by four hexadecimal digits. For example `\u0100` for a capital A with macron (U+0100).

`\UHHHHHH` Character code given by six hexadecimal digits. For example `\U010402`.

The following are examples of string literals:

```
"foo"
"bar plonk"
"Hello World"
"\"Hi\", he said."
```

Три эскейп последовательности `\xHH`, `\uHHHH` и `\UHHHHHH` были выбраны так, чтобы не нарушать совместимость с кодом, написанным для предыдущих версий Guile. Спецификация R6RS предлагает другой, несовместимый синтаксис для экранирования шестнадцатеричных символов: `\xHHHH;` – код символа, за которым следуют от одной до восьми шестнадцатеричных цифр, заканчивающихся точкой с запятой. Если вам надо использовать этот формат для функции чтения, его можно включить с помощью опции `r6rs-hex-escapes`.

```
(read-enable 'r6rs-hex-escapes)
```

For more on reader options, См. Раздел 6.18.2 [Scheme Read], страница 407.

6.6.5.2 Строковые Предикаты

Следующие процедуры могут использваться для проверки того, соответствует ли данная строка некоторым определенным свойствам.

`string?` *obj* [Scheme Procedure]
`scm_string_p` (*obj*) [C Function]
Return `#t` if *obj* is a string, else `#f`.

`int scm_is_string` (*SCM obj*) [C Function]
Returns `1` if *obj* is a string, `0` otherwise.

`string-null?` *str* [Scheme Procedure]
`scm_string_null_p` (*str*) [C Function]
Return `#t` if *str*'s length is zero, and `#f` otherwise.

```
(string-null? "")  ⇒ #t
y                  ⇒ "foo"
(string-null? y)   ⇒ #f
```

`string-any` *char_pred s* [*start* [*end*]] [Scheme Procedure]
`scm_string_any` (*char_pred, s, start, end*) [C Function]
Check if *char_pred* is true for any character in string *s*.

char_pred can be a character to check for any equal to that, or a character set (см. Раздел 6.6.4 [Character Sets], страница 143) to check for any in that set, or a predicate procedure to call.

For a procedure, calls (***char_pred*** `c`) are made successively on the characters from *start* to *end*. If *char_pred* returns true (ie. non-`#f`), `string-any` stops and that return value is the return from `string-any`. The call on the last character (ie. at *end* − 1), if that point is reached, is a tail call.

If there are no characters in *s* (ie. *start* equals *end*) then the return is `#f`.

`string-every` *char_pred s* [*start* [*end*]] [Scheme Procedure]
`scm_string_every` (*char_pred, s, start, end*) [C Function]
Check if *char_pred* is true for every character in string *s*.

char_pred can be a character to check for every character equal to that, or a character set (см. Раздел 6.6.4 [Character Sets], страница 143) to check for every character being in that set, or a predicate procedure to call.

For a procedure, calls (***char_pred*** `c`) are made successively on the characters from *start* to *end*. If *char_pred* returns `#f`, `string-every` stops and returns `#f`. The call on the last character (ie. at *end* − 1), if that point is reached, is a tail call and the return from that call is the return from `string-every`.

If there are no characters in *s* (ie. *start* equals *end*) then the return is `#t`.

6.6.5.3 Конструкторы Строк

Процедуры создания строк создают новые строковые объекты, повозможности инициализируя их некоторыми заданными символьными данными. Смотри также См. Раздел 6.6.5.5 [String Selection], страница 155, для ознакомления со способами создания строк из существующих строк.

`string` *char...* [Scheme Procedure]

Return a newly allocated string made from the given character arguments.

```
(string #\x #\y #\z) ⇒ "xyz"
(string)             ⇒ ""
```

`list->string` *lst* [Scheme Procedure]
`scm_string` (*lst*) [C Function]

Return a newly allocated string made from a list of characters.

```
(list->string '(#\a #\b #\c)) ⇒ "abc"
```

`reverse-list->string` *lst* [Scheme Procedure]
`scm_reverse_list_to_string` (*lst*) [C Function]

Return a newly allocated string made from a list of characters, in reverse order.

```
(reverse-list->string '(#\a #\B #\c)) ⇒ "cBa"
```

`make-string` *k* **[***chr***]** [Scheme Procedure]
`scm_make_string` (*k, chr*) [C Function]

Return a newly allocated string of length *k*. If *chr* is given, then all elements of the string are initialized to *chr*, otherwise the contents of the string are unspecified.

`SCM scm_c_make_string` (*size_t len, SCM chr*) [C Function]

Like `scm_make_string`, but expects the length as a `size_t`.

`string-tabulate` *proc len* [Scheme Procedure]
`scm_string_tabulate` (*proc, len*) [C Function]

proc is an integer->char procedure. Construct a string of size *len* by applying *proc* to each index to produce the corresponding string element. The order in which *proc* is applied to the indices is not specified.

`string-join` *ls* **[***delimiter* **[***grammar***]]** [Scheme Procedure]
`scm_string_join` (*ls, delimiter, grammar*) [C Function]

Append the string in the string list *ls*, using the string *delimiter* as a delimiter between the elements of *ls*. *grammar* is a symbol which specifies how the delimiter is placed between the strings, and defaults to the symbol `infix`.

`infix` Insert the separator between list elements. An empty string will produce an empty list.

`strict-infix`
Like `infix`, but will raise an error if given the empty list.

`suffix` Insert the separator after every list element.

`prefix` Insert the separator before each list element.

6.6.5.4 Преобразование Список/Строка

При обработке строк часто бывает удобно сначала перобразовать их в представление списка с помощью процедуры `string->list`, работать с результирующим списком, а затем преобразовать его обратно в строку. Эти процедуры полезны для подобных задач.

`string->list` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_to_list` (*str*, *start*, *end*) [C Function]
`scm_string_to_list` (*str*) [C Function]
Convert the string *str* into a list of characters.

`string-split` *str char_pred* [Scheme Procedure]
`scm_string_split` (*str*, *char_pred*) [C Function]
Split the string *str* into a list of substrings delimited by appearances of characters that

- equal *char_pred*, if it is a character,
- satisfy the predicate *char_pred*, if it is a procedure,
- are in the set *char_pred*, if it is a character set.

Note that an empty substring between separator characters will result in an empty string in the result list.

```
(string-split "root:x:0:0:root:/root:/bin/bash" #\:)
⇒
("root" "x" "0" "0" "root" "/root" "/bin/bash")

(string-split "::" #\:)
⇒
("" "" "")

(string-split "" #\:)
⇒
("")
```

6.6.5.5 Строковые Селекторы

Этими процедурами могут быть извлечены части строк. `string-ref` предоставляет индивидуальне символы, в то время как `substring` можно использовать для извлечения подстрок из более длинных строк.

`string-length` *string* [Scheme Procedure]
`scm_string_length` (*string*) [C Function]
Return the number of characters in *string*.

`size_t scm_c_string_length` (*SCM str*) [C Function]
Return the number of characters in *str* as a `size_t`.

`string-ref` *str k* [Scheme Procedure]
`scm_string_ref` (*str*, *k*) [C Function]
Return character *k* of *str* using zero-origin indexing. *k* must be a valid index of *str*.

`SCM scm_c_string_ref` (*SCM str*, *size_t k*) [C Function]
Return character *k* of *str* using zero-origin indexing. *k* must be a valid index of *str*.

`string-copy` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_copy` (*str*, *start*, *end*) [C Function]

`scm_string_copy` (*str*) [C Function]
Return a copy of the given string *str*.

The returned string shares storage with *str* initially, but it is copied as soon as one of the two strings is modified.

`substring` *str start* [*end*] [Scheme Procedure]
`scm_substring` (*str, start, end*) [C Function]
Return a new string formed from the characters of *str* beginning with index *start* (inclusive) and ending with index *end* (exclusive). *str* must be a string, *start* and *end* must be exact integers satisfying:

0 <= *start* <= *end* <= (`string-length` *str*).

The returned string shares storage with *str* initially, but it is copied as soon as one of the two strings is modified.

`substring/shared` *str start* [*end*] [Scheme Procedure]
`scm_substring_shared` (*str, start, end*) [C Function]
Like `substring`, but the strings continue to share their storage even if they are modified. Thus, modifications to *str* show up in the new string, and vice versa.

`substring/copy` *str start* [*end*] [Scheme Procedure]
`scm_substring_copy` (*str, start, end*) [C Function]
Like `substring`, but the storage for the new string is copied immediately.

`substring/read-only` *str start* [*end*] [Scheme Procedure]
`scm_substring_read_only` (*str, start, end*) [C Function]
Like `substring`, but the resulting string can not be modified.

`SCM scm_c_substring` (*SCM str, size_t start, size_t end*) [C Function]
`SCM scm_c_substring_shared` (*SCM str, size_t start, size_t end*) [C Function]
`SCM scm_c_substring_copy` (*SCM str, size_t start, size_t end*) [C Function]
`SCM scm_c_substring_read_only` (*SCM str, size_t start, size_t end*) [C Function]
Like `scm_substring`, etc. but the bounds are given as a `size_t`.

`string-take` *s n* [Scheme Procedure]
`scm_string_take` (*s, n*) [C Function]
Return the *n* first characters of *s*.

`string-drop` *s n* [Scheme Procedure]
`scm_string_drop` (*s, n*) [C Function]
Return all but the first *n* characters of *s*.

`string-take-right` *s n* [Scheme Procedure]
`scm_string_take_right` (*s, n*) [C Function]
Return the *n* last characters of *s*.

`string-drop-right` *s n* [Scheme Procedure]
`scm_string_drop_right` (*s, n*) [C Function]
Return all but the last *n* characters of *s*.

`string-pad` *s len* [*chr* [*start* [*end*]]] [Scheme Procedure]
`string-pad-right` *s len* [*chr* [*start* [*end*]]] [Scheme Procedure]
`scm_string_pad` (*s, len, chr, start, end*) [C Function]
`scm_string_pad_right` (*s, len, chr, start, end*) [C Function]

Take characters *start* to *end* from the string *s* and either pad with *chr* or truncate them to give *len* characters.

`string-pad` pads or truncates on the left, so for example

```
(string-pad "x" 3)     ⇒ "  x"
(string-pad "abcde" 3) ⇒ "cde"
```

`string-pad-right` pads or truncates on the right, so for example

```
(string-pad-right "x" 3)     ⇒ "x  "
(string-pad-right "abcde" 3) ⇒ "abc"
```

`string-trim` *s* [*char_pred* [*start* [*end*]]] [Scheme Procedure]
`string-trim-right` *s* [*char_pred* [*start* [*end*]]] [Scheme Procedure]
`string-trim-both` *s* [*char_pred* [*start* [*end*]]] [Scheme Procedure]
`scm_string_trim` (*s, char_pred, start, end*) [C Function]
`scm_string_trim_right` (*s, char_pred, start, end*) [C Function]
`scm_string_trim_both` (*s, char_pred, start, end*) [C Function]

Trim occurrences of *char_pred* from the ends of *s*.

`string-trim` trims *char_pred* characters from the left (start) of the string, `string-trim-right` trims them from the right (end) of the string, `string-trim-both` trims from both ends.

char_pred can be a character, a character set, or a predicate procedure to call on each character. If *char_pred* is not given the default is whitespace as per `char-set:whitespace` (см. Раздел 6.6.4.6 [Standard Character Sets], страница 149).

```
(string-trim " x ")              ⇒ "x "
(string-trim-right "banana" #\a) ⇒ "banan"
(string-trim-both ".,xy:;" char-set:punctuation)
                  ⇒ "xy"
(string-trim-both "xyzzy" (lambda (c)
                            (or (eqv? c #\x)
                                (eqv? c #\y))))
                  ⇒ "zz"
```

6.6.5.6 Модификация Строк

Эти процедуры предназначены для изменения строк на месте. Это означает, что результат операции является не новой строкой; вместо этого изменяется представление памяти исходной строки.

`string-set!` *str k chr* [Scheme Procedure]
`scm_string_set_x` (*str, k, chr*) [C Function]

Store *chr* in element *k* of *str* and return an unspecified value. *k* must be a valid index of *str*.

`void scm_c_string_set_x` (*SCM str, size_t k, SCM chr*) [C Function]
Like `scm_string_set_x`, but the index is given as a `size_t`.

`string-fill!` *str chr* [*start* [*end*]] [Scheme Procedure]
`scm_substring_fill_x` (*str, chr, start, end*) [C Function]
`scm_string_fill_x` (*str, chr*) [C Function]
Stores *chr* in every element of the given *str* and returns an unspecified value.

`substring-fill!` *str start end fill* [Scheme Procedure]
`scm_substring_fill_x` (*str, start, end, fill*) [C Function]
Change every character in *str* between *start* and *end* to *fill*.

```
(define y (string-copy "abcdefg"))
(substring-fill! y 1 3 #\r)
y
⇒ "arrdefg"
```

`substring-move!` *str1 start1 end1 str2 start2* [Scheme Procedure]
`scm_substring_move_x` (*str1, start1, end1, str2, start2*) [C Function]
Copy the substring of *str1* bounded by *start1* and *end1* into *str2* beginning at position *start2*. *str1* and *str2* can be the same string.

`string-copy!` *target tstart s* [*start* [*end*]] [Scheme Procedure]
`scm_string_copy_x` (*target, tstart, s, start, end*) [C Function]
Copy the sequence of characters from index range [*start*, *end*) in string *s* to string *target*, beginning at index *tstart*. The characters are copied left-to-right or right-to-left as needed – the copy is guaranteed to work, even if *target* and *s* are the same string. It is an error if the copy operation runs off the end of the target string.

6.6.5.7 Сравнение строк

Процедуры в этом разделе аналогичны предиката используемым для сравнения символов (см. Раздел 6.6.3 [Characters], страница 139), но определены на последовательностях символов.

Первый набор указан в R5RS и имеет имена, которые заканчиваются `?`. Второй набор указан в SRFI-13 и его имена не заканчиваюься `?`.

Предикаты, заканчивающиеся на `-ci` игнорируют регистр символов когда сравнивают строки. На данный момент, сравнение без учета регистра выполняется с использованием правил R5RS, где каждый строчный символ который имеет одну форму в верхнем регистре преобразуется в верхний регистр перед сравнением. См. См. Раздел 6.25.2 [Text Collation], страница 487, для сравнения строк, зависящих от локали.

`string=?` *s1 s2 s3 ...* [Scheme Procedure]
Lexicographic equality predicate; return `#t` if all strings are the same length and contain the same characters in the same positions, otherwise return `#f`.

The procedure `string-ci=?` treats upper and lower case letters as though they were the same character, but `string=?` treats upper and lower case as distinct characters.

`string<?` *s1 s2 s3 . . .* [Scheme Procedure]
Lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically less than *str_i+1*.

`string<=?` *s1 s2 s3 . . .* [Scheme Procedure]
Lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically less than or equal to *str_i+1*.

`string>?` *s1 s2 s3 . . .* [Scheme Procedure]
Lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically greater than *str_i+1*.

`string>=?` *s1 s2 s3 . . .* [Scheme Procedure]
Lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically greater than or equal to *str_i+1*.

`string-ci=?` *s1 s2 s3 . . .* [Scheme Procedure]
Case-insensitive string equality predicate; return `#t` if all strings are the same length and their component characters match (ignoring case) at each position; otherwise return `#f`.

`string-ci<?` *s1 s2 s3 . . .* [Scheme Procedure]
Case insensitive lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically less than *str_i+1* regardless of case.

`string-ci<=?` *s1 s2 s3 . . .* [Scheme Procedure]
Case insensitive lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically less than or equal to *str_i+1* regardless of case.

`string-ci>?` *s1 s2 s3 . . .* [Scheme Procedure]
Case insensitive lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically greater than *str_i+1* regardless of case.

`string-ci>=?` *s1 s2 s3 . . .* [Scheme Procedure]
Case insensitive lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically greater than or equal to *str_i+1* regardless of case.

`string-compare` *s1 s2 proc_lt proc_eq proc_gt* **[***start1* **[***end1* **[***start2* **[***end2***]]]]** [Scheme Procedure]
`scm_string_compare` (*s1, s2, proc_lt, proc_eq, proc_gt, start1, end1, start2, end2*) [C Function]
Apply *proc_lt*, *proc_eq*, *proc_gt* to the mismatch index, depending upon whether *s1* is less than, equal to, or greater than *s2*. The mismatch index is the largest index *i* such that for every 0 <= *j* < *i*, *s1*[*j*] = *s2*[*j*] – that is, *i* is the first position that does not match.

`string-compare-ci` *s1 s2 proc_lt proc_eq proc_gt* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_compare_ci` (*s1, s2, proc_lt, proc_eq, proc_gt, start1, end1, start2, end2*) [C Function]

Apply *proc_lt*, *proc_eq*, *proc_gt* to the mismatch index, depending upon whether *s1* is less than, equal to, or greater than *s2*. The mismatch index is the largest index *i* such that for every 0 <= *j* < *i*, *s1*[*j*] = *s2*[*j*] – that is, *i* is the first position where the lowercased letters do not match.

`string=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_eq` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* and *s2* are not equal, a true value otherwise.

`string<>` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_neq` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* and *s2* are equal, a true value otherwise.

`string<` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_lt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is greater or equal to *s2*, a true value otherwise.

`string>` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_gt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is less or equal to *s2*, a true value otherwise.

`string<=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_le` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is greater to *s2*, a true value otherwise.

`string>=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ge` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is less to *s2*, a true value otherwise.

`string-ci=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_eq` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* and *s2* are not equal, a true value otherwise. The character comparison is done case-insensitively.

`string-ci<>` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_neq` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* and *s2* are equal, a true value otherwise. The character comparison is done case-insensitively.

`string-ci<` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_lt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is greater or equal to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-ci>` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_gt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is less or equal to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-ci<=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_le` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return `#f` if *s1* is greater to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-ci>=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_ge` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return `#f` if *s1* is less to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-hash` *s* [*bound* [*start* [*end*]]] [Scheme Procedure]
`scm_substring_hash` (*s, bound, start, end*) [C Function]
Compute a hash value for *s*. The optional argument *bound* is a non-negative exact integer specifying the range of the hash function. A positive value restricts the return value to the range [0,bound).

`string-hash-ci` *s* [*bound* [*start* [*end*]]] [Scheme Procedure]
`scm_substring_hash_ci` (*s, bound, start, end*) [C Function]
Compute a hash value for *s*. The optional argument *bound* is a non-negative exact integer specifying the range of the hash function. A positive value restricts the return value to the range [0,bound).

Поскольку один и тот же визуальный вид абстрактного символа Unicode может быть получен через несколько последовательностей символов Unicode, даже при сравнении строк без учета регистра, функции описанные вышее, могут возвращать `#f` когда представлены строки, содержащие разные представления одного и того же символа. Например, символ Unicode “LATIN SMALL LETTER S WITH DOT BELOW AND DOT ABOVE” может быть представлен одним символом (U+1E69) или символом “LATIN SMALL LETTER S” (U+0073), за которым следует комбинирующие метки “COMBINING DOT BELOW” (U+0323) и “COMBINING DOT ABOVE” (U+0307).

По этой причине часто желательно обеспечить, чтобы сопоставляемые строки использовали взаимно согласованное представление для каждого символа. Стандарт Unicode определяет два метода нормализации содержимого строк: разложение, которое разбивает сотавные символы в набор составных символов с упорядочением, определенным стандартом Unicode, и композицию, выполняющую обратное преобразование.

Существуют две операции разложения. “Каноническое разложение” порождает последовательность символов, которые имеют тот же внешний вид, что и исходные символы, тогда как “Совместимое разложение” производит такие последовательности, чьи визуальные отображения могут отличаться от оригиналов, но которые представляют собой абстрактный символ.

Эти операции инкапсулируются в следующем набор нормализационных форм:

NFD Символы разлагаются на их канонические формы.

NFKD Символы разлагаются на их совместимые формы.

NFC Символы разлагаются на их канонические формы, а затем объединяются

NFKC Символы разлагаются на их совместимые формы, а затем объединяются

Приведенные ниже функции приводят аргументы в одну из описанных выше форм.

`string-normalize-nfd` *s* [Scheme Procedure]
`scm_string_normalize_nfd` (*s*) [C Function]
Return the `NFD` normalized form of *s*.

`string-normalize-nfkd` *s* [Scheme Procedure]
`scm_string_normalize_nfkd` (*s*) [C Function]
Return the `NFKD` normalized form of *s*.

`string-normalize-nfc` *s* [Scheme Procedure]
`scm_string_normalize_nfc` (*s*) [C Function]
Return the `NFC` normalized form of *s*.

`string-normalize-nfkc` *s* [Scheme Procedure]
`scm_string_normalize_nfkc` (*s*) [C Function]
Return the `NFKC` normalized form of *s*.

6.6.5.8 Строковый Поиск

`string-index` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_index` (*s, char_pred, start, end*) [C Function]
Search through the string *s* from left to right, returning the index of the first occurrence of a character which

- equals *char_pred*, if it is character,
- satisfies the predicate *char_pred*, if it is a procedure,
- is in the set *char_pred*, if it is a character set.

Return `#f` if no match is found.

`string-rindex` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_rindex` (*s, char_pred, start, end*) [C Function]
Search through the string *s* from right to left, returning the index of the last occurrence of a character which

- equals *char_pred*, if it is character,
- satisfies the predicate *char_pred*, if it is a procedure,
- is in the set if *char_pred* is a character set.

Return `#f` if no match is found.

`string-prefix-length` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_length` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return the length of the longest common prefix of the two strings.

`string-prefix-length-ci` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_length_ci` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return the length of the longest common prefix of the two strings, ignoring character case.

`string-suffix-length` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_length` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return the length of the longest common suffix of the two strings.

`string-suffix-length-ci` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_length_ci` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return the length of the longest common suffix of the two strings, ignoring character case.

`string-prefix?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_p` (*s1, s2, start1, end1, start2, end2*) [C Function]
Is *s1* a prefix of *s2*?

`string-prefix-ci?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_ci_p` (*s1, s2, start1, end1, start2, end2*) [C Function]
Is *s1* a prefix of *s2*, ignoring character case?

`string-suffix?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_p` (*s1, s2, start1, end1, start2, end2*) [C Function]
Is *s1* a suffix of *s2*?

`string-suffix-ci?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_ci_p` (*s1, s2, start1, end1, start2, end2*) [C Function]
Is *s1* a suffix of *s2*, ignoring character case?

`string-index-right` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_index_right` (*s, char_pred, start, end*) [C Function]
Search through the string *s* from right to left, returning the index of the last occurrence of a character which

- equals *char_pred*, if it is character,
- satisfies the predicate *char_pred*, if it is a procedure,
- is in the set if *char_pred* is a character set.

Return `#f` if no match is found.

`string-skip` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_skip` (*s, char_pred, start, end*) [C Function]
Search through the string *s* from left to right, returning the index of the first occurrence of a character which

- does not equal *char_pred*, if it is character,
- does not satisfy the predicate *char_pred*, if it is a procedure,
- is not in the set if *char_pred* is a character set.

`string-skip-right` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_skip_right` (*s, char_pred, start, end*) [C Function]
Search through the string *s* from right to left, returning the index of the last occurrence of a character which

- does not equal *char_pred*, if it is character,

- does not satisfy the predicate *char_pred*, if it is a procedure,
- is not in the set if *char_pred* is a character set.

`string-count` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_count` (*s, char_pred, start, end*) [C Function]
Return the count of the number of characters in the string *s* which

- equals *char_pred*, if it is character,
- satisfies the predicate *char_pred*, if it is a procedure.
- is in the set *char_pred*, if it is a character set.

`string-contains` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_contains` (*s1, s2, start1, end1, start2, end2*) [C Function]
Does string *s1* contain string *s2*? Return the index in *s1* where *s2* occurs as a substring, or false. The optional start/end indices restrict the operation to the indicated substrings.

`string-contains-ci` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_contains_ci` (*s1, s2, start1, end1, start2, end2*) [C Function]
Does string *s1* contain string *s2*? Return the index in *s1* where *s2* occurs as a substring, or false. The optional start/end indices restrict the operation to the indicated substrings. Character comparison is done case-insensitively.

6.6.5.9 Алфавитное Преобразование

Это процедуры дле преобразования строк в их эквиваленты в верхнем или нижнем регистре, соотвественно, или для заглавных строк.

Они используют основные правила отображение регистров для символов Unicode. Нет специального языка или рассматриваются контекстные правила. Полученные строки будут гарантированно иметь ту же длину, что и входные строки.

См. Раздел 6.25.3 [Character Case Mapping], страница 488, для преобразований зависящих от локали.

`string-upcase` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_upcase` (*str, start, end*) [C Function]
`scm_string_upcase` (*str*) [C Function]
Upcase every character in `str`.

`string-upcase!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_upcase_x` (*str, start, end*) [C Function]
`scm_string_upcase_x` (*str*) [C Function]
Destructively upcase every character in `str`.

```
(string-upcase! y)
⇒ "ARRDEFG"
y
⇒ "ARRDEFG"
```

`string-downcase` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_downcase` (*str, start, end*) [C Function]
`scm_string_downcase` (*str*) [C Function]
Downcase every character in *str*.

`string-downcase!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_downcase_x` (*str*, *start*, *end*) [C Function]
`scm_string_downcase_x` (*str*) [C Function]
Destructively downcase every character in *str*.

```
y
⇒ "ARRDEFG"
(string-downcase! y)
⇒ "arrdefg"
y
⇒ "arrdefg"
```

`string-capitalize` *str* [Scheme Procedure]
`scm_string_capitalize` (*str*) [C Function]
Return a freshly allocated string with the characters in *str*, where the first character of every word is capitalized.

`string-capitalize!` *str* [Scheme Procedure]
`scm_string_capitalize_x` (*str*) [C Function]
Upcase the first character of every word in *str* destructively and return *str*.

```
y                      ⇒ "hello world"
(string-capitalize! y) ⇒ "Hello World"
y                      ⇒ "Hello World"
```

`string-titlecase` *str* [*start* [*end*]] [Scheme Procedure]
`scm_string_titlecase` (*str*, *start*, *end*) [C Function]
Titlecase every first character in a word in *str*.

`string-titlecase!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_string_titlecase_x` (*str*, *start*, *end*) [C Function]
Destructively titlecase every first character in a word in *str*.

6.6.5.10 Поворот и Добавление Строк

`string-reverse` *str* [*start* [*end*]] [Scheme Procedure]
`scm_string_reverse` (*str*, *start*, *end*) [C Function]
Reverse the string *str*. The optional arguments *start* and *end* delimit the region of *str* to operate on.

`string-reverse!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_string_reverse_x` (*str*, *start*, *end*) [C Function]
Reverse the string *str* in-place. The optional arguments *start* and *end* delimit the region of *str* to operate on. The return value is unspecified.

`string-append` *arg* ... [Scheme Procedure]
`scm_string_append` (*args*) [C Function]
Return a newly allocated string whose characters form the concatenation of the given strings, *arg* ....

```
(let ((h "hello "))
  (string-append h "world"))
⇒ "hello world"
```

`string-append/shared` *arg ...* [Scheme Procedure]
`scm_string_append_shared` (*args*) [C Function]
Like `string-append`, but the result may share memory with the argument strings.

`string-concatenate` *ls* [Scheme Procedure]
`scm_string_concatenate` (*ls*) [C Function]
Append the elements (which must be strings) of *ls* together into a single string. Guaranteed to return a freshly allocated string.

`string-concatenate-reverse` *ls* [*final_string* [*end*]] [Scheme Procedure]
`scm_string_concatenate_reverse` (*ls, final_string, end*) [C Function]
Without optional arguments, this procedure is equivalent to

```
(string-concatenate (reverse ls))
```

If the optional argument *final_string* is specified, it is consed onto the beginning to *ls* before performing the list-reverse and string-concatenate operations. If *end* is given, only the characters of *final_string* up to index *end* are used.

Guaranteed to return a freshly allocated string.

`string-concatenate/shared` *ls* [Scheme Procedure]
`scm_string_concatenate_shared` (*ls*) [C Function]
Like `string-concatenate`, but the result may share memory with the strings in the list *ls*.

`string-concatenate-reverse/shared` *ls* [*final_string* [*end*]] [Scheme Procedure]
`scm_string_concatenate_reverse_shared` (*ls, final_string, end*) [C Function]
Like `string-concatenate-reverse`, but the result may share memory with the strings in the *ls* arguments.

6.6.5.11 Отображение(Map), Сворачивание(Fold) и Разворачивание(Unfold) Строк

`string-map` *proc s* [*start* [*end*]] [Scheme Procedure]
`scm_string_map` (*proc, s, start, end*) [C Function]
proc is a char->char procedure, it is mapped over *s*. The order in which the procedure is applied to the string elements is not specified.

`string-map!` *proc s* [*start* [*end*]] [Scheme Procedure]
`scm_string_map_x` (*proc, s, start, end*) [C Function]
proc is a char->char procedure, it is mapped over *s*. The order in which the procedure is applied to the string elements is not specified. The string *s* is modified in-place, the return value is not specified.

`string-for-each` *proc s* [*start* [*end*]] [Scheme Procedure]
`scm_string_for_each` (*proc, s, start, end*) [C Function]
proc is mapped over *s* in left-to-right order. The return value is not specified.

`string-for-each-index` *proc s* [*start* [*end*]] [Scheme Procedure]
`scm_string_for_each_index` (*proc, s, start, end*) [C Function]
Call (*proc* `i`) for each index i in *s*, from left to right.

For example, to change characters to alternately upper and lower case,

```
(define str (string-copy "studly"))
(string-for-each-index
    (lambda (i)
      (string-set! str i
        ((if (even? i) char-upcase char-downcase)
         (string-ref str i))))
    str)
str ⇒ "StUdLy"
```

`string-fold` *kons knil s* [*start* [*end*]] [Scheme Procedure]
`scm_string_fold` (*kons, knil, s, start, end*) [C Function]

Fold *kons* over the characters of *s*, with *knil* as the terminating element, from left to right. *kons* must expect two arguments: The actual character and the last result of *kons*' application.

`string-fold-right` *kons knil s* [*start* [*end*]] [Scheme Procedure]
`scm_string_fold_right` (*kons, knil, s, start, end*) [C Function]

Fold *kons* over the characters of *s*, with *knil* as the terminating element, from right to left. *kons* must expect two arguments: The actual character and the last result of *kons*' application.

`string-unfold` *p f g seed* [*base* [*make_final*]] [Scheme Procedure]
`scm_string_unfold` (*p, f, g, seed, base, make_final*) [C Function]

- *g* is used to generate a series of *seed* values from the initial *seed*: *seed*, (*g seed*), (*g*^2 *seed*), (*g*^3 *seed*), ...
- *p* tells us when to stop – when it returns true when applied to one of these seed values.
- *f* maps each seed value to the corresponding character in the result string. These chars are assembled into the string in a left-to-right order.
- *base* is the optional initial/leftmost portion of the constructed string; it default to the empty string.
- *make_final* is applied to the terminal seed value (on which *p* returns true) to produce the final/rightmost portion of the constructed string. The default is nothing extra.

`string-unfold-right` *p f g seed* [*base* [*make_final*]] [Scheme Procedure]
`scm_string_unfold_right` (*p, f, g, seed, base, make_final*) [C Function]

- *g* is used to generate a series of *seed* values from the initial *seed*: *seed*, (*g seed*), (*g*^2 *seed*), (*g*^3 *seed*), ...
- *p* tells us when to stop – when it returns true when applied to one of these seed values.
- *f* maps each seed value to the corresponding character in the result string. These chars are assembled into the string in a right-to-left order.
- *base* is the optional initial/rightmost portion of the constructed string; it default to the empty string.

- *make_final* is applied to the terminal seed value (on which *p* returns true) to produce the final/leftmost portion of the constructed string. It defaults to `(lambda (x) )`.

6.6.5.12 Различные строковые операции

`xsubstring` *s from* [*to* [*start* [*end*]]] [Scheme Procedure]
`scm_xsubstring` (*s, from, to, start, end*) [C Function]

This is the *extended substring* procedure that implements replicated copying of a substring of some string.

s is a string, *start* and *end* are optional arguments that demarcate a substring of *s*, defaulting to 0 and the length of *s*. Replicate this substring up and down index space, in both the positive and negative directions. `xsubstring` returns the substring of this string beginning at index *from*, and ending at *to*, which defaults to *from* + (*end* - *start*).

`string-xcopy!` *target tstart s sfrom* [*sto* [*start* [*end*]]] [Scheme Procedure]
`scm_string_xcopy_x` (*target, tstart, s, sfrom, sto, start, end*) [C Function]

Exactly the same as `xsubstring`, but the extracted text is written into the string *target* starting at index *tstart*. The operation is not defined if `(eq? target s)` or these arguments share storage – you cannot copy a string on top of itself.

`string-replace` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_replace` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return the string *s1*, but with the characters *start1* ... *end1* replaced by the characters *start2* ... *end2* from *s2*.

`string-tokenize` *s* [*token_set* [*start* [*end*]]] [Scheme Procedure]
`scm_string_tokenize` (*s, token_set, start, end*) [C Function]

Split the string *s* into a list of substrings, where each substring is a maximal non-empty contiguous sequence of characters from the character set *token_set*, which defaults to `char-set:graphic`. If *start* or *end* indices are provided, they restrict `string-tokenize` to operating on the indicated substring of *s*.

`string-filter` *char_pred s* [*start* [*end*]] [Scheme Procedure]
`scm_string_filter` (*char_pred, s, start, end*) [C Function]

Filter the string *s*, retaining only those characters which satisfy *char_pred*.

If *char_pred* is a procedure, it is applied to each character as a predicate, if it is a character, it is tested for equality and if it is a character set, it is tested for membership.

`string-delete` *char_pred s* [*start* [*end*]] [Scheme Procedure]
`scm_string_delete` (*char_pred, s, start, end*) [C Function]

Delete characters satisfying *char_pred* from *s*.

If *char_pred* is a procedure, it is applied to each character as a predicate, if it is a character, it is tested for equality and if it is a character set, it is tested for membership.

6.6.5.13 Представление строк как байтов.

В холодном мире за пределами Guile не все строки обрабатываются одинаково. Снаружи есть только байты, и существует много способов представления строк (последовательностей символов) как двоичных данных (последовательностей байтов).

Как пользователю, вам обычно не надо думать. Когда вы вводите, ваша клавиатура, ваша система кодирует все ваши нажатия клавиш в виде байтов в соответствии с языком, который вы настроили на своем компьютере. Guile использует локаль для декодирования этих байтов назад в символы — мы надеемся, в те же символы, что вы ввели.

Все становиться менее понятным при работе с системой с несколькими пользователями, например с веб-сервером. Ваш веб-сервер может получить запрос от одного пользователя для данных закодированных в наборе символов ISO-8859-1, а затем получить запрос от другого пользователя дя данных UTF-8.

Guile предоставляет модуль *iconv* для преобразования строк и последовательностей байтов. См. Раздел 6.6.12 [Bytevectors], страница 205, для получения дополнительной информации о том как Guile представляет собой байты исходной последовательности. Этот модуль получил свое имя от общей команды unix с тем же именем.

Обратите внимание, что часто достаточно просто читать и писать строки из портов вместо использования этой функции. Для этого укажите кодировку порта с помощью `set-port-encoding!`. См. Раздел 6.14.1 [Ports], страница 352, еще больше о портаи и кодировках символов.

В отличие от остальных процедур в этом разделе вам нужно загрузить модуль `iconv` перед доступом к этим процедурам:

```
(use-modules (ice-9 iconv))
```

`string->bytevector` *string encoding* **[***conversion-strategy***]** [Scheme Procedure]
Encode *string* as a sequence of bytes.

The string will be encoded in the character set specified by the *encoding* string. If the string has characters that cannot be represented in the encoding, by default this procedure raises an `encoding-error`. Pass a *conversion-strategy* argument to specify other behaviors.

The return value is a bytevector. См. Раздел 6.6.12 [Bytevectors], страница 205, for more on bytevectors. См. Раздел 6.14.1 [Ports], страница 352, for more on character encodings and conversion strategies.

`bytevector->string` *bytevector encoding* **[***conversion-strategy***]** [Scheme Procedure]
Decode *bytevector* into a string.

The bytes will be decoded from the character set by the *encoding* string. If the bytes do not form a valid encoding, by default this procedure raises an `decoding-error`. As with `string->bytevector`, pass the optional *conversion-strategy* argument to modify this behavior. См. Раздел 6.14.1 [Ports], страница 352, for more on character encodings and conversion strategies.

`call-with-output-encoded-string` *encoding proc* [*conversion-strategy*] [Scheme Procedure]

Like `call-with-output-string`, but instead of returning a string, returns a encoding of the string according to *encoding*, as a bytevector. This procedure can be more efficient than collecting a string and then converting it via `string->bytevector`.

6.6.5.14 Преобразование в/из Си

При создании строки Scheme из строки Си или при перобразовании строки Scheme в строку Си, концепция кодирования символов становиться важной.

В Си строка представляет собой всего лишь последовательность байтов, а кодировка символов описывает отношение между этими байтами и фактическми символами, которые составляют сторку. Для строк Scheme, кодировка символов не является проблемой (в основном), так как на Scheme вы обычно используете строки как последовательности символов, а не как последовательности байтов.

Преобразование в Си И преобразование из Си имеют свои проблемы.

При преобразовании из Си в Scheme важно, чтобы последовательность байтов в строке Си была действительной впо отношению к ее кодированию. Например, сторки ASCII не могут иметь байтов превышающих значение 127. Байт ASCII превышающий 127 считается *плохо сформированным* и не может преобразовываться в символы Scheme.

Проблемы могут возникнуть и при обратном преобразовании. Не все кодировки символов могут содержать все возможные символы Scheme. Например, некоторые кодировки, наприер ASCII, могут описать небольшое подмножество всех возможных символов. Итак, при преобразовании в Си сначала необходимо решить, что деалть с символами Scheme, которые не могут быть представлены в строке Си.

Преобразуя сторoку Scheme в строку Си часто выделяют свежую память для хранения результата. Вы должны позаботиться о том, что бы эта память была освобождена должным образом. Во многих случаях, это может быть сделано с помощью исользования `scm_dynwind_free` в соответствующем контексте dynwind, См. Раздел 6.13.10 [Dynamic Wind], страница 339.

`SCM scm_from_locale_string` (*const char *str*) [C Function]
`SCM scm_from_locale_stringn` (*const char *str, size_t len*) [C Function]

Creates a new Scheme string that has the same contents as *str* when interpreted in the character encoding of the current locale.

For `scm_from_locale_string`, *str* must be null-terminated.

For `scm_from_locale_stringn`, *len* specifies the length of *str* in bytes, and *str* does not need to be null-terminated. If *len* is `(size_t)-1`, then *str* does need to be null-terminated and the real length will be found with `strlen`.

If the C string is ill-formed, an error will be raised.

Note that these functions should *not* be used to convert C string constants, because there is no guarantee that the current locale will match that of the execution character set, used for string and character constants. Most modern C compilers use UTF-8 by default, so to convert C string constants we recommend `scm_from_utf8_string`.

`SCM scm_take_locale_string` (*char *str*) [C Function]
`SCM scm_take_locale_stringn` (*char *str, size_t len*) [C Function]
Like `scm_from_locale_string` and `scm_from_locale_stringn`, respectively, but also frees *str* with `free` eventually. Thus, you can use this function when you would free *str* anyway immediately after creating the Scheme string. In certain cases, Guile can then use *str* directly as its internal representation.

`char * scm_to_locale_string` (*SCM str*) [C Function]
`char * scm_to_locale_stringn` (*SCM str, size_t *lenp*) [C Function]
Returns a C string with the same contents as *str* in the character encoding of the current locale. The C string must be freed with `free` eventually, maybe by using `scm_dynwind_free`, См. Раздел 6.13.10 [Dynamic Wind], страница 339.

For `scm_to_locale_string`, the returned string is null-terminated and an error is signalled when *str* contains `#\nul` characters.

For `scm_to_locale_stringn` and *lenp* not `NULL`, *str* might contain `#\nul` characters and the length of the returned string in bytes is stored in `*lenp`. The returned string will not be null-terminated in this case. If *lenp* is `NULL`, `scm_to_locale_stringn` behaves like `scm_to_locale_string`.

If a character in *str* cannot be represented in the character encoding of the current locale, the default port conversion strategy is used. См. Раздел 6.14.1 [Ports], страница 352, for more on conversion strategies.

If the conversion strategy is `error`, an error will be raised. If it is `substitute`, a replacement character, such as a question mark, will be inserted in its place. If it is `escape`, a hex escape will be inserted in its place.

`size_t scm_to_locale_stringbuf` (*SCM str, char *buf, size_t max_len*) [C Function]
Puts *str* as a C string in the current locale encoding into the memory pointed to by *buf*. The buffer at *buf* has room for *max_len* bytes and `scm_to_local_stringbuf` will never store more than that. No terminating `'\0'` will be stored.

The return value of `scm_to_locale_stringbuf` is the number of bytes that are needed for all of *str*, regardless of whether *buf* was large enough to hold them. Thus, when the return value is larger than *max_len*, only *max_len* bytes have been stored and you probably need to try again with a larger buffer.

For most situations, string conversion should occur using the current locale, such as with the functions above. But there may be cases where one wants to convert strings from a character encoding other than the locale's character encoding. For these cases, the lower-level functions `scm_to_stringn` and `scm_from_stringn` are provided. These functions should seldom be necessary if one is properly using locales.

`scm_t_string_failed_conversion_handler` [C Type]
This is an enumerated type that can take one of three values: `SCM_FAILED_CONVERSION_ERROR`, `SCM_FAILED_CONVERSION_QUESTION_MARK`, and `SCM_FAILED_CONVERSION_ESCAPE_SEQUENCE`. They are used to indicate a strategy for handling characters that cannot be converted to or from a given character encoding. `SCM_FAILED_CONVERSION_ERROR` indicates that a conversion should

throw an error if some characters cannot be converted. `SCM_FAILED_CONVERSION_QUESTION_MARK` indicates that a conversion should replace unconvertable characters with the question mark character. And, `SCM_FAILED_CONVERSION_ESCAPE_SEQUENCE` requests that a conversion should replace an unconvertable character with an escape sequence.

While all three strategies apply when converting Scheme strings to C, only `SCM_FAILED_CONVERSION_ERROR` and `SCM_FAILED_CONVERSION_QUESTION_MARK` can be used when converting C strings to Scheme.

`char *scm_to_stringn` (*SCM str, size_t *lenp, const char *encoding, scm_t_string_failed_conversion_handler handler*) [C Function]

This function returns a newly allocated C string from the Guile string *str*. The length of the returned string in bytes will be returned in *lenp*. The character encoding of the C string is passed as the ASCII, null-terminated C string *encoding*. The *handler* parameter gives a strategy for dealing with characters that cannot be converted into *encoding*.

If *lenp* is `NULL`, this function will return a null-terminated C string. It will throw an error if the string contains a null character.

The Scheme interface to this function is `string->bytevector`, from the `ice-9 iconv` module. См. Раздел 6.6.5.13 [Representing Strings as Bytes], страница 169.

`SCM scm_from_stringn` (*const char *str, size_t len, const char *encoding, scm_t_string_failed_conversion_handler handler*) [C Function]

This function returns a scheme string from the C string *str*. The length in bytes of the C string is input as *len*. The encoding of the C string is passed as the ASCII, null-terminated C string `encoding`. The *handler* parameters suggests a strategy for dealing with unconvertable characters.

The Scheme interface to this function is `bytevector->string`. См. Раздел 6.6.5.13 [Representing Strings as Bytes], страница 169.

The following conversion functions are provided as a convenience for the most commonly used encodings.

`SCM scm_from_latin1_string` (*const char *str*) [C Function]
`SCM scm_from_utf8_string` (*const char *str*) [C Function]
`SCM scm_from_utf32_string` (*const scm_t_wchar *str*) [C Function]

Return a scheme string from the null-terminated C string *str*, which is ISO-8859-1-, UTF-8-, or UTF-32-encoded. These functions should be used to convert hard-coded C string constants into Scheme strings.

`SCM scm_from_latin1_stringn` (*const char *str, size_t len*) [C Function]
`SCM scm_from_utf8_stringn` (*const char *str, size_t len*) [C Function]
`SCM scm_from_utf32_stringn` (*const scm_t_wchar *str, size_t len*) [C Function]

Return a scheme string from C string *str*, which is ISO-8859-1-, UTF-8-, or UTF-32-encoded, of length *len*. *len* is the number of bytes pointed to by *str* for `scm_from_latin1_stringn` and `scm_from_utf8_stringn`; it is the number of elements (code points) in *str* in the case of `scm_from_utf32_stringn`.

`char *scm_to_latin1_stringn` (*SCM str, size_t *lenp*) [C function]
`char *scm_to_utf8_stringn` (*SCM str, size_t *lenp*) [C function]
`scm_t_wchar *scm_to_utf32_stringn` (*SCM str, size_t *lenp*) [C function]
Return a newly allocated, ISO-8859-1-, UTF-8-, or UTF-32-encoded C string from Scheme string *str*. An error is thrown when *str* cannot be converted to the specified encoding. If *lenp* is `NULL`, the returned C string will be null terminated, and an error will be thrown if the C string would otherwise contain null characters. If *lenp* is not `NULL`, the string is not null terminated, and the length of the returned string is returned in *lenp*. The length returned is the number of bytes for `scm_to_latin1_stringn` and `scm_to_utf8_stringn`; it is the number of elements (code points) for `scm_to_utf32_stringn`.

It is not often the case, but sometimes when you are dealing with the implementation details of a port, you need to encode and decode strings according to the encoding and conversion strategy of the port. There are some convenience functions for that purpose as well.

`SCM scm_from_port_string` (*const char *str, SCM port*) [C Function]
`SCM scm_from_port_stringn` (*const char *str, size_t len, SCM port*) [C Function]
`char* scm_to_port_string` (*SCM str, SCM port*) [C Function]
`char* scm_to_port_stringn` (*SCM str, size_t *lenp, SCM port*) [C Function]
Like `scm_from_stringn` and friends, except they take their encoding and conversion strategy from a given port object.

6.6.5.15 Внутренности строки

Guile хранит каждую строку в памяти как непрерывный массив кодовых точек Unicode связанных с набором атрибутов. Если все кодовые точки имеют целочисленный диапазон от 0 до 255 включительно, массив кодовых точек храниться как один байт на кодовую точку: такое сохранение является строкой в коде ISO-8859-1 (т.е Latin-1). Если какая-либо из кодовых точек стороки имеет целочисленное значение больше 255, мвссив кодовых точек храниться как четыре байта на кодовую точку: он сохраняется как строка UTF-32.

Преобразование между представлениями в один байт на код и четыре байта на кодовую точку происходит автоматически по мере необходимости.

API для установки внутреннего представления строк не предоставляется, однако существуют две процедуры для доступа к ней. Это процедуры отладки. Использовать их в рабочем коде крайне не рекомендуется, так как детали внутреннего представления строк Guile могут измениться в последующих выпусках.

`string-bytes-per-char` *str* [Scheme Procedure]
`scm_string_bytes_per_char` (*str*) [C Function]
Return the number of bytes used to encode a Unicode code point in string *str*. The result is one or four.

`%string-dump` *str* [Scheme Procedure]
`scm_sys_string_dump` (*str*) [C Function]
Returns an association list containing debugging information for *str*. The association list has the following entries.

`string` The string itself.

`start` The start index of the string into its stringbuf

`length` The length of the string

`shared` If this string is a substring, it returns its parent string. Otherwise, it returns `#f`

`read-only`
`#t` if the string is read-only

`stringbuf-chars`
A new string containing this string's stringbuf's characters

`stringbuf-length`
The number of characters in this stringbuf

`stringbuf-shared`
`#t` if this stringbuf is shared

`stringbuf-wide`
`#t` if this stringbuf's characters are stored in a 32-bit buffer, or `#f` if they are stored in an 8-bit buffer

6.6.6 Символы

Символы в Scheme широко используются тремя способами: как элементы дискретных данных, как ключи для поиска в ассоциативных списках alist и хеш-таблиц, а также для обозначения ссылок на переменные.

Символ похож на строку, поскольку она определяется последовательностью символов. Последовательность символов называется *Именем* символа. В обычном случае, т.е когда имя символа не содержит букв, которые можно было бы спутать с другими элементами синтаксиса Scheme — символ записывается в программе Scheme последовательностью букв(цифр и др. символов), которые составляют имя, *без кавычек* или какого другого специального синтаксиса. Например, символ имя которого "multiply-by-2" записывается, просто:

```
multiply-by-2
```

Обратите внимание, что объявление отличается от *строки* с содержимым "multiply-by-2", которое записано с двойными кавычками, например:

```
"multiply-by-2"
```

Выходя за рамки того, как они описаны, символы отличаются от строк в двух важных отношениях.

Первым важным отличием является уникальность. Если однотипная строка читается дважды в двух разных местах программы, результатом будут два разных строковых объекта, содержимое которых будет одно и тоже. Если, с другой стороны, один и тот же символ читается дважды из двух разных мест программы, результат будет один и тот же символьный объект.

Получив два прочитанных символа, вы можете использовать `eq?` для проверки их идентичности (т.е имеют ли они одно и тоже имя). Оператор `eq?` является наиболее эффективным оператором сравнения в Scheme, и сравнивает два таких символа так же

быстро как сравнивают, например числа. Получая две строки, с другой стороны, вы должны использовать оператор `equal?` или `string=?`, которые значительно медленне оператора eq, что бы определить имеют строки одно и тоже содержимое.

```
(define sym1 (quote hello))
(define sym2 (quote hello))
(eq? sym1 sym2) ⇒ #t

(define str1 "hello")
(define str2 "hello")
(eq? str1 str2) ⇒ #f
(equal? str1 str2) ⇒ #t
```

Второе важное отличие состоит в том, что символы, в отличие от строк, не являются самовычисляемыми объектами. Вот почему нам нужена операция `(quote ...)`s в примере выше: `(quote hello)` блокирует вычисление символа с именем "hello", тогда как бескавычек `hello` *читается* как символ с именем "hello" и вычисляется как ссылка на переменную ... о чем мы расскажем ниже (см. Раздел 6.6.6.3 [Symbol Variables], страница 177).

6.6.6.1 Символ как дискретные данные

Числа и символы сходны в том отношении, что оба типа поддаются сравнению с помощью `eq?`. Но символы более наглядны, чем цифры, поэтому имя символа может использоваться непосредственно для описания концепции, для которой используется этот символ.

Например, представьте, что вам нужно представить несколько цветов в компьютерной программе. Используя числа, вы должны выбрать произвольно некоторое сопоставление между числами и цветом, а затем позаботиться о том, чтобы использовать это сопоставление корректно.

```
;; 1=red, 2=green, 3=purple

(if (eq? (colour-of vehicle) 1)
    ...)
```

Вы можете сделать это сопоставление более явным, а код более читабельным, определяя константы:

```
(define red 1)
(define green 2)
(define purple 3)

(if (eq? (colour-of vehicle) red)
    ...)
```

Но самый простой и ясный подход - не испоьзовать числа вообще, а только символы, чьи имена определяют цвета, на которые они ссылаются:

```
(if (eq? (colour-of vehicle) 'red)
    ...)
```

Описательные преимущества символов над числами возрастают как набор понятий, которые вы хотите описать. Предположим, что объект автомобиль, может иметь и другие свойтсва, например имеет или использует:

- automatic or manual transmission
- leaded or unleaded fuel
- power steering (or not).

Тогда совокупный набор свойств автомобиля может быть представлен естественным образом и управляться как список символов:

```
(properties-of vehicle1)
⇒
(red manual unleaded power-steering)

(if (memq 'power-steering (properties-of vehicle1))
    (display "Unfit people can drive this vehicle.\n")
    (display "You'll need strong arms to drive this vehicle!\n"))
⊣
Unfit people can drive this vehicle.
```

Запомните, что основное свойство символов, на которое мы полагаеся, состоит в том что появление `'red` в одной части программы является неотличимым символом от появления `'red` в другой части программы; это означает, что символы можно эффективно сравнивать с использованием `eq?`. В то же время символы имеют естественные описательные имена. Это сочетание эффективности и описательной мощи делает их идельными для использования в качестве дискретных данных.

6.6.6.2 Символы как Ключи Поиска

Учитывая их эффективность и описательную силу, естественно использовать символы в качестве ключей в ассоциативном списке или хеш-таблицах.

Чтобы проилюстрировать это, рассмотрим более структурированное представление свойств автомобиля из предыдущего подраздела. Вместо того, чтобы смешивать все свойства вместе в простом списке, мы могли бы использовать ассоциативный список следующим образом:

```
(define car1-properties '((colour . red)
                          (transmission . manual)
                          (fuel . unleaded)
                          (steering . power-assisted)))
```

Обратите внимание, что эта структура более ясная и расширяемая, чем плоский список. Например он ясно показывает, что ручное относиться к передаче, а не к окнам или блокировке автомобиля. Он также позволяет другим свойствам использовать одни и те же символы среди своих возможных значений, не становясь двусмысленными.

```
(define car1-properties '((colour . red)
                          (transmission . manual)
                          (fuel . unleaded)
                          (steering . power-assisted)
                          (seat-colour . red)
```

```
                                (locking . manual)))
```

С таким представлением легко использовать эффективное семейство процедур `assq-XXX` из списка операций с ассоцированными списками (см. Раздел 6.6.20 [Association Lists], страница 243) для извлечения или изменения отдельных фрагментов информации:

```
(assq-ref car1-properties 'fuel) ⇒ unleaded
(assq-ref car1-properties 'transmission) ⇒ manual

(assq-set! car1-properties 'seat-colour 'black)
⇒
((colour . red)
 (transmission . manual)
 (fuel . unleaded)
 (steering . power-assisted)
 (seat-colour . black)
 (locking . manual)))
```

Хеш-таблицы также имеют ключи, и точно такие же аргументы применяются при использовании символов в хеш-таблицах, как и в ассоциативных списках. Хэш-значение, которое использует Guile, чтобы решить, где добавить запись в виде символа в хеш-таблицу, можно получить, вызвав процедуру `symbol-hash`

`symbol-hash` *symbol* [Scheme Procedure]
`scm_symbol_hash` (*symbol*) [C Function]
Return a hash value for *symbol*.

Смотри Раздел 6.6.22 [Hash Tables], страница 251, для получения информации о хеш-таблицах и почему лучше использовать хеш-таблицу, а не ассоциированный список.

6.6.6.3 Символы, обозначающие Переменные

Когда вычисляется неквотируемый символ в пограмме Scheme, он интерпретируется как ссылка на переменную, а результат вычисления — значение соответствующей переменной.

Например, когда выражение `(string-length "abcd")` считывается и вычисляется, последовательность символов `string-length` считывается как символ, имя которого "string-length". Этот символ связан с переменной, значением которой является процедура, которая выполняет вычисление длины стоки. Поэтому вычисление `string-length` приводит к выполнению этой процедуры.

Детали объединения символа без кавычек и переменной, ностися к другому разделу. Смотри Раздел 6.12 [Binding Constructs], страница 312, для ознакомления как создаются ассоциации между символами и переменными, и раздел Раздел 6.20 [Modules], страница 431, для ознакомления как эти ассоциации влияют на модульную систему Guile.

6.6.6.4 Операции связанные с символами

В любой Scheme, вы можете определить, является ли некий объект символом используя примитив `symbol?`:

`symbol?` *obj* [Scheme Procedure]
`scm_symbol_p` (*obj*) [C Function]
Return `#t` if *obj* is a symbol, otherwise return `#f`.

`int scm_is_symbol` (*SCM val*) [C Function]
Equivalent to `scm_is_true (scm_symbol_p (val))`.

Once you know that you have a symbol, you can obtain its name as a string by calling `symbol->string`. Note that Guile differs by default from R5RS on the details of `symbol->string` as regards case-sensitivity:

`symbol->string` *s* [Scheme Procedure]
`scm_symbol_to_string` (*s*) [C Function]
Return the name of symbol *s* as a string. By default, Guile reads symbols case-sensitively, so the string returned will have the same case variation as the sequence of characters that caused *s* to be created.

If Guile is set to read symbols case-insensitively (as specified by R5RS), and *s* comes into being as part of a literal expression (см. Раздел "Literal expressions" в *The Revised^5 Report on Scheme*) or by a call to the `read` or `string-ci->symbol` procedures, Guile converts any alphabetic characters in the symbol's name to lower case before creating the symbol object, so the string returned here will be in lower case.

If *s* was created by `string->symbol`, the case of characters in the string returned will be the same as that in the string that was passed to `string->symbol`, regardless of Guile's case-sensitivity setting at the time *s* was created.

It is an error to apply mutation procedures like `string-set!` to strings returned by this procedure.

Most symbols are created by writing them literally in code. However it is also possible to create symbols programmatically using the following procedures:

`symbol` *char...* [Scheme Procedure]
Return a newly allocated symbol made from the given character arguments.

```
(symbol #\x #\y #\z) ⇒ xyz
```

`list->symbol` *lst* [Scheme Procedure]
Return a newly allocated symbol made from a list of characters.

```
(list->symbol '(#\a #\b #\c)) ⇒ abc
```

`symbol-append` *arg ...* [Scheme Procedure]
Return a newly allocated symbol whose characters form the concatenation of the given symbols, *arg ...*.

```
(let ((h 'hello))
  (symbol-append h 'world))
⇒ helloworld
```

`string->symbol` *string* [Scheme Procedure]
`scm_string_to_symbol` (*string*) [C Function]
Return the symbol whose name is *string*. This procedure can create symbols with names containing special characters or letters in the non-standard case, but it is usually a bad idea to create such symbols because in some implementations of Scheme they cannot be read as themselves.

`string-ci->symbol` *str* [Scheme Procedure]
`scm_string_ci_to_symbol` (*str*) [C Function]
Return the symbol whose name is *str*. If Guile is currently reading symbols case-insensitively, *str* is converted to lowercase before the returned symbol is looked up or created.

The following examples illustrate Guile's detailed behaviour as regards the case-sensitivity of symbols:

```
(read-enable 'case-insensitive)   ; R5RS compliant behaviour

(symbol->string 'flying-fish)    ⇒ "flying-fish"
(symbol->string 'Martin)         ⇒ "martin"
(symbol->string
   (string->symbol "Malvina"))   ⇒ "Malvina"

(eq? 'mISSISSIppi 'mississippi)  ⇒ #t
(string->symbol "mISSISSIppi")   ⇒ mISSISSIppi
(eq? 'bitBlt (string->symbol "bitBlt")) ⇒ #f
(eq? 'LolliPop
  (string->symbol (symbol->string 'LolliPop))) ⇒ #t
(string=? "K. Harper, M.D."
  (symbol->string
    (string->symbol "K. Harper, M.D."))) ⇒ #t

(read-disable 'case-insensitive)   ; Guile default behaviour

(symbol->string 'flying-fish)    ⇒ "flying-fish"
(symbol->string 'Martin)         ⇒ "Martin"
(symbol->string
   (string->symbol "Malvina"))   ⇒ "Malvina"

(eq? 'mISSISSIppi 'mississippi)  ⇒ #f
(string->symbol "mISSISSIppi")   ⇒ mISSISSIppi
(eq? 'bitBlt (string->symbol "bitBlt")) ⇒ #t
(eq? 'LolliPop
  (string->symbol (symbol->string 'LolliPop))) ⇒ #t
(string=? "K. Harper, M.D."
  (symbol->string
    (string->symbol "K. Harper, M.D."))) ⇒ #t
```

From C, there are lower level functions that construct a Scheme symbol from a C string in the current locale encoding.

When you want to do more from C, you should convert between symbols and strings using `scm_symbol_to_string` and `scm_string_to_symbol` and work with the strings.

`SCM scm_from_latin1_symbol` (*const char *name*) [C Function]
`SCM scm_from_utf8_symbol` (*const char *name*) [C Function]
Construct and return a Scheme symbol whose name is specified by the null-terminated C string *name*. These are appropriate when the C string is hard-coded in the source code.

`SCM scm_from_locale_symbol` (*const char *name*) [C Function]
`SCM scm_from_locale_symboln` (*const char *name, size_t len*) [C Function]
Construct and return a Scheme symbol whose name is specified by *name*. For `scm_from_locale_symbol`, *name* must be null terminated; for `scm_from_locale_symboln` the length of *name* is specified explicitly by *len*.

Note that these functions should *not* be used when *name* is a C string constant, because there is no guarantee that the current locale will match that of the execution character set, used for string and character constants. Most modern C compilers use UTF-8 by default, so in such cases we recommend `scm_from_utf8_symbol`.

`SCM scm_take_locale_symbol` (*char *str*) [C Function]
`SCM scm_take_locale_symboln` (*char *str, size_t len*) [C Function]
Like `scm_from_locale_symbol` and `scm_from_locale_symboln`, respectively, but also frees *str* with `free` eventually. Thus, you can use this function when you would free *str* anyway immediately after creating the Scheme string. In certain cases, Guile can then use *str* directly as its internal representation.

The size of a symbol can also be obtained from C:

`size_t scm_c_symbol_length` (*SCM sym*) [C Function]
Return the number of characters in *sym*.

Наконец, некоторым приложениям, особенно тем, кторые генерируют новый код схемы Scheme динамически, необходимо создавать символы для использования в сгенерированном коде. Примитив `gensym` позволяет это сделать:

`gensym` [*prefix*] [Scheme Procedure]
`scm_gensym` (*prefix*) [C Function]
Create a new symbol with a name constructed from a prefix and a counter value. The string *prefix* can be specified as an optional argument. Default prefix is ‘ `g`’. The counter is increased by 1 at each call. There is no provision for resetting the counter.

Символы, генерируемые `gensym` скорее всего будут уникальными, поскольку их имена начинаются с пробелов, и неуникальными если программист найдет способ создать подобные символы. Уникальность может быть гарантирована, вместо этого, использванием неинтерминированных символов (см. Раздел 6.6.6.7 [Symbol Uninterned], страница 183), хоти они не могут быть корректно выведены и считаны обратно.

6.6.6.5 Функциональные слоты и Списки Свойств

В традиционных диалектах Lisp, символы часто понимаются как имеющие одновременно три вида значений:

- значение переменной*variable*, которое используется, когда символ отображается в контексте кода как ссылка на переменную.
- Значение функции *function*, которое используется, когда символ появляется в коде в имени функции (т.е как первый элемент в неквотируемом списке)
- Значение списка свойств*property list*, которое используется, когда символ указан в качестве первого аргумента для функции Lisp `put` или `get`.

Хотя Scheme (как одно из упрощений в отошении Lisp) убирает различия между пространствами имен переменных и функций, в настоящее время Guile сохраняет некоторые элементы традиционной структуры в случае если они окажутся полезными при реализации перевода для других языков, в частности Emacs Lisp.

В частности, символы Guile имеют два дополнительных слота: один для символа списка свойств и один для "значения функции". Для доступа к этим слотам предоставляются следующие процедуры.

`symbol-fref` *symbol* [Scheme Procedure]
`scm_symbol_fref` (*symbol*) [C Function]
Return the contents of *symbol*'s *function slot*.

`symbol-fset!` *symbol value* [Scheme Procedure]
`scm_symbol_fset_x` (*symbol, value*) [C Function]
Set the contents of *symbol*'s function slot to *value*.

`symbol-pref` *symbol* [Scheme Procedure]
`scm_symbol_pref` (*symbol*) [C Function]
Return the *property list* currently associated with *symbol*.

`symbol-pset!` *symbol value* [Scheme Procedure]
`scm_symbol_pset_x` (*symbol, value*) [C Function]
Set *symbol*'s property list to *value*.

`symbol-property` *sym prop* [Scheme Procedure]
From *sym*'s property list, return the value for property *prop*. The assumption is that *sym*'s property list is an association list whose keys are distinguished from each other using `equal?`; *prop* should be one of the keys in that list. If the property list has no entry for *prop*, `symbol-property` returns `#f`.

`set-symbol-property!` *sym prop val* [Scheme Procedure]
In *sym*'s property list, set the value for property *prop* to *val*, or add a new entry for *prop*, with value *val*, if none already exists. For the structure of the property list, see `symbol-property`.

`symbol-property-remove!` *sym prop* [Scheme Procedure]
From *sym*'s property list, remove the entry for property *prop*, if there is one. For the structure of the property list, see `symbol-property`.

Поддержка этих дополнительных слотов может быть удалена в будущих версиях, поэтому лучше избегать их использования. Для более современного и комплексного подхода к свойствам см Раздел 6.11.2 [Object Properties], страница 303.

6.6.6.6 Синтаксис Расширенного Чтения для Символов

Синтаксисом для чтения символов является последовательность букв, цифр и расширенного алфавита символов, начинающегося с символа с которого не может начинаться число. Кроме того, символы `+`, `-`, и `...` представляют собой особый случай, и считаются символами, хотя числа могут начинаться с `+`, `-` или `..`

Расширенные буквенные символы могут использваться в идентификаторах, как если бы они были буквами. Набор расширенных буквенных символов:

```
! $ % & * + - . / : < = > ? @ ^ _ ~
```

В дополнение к стандартному синтаксису чтения определенному выше(который взят из R5RS (см. Раздел “Formal syntax” в *The Revised^5 Report on Scheme*)), Guile предоставляет расширенный синтаксис чтения символов, который позволяет включать необычные символы, такие как символы пробела, новой строки и круглые скобки. Если (по какой-либо причине) вам нужно написать символ, содержащий буквы, не упомянутые выше, вы можете сделать это следующим образом.

- Начните символ с сочетания симолов `#{`,
- напишите буквы имени символа и
- завершите имя символа сочетанием символов `}#`.

Вот несколько примеров использования этой формы синтаксиса чтения. Первый символ должен использовать расширенный синтаксис, поскольку он содержит пробельный символ. Второй - потому что он содержит символ line break, и последний, потому что он похож на число.

```
#{foo bar}#

#{what
ever}#

#{4242}#
```

Although Guile provides this extended read syntax for symbols, widespread usage of it is discouraged because it is not portable and not very readable.

Alternatively, if you enable the `r7rs-symbols` read option (see см. Раздел 6.18.2 [Scheme Read], страница 407), you can write arbitrary symbols using the same notation used for strings, except delimited by vertical bars instead of double quotes.

```
|foo bar|
|\x3BB; is a greek lambda|
|\| is a vertical bar|
```

Note that there's also an `r7rs-symbols` print option (см. Раздел 6.18.3 [Scheme Write], страница 408). To enable the use of this notation, evaluate one or both of the following expressions:

```
(read-enable  'r7rs-symbols)
(print-enable 'r7rs-symbols)
```

6.6.6.7 Уникальные(uninterned) символы

Что делает символы полезными, так это то, что они автоматически сохраняются уникальными. нет двух символов, которые являются отдельными объектами и имеют одинаковое имя. Но конечно нет правила без исключения. В дополнении к обычным символам, которые обсуждались до сих пор, вы так же можете создавать специальные неограниченные символы, которые ведут себя несколько иначе.

Чтобы понять, чем эти символы отличаются и почему они полезны, мы посмотрим, как обычные символы сохраняют свою уникальность.

Всякий раз когда Guile необходимо найти символ с определенным именем, например во время четения `read` или при выполнении функции `string->symbol`, он сначала просматривает таблицу всех существующих символов, чтобы найти символ с указанным именем. Когда такой символ найден, Guile просто возвращает этот символ. Когда нет, создается новый символ с именем и вводиться в таблицу, чтобы его можно было найти позже.

Иногда вам может понадобиться создать символ, который был бы гарантированно 'новым', т.е ранее не существовал. Вы также можете както гарантировать, что никто другой когда-нибудь непреднамеренно не пересечется с вашим символом в будущем. Эти свойства символу часто необходимы при создании кода во время расширения макроса. При введении новых временных переменных, вы хотите гарантировать, что они не будут конфликтовать с переменными в коде других людей.

Самый простой способ организовать это - создать новый символ, но не вводить его в глобальную таблицу всех символов. Таким образом, никто никогда не получит доступ к вашему символу случайно. Символы которые не указаны в таблице, называются *uninterned*. Конечно символы находящиеся в таблице называются *интернированными(interned)*.

Вы можете создать новый uninterned символ функцией `make-symbol`. Вы можете проверить является ли символ интернированным или нет функцией `symbol-interned?`.

Uninterned символы нарушают правило, согласно которому имя символа однозначно идентифицирует символьный объект. Из-за этого они не могуть быть выведены и прочитаны обратно, как интернированные символы. В настоящее время Guile не поддерживает чтение uninterned символов. Обратите внимание, что функция `gensym` не возвращает uninterned символов по этой причине.

`make-symbol` *name* [Scheme Procedure]
`scm_make_symbol` (*name*) [C Function]

Return a new uninterned symbol with the name *name*. The returned symbol is guaranteed to be unique and future calls to `string->symbol` will not return it.

`symbol-interned?` *symbol* [Scheme Procedure]
`scm_symbol_interned_p` (*symbol*) [C Function]

Return `#t` if *symbol* is interned, otherwise return `#f`.

For example:

```
(define foo-1 (string->symbol "foo"))
(define foo-2 (string->symbol "foo"))
```

```
(define foo-3 (make-symbol "foo"))
(define foo-4 (make-symbol "foo"))

(eq? foo-1 foo-2)
⇒ #t
; Two interned symbols with the same name are the same object,

(eq? foo-1 foo-3)
⇒ #f
; but a call to make-symbol with the same name returns a
; distinct object.

(eq? foo-3 foo-4)
⇒ #f
; A call to make-symbol always returns a new object, even for
; the same name.

foo-3
⇒ #<uninterned-symbol foo 8085290>
; Uninterned symbols print differently from interned symbols,

(symbol? foo-3)
⇒ #t
; but they are still symbols,

(symbol-interned? foo-3)
⇒ #f
; just not interned.
```

6.6.7 Ключевые слова

Ключевые слова являются самовычислимым объектом с удобным синтаксисом чтения, что упращает их печать.

Поддержка ключевых слов Guile соответствует R5RS, и добавляет (с возможностью переключения) расширение синтаксиса чтения чтобы разрешить начинаться ключевым словам с `:` а так же с `#:`, или завершаться `:`.

6.6.7.1 Зачем использовать ключевые слова?

Ключевые слова полезны в контексте, где программе или процедуре необходимо принимать большое количество необязательных аргументов, не делая интерфес к процедуре неуправляемым.

Чтобы проилюстрировать это, рассмотрим гипотетическую процедуру создания окна `make-window`, которая создает новое окно на экране для рисования с использованием какого либо графического инструментария. Здесь очень много параметров, которые вызывающий может пожелать указать, но которые также могуть быть установлены поумолчанию, Например:

- color depth – Default: the color depth for the screen

- background color – Default: white
- width – Default: 600
- height – Default: 400

Если в процедуре `make-window` не использовались ключевые слова, вызывающий должен был бы передать значение для каждого возможного аргумента, соблюдая правильный порядок аргументов и используя специальные значения для указания на значение по умолчанию для этого аргумента.

```
(make-window 'default              ;; Color depth
             'default              ;; Background color
             800                   ;; Width
             100                   ;; Height
             ...)                  ;; More make-window arguments
```

С ключевыми словами, с другой стороны, дефолтные аргументы опускаются, а аргументы не по умолчанию четко помечены соответствующими ключевыми словами. В результате вызов становиться намного яснее:

```
(make-window #:width 800 #:height 100)
```

С другой стороны, для простой процедуры с несколькими аргументами использование ключевых слов было бы помехой, а не помощью. Например, примитивные процедуры, не следует улучшать вот так:

```
(cons #:car x #:cdr y)
```

Поэтому решение о том, использовать ли ключевые слова или нет, чисто прогматичное: используйте их, если они будут разъяснять вызов процедуры в точке вызова.

6.6.7.2 Кодирование с использованием ключевых слов

Если процедуре необходимо поддерживать ключевые слова, она должна принять последний аргумент, а затем использовать любые удобные средства для извлечения ключевых слов и их соответствующих аргументов из содержимого этого последнего аргумента.

The following example illustrates the principle: the code for `make-window` uses a helper procedure called `get-keyword-value` to extract individual keyword arguments from the rest argument.

```
(define (get-keyword-value args keyword default)
  (let ((kv (memq keyword args)))
    (if (and kv (>= (length kv) 2))
        (cadr kv)
        default)))

(define (make-window . args)
  (let ((depth  (get-keyword-value args #:depth  screen-depth))
        (bg     (get-keyword-value args #:bg     "white"))
        (width  (get-keyword-value args #:width  800))
        (height (get-keyword-value args #:height 100))
        ...)
    ...))
```

Но вам не нужно создавать `get-keyword-value`. Модуль `(ice-9 optargs)` поставляет набор мощных макросов, которые можно использовать для реализации процедур поддержки работы с ключевыми словами, таких как:

```
(use-modules (ice-9 optargs))

(define (make-window . args)
  (let-keywords args #f ((depth  screen-depth)
                         (bg     "white")
                         (width  800)
                         (height 100))
    ...))
```

Или, еще более экономно, вот так:

```
(use-modules (ice-9 optargs))

(define* (make-window #:key (depth  screen-depth)
                            (bg     "white")
                            (width  800)
                            (height 100))
  ...)
```

Для получения дополнительной информации о `let-keywords`, `define*` и других возможностях поставляемых модулем `(ice-9 optargs)`, смотри Раздел 6.9.4 [Optional Arguments], страница 267.

Чтобы обрабатывать аргументы ключевых слов из процедур реализованных на Си, используйте `scm_c_bind_keyword_arguments` (см. Раздел 6.6.7.4 [Keyword Procedures], страница 187).

6.6.7.3 Синтаксис чтения Ключевых Слов

Guile, по умолчанию, распознает только синтаксис ключевых слов, совместимый с R5RS. Токен формы `#:NAME`, где `NAME` имеет тот же синтаксис, что и символы Scheme (см. Раздел 6.6.6.6 [Symbol Read Syntax], страница 182), является внешним представлением ключевого слова с именем `NAME`. Кроме того, объекты ключевые слова печатются с использованием этого синтаксиса, поэтому значения, содержащие объекты ключых слов, могут быть прочитаными Guile. При использовании в выражении ключевые слова являются самоквотирующимися выражениями.

Если для параметра чтения ключевых слов `keywords` устновлено значение префикс `'prefix`, Guile также распрознает альтернативный синтаксис чтения `:NAME`. В противном случае маркеры формы `:NAME` считаются символами, как это требует R5RS.

Если для параметра чтения ключевого слова `keywords` установлено значение `'postfix`, Guile распознает синтаксис чтения SRFI-88 `NAME:` (см. Раздел 7.5.41 [SRFI-88], страница 675). В противном случае токены этой формы читаются как символы.

Чтобы включить или отключить альтернативный синтаксис чтения ключевых слов, отличный от R5RS, используйте процедуру `read-set!` описанную Раздел 6.18.2 [Scheme Read], страница 407. Обратите внимание синтаксисы `prefix` и `postfix` являются взаимоисключающими.

```
(read-set! keywords 'prefix)

#:type
⇒
#:type

:type
⇒
#:type

(read-set! keywords 'postfix)

type:
⇒
#:type

:type
⇒
:type

(read-set! keywords #f)

#:type
⇒
#:type

:type
⊣
ERROR: In expression :type:
ERROR: Unbound variable: :type
ABORT: (unbound-variable)
```

6.6.7.4 Процедуры работы с Ключевыми Словами

`keyword?` *obj* [Scheme Procedure]
`scm_keyword_p` (*obj*) [C Function]
Return `#t` if the argument *obj* is a keyword, else `#f`.

`keyword->symbol` *keyword* [Scheme Procedure]
`scm_keyword_to_symbol` (*keyword*) [C Function]
Return the symbol with the same name as *keyword*.

`symbol->keyword` *symbol* [Scheme Procedure]
`scm_symbol_to_keyword` (*symbol*) [C Function]
Return the keyword with the same name as *symbol*.

`int scm_is_keyword` (*SCM obj*) [C Function]
Equivalent to `scm_is_true (scm_keyword_p (obj))`.

`SCM scm_from_locale_keyword` (*const char *name*) [C Function]
`SCM scm_from_locale_keywordn` (*const char *name, size_t len*) [C Function]
Equivalent to `scm_symbol_to_keyword (scm_from_locale_symbol (name))` and `scm_symbol_to_keyword (scm_from_locale_symboln (name, len))`, respectively.

Note that these functions should *not* be used when *name* is a C string constant, because there is no guarantee that the current locale will match that of the execution character set, used for string and character constants. Most modern C compilers use UTF-8 by default, so in such cases we recommend `scm_from_utf8_keyword`.

`SCM scm_from_latin1_keyword` (*const char *name*) [C Function]
`SCM scm_from_utf8_keyword` (*const char *name*) [C Function]
Equivalent to `scm_symbol_to_keyword (scm_from_latin1_symbol (name))` and `scm_symbol_to_keyword (scm_from_utf8_symbol (name))`, respectively.

`void scm_c_bind_keyword_arguments` (*const char *subr, SCM rest,* [C Function]
*scm_t_keyword_arguments_flags flags, SCM keyword1, SCM *argp1, ..., SCM keywordN, SCM *argpN,* `SCM_UNDEFINED`)
Extract the specified keyword arguments from *rest*, which is not modified. If the keyword argument *keyword1* is present in *rest* with an associated value, that value is stored in the variable pointed to by *argp1*, otherwise the variable is left unchanged. Similarly for the other keywords and argument pointers up to *keywordN* and *argpN*. The argument list to `scm_c_bind_keyword_arguments` must be terminated by `SCM_UNDEFINED`.

Note that since the variables pointed to by *argp1* through *argpN* are left unchanged if the associated keyword argument is not present, they should be initialized to their default values before calling `scm_c_bind_keyword_arguments`. Alternatively, you can initialize them to `SCM_UNDEFINED` before the call, and then use `SCM_UNBNDP` after the call to see which ones were provided.

If an unrecognized keyword argument is present in *rest* and *flags* does not contain `SCM_ALLOW_OTHER_KEYS`, or if non-keyword arguments are present and *flags* does not contain `SCM_ALLOW_NON_KEYWORD_ARGUMENTS`, an exception is raised. *subr* should be the name of the procedure receiving the keyword arguments, for purposes of error reporting.

For example:

```
SCM k_delimiter;
SCM k_grammar;
SCM sym_infix;

SCM my_string_join (SCM strings, SCM rest)
{
  SCM delimiter = SCM_UNDEFINED;
  SCM grammar   = sym_infix;

  scm_c_bind_keyword_arguments ("my-string-join", rest, 0,
                                k_delimiter, &delimiter,
                                k_grammar, &grammar,
```

```
                                    SCM_UNDEFINED);

  if (SCM_UNBNDP (delimiter))
    delimiter = scm_from_utf8_string (" ");

  return scm_string_join (strings, delimiter, grammar);
}

void my_init ()
{
  k_delimiter = scm_from_utf8_keyword ("delimiter");
  k_grammar   = scm_from_utf8_keyword ("grammar");
  sym_infix   = scm_from_utf8_symbol  ("infix");
  scm_c_define_gsubr ("my-string-join", 1, 0, 1, my_string_join);
}
```

6.6.8 Пары

Пары используются для объединения двух объектов Scheme в один составной объект. Отсюда и название: Пара хранит пару объектов.

Тип данных Пара *pair* чрезвычайно важен в Scheme, как и в любом другом диалекте Lisp. Причина в том, что пары используются не только для того чтобы сделать два значения доступными как один объект, но эти пары используются для построения списков значений. Поскольку списки так важны в Scheme, они описаны в отдельном разделе (см. Раздел 6.6.9 [Lists], страница 192).

Пары могут буквально вводиться в исходный код или в REPL, в так называемом точечном синтаксисе списка *dotted list*. Этот синтаксис состоит из открывающей круглой скобки, первого элемента пары, точки, второго элемента, закрывающей круглой скобки. В следующем примере показано, формирование пары из двух чисел 1 и 2, и пары содержащей символы `foo` и `bar`. Очень важно писать пробелы до и после точки, потому что в противном случае анализатор Scheme не сможет определить, где разделять токены.

```
(1 . 2)
(foo . bar)
```

Но будьте осторожны, если вы хотите попробовать эти примеры, вам нужно квотировать*quote* эти выражения. Более подробную информацию можно найти в разделе Раздел 6.18.1.1 [Expression Syntax], страница 404. Правильный способ использовать эти примеры состоит в следующем.

```
'(1 . 2)
⇒
(1 . 2)
'(foo . bar)
⇒
(foo . bar)
```

Новая пара создается путем вызова процедуры `cons` с двумя аргументами. Затем значения аргументов сохраняются во вновь выделенную пару, и пара возвращается.

Имя `cons` означает "Конструировать(construct)". Используйте процедуру `pair?` что бы проверить является ли данный объект Scheme парой или нет.

`cons` *x y* [Scheme Procedure]
`scm_cons` (*x*, *y*) [C Function]
Return a newly allocated pair whose car is *x* and whose cdr is *y*. The pair is guaranteed to be different (in the sense of `eq?`) from every previously existing object.

`pair?` *x* [Scheme Procedure]
`scm_pair_p` (*x*) [C Function]
Return `#t` if *x* is a pair; otherwise return `#f`.

`int scm_is_pair` (*SCM x*) [C Function]
Return 1 when *x* is a pair; otherwise return 0.

The two parts of a pair are traditionally called *car* and *cdr*. They can be retrieved with procedures of the same name (`car` and `cdr`), and can be modified with the procedures `set-car!` and `set-cdr!`.

Since a very common operation in Scheme programs is to access the car of a car of a pair, or the car of the cdr of a pair, etc., the procedures called `caar`, `cadr` and so on are also predefined. However, using these procedures is often detrimental to readability, and error-prone. Thus, accessing the contents of a list is usually better achieved using pattern matching techniques (см. Раздел 7.7 [Pattern Matching], страница 720).

`car` *pair* [Scheme Procedure]
`cdr` *pair* [Scheme Procedure]
`scm_car` (*pair*) [C Function]
`scm_cdr` (*pair*) [C Function]
Return the car or the cdr of *pair*, respectively.

`SCM SCM_CAR` (*SCM pair*) [C Macro]
`SCM SCM_CDR` (*SCM pair*) [C Macro]
These two macros are the fastest way to access the car or cdr of a pair; they can be thought of as compiling into a single memory reference.

These macros do no checking at all. The argument *pair* must be a valid pair.

`cddr` *pair* [Scheme Procedure]
`cdar` *pair* [Scheme Procedure]
`cadr` *pair* [Scheme Procedure]
`caar` *pair* [Scheme Procedure]
`cdddr` *pair* [Scheme Procedure]
`cddar` *pair* [Scheme Procedure]
`cdadr` *pair* [Scheme Procedure]
`cdaar` *pair* [Scheme Procedure]
`caddr` *pair* [Scheme Procedure]
`cadar` *pair* [Scheme Procedure]
`caadr` *pair* [Scheme Procedure]
`caaar` *pair* [Scheme Procedure]
`cddddr` *pair* [Scheme Procedure]

`cdddar` *pair* [Scheme Procedure]
`cddadr` *pair* [Scheme Procedure]
`cddaar` *pair* [Scheme Procedure]
`cdaddr` *pair* [Scheme Procedure]
`cdadar` *pair* [Scheme Procedure]
`cdaadr` *pair* [Scheme Procedure]
`cdaaar` *pair* [Scheme Procedure]
`cadddr` *pair* [Scheme Procedure]
`caddar` *pair* [Scheme Procedure]
`cadadr` *pair* [Scheme Procedure]
`cadaar` *pair* [Scheme Procedure]
`caaddr` *pair* [Scheme Procedure]
`caadar` *pair* [Scheme Procedure]
`caaadr` *pair* [Scheme Procedure]
`caaaar` *pair* [Scheme Procedure]
`scm_cddr` (*pair*) [C Function]
`scm_cdar` (*pair*) [C Function]
`scm_cadr` (*pair*) [C Function]
`scm_caar` (*pair*) [C Function]
`scm_cdddr` (*pair*) [C Function]
`scm_cddar` (*pair*) [C Function]
`scm_cdadr` (*pair*) [C Function]
`scm_cdaar` (*pair*) [C Function]
`scm_caddr` (*pair*) [C Function]
`scm_cadar` (*pair*) [C Function]
`scm_caadr` (*pair*) [C Function]
`scm_caaar` (*pair*) [C Function]
`scm_cddddr` (*pair*) [C Function]
`scm_cdddar` (*pair*) [C Function]
`scm_cddadr` (*pair*) [C Function]
`scm_cddaar` (*pair*) [C Function]
`scm_cdaddr` (*pair*) [C Function]
`scm_cdadar` (*pair*) [C Function]
`scm_cdaadr` (*pair*) [C Function]
`scm_cdaaar` (*pair*) [C Function]
`scm_cadddr` (*pair*) [C Function]
`scm_caddar` (*pair*) [C Function]
`scm_cadadr` (*pair*) [C Function]
`scm_cadaar` (*pair*) [C Function]
`scm_caaddr` (*pair*) [C Function]
`scm_caadar` (*pair*) [C Function]
`scm_caaadr` (*pair*) [C Function]
`scm_caaaar` (*pair*) [C Function]

These procedures are compositions of `car` and `cdr`, where for example `caddr` could be defined by

```
(define caddr (lambda (x) (car (cdr (cdr x)))))
```

`cadr`, `caddr` and `cadddr` pick out the second, third or fourth elements of a list, respectively. SRFI-1 provides the same under the names `second`, `third` and `fourth` (см. Раздел 7.5.3.3 [SRFI-1 Selectors], страница 609).

`set-car!` *pair value* [Scheme Procedure]
`scm_set_car_x` (*pair, value*) [C Function]
Stores *value* in the car field of *pair*. The value returned by `set-car!` is unspecified.

`set-cdr!` *pair value* [Scheme Procedure]
`scm_set_cdr_x` (*pair, value*) [C Function]
Stores *value* in the cdr field of *pair*. The value returned by `set-cdr!` is unspecified.

6.6.9 Списки

Очень важным типом данных в Scheme, а также во всех других диалектах Lisp, является тип данных типа Список*list*.[1]

Это краткое определение того, что такое список:

- Либо пустой список `()`,
- либо пара, которая имеет списко в своем cdr.

6.6.9.1 Синтаксис чтения Списка

Синтаксис списка - это открывающая круглая скобка, затем все элементы списка (разделяемые пробелами) и наконец, закрывающая круглая скобка.[2].

```
(1 2 3)             ; a list of the numbers 1, 2 and 3
("foo" bar 3.1415) ; a string, a symbol and a real number
()                  ; the empty list
```

В последнем примере требуется немного объяснений. Список без элементов, называемый Пустой список *empty list*, используется особенным образом. Он используется для завершения списков, сохраняя его в cdr последнй пары, составляющих список. Приведем пример:

```
(car '(1))
⇒
1
(cdr '(1))
⇒
()
```

Этот пример также показывает. что списки должны квотироваться при записи(см. Раздел 6.18.1.1 [Expression Syntax], страница 404), потому что иначе они были бы ошибочно приняты как вызов процедуры (см. Раздел 3.2.2 [Simple Invocation], страница 18).

---

[1] Строго говоря, Scheme не имеет реально сущетствующего типа данных *list*. Список представляет собой *цепочки пар(chained pairs)*, и используется только как определение — список это цепочка пар выглядящая как список.

[2] Обратите внимание, что между элементами списка нет разделительного символа, например запятой или точки с запятой.

6.6.9.2 Списковые предикаты

Часто полезно проверить, является ли данный объект Scheme списком или нет. Процедуры Обработки Списков могут использовать эту информацию, чтобы проверить, действительно ли им на вход подали список, или они могут выполнять различные действия в зависимости от типа данных аргументов.

`list?` *x* [Scheme Procedure]
`scm_list_p` (*x*) [C Function]
Return `#t` if *x* is a proper list, else `#f`.

The predicate `null?` is often used in list-processing code to tell whether a given list has run out of elements. That is, a loop somehow deals with the elements of a list until the list satisfies `null?`. Then, the algorithm terminates.

`null?` *x* [Scheme Procedure]
`scm_null_p` (*x*) [C Function]
Return `#t` if *x* is the empty list, else `#f`.

`int scm_is_null` (*SCM x*) [C Function]
Return 1 when *x* is the empty list; otherwise return 0.

6.6.9.3 Конструкторы Списков

В этом разделе описываются процедуры построения новых списков. Функция `list` просто возвращает список, переданных ей аргументов, аналогично работает `cons*`, но последний аргумент сохраняется в cdr последней пары списка.

`list` *elem ...* [Scheme Procedure]
`scm_list_1` (*elem1*) [C Function]
`scm_list_2` (*elem1, elem2*) [C Function]
`scm_list_3` (*elem1, elem2, elem3*) [C Function]
`scm_list_4` (*elem1, elem2, elem3, elem4*) [C Function]
`scm_list_5` (*elem1, elem2, elem3, elem4, elem5*) [C Function]
`scm_list_n` (*elem1, ..., elemN*, `SCM_UNDEFINED`) [C Function]
Return a new list containing elements *elem ...*.

`scm_list_n` takes a variable number of arguments, terminated by the special `SCM_UNDEFINED`. That final `SCM_UNDEFINED` is not included in the list. None of *elem ...* can themselves be `SCM_UNDEFINED`, or `scm_list_n` will terminate at that point.

`cons*` *arg1 arg2 ...* [Scheme Procedure]
Like `list`, but the last arg provides the tail of the constructed list, returning `(cons` *arg1* `(cons` *arg2* `(cons ...` *argn*`)))`. Requires at least one argument. If given one argument, that argument is returned as result. This function is called `list*` in some other Schemes and in Common LISP.

`list-copy` *lst* [Scheme Procedure]
`scm_list_copy` (*lst*) [C Function]
Return a (newly-created) copy of *lst*.

`make-list` *n* [*init*] [Scheme Procedure]
: Create a list containing of *n* elements, where each element is initialized to *init*. *init* defaults to the empty list `()` if not given.

Обратите внимание, что в процедуре `list-copy` делается только копирование пар, составляющих основу списка. Элементы списка не копируются, что означает, что изменение элементов нового списка, также изменит и элементы старого списка. С другой стороны, применяя такие процедуры как `set-cdr!` или `delv!` для нового списка, старый список не будет изменен. Если вам также необходимо скопировать элементы списка(сделать глубокую копию), используйте процедуру `copy-tree` (см. Раздел 6.11.4 [Copying], страница 306).

6.6.9.4 Операции Выбора в Списке

Эти процедуры используются для получения некоторой информации о списке или для извлечения одного или нескольких элементов списка.

`length` *lst* [Scheme Procedure]
`scm_length` (*lst*) [C Function]
: Return the number of elements in list *lst*.

`last-pair` *lst* [Scheme Procedure]
`scm_last_pair` (*lst*) [C Function]
: Return the last pair in *lst*, signalling an error if *lst* is circular.

`list-ref` *list k* [Scheme Procedure]
`scm_list_ref` (*list, k*) [C Function]
: Return the *k*th element from *list*.

`list-tail` *lst k* [Scheme Procedure]
`list-cdr-ref` *lst k* [Scheme Procedure]
`scm_list_tail` (*lst, k*) [C Function]
: Return the "tail" of *lst* beginning with its *k*th element. The first element of the list is considered to be element 0.

  `list-tail` and `list-cdr-ref` are identical. It may help to think of `list-cdr-ref` as accessing the *k*th cdr of the list, or returning the results of cdring *k* times down *lst*.

`list-head` *lst k* [Scheme Procedure]
`scm_list_head` (*lst, k*) [C Function]
: Copy the first *k* elements from *lst* into a new list, and return it.

6.6.9.5 Добавление и Переворачивание

`append` и `append!` используются для объединения двух и более списков, что бы сформировать новый список. `reverse` и `reverse!` возвращаеют списки с теми же элементами, но идущими в обратном порядке. Варианты процедур с `!` непосредственно изменяют пары, которые образуют список, а другие процедуры создают новые пары. Вот почему вы должны быть осторожны, когда используете побочные эффекты процедур.

`append` *lst ... obj* [Scheme Procedure]
`append` [Scheme Procedure]
`append!` *lst ... obj* [Scheme Procedure]
`append!` [Scheme Procedure]
`scm_append` (*lstlst*) [C Function]
`scm_append_x` (*lstlst*) [C Function]

Return a list comprising all the elements of lists *lst ... obj*. If called with no arguments, return the empty list.

```
(append '(x) '(y))          ⇒  (x y)
(append '(a) '(b c d))      ⇒  (a b c d)
(append '(a (b)) '((c)))    ⇒  (a (b) (c))
```

The last argument *obj* may actually be any object; an improper list results if the last argument is not a proper list.

```
(append '(a b) '(c . d))    ⇒  (a b c . d)
(append '() 'a)             ⇒  a
```

`append` doesn't modify the given lists, but the return may share structure with the final *obj*. `append!` is permitted, but not required, to modify the given lists to form its return.

For `scm_append` and `scm_append_x`, *lstlst* is a list of the list operands *lst ... obj*. That *lstlst* itself is not modified or used in the return.

`reverse` *lst* [Scheme Procedure]
`reverse!` *lst* [*newtail*] [Scheme Procedure]
`scm_reverse` (*lst*) [C Function]
`scm_reverse_x` (*lst, newtail*) [C Function]

Return a list comprising the elements of *lst*, in reverse order.

`reverse` constructs a new list. `reverse!` is permitted, but not required, to modify *lst* in constructing its return.

For `reverse!`, the optional *newtail* is appended to the result. *newtail* isn't reversed, it simply becomes the list tail. For `scm_reverse_x`, the *newtail* parameter is mandatory, but can be `SCM_EOL` if no further tail is required.

6.6.9.6 Изменение Списка

Следующие процедуры изменяют существующий список либо путем изменения элементов списка, либо путем изменения самой структуры списка.

`list-set!` *list k val* [Scheme Procedure]
`scm_list_set_x` (*list, k, val*) [C Function]

Устанавливает *k*-тый элемент списка *list* значением *val*.

```
;;ВНИМАНИЕ!!! меня ждал тут сюрприз!!!
(define t1 '(1 2 3 4 5))
(list-set! t1 0 100)  ⇒
⇒ ERROR: In procedure list-set!:
 set-car!: Wrong type argument in position 1 (expecting mutable pair)
;; А так РАБОТАЕТ!!! значит важно различить mutable и imutable list
```

```
(define t1 (list 1 2 3 4 5))
(list-set! t1 0 100) (display t1)  ⇒  (100 2 3 4 5)
```

`list-cdr-set!` *list k val* [Scheme Procedure]
`scm_list_cdr_set_x` (*list, k, val*) [C Function]
Set the *k*th cdr of *list* to *val*.

`delq` *item lst* [Scheme Procedure]
`scm_delq` (*item, lst*) [C Function]
Return a newly-created copy of *lst* with elements `eq?` to *item* removed. This procedure mirrors `memq`: `delq` compares elements of *lst* against *item* with `eq?`.

`delv` *item lst* [Scheme Procedure]
`scm_delv` (*item, lst*) [C Function]
Return a newly-created copy of *lst* with elements `eqv?` to *item* removed. This procedure mirrors `memv`: `delv` compares elements of *lst* against *item* with `eqv?`.

`delete` *item lst* [Scheme Procedure]
`scm_delete` (*item, lst*) [C Function]
Return a newly-created copy of *lst* with elements `equal?` to *item* removed. This procedure mirrors `member`: `delete` compares elements of *lst* against *item* with `equal?`.

See also SRFI-1 which has an extended `delete` (Раздел 7.5.3.8 [SRFI-1 Deleting], страница 616), and also an `lset-difference` which can delete multiple *item*s in one call (Раздел 7.5.3.10 [SRFI-1 Set Operations], страница 617).

`delq!` *item lst* [Scheme Procedure]
`delv!` *item lst* [Scheme Procedure]
`delete!` *item lst* [Scheme Procedure]
`scm_delq_x` (*item, lst*) [C Function]
`scm_delv_x` (*item, lst*) [C Function]
`scm_delete_x` (*item, lst*) [C Function]
These procedures are destructive versions of `delq`, `delv` and `delete`: they modify the pointers in the existing *lst* rather than creating a new list. Caveat evaluator: Like other destructive list functions, these functions cannot modify the binding of *lst*, and so cannot be used to delete the first element of *lst* destructively.

`delq1!` *item lst* [Scheme Procedure]
`scm_delq1_x` (*item, lst*) [C Function]
Like `delq!`, but only deletes the first occurrence of *item* from *lst*. Tests for equality using `eq?`. See also `delv1!` and `delete1!`.

`delv1!` *item lst* [Scheme Procedure]
`scm_delv1_x` (*item, lst*) [C Function]
Like `delv!`, but only deletes the first occurrence of *item* from *lst*. Tests for equality using `eqv?`. See also `delq1!` and `delete1!`.

`delete1!` *item lst* [Scheme Procedure]
`scm_delete1_x` (*item, lst*) [C Function]
Like `delete!`, but only deletes the first occurrence of *item* from *lst*. Tests for equality using `equal?`. See also `delq1!` and `delv1!`.

`filter` *pred lst* [Scheme Procedure]
`filter!` *pred lst* [Scheme Procedure]

Return a list containing all elements from *lst* which satisfy the predicate *pred*. The elements in the result list have the same order as in *lst*. The order in which *pred* is applied to the list elements is not specified.

`filter` does not change *lst*, but the result may share a tail with it. `filter!` may modify *lst* to construct its return.

6.6.9.7 Поиск по Списку

Следующие процедуры выполняют поиск по конкретным элементам. Они используют различные типы предикатов сравнения элементов списка с объектом, подлежащем поиску. Когда поиск неудачен, функции возвращают `#f`, иначе они возвращают подсписок, car для которого равен искомому объекту, результат сравнения(и поиска) зависит от используемого предиката равенства.

`memq` *x lst* [Scheme Procedure]
`scm_memq` (*x, lst*) [C Function]

Return the first sublist of *lst* whose car is `eq?` to *x* where the sublists of *lst* are the non-empty lists returned by `(list-tail lst k)` for *k* less than the length of *lst*. If *x* does not occur in *lst*, then `#f` (not the empty list) is returned.

`memv` *x lst* [Scheme Procedure]
`scm_memv` (*x, lst*) [C Function]

Return the first sublist of *lst* whose car is `eqv?` to *x* where the sublists of *lst* are the non-empty lists returned by `(list-tail lst k)` for *k* less than the length of *lst*. If *x* does not occur in *lst*, then `#f` (not the empty list) is returned.

`member` *x lst* [Scheme Procedure]
`scm_member` (*x, lst*) [C Function]

Return the first sublist of *lst* whose car is `equal?` to *x* where the sublists of *lst* are the non-empty lists returned by `(list-tail lst k)` for *k* less than the length of *lst*. If *x* does not occur in *lst*, then `#f` (not the empty list) is returned.

See also SRFI-1 which has an extended `member` function (Раздел 7.5.3.7 [SRFI-1 Searching], страница 614).

6.6.9.8 Поэлементная обработка Списка

Обработка списков очень удобна в Scheme, потому что, процесс итерации по элементам списка может быть сильно абстрагирован. Процедуры описанные в этом разделе являются базовыми процедурами итерации для списков. Они принимают процедуру и один или несколько списков в качесве аргументов, и применяют процедуру к каждому элементу списка. Процедуры отличаются значением которое они возвращают.

`map` *proc arg1 arg2 ...* [Scheme Procedure]
`map-in-order` *proc arg1 arg2 ...* [Scheme Procedure]
`scm_map` (*proc, arg1, args*) [C Function]

Apply *proc* to each element of the list *arg1* (if only two arguments are given), or to the corresponding elements of the argument lists (if more than two arguments are

given). The result(s) of the procedure applications are saved and returned in a list. For `map`, the order of procedure applications is not specified, `map-in-order` applies the procedure from left to right to the list elements.

`for-each` *proc arg1 arg2 ...* [Scheme Procedure]
Like `map`, but the procedure is always applied from left to right, and the result(s) of the procedure applications are thrown away. The return value is not specified.

See also SRFI-1 which extends these functions to take lists of unequal lengths (Раздел 7.5.3.5 [SRFI-1 Fold and Map], страница 611).

6.6.10 Векторы

Векторы - это последовательности объектов Scheme. В отличие от списков длина вектора, после создания вектора, не может быть изменена. Преимущество векторов над списками заключается в том, что требуемое время для доступа к одному элементу вектора, с учетом его положения(синоним индекса) относительно нулевой позиции, является константой, тогда как списки имеют время доступа линейное по отношению к позиции доступа элемета в списке.

Векторы могут содержать любой объект Scheme; возможно даже иметь разные типы объектов в одном и том же векторе. Для векторов, содержащих векторы, вместо них вы можете использовать Массивы. Заметим также, что векторы являются частным случаем одномерных неоднородных массивов и большинство процедур для Массивов успешно работают и на Векорах. (см. Раздел 6.6.13 [Arrays], страница 212)

Also see Раздел 7.5.30 [SRFI-43], страница 663, for a comprehensive vector library.

6.6.10.1 Синтаксис чтения для Векторов

Векторы можно определить в исходном коде, подобно строкам, символам или некоторым другим типам данных. Синтаксис чтения для векторов слеюдующий: символ решетки (`#`), за которым следует открывающая круглая скобка, далее все элементы вектора в соотвествующем им синтаксисе чтения и, наконец, закрывающая круглая скобка. Подобно строкам векторам не нужно указывать знак квотирования.

Ниже приведены примеры синтаксиса чтения для векторов; где только первый вектор содержит числа и три разных типа объекта: строку, символ и число в шестнадцатеричной форме.

```
#(1 2 3)
#("Hello" foo #xdeadbeef)
```

6.6.10.2 Динамическое создание и проверка Вектора

Вместо того, чтобы создавать векторы неявно, используя только что описанный синтаксис чтения, вы можете создать вектор динамически, вызывая одну из процедур `vector` и `list->vector` со списком значений Scheme, которые вы хотите поместить в вектор. Размер вектора созданный таким образом, определяется неявно количеством доступных аргументов.

`vector` *arg ...* [Scheme Procedure]
`list->vector` *l* [Scheme Procedure]

`scm_vector` (*l*) [C Function]
Return a newly allocated vector composed of the given arguments. Analogous to `list`.

```
(vector 'a 'b 'c) ⇒ #(a b c)
```

The inverse operation is `vector->list`:

`vector->list` *v* [Scheme Procedure]
`scm_vector_to_list` (*v*) [C Function]
Return a newly allocated list composed of the elements of *v*.

```
(vector->list #(dah dah didah)) ⇒  (dah dah didah)
(list->vector '(dididit dah)) ⇒  #(dididit dah)
```

To allocate a vector with an explicitly specified size, use `make-vector`. With this primitive you can also specify an initial value for the vector elements (the same value for all elements, that is):

`make-vector` *len* [*fill*] [Scheme Procedure]
`scm_make_vector` (*len, fill*) [C Function]
Return a newly allocated vector of *len* elements. If a second argument is given, then each position is initialized to *fill*. Otherwise the initial contents of each position is unspecified.

`SCM scm_c_make_vector` (*size_t k, SCM fill*) [C Function]
Like `scm_make_vector`, but the length is given as a `size_t`.

To check whether an arbitrary Scheme value *is* a vector, use the `vector?` primitive:

`vector?` *obj* [Scheme Procedure]
`scm_vector_p` (*obj*) [C Function]
Return `#t` if *obj* is a vector, otherwise return `#f`.

`int scm_is_vector` (*SCM obj*) [C Function]
Return non-zero when *obj* is a vector, otherwise return `zero`.

6.6.10.3 Доступ и Модификация содержимого Вектора

процедуры `vector-length` и `vector-ref` возвращают информацию об определенном векторе, соответственно длину вектора и элементы, которые содержит вектор.

`vector-length` *vector* [Scheme Procedure]
`scm_vector_length` (*vector*) [C Function]
Return the number of elements in *vector* as an exact integer.

`size_t scm_c_vector_length` (*SCM vec*) [C Function]
Return the number of elements in *vec* as a `size_t`.

`vector-ref` *vec k* [Scheme Procedure]
`scm_vector_ref` (*vec, k*) [C Function]
Return the contents of position *k* of *vec*. *k* must be a valid index of *vec*.

```
(vector-ref #(1 1 2 3 5 8 13 21) 5) ⇒ 8
```

```
(vector-ref #(1 1 2 3 5 8 13 21)
    (let ((i (round (* 2 (acos -1)))))
      (if (inexact? i)
        (inexact->exact i)
           i))) ⇒ 13
```

`SCM scm_c_vector_ref` (*SCM vec, size_t k*) [C Function]
Return the contents of position *k* (a `size_t`) of *vec*.

A vector created by one of the dynamic vector constructor procedures (см. Раздел 6.6.10.2 [Vector Creation], страница 198) can be modified using the following procedures.

NOTE: According to R5RS, it is an error to use any of these procedures on a literally read vector, because such vectors should be considered as constants. Currently, however, Guile does not detect this error.

`vector-set!` *vec k obj* [Scheme Procedure]
`scm_vector_set_x` (*vec, k, obj*) [C Function]
Store *obj* in position *k* of *vec*. *k* must be a valid index of *vec*. The value returned by '`vector-set!`' is unspecified.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec) ⇒  #(0 ("Sue" "Sue") "Anna")
```

`void scm_c_vector_set_x` (*SCM vec, size_t k, SCM obj*) [C Function]
Store *obj* in position *k* (a `size_t`) of *vec*.

`vector-fill!` *vec fill* [Scheme Procedure]
`scm_vector_fill_x` (*vec, fill*) [C Function]
Store *fill* in every position of *vec*. The value returned by `vector-fill!` is unspecified.

`vector-copy` *vec* [Scheme Procedure]
`scm_vector_copy` (*vec*) [C Function]
Return a copy of *vec*.

`vector-move-left!` *vec1 start1 end1 vec2 start2* [Scheme Procedure]
`scm_vector_move_left_x` (*vec1, start1, end1, vec2, start2*) [C Function]
Copy elements from *vec1*, positions *start1* to *end1*, to *vec2* starting at position *start2*. *start1* and *start2* are inclusive indices; *end1* is exclusive.

`vector-move-left!` copies elements in leftmost order. Therefore, in the case where *vec1* and *vec2* refer to the same vector, `vector-move-left!` is usually appropriate when *start1* is greater than *start2*.

`vector-move-right!` *vec1 start1 end1 vec2 start2* [Scheme Procedure]
`scm_vector_move_right_x` (*vec1, start1, end1, vec2, start2*) [C Function]
Copy elements from *vec1*, positions *start1* to *end1*, to *vec2* starting at position *start2*. *start1* and *start2* are inclusive indices; *end1* is exclusive.

`vector-move-right!` copies elements in rightmost order. Therefore, in the case where *vec1* and *vec2* refer to the same vector, `vector-move-right!` is usually appropriate when *start1* is less than *start2*.

6.6.10.4 Доступк в Вектору из Си

Вектор можно считывать и изменять из Си ,например, с помощью функций `scm_c_vector_ref` и `scm_c_vector_set_x`. В дополнение к этим функциям есть еще два способа доступа к векторам из Си, которые могут быть эффективными в определенных ситуациях: вы можете ограничить себя работой с простыми векторами *simple vectors* и затем использовать очень быстрые простые векторные макросы; или вы можете воспользоваться базовым доступом ко всем типам Массивов (см. Раздел 6.6.13.5 [Accessing Arrays from C], страница 223), более подробно, но может эффективно работать со всеми видами векторов(и массивов). Для векторов, вы можете использовать функции `scm_vector_elements` и `scm_vector_writable_elements`.

`int scm_is_simple_vector` (*SCM obj*) [C Function]

Return non-zero if *obj* is a simple vector, else return zero. A simple vector is a vector that can be used with the `SCM_SIMPLE_*` macros below.

The following functions are guaranteed to return simple vectors: `scm_make_vector`, `scm_c_make_vector`, `scm_vector`, `scm_list_to_vector`.

`size_t SCM_SIMPLE_VECTOR_LENGTH` (*SCM vec*) [C Macro]

Evaluates to the length of the simple vector *vec*. No type checking is done.

`SCM SCM_SIMPLE_VECTOR_REF` (*SCM vec, size_t idx*) [C Macro]

Evaluates to the element at position *idx* in the simple vector *vec*. No type or range checking is done.

`void SCM_SIMPLE_VECTOR_SET` (*SCM vec, size_t idx, SCM val*) [C Macro]

Sets the element at position *idx* in the simple vector *vec* to *val*. No type or range checking is done.

`const SCM * scm_vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

Acquire a handle for the vector *vec* and return a pointer to the elements of it. This pointer can only be used to read the elements of *vec*. When *vec* is not a vector, an error is signaled. The handle must eventually be released with `scm_array_handle_release`.

The variables pointed to by *lenp* and *incp* are filled with the number of elements of the vector and the increment (number of elements) between successive elements, respectively. Successive elements of *vec* need not be contiguous in their underlying "root vector" returned here; hence the increment is not necessarily equal to 1 and may well be negative too (см. Раздел 6.6.13.3 [Shared Arrays], страница 218).

The following example shows the typical way to use this function. It creates a list of all elements of *vec* (in reverse order).

```
scm_t_array_handle handle;
size_t i, len;
ssize_t inc;
const SCM *elt;
SCM list;
```

```
elt = scm_vector_elements (vec, &handle, &len, &inc);
list = SCM_EOL;
for (i = 0; i < len; i++, elt += inc)
  list = scm_cons (*elt, list);
scm_array_handle_release (&handle);
```

`SCM * scm_vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_vector_elements` but the pointer can be used to modify the vector.

The following example shows the typical way to use this function. It fills a vector with `#t`.

```
scm_t_array_handle handle;
size_t i, len;
ssize_t inc;
SCM *elt;

elt = scm_vector_writable_elements (vec, &handle, &len, &inc);
for (i = 0; i < len; i++, elt += inc)
  *elt = SCM_BOOL_T;
scm_array_handle_release (&handle);
```

6.6.10.5 Унифицированные Числовые Вектора

Унифицированным числовым вектором является вектор, элементы которого представляют собой один числовой тип. Guile предоставляет унифицированные числовые вектора для знаковых и бесзнаковых 8-битных, 16-битных, 32 и 64 битных целых, чисел с плавающей точкой, комплексных чисел. Дополнительная информация См. Раздел 7.5.5 [SRFI-4], страница 621,

Для многих целей, байтовые векторы работают также, как и унифицированные векторы, и имеют преимущество тем что хорошо интегрируются с двоичным вводом и выводом. Для получения дополнительной информации о байтовых векторах См. Раздел 6.6.12 [Bytevectors], страница 205.

6.6.11 Битовые Вектора

Битовыми векторами являются с нулевого индекса одномерные булевые массивы. Они отображаются как последовательность `0` и `1` с префиксом `#*`, например,

```
(make-bitvector 8 #f) ⇒
#*00000000
```

Битовые векторы являются частным случаем одномерных битовых массивов и поэтому к ним могут быть применены процедуры для работы с Массивами, См. Раздел 6.6.13 [Arrays], страница 212.

`bitvector?` *obj* [Scheme Procedure]
`scm_bitvector_p` (*obj*) [C Function]

Return `#t` when *obj* is a bitvector, else return `#f`.

`int scm_is_bitvector` (*SCM obj*) [C Function]

Return `1` when *obj* is a bitvector, else return `0`.

`make-bitvector` *len* **[***fill***]** [Scheme Procedure]
`scm_make_bitvector` (*len, fill*) [C Function]
Create a new bitvector of length *len* and optionally initialize all elements to *fill*.

`SCM scm_c_make_bitvector` (*size_t len, SCM fill*) [C Function]
Like `scm_make_bitvector`, but the length is given as a `size_t`.

`bitvector` *bit ...* [Scheme Procedure]
`scm_bitvector` (*bits*) [C Function]
Create a new bitvector with the arguments as elements.

`bitvector-length` *vec* [Scheme Procedure]
`scm_bitvector_length` (*vec*) [C Function]
Return the length of the bitvector *vec*.

`size_t scm_c_bitvector_length` (*SCM vec*) [C Function]
Like `scm_bitvector_length`, but the length is returned as a `size_t`.

`bitvector-ref` *vec idx* [Scheme Procedure]
`scm_bitvector_ref` (*vec, idx*) [C Function]
Return the element at index *idx* of the bitvector *vec*.

`SCM scm_c_bitvector_ref` (*SCM vec, size_t idx*) [C Function]
Return the element at index *idx* of the bitvector *vec*.

`bitvector-set!` *vec idx val* [Scheme Procedure]
`scm_bitvector_set_x` (*vec, idx, val*) [C Function]
Set the element at index *idx* of the bitvector *vec* when *val* is true, else clear it.

`SCM scm_c_bitvector_set_x` (*SCM vec, size_t idx, SCM val*) [C Function]
Set the element at index *idx* of the bitvector *vec* when *val* is true, else clear it.

`bitvector-fill!` *vec val* [Scheme Procedure]
`scm_bitvector_fill_x` (*vec, val*) [C Function]
Set all elements of the bitvector *vec* when *val* is true, else clear them.

`list->bitvector` *list* [Scheme Procedure]
`scm_list_to_bitvector` (*list*) [C Function]
Return a new bitvector initialized with the elements of *list*.

`bitvector->list` *vec* [Scheme Procedure]
`scm_bitvector_to_list` (*vec*) [C Function]
Return a new list initialized with the elements of the bitvector *vec*.

`bit-count` *bool bitvector* [Scheme Procedure]
`scm_bit_count` (*bool, bitvector*) [C Function]
Return a count of how many entries in *bitvector* are equal to *bool*. For example,

```
(bit-count #f #*000111000)  ⇒ 6
```

`bit-position` *bool bitvector start* [Scheme Procedure]
`scm_bit_position` (*bool, bitvector, start*) [C Function]
Return the index of the first occurrence of *bool* in *bitvector*, starting from *start*. If there is no *bool* entry between *start* and the end of *bitvector*, then return `#f`. For example,

```
(bit-position #t #*000101 0)  ⇒ 3
(bit-position #f #*0001111 3) ⇒ #f
```

`bit-invert!` *bitvector* [Scheme Procedure]
`scm_bit_invert_x` (*bitvector*) [C Function]
Modify *bitvector* by replacing each element with its negation.

`bit-set*!` *bitvector uvec bool* [Scheme Procedure]
`scm_bit_set_star_x` (*bitvector, uvec, bool*) [C Function]
Set entries of *bitvector* to *bool*, with *uvec* selecting the entries to change. The return value is unspecified.

If *uvec* is a bit vector, then those entries where it has `#t` are the ones in *bitvector* which are set to *bool*. *uvec* and *bitvector* must be the same length. When *bool* is `#t` it's like *uvec* is OR'ed into *bitvector*. Or when *bool* is `#f` it can be seen as an ANDNOT.

```
(define bv #*01000010)
(bit-set*! bv #*10010001 #t)
bv
⇒ #*11010011
```

If *uvec* is a uniform vector of unsigned long integers, then they're indexes into *bitvector* which are set to *bool*.

```
(define bv #*01000010)
(bit-set*! bv #u(5 2 7) #t)
bv
⇒ #*01100111
```

`bit-count*` *bitvector uvec bool* [Scheme Procedure]
`scm_bit_count_star` (*bitvector, uvec, bool*) [C Function]
Return a count of how many entries in *bitvector* are equal to *bool*, with *uvec* selecting the entries to consider.

uvec is interpreted in the same way as for `bit-set*!` above. Namely, if *uvec* is a bit vector then entries which have `#t` there are considered in *bitvector*. Or if *uvec* is a uniform vector of unsigned long integers then it's the indexes in *bitvector* to consider.

For example,

```
(bit-count* #*01110111 #*11001101 #t) ⇒ 3
(bit-count* #*01110111 #u32(7 0 4) #f)  ⇒ 2
```

`const scm_t_uint32 * scm_bitvector_elements` (*SCM vec,* [C Function]
*scm_t_array_handle *handle, size_t *offp, size_t *lenp, ssize_t *incp*)
Like `scm_vector_elements` (см. Раздел 6.6.10.4 [Vector Accessing from C], страница 201), but for bitvectors. The variable pointed to by *offp* is set to the value

returned by `scm_array_handle_bit_elements_offset`. See `scm_array_handle_bit_elements` for how to use the returned pointer and the offset.

`scm_t_uint32 * scm_bitvector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *offp, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_bitvector_elements`, but the pointer is good for reading and writing.

6.6.12 Байтовые вектора

A *bytevector*байтовый вектор это чистая байтовая строка. Модуль `(rnrs bytevectors)` обеспечивает программный интерфес, указанный в Revised^6 Report on the Algorithmic Language Scheme (R6RS) (`http://www.r6rs.org/`). Он содержит процедуры манипулирования байтовыми векторами и функции их интерпретации различными способами: байтовый вектор может представить как знаковое или беззнаковое целое различной длины и порядка следования байт, как число с плавающей точкой представленное по стандарту IEEE-754, или как стороку. Это полезный инструмент для кодирования и декодирования двоичных данных.

R6RS (Section 4.3.4) определяет внешнее представление для байтовых векторов, в соответствии с которым октеты (целые числа в пределах 0–255) содержащиеся в байтовом векторе представляются в виде списка с префиксом `#vu8`:

```
#vu8(1 53 204)
```

обозначает 3-байтовый байтовый вектор содержащий октеты 1, 53, и 204. Подобно строковым литералам, булевым знчениям и др., байтовые векторы являюстя “самоквотируемыми”, т.е. им не нужно указвать квотирование:

```
#vu8(1 53 204)
⇒ #vu8(1 53 204)
```

Байтовые вектора могут использоваться с двоичными входными/выходными примитивами (см. Раздел 6.14.2 [Binary I/O], страница 353).

6.6.12.1 Порядок байтов

Некоторые из следующих процедур принимают параметр порядок байтов *endianness*. Порядок байтов *endianness* определяет какой порядок у байтов в многобайтовых числах: числа кодированные как *big endian* имеют свои самые значащие байты записанными первыми, тогда как числа кодированные как *little endian* первыми имеют самые младшие байты[3].

Little-endian -это внутренняя спецификация порядка байтов архитектуры IA32 и ее производных, в то время как big-endian является родным для SPARC и PowerPC, среди прочих. Процедура `native-endianness` возвращает внутреннюю спецификацию машины, на которой она работает.

`native-endianness` [Scheme Procedure]

`scm_native_endianness ()` [C Function]

Return a value denoting the native endianness of the host machine.

[3] Big-endian и little-endian являются наиболее распространенными способами упорядочения байт в многобайтовых числах, но существуют и другие. Например библиотека GNU MP допускает *порядок слов* независимый от *порядка байтов* (см. Раздел “Integer Import and Export” в *The GNU Multiple Precision Arithmetic Library Manual*).

`endianness` *symbol* [Scheme Macro]
Return an object denoting the endianness specified by *symbol*. If *symbol* is neither `big` nor `little` then an error is raised at expand-time.

`scm_endianness_big` [C Variable]
`scm_endianness_little` [C Variable]
The objects denoting big- and little-endianness, respectively.

6.6.12.2 Манипулирование Байтовыми Векторами

Байтовый вектор может быть создан, скопирован и проанализирован и передан процедурам и функциям Си.

`make-bytevector` *len* [*fill*] [Scheme Procedure]
`scm_make_bytevector` (*len, fill*) [C Function]
`scm_c_make_bytevector` (*size_t len*) [C Function]
Return a new bytevector of *len* bytes. Optionally, if *fill* is given, fill it with *fill*; *fill* must be in the range [-128,255].

`bytevector?` *obj* [Scheme Procedure]
`scm_bytevector_p` (*obj*) [C Function]
Return true if *obj* is a bytevector.

`int scm_is_bytevector` (*SCM obj*) [C Function]
Equivalent to `scm_is_true (scm_bytevector_p (obj))`.

`bytevector-length` *bv* [Scheme Procedure]
`scm_bytevector_length` (*bv*) [C Function]
Return the length in bytes of bytevector *bv*.

`size_t scm_c_bytevector_length` (*SCM bv*) [C Function]
Likewise, return the length in bytes of bytevector *bv*.

`bytevector=?` *bv1 bv2* [Scheme Procedure]
`scm_bytevector_eq_p` (*bv1, bv2*) [C Function]
Return is *bv1* equals to *bv2*—i.e., if they have the same length and contents.

`bytevector-fill!` *bv fill* [Scheme Procedure]
`scm_bytevector_fill_x` (*bv, fill*) [C Function]
Fill bytevector *bv* with *fill*, a byte.

`bytevector-copy!` *source source-start target target-start len* [Scheme Procedure]
`scm_bytevector_copy_x` (*source, source_start, target, target_start, len*) [C Function]
Copy *len* bytes from *source* into *target*, starting reading from *source-start* (a positive index within *source*) and start writing at *target-start*. It is permitted for the *source* and *target* regions to overlap.

`bytevector-copy` *bv* [Scheme Procedure]
`scm_bytevector_copy` (*bv*) [C Function]
Return a newly allocated copy of *bv*.

`scm_t_uint8 scm_c_bytevector_ref` (*SCM bv*, *size_t index*) [C Function]
Return the byte at *index* in bytevector *bv*.

`void scm_c_bytevector_set_x` (*SCM bv*, *size_t index*, *scm_t_uint8 value*) [C Function]
Set the byte at *index* in *bv* to *value*.

Low-level C macros are available. They do not perform any type-checking; as such they should be used with care.

`size_t SCM_BYTEVECTOR_LENGTH` (*bv*) [C Macro]
Return the length in bytes of bytevector *bv*.

`signed char * SCM_BYTEVECTOR_CONTENTS` (*bv*) [C Macro]
Return a pointer to the contents of bytevector *bv*.

6.6.12.3 Интерпретация Байтового Вектора как Целого

Содержимое байтовго вектора можно интерпретировать как последовательность целых чисел любого заданного размера, знака и порядка следования байтов.

```
(let ((bv (make-bytevector 4)))
  (bytevector-u8-set! bv 0 #x12)
  (bytevector-u8-set! bv 1 #x34)
  (bytevector-u8-set! bv 2 #x56)
  (bytevector-u8-set! bv 3 #x78)

  (map (lambda (number)
         (number->string number 16))
       (list (bytevector-u8-ref bv 0)
             (bytevector-u16-ref bv 0 (endianness big))
             (bytevector-u32-ref bv 0 (endianness little)))))
```

⇒ `("12" "1234" "78563412")`

Ниже описаны наиболее общие процедуры для интерпретации содержимого в виде целых чисел.

`bytevector-uint-ref` *bv index endianness size* [Scheme Procedure]
`scm_bytevector_uint_ref` (*bv*, *index*, *endianness*, *size*) [C Function]
Return the *size*-byte long unsigned integer at index *index* in *bv*, decoded according to *endianness*.

`bytevector-sint-ref` *bv index endianness size* [Scheme Procedure]
`scm_bytevector_sint_ref` (*bv*, *index*, *endianness*, *size*) [C Function]
Return the *size*-byte long signed integer at index *index* in *bv*, decoded according to *endianness*.

`bytevector-uint-set!` *bv index value endianness size* [Scheme Procedure]
`scm_bytevector_uint_set_x` (*bv*, *index*, *value*, *endianness*, *size*) [C Function]
Set the *size*-byte long unsigned integer at *index* to *value*, encoded according to *endianness*.

`bytevector-sint-set!` *bv index value endianness size* [Scheme Procedure]
`scm_bytevector_sint_set_x` (*bv*, *index*, *value*, *endianness*, *size*) [C Function]
Set the *size*-byte long signed integer at *index* to *value*, encoded according to *endianness*.

The following procedures are similar to the ones above, but specialized to a given integer size:

`bytevector-u8-ref` *bv index* [Scheme Procedure]
`bytevector-s8-ref` *bv index* [Scheme Procedure]
`bytevector-u16-ref` *bv index endianness* [Scheme Procedure]
`bytevector-s16-ref` *bv index endianness* [Scheme Procedure]
`bytevector-u32-ref` *bv index endianness* [Scheme Procedure]
`bytevector-s32-ref` *bv index endianness* [Scheme Procedure]
`bytevector-u64-ref` *bv index endianness* [Scheme Procedure]
`bytevector-s64-ref` *bv index endianness* [Scheme Procedure]
`scm_bytevector_u8_ref` (*bv*, *index*) [C Function]
`scm_bytevector_s8_ref` (*bv*, *index*) [C Function]
`scm_bytevector_u16_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_s16_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_u32_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_s32_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_u64_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_s64_ref` (*bv*, *index*, *endianness*) [C Function]
Return the unsigned *n*-bit (signed) integer (where *n* is 8, 16, 32 or 64) from *bv* at *index*, decoded according to *endianness*.

`bytevector-u8-set!` *bv index value* [Scheme Procedure]
`bytevector-s8-set!` *bv index value* [Scheme Procedure]
`bytevector-u16-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-s16-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-u32-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-s32-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-u64-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-s64-set!` *bv index value endianness* [Scheme Procedure]
`scm_bytevector_u8_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_s8_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_u16_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_s16_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_u32_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_s32_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_u64_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_s64_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
Store *value* as an *n*-bit (signed) integer (where *n* is 8, 16, 32 or 64) in *bv* at *index*, encoded according to *endianness*.

Finally, a variant specialized for the host's endianness is available for each of these functions (with the exception of the `u8` and `s8` accessors, as endianness is about byte order and there is only 1 byte):

`bytevector-u16-native-ref` *bv index* [Scheme Procedure]
`bytevector-s16-native-ref` *bv index* [Scheme Procedure]
`bytevector-u32-native-ref` *bv index* [Scheme Procedure]
`bytevector-s32-native-ref` *bv index* [Scheme Procedure]
`bytevector-u64-native-ref` *bv index* [Scheme Procedure]
`bytevector-s64-native-ref` *bv index* [Scheme Procedure]
`scm_bytevector_u16_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_s16_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_u32_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_s32_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_u64_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_s64_native_ref` (*bv*, *index*) [C Function]
Return the unsigned *n*-bit (signed) integer (where *n* is 8, 16, 32 or 64) from *bv* at *index*, decoded according to the host's native endianness.

`bytevector-u16-native-set!` *bv index value* [Scheme Procedure]
`bytevector-s16-native-set!` *bv index value* [Scheme Procedure]
`bytevector-u32-native-set!` *bv index value* [Scheme Procedure]
`bytevector-s32-native-set!` *bv index value* [Scheme Procedure]
`bytevector-u64-native-set!` *bv index value* [Scheme Procedure]
`bytevector-s64-native-set!` *bv index value* [Scheme Procedure]
`scm_bytevector_u16_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_s16_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_u32_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_s32_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_u64_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_s64_native_set_x` (*bv*, *index*, *value*) [C Function]
Store *value* as an *n*-bit (signed) integer (where *n* is 8, 16, 32 or 64) in *bv* at *index*, encoded according to the host's native endianness.

6.6.12.4 Преобразование байтового вектора в/из списка целых чисел

Содержимое байтового вектора может быть легко преобразовано в/из спис(ок/ка) знаковых или беззнаковых целых чисел:

```
(bytevector->sint-list (u8-list->bytevector (make-list 4 255))
                       (endianness little) 2)
⇒ (-1 -1)
```

`bytevector->u8-list` *bv* [Scheme Procedure]
`scm_bytevector_to_u8_list` (*bv*) [C Function]
Return a newly allocated list of unsigned 8-bit integers from the contents of *bv*.

`u8-list->bytevector` *lst* [Scheme Procedure]
`scm_u8_list_to_bytevector` (*lst*) [C Function]
Return a newly allocated bytevector consisting of the unsigned 8-bit integers listed in *lst*.

`bytevector->uint-list` *bv endianness size* [Scheme Procedure]
`scm_bytevector_to_uint_list` (*bv*, *endianness*, *size*) [C Function]
Return a list of unsigned integers of *size* bytes representing the contents of *bv*, decoded according to *endianness*.

`bytevector->sint-list` *bv endianness size* [Scheme Procedure]
`scm_bytevector_to_sint_list` (*bv*, *endianness*, *size*) [C Function]
Return a list of signed integers of *size* bytes representing the contents of *bv*, decoded according to *endianness*.

`uint-list->bytevector` *lst endianness size* [Scheme Procedure]
`scm_uint_list_to_bytevector` (*lst*, *endianness*, *size*) [C Function]
Return a new bytevector containing the unsigned integers listed in *lst* and encoded on *size* bytes according to *endianness*.

`sint-list->bytevector` *lst endianness size* [Scheme Procedure]
`scm_sint_list_to_bytevector` (*lst*, *endianness*, *size*) [C Function]
Return a new bytevector containing the signed integers listed in *lst* and encoded on *size* bytes according to *endianness*.

6.6.12.5 Интерпретация Байтового Вектора как Числа с плавающей точкой

Содержимое Байтового Вектора можно получить в виде числа с плавающей точкой одиночной или двойной точности IEEE-754 (соответственно длиной 32 и 64-бита) используя процедуры описанные здесь.

`bytevector-ieee-single-ref` *bv index endianness* [Scheme Procedure]
`bytevector-ieee-double-ref` *bv index endianness* [Scheme Procedure]
`scm_bytevector_ieee_single_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_ieee_double_ref` (*bv*, *index*, *endianness*) [C Function]
Return the IEEE-754 single-precision floating point number from *bv* at *index* according to *endianness*.

`bytevector-ieee-single-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-ieee-double-set!` *bv index value endianness* [Scheme Procedure]
`scm_bytevector_ieee_single_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_ieee_double_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
Store real number *value* in *bv* at *index* according to *endianness*.

Specialized procedures are also available:

`bytevector-ieee-single-native-ref` *bv index* [Scheme Procedure]
`bytevector-ieee-double-native-ref` *bv index* [Scheme Procedure]
`scm_bytevector_ieee_single_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_ieee_double_native_ref` (*bv*, *index*) [C Function]
Return the IEEE-754 single-precision floating point number from *bv* at *index* according to the host's native endianness.

`bytevector-ieee-single-native-set!` *bv index value* [Scheme Procedure]
`bytevector-ieee-double-native-set!` *bv index value* [Scheme Procedure]
`scm_bytevector_ieee_single_native_set_x` (*bv, index, value*) [C Function]
`scm_bytevector_ieee_double_native_set_x` (*bv, index, value*) [C Function]
Store real number *value* in *bv* at *index* according to the host's native endianness.

6.6.12.6 Итерпретация содержимого Байтового Вектора как строки Unicode

Содержимое байтового вектора также может быть интерпретировано как строки Unicode, закодированные в одном из самых широко распространенных форматов кодирования. См. Раздел 6.6.5.13 [Representing Strings as Bytes], страница 169, для более общего интерфейса.

```
(utf8->string (u8-list->bytevector '(99 97 102 101)))
⇒ "cafe"

(string->utf8 "cafe") ;; SMALL LATIN LETTER E WITH ACUTE ACCENT
⇒ #vu8(99 97 102 195 169)
```

`string-utf8-length` *str* [Scheme Procedure]
`SCM scm_string_utf8_length` (*str*) [C function]
`size_t scm_c_string_utf8_length` (*str*) [C function]
Return the number of bytes in the UTF-8 representation of *str*.

`string->utf8` *str* [Scheme Procedure]
`string->utf16` *str* [*endianness*] [Scheme Procedure]
`string->utf32` *str* [*endianness*] [Scheme Procedure]
`scm_string_to_utf8` (*str*) [C Function]
`scm_string_to_utf16` (*str, endianness*) [C Function]
`scm_string_to_utf32` (*str, endianness*) [C Function]
Return a newly allocated bytevector that contains the UTF-8, UTF-16, or UTF-32 (aka. UCS-4) encoding of *str*. For UTF-16 and UTF-32, *endianness* should be the symbol `big` or `little`; when omitted, it defaults to big endian.

`utf8->string` *utf* [Scheme Procedure]
`utf16->string` *utf* [*endianness*] [Scheme Procedure]
`utf32->string` *utf* [*endianness*] [Scheme Procedure]
`scm_utf8_to_string` (*utf*) [C Function]
`scm_utf16_to_string` (*utf, endianness*) [C Function]
`scm_utf32_to_string` (*utf, endianness*) [C Function]
Return a newly allocated string that contains from the UTF-8-, UTF-16-, or UTF-32-decoded contents of bytevector *utf*. For UTF-16 and UTF-32, *endianness* should be the symbol `big` or `little`; when omitted, it defaults to big endian.

6.6.12.7 Доступ к Байтовому вектору через API работы с Массивами

В качестве расширения для R6RS, Guile позволяет управлять байтовыми векторами с помощью процедур работы с массивами (см. Раздел 6.6.13 [Arrays], страница 212).

При использовании этого API, байты доступны по одному как 8-битовы целые числа без знака:

```
(define bv #vu8(0 1 2 3))

(array? bv)
⇒ #t

(array-rank bv)
⇒ 1

(array-ref bv 2)
⇒ 2

;; Note the different argument order on array-set!.
(array-set! bv 77 2)
(array-ref bv 2)
⇒ 77

(array-type bv)
⇒ vu8
```

6.6.12.8 Доступ к Байтовому Вектору через SRFI-4 API

Доступ к байтовым векторам также возможен с помощью SRFI-4 API. См. Раздел 7.5.5.3 [SRFI-4 and Bytevectors], страница 629, для получения более подробной информации.

6.6.13 Массивы(Arrays)

Arrays Массивы представляют собой набор ячеек, организованных в произвольное число измерений. Каждую ячейку можно получить в течении фиксированного времени, указав индекс для каждого измерения.

В текущей реализации массив использует какой либо вектор для реального хранения своих элементов. Любой вектор будет работать, поэтому вы можете иметь массивы единых числовых значений, массивы символов, массивы бит и конечно массивы произвольных значений Scheme. Например, массивы с базовым типом `c64vector` могут быть хороши для обработки цифровых сигналов, в то время как массивы созданные из `u8vector`, могут использоваться для хранения изображений в серой шкалой цветности.

Число измерений массива называется его рангом *rank*. Таким образом, матрица представляет собой массив ранга 2, а вектор имеет ранг 1. При доступе к элементу массива вы должны указать одно точное целое для каждого его измерения. Эти целые числа называются индексами элемента массива. Массив задает допустимый диапазон индексов для каждого измерения через инклюзивные нижние и верхнии границы. Эти границы могут быть отритцательными, но верхняя граница должна быть больше нижней границы минус 1. Когад все нижние границы равны нулю, он называется массивом с нулевым началом. *zero-origin*.

Массивы могут иметь ранг 0, который можно интерпретировать как скаляр. Таким образом, массив нулевого ранга может хранить ровно один объект, а список индексов этого элемента - пустой список.

Массивы содержат 0 элементов, когда одна из размерностей равна нулю. Эти пустые массивы сохраняют информацию об их форме: матрица с нулевыми столбцами и 3 строками отличается от матрицы с тремя столбцами и нулем строк, что в свою очередь отличается от вектора нулевой длины.

Процедуры работающие с массивами являются полиморфными, они обрабатывают строки, унифицированные числовые вектора, байтовые вектора, и обычные вектора как одномерные массивы.

6.6.13.1 Синтаксис определения Массивов

Массив отображается как решетка `#`, за которой следует ранг, за которым следует тег, который описывает основной вектор, необязательно сопровождаемый информацией о его форме, и наконце следуют ячейки, организованные по размерностям с помощью круглых скобок.

Другими словами, тег массива имеет вид:

```
#<rank><vectag><@lower><:len><@lower><:len>...
```

где `<rank>` - положительное целое число в десятичном значении, задающее ранг массива. Он может быть опущен когда ранг равен 1, а массив не разделен и имеет начальный элемент с нулевым индексом (см. ниже). Для разделяемых массивов и массивов с ненулевым начальным индексом, ранг указывается всегда, даже когда он единичный, и их отличают от обычных векторов.

Часть `<vectag>` является тегом для унифицированного числовго вектора, такого как `u8`, `s16`, и др., `b` для битового вектора, или `a` для стороки. Он пуст для обычных векторов.

Часть `<@lower>` начинается с символа '`@`' за которым следует целое число со знаком десятичном значении, которое дает нижнюю границу размерности массива. Для каждого измерения существует один `<@lower>` Когда все нижние границы равны нулю, все части `<@lower>` опускаются.

Часть `<:len>` начинается символом '`:`', за которым следует целое число без знака в десятичном значении, дающее длину размерности. Как и для нижних границ, для каждого измерения существует только одна длина `<:len>`, и часть `<:len>` всегда следует за частью `<@lower>` для каждого измерения. Только длины печатаются, когда массив не может быть выведен из за вложенных списков элементов массива, что может произойти когда длина хотя бы одной размерности равна нулю.

В качестве специального случая, массив с рангом равным 0 печтатется как `#0<vectag>(<scalar>)`, где `<scalar>` - это результат печати одного элемента массива.

Таким образом,

`#(1 2 3)` это обычный массив ранга 1 с нижней границей 0 в размерности 0 (т.е, а это обычный вектор)

`#@2(1 2 3)`
это обычный массив ранга 1 с нижней границей равной 2 в размерности 0.

`#2((1 2 3) (4 5 6))`
: это не унифицированный(содержащий элементы различных типов) массив ранга 2; матрица 2×3 с индексами в области 0..1 и 0..2.

`#u32(0 1 2)`
: унифицированный массив типа u32 ранга 1.

`#2u32@2@3((1 2) (2 3))`
: унифицированный массив с элементами типа u32 ранга 2 с индексами в пределах 2..3 и 3..4.

`#2()` представляет собой двухмерный массив с диапазонами индексов 0..-1 и 0..-1, т.е обе размерности имеют нулевую длину.

`#2:0:2()` представляет собой двумерный массив с индексами 0..-1 и 0..1, т.е. первая размрность имеет нулевую длину, а вторая имеет длину 2.

`#0(12)` это массив с рангом 0 содержащий число 12.

Кроме того, байтовый вектор (bytevectors) также является массивом, но использует другой синтаксис: (см. Раздел 6.6.12 [Bytevectors], страница 205):

`#vu8(1 2 3)`
: представляет собой байтовый вектор длиной в 3-байта, содержащий 1, 2, 3.

6.6.13.2 Процедуры работающие с Массивами

Когда создается массив, диапазон каждго измерения должен быть определен, например, для создания массива 2×3 с индексами начинающимися с нуля:

```
(make-array 'ho 2 3) ⇒ #2((ho ho ho) (ho ho ho))
```

Диапазон каждого измерения также может быть задан явно, другой способ создания тогоже массива:

```
(make-array 'ho '(0 1) '(0 2)) ⇒ #2((ho ho ho) (ho ho ho))
```

Следующие процедуры могут использоваться с массивами(или векторами). Аргумент, показанный как *idx*... означает один параметр для каждого измерения в массива. Аргумент *idxlist* означает список таких значений, по одному для каждого измерения.

`array?` *obj* [Scheme Procedure]
`scm_array_p` (*obj, unused*) [C Function]
: Return `#t` if the *obj* is an array, and `#f` if not.

  The second argument to scm_array_p is there for historical reasons, but it is not used. You should always pass `SCM_UNDEFINED` as its value.

`typed-array?` *obj type* [Scheme Procedure]
`scm_typed_array_p` (*obj, type*) [C Function]
: Return `#t` if the *obj* is an array of type *type*, and `#f` if not.

`int scm_is_array` (*SCM obj*) [C Function]
: Return `1` if the *obj* is an array and `0` if not.

`int scm_is_typed_array` (*SCM obj, SCM type*) [C Function]
Return `0` if the *obj* is an array of type *type*, and `1` if not.

`make-array` *fill bound . . .* [Scheme Procedure]
`scm_make_array` (*fill, bounds*) [C Function]
Equivalent to `(make-typed-array #t` *`fill bound`* `...)`.

`make-typed-array` *type fill bound . . .* [Scheme Procedure]
`scm_make_typed_array` (*type, fill, bounds*) [C Function]
Create and return an array that has as many dimensions as there are *bound*s and (maybe) fill it with *fill*.

The underlying storage vector is created according to *type*, which must be a symbol whose name is the 'vectag' of the array as explained above, or `#t` for ordinary, non-specialized arrays.

For example, using the symbol `f64` for *type* will create an array that uses a `f64vector` for storing its elements, and `a` will use a string.

When *fill* is not the special *unspecified* value, the new array is filled with *fill*. Otherwise, the initial contents of the array is unspecified. The special *unspecified* value is stored in the variable `*unspecified*` so that for example `(make-typed-array 'u32 *unspecified* 4)` creates a uninitialized `u32` vector of length 4.

Each *bound* may be a positive non-zero integer *n*, in which case the index for that dimension can range from 0 through *n*-1; or an explicit index range specifier in the form `(LOWER UPPER)`, where both *lower* and *upper* are integers, possibly less than zero, and possibly the same number (however, *lower* cannot be greater than *upper*).

`list->array` *dimspec list* [Scheme Procedure]
Equivalent to `(list->typed-array #t` *`dimspec list`*`)`.

`list->typed-array` *type dimspec list* [Scheme Procedure]
`scm_list_to_typed_array` (*type, dimspec, list*) [C Function]
Return an array of the type indicated by *type* with elements the same as those of *list*.

The argument *dimspec* determines the number of dimensions of the array and their lower bounds. When *dimspec* is an exact integer, it gives the number of dimensions directly and all lower bounds are zero. When it is a list of exact integers, then each element is the lower index bound of a dimension, and there will be as many dimensions as elements in the list.

`array-type` *array* [Scheme Procedure]
`scm_array_type` (*array*) [C Function]
Return the type of *array*. This is the 'vectag' used for printing *array* (or `#t` for ordinary arrays) and can be used with `make-typed-array` to create an array of the same kind as *array*.

`array-ref` *array idx . . .* [Scheme Procedure]
`scm_array_ref` (*array, idxlist*) [C Function]
Return the element at `(idx ...)` in *array*.

```
(define a (make-array 999 '(1 2) '(3 4)))
(array-ref a 2 4) ⇒ 999
```

`array-in-bounds?` *array idx ...* [Scheme Procedure]
`scm_array_in_bounds_p` (*array*, *idxlist*) [C Function]
Return `#t` if the given indices would be acceptable to `array-ref`.

```
(define a (make-array #f '(1 2) '(3 4)))
(array-in-bounds? a 2 3) ⇒ #t
(array-in-bounds? a 0 0) ⇒ #f
```

`array-set!` *array obj idx ...* [Scheme Procedure]
`scm_array_set_x` (*array*, *obj*, *idxlist*) [C Function]
Set the element at `(idx ...)` in *array* to *obj*. The return value is unspecified.

```
(define a (make-array #f '(0 1) '(0 1)))
(array-set! a #t 1 1)
a ⇒ #2((#f #f) (#f #t))
```

`array-shape` *array* [Scheme Procedure]
`array-dimensions` *array* [Scheme Procedure]
`scm_array_dimensions` (*array*) [C Function]
Return a list of the bounds for each dimension of *array*.

`array-shape` gives `(`*`lower upper`*`)` for each dimension. `array-dimensions` instead returns just *upper* + 1 for dimensions with a 0 lower bound. Both are suitable as input to `make-array`.

For example,

```
(define a (make-array 'foo '(-1 3) 5))
(array-shape a)      ⇒ ((-1 3) (0 4))
(array-dimensions a) ⇒ ((-1 3) 5)
```

`array-length` *array* [Scheme Procedure]
`scm_array_length` (*array*) [C Function]
`size_t` *scm_c_array_length* (*array*) [C Function]
Return the length of an array: its first dimension. It is an error to ask for the length of an array of rank 0.

`array-rank` *array* [Scheme Procedure]
`scm_array_rank` (*array*) [C Function]
Return the rank of *array*.

`size_t scm_c_array_rank` (*SCM array*) [C Function]
Return the rank of *array* as a `size_t`.

`array->list` *array* [Scheme Procedure]
`scm_array_to_list` (*array*) [C Function]
Return a list consisting of all the elements, in order, of *array*.

`array-copy!` *src dst* [Scheme Procedure]
`array-copy-in-order!` *src dst* [Scheme Procedure]
`scm_array_copy_x` (*src*, *dst*) [C Function]
Copy every element from vector or array *src* to the corresponding element of *dst*. *dst* must have the same rank as *src*, and be at least as large in each dimension. The return value is unspecified.

`array-fill!` *array fill* [Scheme Procedure]
`scm_array_fill_x` (*array*, *fill*) [C Function]
Store *fill* in every element of *array*. The value returned is unspecified.

`array-equal?` *array ...* [Scheme Procedure]
Return `#t` if all arguments are arrays with the same shape, the same type, and have corresponding elements which are either `equal?` or `array-equal?`. This function differs from `equal?` (см. Раздел 6.11.1 [Equality], страница 301) in that all arguments must be arrays.

`array-map!` *dst proc src ...* [Scheme Procedure]
`array-map-in-order!` *dst proc src ...* [Scheme Procedure]
`scm_array_map_x` (*dst*, *proc*, *srclist*) [C Function]
Set each element of the *dst* array to values obtained from calls to *proc*. The list of *src* arguments may be empty. The value returned is unspecified.

Each call is (*proc elem ...*), where each *elem* is from the corresponding *src* array, at the *dst* index. `array-map-in-order!` makes the calls in row-major order, `array-map!` makes them in an unspecified order.

The *src* arrays must have the same number of dimensions as *dst*, and must have a range for each dimension which covers the range in *dst*. This ensures all *dst* indices are valid in each *src*.

`array-for-each` *proc src1 src2 ...* [Scheme Procedure]
`scm_array_for_each` (*proc*, *src1*, *srclist*) [C Function]
Apply *proc* to each tuple of elements of *src1 src2 ...*, in row-major order. The value returned is unspecified.

`array-index-map!` *dst proc* [Scheme Procedure]
`scm_array_index_map_x` (*dst*, *proc*) [C Function]
Set each element of the *dst* array to values returned by calls to *proc*. The value returned is unspecified.

Each call is (*proc i1 ... iN*), where *i1...iN* is the destination index, one parameter for each dimension. The order in which the calls are made is unspecified.

For example, to create a 4×4 matrix representing a cyclic group,

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{pmatrix}$$

```
(define a (make-array #f 4 4))
(array-index-map! a (lambda (i j)
                      (modulo (+ i j) 4)))
```

An additional array function is available in the module `(ice-9 arrays)`. It can be used with:

```
(use-modules (ice-9 arrays))
```

`array-copy` *src* [Scheme Procedure]

Return a new array with the same elements, type and shape as *src*. However, the array increments may not be the same as those of *src*. In the current implementation, the returned array will be in row-major order, but that might change in the future. Use `array-copy!` on an array of known order if that is a concern.

6.6.13.3 Общие Массивы

`make-shared-array` *oldarray mapfunc bound ...* [Scheme Procedure]
`scm_make_shared_array` (*oldarray, mapfunc, boundlist*) [C Function]

Возвращает новый массив, который разделяет хранилище *oldarray*. Изменения, произведенные в нем, влияют на общее хранилище. Аргументы *bound ...* определяют границы нового массива, такие же как и в `make-array` (см. Раздел 6.6.13.2 [Array Procedures], страница 214).

mapfunc переводит координаты из нового массива в старый *oldarray*. Она вызывается как (***mapfunc*** `newidx1 ...`) с одним параметром для каждого измерения нового массива, и должна возвращать список индексов для старого массива *oldarray*, по одному для каждого измерения *oldarray*.

mapfunc должна быть афинно линейной, что означает, что каждый индекс в старом массиве *oldarray* должен быть сформирован добавлением (возможно отритцательным) целочисленных множителей, некоторых или всех новых индексов *newidx1* etc, плюс возможно целочисленное мещение. В каждом вызове множители и смещения должны быть одинаковыми.

Одно полезное использование для разделяемого массива - это ограничение диапазона некоторых измерений, чтобы применить функцию `array-for-each` или `array-fill!` только к части массива. Функции `list` может использоваться как *mapfunc*, в этом случае не изменяются значения индексов массива. Например,

```
(make-shared-array #2((a b c) (d e f) (g h i)) list 3 2)
⇒ #2((a b) (d e) (g h))
```

The new array can have fewer dimensions than *oldarray*, for example to take a column from an array.

```
(make-shared-array #2((a b c) (d e f) (g h i))
                   (lambda (i) (list i 2))
                   '(0 2))
⇒ #1(c f i)
```

A diagonal can be taken by using the single new array index for both row and column in the old array. For example,

```
(make-shared-array #2((a b c) (d e f) (g h i))
                   (lambda (i) (list i i))
                   '(0 2))
⇒ #1(a e i)
```

Dimensions can be increased by for instance considering portions of a one dimensional array as rows in a two dimensional array. (`array-contents` below can do the opposite, flattening an array.)

```
(make-shared-array #1(a b c d e f g h i j k l)
```

```
                    (lambda (i j) (list (+ (* i 3) j)))
                    4 3)
⇒ #2((a b c) (d e f) (g h i) (j k l))
```

By negating an index the order that elements appear can be reversed. The following just reverses the column order,

```
(make-shared-array #2((a b c) (d e f) (g h i))
                   (lambda (i j) (list i (- 2 j)))
                   3 3)
⇒ #2((c b a) (f e d) (i h g))
```

A fixed offset on indexes allows for instance a change from a 0 based to a 1 based array,

```
(define x #2((a b c) (d e f) (g h i)))
(define y (make-shared-array x
                             (lambda (i j) (list (1- i) (1- j)))
                             '(1 3) '(1 3)))
(array-ref x 0 0) ⇒ a
(array-ref y 1 1) ⇒ a
```

A multiple on an index allows every Nth element of an array to be taken. The following is every third element,

```
(make-shared-array #1(a b c d e f g h i j k l)
                   (lambda (i) (list (* i 3)))
                   4)
⇒ #1(a d g j)
```

The above examples can be combined to make weird and wonderful selections from an array, but it's important to note that because *mapfunc* must be affine linear, arbitrary permutations are not possible.

In the current implementation, *mapfunc* is not called for every access to the new array but only on some sample points to establish a base and stride for new array indices in *oldarray* data. A few sample points are enough because *mapfunc* is linear.

`shared-array-increments` *array* [Scheme Procedure]
`scm_shared_array_increments` (*array*) [C Function]

For each dimension, return the distance between elements in the root vector.

`shared-array-offset` *array* [Scheme Procedure]
`scm_shared_array_offset` (*array*) [C Function]

Return the root vector index of the first element in the array.

`shared-array-root` *array* [Scheme Procedure]
`scm_shared_array_root` (*array*) [C Function]

Return the root vector of a shared array.

`array-contents` *array* [*strict*] [Scheme Procedure]
`scm_array_contents` (*array, strict*) [C Function]

If *array* may be *unrolled* into a one dimensional shared array without changing their order (last subscript changing fastest), then `array-contents` returns that shared

array, otherwise it returns `#f`. All arrays made by `make-array` and `make-typed-array` may be unrolled, some arrays made by `make-shared-array` may not be.

If the optional argument *strict* is provided, a shared array will be returned only if its elements are stored internally contiguous in memory.

`transpose-array` *array dim1 dim2 ...* [Scheme Procedure]
`scm_transpose_array` (*array, dimlist*) [C Function]

Return an array sharing contents with *array*, but with dimensions arranged in a different order. There must be one *dim* argument for each dimension of *array*. *dim1*, *dim2*, ... should be integers between 0 and the rank of the array to be returned. Each integer in that range must appear at least once in the argument list.

The values of *dim1*, *dim2*, ... correspond to dimensions in the array to be returned, and their positions in the argument list to dimensions of *array*. Several *dim*s may have the same value, in which case the returned array will have smaller rank than *array*.

```
(transpose-array '#2((a b) (c d)) 1 0) ⇒ #2((a c) (b d))
(transpose-array '#2((a b) (c d)) 0 0) ⇒ #1(a d)
(transpose-array '#3(((a b c) (d e f)) ((1 2 3) (4 5 6))) 1 1 0) ⇒
                 #2((a 4) (b 5) (c 6))
```

6.6.13.4 Массивы как массивы массивов

Математически, можно увидеть массив ранга n (н-мерный массив n-array) как массив более низкого ранга, где сами элементы представляют собой массивы (‘ячейки’).

Мы говорим о первых $n - k$ размерностях массива как о $n - k$-‘каркасе’ массива, тогда как последние k размерностей являются k- измерениями ‘ячеек’. Например, можно видеть 3х мерный массив может быть рассмотрен как 2 мерный -массив векторов (1-мерный массив) или как 1-мерный массив матриц размером(2-мерный массив). В каждом случае векторы или матрицы являются 1 мерными ячеками или двумерными ячеками массива. Эта терминология возникает в J языке.

Более неопределенная концепция ‘среза’ относится к подмножеству массвов, где некоторые индексы фиксируются, а другие остаются свбодными. В качестве объекта данных Guile ячейка совпадает с префиксом срез ‘prefix slice’ (первые $n-k$ индексов в исходном массиве фиксируются), за исключением того, что 0-ячека не может являтся общим массивом исходного массива, но 0-срез (где все индексы в исходном массиве фиксированы) является.

До версси version 2.0, у Guile была функция называемая закрытытием масссива ‘enclosed arrays’ для создания специального объекта массива массивов. Функции в этом разделе не нуждаются в специальных типах, весто этого ранг каркаса(frame) указывается в каждом вызове функции, либо неявно либо явно.

`array-cell-ref` *array idx ...* [Scheme Procedure]
`scm_array_cell_ref` (*array, idxlist*) [C Function]

If the length of *idxlist* equals the rank n of *array*, return the element at `(idx ...)`, just like `(array-ref array idx ...)`. If, however, the length k of *idxlist* is smaller than n, then return the $(n - k)$-cell of *array* given by *idxlist*, as a shared array.

For example:

```
(array-cell-ref #2((a b) (c d)) 0) ⇒ #(a b)
(array-cell-ref #2((a b) (c d)) 1) ⇒ #(c d)
(array-cell-ref #2((a b) (c d)) 1 1) ⇒ d
(array-cell-ref #2((a b) (c d))) ⇒ #2((a b) (c d))
```

`(apply array-cell-ref array indices)` is equivalent to

```
(let ((len (length indices)))
  (if (= (array-rank a) len)
    (apply array-ref a indices)
    (apply make-shared-array a
           (lambda t (append indices t))
           (drop (array-dimensions a) len))))
```

`array-slice` *array idx ...* [Scheme Procedure]
`scm_array_slice` (*array*, *idxlist*) [C Function]

Like `(array-cell-ref array idx ...)`, but return a 0-rank shared array into *ARRAY* if the length of *idxlist* matches the rank of *array*. This can be useful when using *ARRAY* as a place to write to.

Compare:

```
(array-cell-ref #2((a b) (c d)) 1 1) ⇒ d
(array-slice #2((a b) (c d)) 1 1) ⇒ #0(d)
(define a (make-array 'a 2 2))
(array-fill! (array-slice a 1 1) 'b)
a ⇒ #2((a a) (a b)).
(array-fill! (array-cell-ref a 1 1) 'b) ⇒ error: not an array
```

`(apply array-slice array indices)` is equivalent to

```
(apply make-shared-array a
  (lambda t (append indices t))
  (drop (array-dimensions a) (length indices)))
```

`array-cell-set!` *array x idx ...* [Scheme Procedure]
`scm_array_cell_set_x` (*array*, *x*, *idxlist*) [C Function]

If the length of *idxlist* equals the rank n of *array*, set the element at `(idx ...)` of *array* to *x*, just like `(array-set! array x idx ...)`. If, however, the length k of *idxlist* is smaller than n, then copy the $(n-k)$-rank array *x* into the $(n-k)$-cell of *array* given by *idxlist*. In this case, the last $(n-k)$ dimensions of *array* and the dimensions of *x* must match exactly.

This function returns the modified *array*.

For example:

```
(array-cell-set! (make-array 'a 2 2) b 1 1)
  ⇒ #2((a a) (a b))
(array-cell-set! (make-array 'a 2 2) #(x y) 1)
  ⇒ #2((a a) (x y))
```

Note that `array-cell-set!` will expect elements, not arrays, when the destination has rank 0. Use `array-slice` for the opposite behavior.

```
(array-cell-set! (make-array 'a 2 2) #0(b) 1 1)
```

```
  ⇒ #2((a a) (a #0(b)))
(let ((a (make-array 'a 2 2)))
  (array-copy! #0(b) (array-slice a 1 1)) a)
  ⇒ #2((a a) (a b))
```

`(apply array-cell-set! array x indices)` is equivalent to

```
(let ((len (length indices)))
  (if (= (array-rank array) len)
    (apply array-set! array x indices)
    (array-copy! x (apply array-cell-ref array indices)))
  array)
```

`array-slice-for-each` *frame-rank op x ...* [Scheme Procedure]
`scm_array_slice_for_each` (*array, frame_rank, op, xlist*) [C Function]

Each *x* must be an array of rank ≥ *frame-rank*, and the first *frame-rank* dimensions of each *x* must all be the same. *array-slice-for-each* calls *op* with each set of (rank(*x*) - *frame-rank*)-cells from *x*, in unspecified order.

array-slice-for-each allows you to loop over cells of any rank without having to carry an index list or construct shared arrays manually. The slices passed to *op* are always shared arrays of *X*, even if they are of rank 0, so it is possible to write to them.

This function returns an unspecified value.

For example, to sort the rows of rank-2 array `a`:

```
(array-slice-for-each 1 (lambda (x) (sort! x <)) a)
```

As another example, let `a` be a rank-2 array where each row is a 2-element vector (x, y). Let's compute the arguments of these vectors and store them in rank-1 array `b`.

```
(array-slice-for-each 1
  (lambda (a b)
    (array-set! b (atan (array-ref a 1) (array-ref a 0))))
  a b)
```

`(apply array-slice-for-each frame-rank op x)` is equivalent to

```
(let ((frame (take (array-dimensions (car x)) frank)))
  (unless (every (lambda (x)
                   (equal? frame (take (array-dimensions x) frank)))
                 (cdr x))
    (error))
  (array-index-map!
    (apply make-shared-array (make-array #t) (const '()) frame)
    (lambda i (apply op (map (lambda (x) (apply array-slice x i)) x)))))
```

`array-slice-for-each-in-order` *frame-rank op x ...* [Scheme Procedure]
`scm_array_slice_for_each_in_order` (*array, frame_rank, op, xlist*) [C Function]

Same as `array-slice-for-each`, but the arguments are traversed sequentially and in row-major order.

6.6.13.5 Доступ к Массивам из Си

Для взаимодействия с внешним кодом Си Guile предоставляет API, позволяющий Си-коду получать доступ к элементам массивов Scheme. В частности, для унифицированных числовых массивов API предоставляет лежащие в их основе данные как Си массив чисел соответствующего типа.

Хотя указатели на элементы массива используются, сам массив должен быть защищен, так что указатель остается в силе. Говорят что такой защищенный массив зарезервирован. Зарезервированный массив может быть прочитан, но изменения в нем, указываемых ссылкой элементов, недопустимы. При попытке такой модификации выдается сигнал ошибки.

(Это похоже на блокировку массива во время его использования, но без опасности "смертельной" блокировки. В многопоточной программе вам потребуется дополнительная синхронизация, что бы избежать изменения зарезервированных массивов.)

Вы должны позаботиться о том что бы всегда востанавливать из резерва массив после его резервирования, даже если имеются нелокальные выходы. Если между двумя этими вызовами может произойти нелокальный выход, вы должны установить контекст dynwind, который освобождает массив когда он оставлен. (см. Раздел 6.13.10 [Dynamic Wind], страница 339).

Кроме того, резрвирование и восстановление из резерва должны быть сопряжены. Например при резервировании двух и более массивов в определенном порядке вам необходимо осовбодить (востановить из резерва) их в противоположном порядке.

После того как вы зарезервировали массив и извлекли указатель на его элементы, вы должны определить расположение элементов в памяти. Guile позволяет вырезать фрагменты из массива без фактического создания копии, например, создание псевдонима для диагонали матрицы который можно рассматривать как вектор. Массивы, которые не являются результатом такой операции, не сохраняются в памяти и при работе с их элементами на прямую, нужно обращать на это внимание.

Размещение элементов массива в памяти можно определить с помощью функции *mapping function* отображения, которая вычисляет позицию из вектора индексов. Тогда скалярное положение будет смещением элемента с указанными индексами от начала блока хранилища массива.

В Guile, эта функция отображения ограниченна аффинным преобразованием *affine*: все функции отображения массивов в Guile могут быть записаны как `p = b + c[0]*i[0] + c[1]*i[1] + ... + c[n-1]*i[n-1]` где `i[k]` это `k`й индекс и `n` это ранг массива. Например, матрица размера 3x3 будет иметь `b == 0`, `c[0] == 3` и `c[1] == 1`. Когда вы перенесете эту матрицу (с помощью скажем `transpose-array`), вы получите массив, функция отображения которого будет `b == 0`, `c[0] == 1` и `c[1] == 3`.

Функция `scm_array_handle_dims` дает вамu (косвенный) доступ к коэффциэнтам `c[k]`.

Обратите внимание: нет функций для доступа к элементам массива символов. После того, как строковая реализация в Guile была изменена для использования Unicode, мы используем ее.

`scm_t_array_handle` [C Type]

This is a structure type that holds all information necessary to manage the reservation of arrays as explained above. Structures of this type must be allocated on the stack and must only be accessed by the functions listed below.

`void scm_array_get_handle` (*SCM array, scm_t_array_handle *handle*) [C Function]

Reserve *array*, which must be an array, and prepare *handle* to be used with the functions below. You must eventually call `scm_array_handle_release` on *handle*, and do this in a properly nested fashion, as explained above. The structure pointed to by *handle* does not need to be initialized before calling this function.

`void scm_array_handle_release` (*scm_t_array_handle *handle*) [C Function]

End the array reservation represented by *handle*. After a call to this function, *handle* might be used for another reservation.

`size_t scm_array_handle_rank` (*scm_t_array_handle *handle*) [C Function]

Return the rank of the array represented by *handle*.

`scm_t_array_dim` [C Type]

This structure type holds information about the layout of one dimension of an array. It includes the following fields:

`ssize_t lbnd`
`ssize_t ubnd`

The lower and upper bounds (both inclusive) of the permissible index range for the given dimension. Both values can be negative, but *lbnd* is always less than or equal to *ubnd*.

`ssize_t inc`

The distance from one element of this dimension to the next. Note, too, that this can be negative.

`const scm_t_array_dim * scm_array_handle_dims` [C Function]
(*scm_t_array_handle *handle*)

Return a pointer to a C vector of information about the dimensions of the array represented by *handle*. This pointer is valid as long as the array remains reserved. As explained above, the `scm_t_array_dim` structures returned by this function can be used calculate the position of an element in the storage block of the array from its indices.

This position can then be used as an index into the C array pointer returned by the various `scm_array_handle_<foo>_elements` functions, or with `scm_array_handle_ref` and `scm_array_handle_set`.

Here is how one can compute the position *pos* of an element given its indices in the vector *indices*:

```
ssize_t indices[RANK];
scm_t_array_dim *dims;
ssize_t pos;
size_t i;
```

```
pos = 0;
for (i = 0; i < RANK; i++)
  {
    if (indices[i] < dims[i].lbnd || indices[i] > dims[i].ubnd)
      out_of_range ();
    pos += (indices[i] - dims[i].lbnd) * dims[i].inc;
  }
```

`ssize_t scm_array_handle_pos` (*scm_t_array_handle *handle, SCM indices*) [C Function]

Compute the position corresponding to *indices*, a list of indices. The position is computed as described above for `scm_array_handle_dims`. The number of the indices and their range is checked and an appropriate error is signalled for invalid indices.

`SCM scm_array_handle_ref` (*scm_t_array_handle *handle, ssize_t pos*) [C Function]

Return the element at position *pos* in the storage block of the array represented by *handle*. Any kind of array is acceptable. No range checking is done on *pos*.

`void scm_array_handle_set` (*scm_t_array_handle *handle, ssize_t pos, SCM val*) [C Function]

Set the element at position *pos* in the storage block of the array represented by *handle* to *val*. Any kind of array is acceptable. No range checking is done on *pos*. An error is signalled when the array can not store *val*.

`const SCM * scm_array_handle_elements` (*scm_t_array_handle *handle*) [C Function]

Return a pointer to the elements of a ordinary array of general Scheme values (i.e., a non-uniform array) for reading. This pointer is valid as long as the array remains reserved.

`SCM * scm_array_handle_writable_elements` (*scm_t_array_handle *handle*) [C Function]

Like `scm_array_handle_elements`, but the pointer is good for reading and writing.

`const void * scm_array_handle_uniform_elements` (*scm_t_array_handle *handle*) [C Function]

Return a pointer to the elements of a uniform numeric array for reading. This pointer is valid as long as the array remains reserved. The size of each element is given by `scm_array_handle_uniform_element_size`.

`void * scm_array_handle_uniform_writable_elements` (*scm_t_array_handle *handle*) [C Function]

Like `scm_array_handle_uniform_elements`, but the pointer is good reading and writing.

`size_t scm_array_handle_uniform_element_size` (*scm_t_array_handle *handle*) [C Function]

Return the size of one element of the uniform numeric array represented by *handle*.

`const scm_t_uint8 * scm_array_handle_u8_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_int8 * scm_array_handle_s8_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_uint16 * scm_array_handle_u16_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_int16 * scm_array_handle_s16_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_uint32 * scm_array_handle_u32_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_int32 * scm_array_handle_s32_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_uint64 * scm_array_handle_u64_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_int64 * scm_array_handle_s64_elements` [C Function]
(*scm_t_array_handle *handle*)
`const float * scm_array_handle_f32_elements` (*scm_t_array_handle* [C Function]
**handle*)
`const double * scm_array_handle_f64_elements` [C Function]
(*scm_t_array_handle *handle*)
`const float * scm_array_handle_c32_elements` (*scm_t_array_handle* [C Function]
**handle*)
`const double * scm_array_handle_c64_elements` [C Function]
(*scm_t_array_handle *handle*)

Return a pointer to the elements of a uniform numeric array of the indicated kind for reading. This pointer is valid as long as the array remains reserved.

The pointers for `c32` and `c64` uniform numeric arrays point to pairs of floating point numbers. The even index holds the real part, the odd index the imaginary part of the complex number.

`scm_t_uint8 * scm_array_handle_u8_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int8 * scm_array_handle_s8_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_uint16 * scm_array_handle_u16_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int16 * scm_array_handle_s16_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_uint32 * scm_array_handle_u32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int32 * scm_array_handle_s32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_uint64 * scm_array_handle_u64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int64 * scm_array_handle_s64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

`float * scm_array_handle_f32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

`double * scm_array_handle_f64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

`float * scm_array_handle_c32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

`double * scm_array_handle_c64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

Like `scm_array_handle_<kind>_elements`, but the pointer is good for reading and writing.

`const scm_t_uint32 * scm_array_handle_bit_elements` [C Function]
(*scm_t_array_handle *handle*)

Return a pointer to the words that store the bits of the represented array, which must be a bit array.

Unlike other arrays, bit arrays have an additional offset that must be figured into index calculations. That offset is returned by `scm_array_handle_bit_elements_offset`.

To find a certain bit you first need to calculate its position as explained above for `scm_array_handle_dims` and then add the offset. This gives the absolute position of the bit, which is always a non-negative integer.

Each word of the bit array storage block contains exactly 32 bits, with the least significant bit in that word having the lowest absolute position number. The next word contains the next 32 bits.

Thus, the following code can be used to access a bit whose position according to `scm_array_handle_dims` is given in *pos*:

```
SCM bit_array;
scm_t_array_handle handle;
scm_t_uint32 *bits;
ssize_t pos;
size_t abs_pos;
size_t word_pos, mask;

scm_array_get_handle (&bit_array, &handle);
bits = scm_array_handle_bit_elements (&handle);

pos = ...
abs_pos = pos + scm_array_handle_bit_elements_offset (&handle);
word_pos = abs_pos / 32;
mask = 1L << (abs_pos % 32);

if (bits[word_pos] & mask)
  /* bit is set. */

scm_array_handle_release (&handle);
```

`scm_t_uint32 * scm_array_handle_bit_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

Like `scm_array_handle_bit_elements` but the pointer is good for reading and writing. You must take care not to modify bits outside of the allowed index range of the array, even for contiguous arrays.

6.6.14 VLists

Модуль `(ice-9 vlist)` обеспечивает реализацию структуры данных *VList*, разработанной Phil Bagwell в 2002. VLists это неизменяемые списки, которые могут содержать любой объект Scheme. Он улучшает стандартные связаные списки Scheme в нескольких областях:

- Произвольный доступ выполняется за постоянное время.
- Вычисление длины VList имеет логарифмическую сложность от числа элементов.
- VLists использует меньше места для хранения, чем стандартные списки.
- Элементы VList хранятся в смежных областях, что улучшает распределение памяти и приводит к более эффективному использованию аппаратных кэшей.

Идея VLists заключается в том, чтобы хранить элементы vlist в более крупных смежных блоках (реализованных здесь как векторы). Эти блоки связаны друг с другом с помощью указателя на следующий блок и смещения внутри этого блока. Размер этих блоков формирует геометрический ряд с коэффициэнтом роста: `block-growth-factor` (по умолчанию 2).

Структура VList также служит основой для хэш-списков на основе *хэш-список на осонове VList* или “vhashes”, неизменный тип словаря (см. Раздел 6.6.21 [VHashes], страница 249).

Однако текущая реализация `(ice-9 vlist)` имеет несколько примечательных недостатков:

- Он не является потокобезопасным. Хотя операции с vlists все *ссылочно прозрачны* (т.е. чисто функциональны), добавление элементов в vlist с помощью `vlist-cons` изменяет часть внутренней структуры, которая делает его потоко небезопасным. Это может быть исправлено, но это замедлит `vlist-cons`.
- `vlist-cons` всегда выделяет как минимум столько же памяти, сколько и `cons`. Опять же, Phil Bagwell описывает как это исправить, но это потребует настройки сборщика мусора способом который может быть не совсем полезен.
- `vlist-cons` это процедура Scheme компилируемая в байт-код, и оне не может конкурировать с простой Си реализациейand операции `cons`, и тем, что VM имеет специальную инструкцию `cons`.

Мы надеемся их решить в будущем.

Интерфейс программирования, экспортируемый `(ice-9 vlist)` определен ниже. Большая часть так же как SRFI-1 с добавлением префикса `vlist-` в имена функций.

`vlist?` *obj* [Scheme Procedure]

Return true if *obj* is a VList.

`vlist-null` [Scheme Variable]
The empty VList. Note that it's possible to create an empty VList not `eq?` to `vlist-null`; thus, callers should always use `vlist-null?` when testing whether a VList is empty.

`vlist-null?` *vlist* [Scheme Procedure]
Return true if *vlist* is empty.

`vlist-cons` *item vlist* [Scheme Procedure]
Return a new vlist with *item* as its head and *vlist* as its tail.

`vlist-head` *vlist* [Scheme Procedure]
Return the head of *vlist*.

`vlist-tail` *vlist* [Scheme Procedure]
Return the tail of *vlist*.

`block-growth-factor` [Scheme Variable]
A fluid that defines the growth factor of VList blocks, 2 by default.

The functions below provide the usual set of higher-level list operations.

`vlist-fold` *proc init vlist* [Scheme Procedure]
`vlist-fold-right` *proc init vlist* [Scheme Procedure]
Fold over *vlist*, calling *proc* for each element, as for SRFI-1 `fold` and `fold-right` (см. Раздел 7.5.3 [SRFI-1], страница 607).

`vlist-ref` *vlist index* [Scheme Procedure]
Return the element at index *index* in *vlist*. This is typically a constant-time operation.

`vlist-length` *vlist* [Scheme Procedure]
Return the length of *vlist*. This is typically logarithmic in the number of elements in *vlist*.

`vlist-reverse` *vlist* [Scheme Procedure]
Return a new *vlist* whose content are those of *vlist* in reverse order.

`vlist-map` *proc vlist* [Scheme Procedure]
Map *proc* over the elements of *vlist* and return a new vlist.

`vlist-for-each` *proc vlist* [Scheme Procedure]
Call *proc* on each element of *vlist*. The result is unspecified.

`vlist-drop` *vlist count* [Scheme Procedure]
Return a new vlist that does not contain the *count* first elements of *vlist*. This is typically a constant-time operation.

`vlist-take` *vlist count* [Scheme Procedure]
Return a new vlist that contains only the *count* first elements of *vlist*.

`vlist-filter` *pred vlist* [Scheme Procedure]
Return a new vlist containing all the elements from *vlist* that satisfy *pred*.

`vlist-delete` *x vlist* **[***equal?***]** [Scheme Procedure]
Return a new vlist corresponding to *vlist* without the elements *equal?* to *x*.

`vlist-unfold` *p f g seed* **[***tail-gen***]** [Scheme Procedure]
`vlist-unfold-right` *p f g seed* **[***tail***]** [Scheme Procedure]
Return a new vlist, as for SRFI-1 `unfold` and `unfold-right` (см. Раздел 7.5.3 [SRFI-1], страница 607).

`vlist-append` *vlist ...* [Scheme Procedure]
Append the given vlists and return the resulting vlist.

`list->vlist` *lst* [Scheme Procedure]
Return a new vlist whose contents correspond to *lst*.

`vlist->list` *vlist* [Scheme Procedure]
Return a new list whose contents match those of *vlist*.

6.6.15 Записи Обзор(Record Overview)

Записи(Records), также называемые *Структурами(structures)*, являются основным механизмом Scheme для определения новых типов. *Тип записи* определяет списко полей, из которых состоят экземпляры этого типа. Это похоже на структыры Си.

Исторически сложилось так, что Guile предлагае несколько различных способов определения типов записей и создания записи, предлагая разные функции и идя на разные компромисы. На протяжении многих лет каждый "Стандарт" также имеет свой собственный новый интерфес работы с записями. что привело к лабиринту API интерфейсов для работы с записями.

На самом высшем уровне стоит SRFI-9, высокоуровневый интерфес работы с записями, реализованный в большинстве реализаций Scheme (см. Раздел 6.6.16 [SRFI-9 Records], страница 231). Он определяет простую и эффективную синтаксическую абстракцию типо записей и связанных ними типов полей и функцй доступа к этим полям. SRFI-9 подходит для большинства применений, и рекомендуется для создания записей в Guile. Аналогичные высокоуровневые интерфесы включают SRFI-35 (см. Раздел 7.5.24 [SRFI-35], страница 648) и записи R6RS (см. Раздел 7.6.2.9 [rnrs records syntactic], страница 691).

Затем(исторически) появился Guile API "records" (см. Раздел 6.6.17 [Records], страница 234). Типы записей определенные таким образом, являются первыми классовыми объектами. Имеются интроспективные объекты позволяющие пользователям запрашивать список полей или значение определенного поля во время выполнения без предварительного знания его типа.

Наконец, общим знаменателем этих интерфесов является Guile API *Структуры(structure)* (см. Раздел 6.6.18 [Structures], страница 235). Структуры Guile это низкоуровневый строительный блок для всех других API работы с записями. Пользователям приложений как правило не нужно будет их использовать.

Записи, созданные с помощью этих APIs могут быть все сопоставлены с образцом используемым Guile стандартным поиском по шаблону. (см. Раздел 7.7 [Pattern Matching], страница 720).

6.6.16 Записи по спецификации (SRFI-9)

SRFI-9 стандартиризует синтаксис для определения новых типов записей и создания предикатов, конструкторов и функций доступа к полям(get, set). В Guile это рекомендуемый вариант создания новых типов записей (см. Раздел 6.6.15 [Record Overview], страница 230). Его можно использовать загрузив модуль:

```
(use-modules (srfi srfi-9))
```

`define-record-type` *type* [Scheme Syntax]
(*constructor fieldname* ...)
predicate
(*fieldname accessor* [*modifier*]) ...

Создает новый тип записи и создает для этого различные определения. Этот синтаксис работает только на верхнем уровне, и не подходит для вложения в какую либо другую форму.

type связь с символом типа записи, который соотвествует возвращаемому из функции `make-record-type`. *type* также предоставляет имя записи, согласно `record-type-name`.

constructor связь с функцией, которая будет вызываться как (***constructor*** `fieldval ...`), чтобы создать новую запись этого типа. Аргументы представляют собой начальные значения для полей, один аргумент для каждого поля, в том порядке, в котором они появляются в форме определения типа записи `define-record-type`.

Имена полей *fieldname*s представляют имена для полей записи, в соответствии с `record-type-fields` и т.д., и упоминаются в последующих формах доступа/изменения (accessor/modifier).

predicate связан с функцией, которая будет вызываться как: (***predicate*** `obj`). Он возвращает `#t` или `#f` в зависимости от того, является ли *obj* записью этого типа или нет.

Каждый *accessor* привязан к функции, которая должна быть вызвана как: (***accessor*** `record`), что бы получить соответствующее поле из *record*. Аналогично, каждый *modifier* привязан к функции вызываемой как: (***modifier*** `record val`), чтобы установить соответствующее значение поля в записи *record*.

Пример иллюстрирует типичное использование,

```
(define-record-type <employee>
  (make-employee name age salary)
  employee?
  (name    employee-name)
  (age     employee-age    set-employee-age!)
  (salary  employee-salary set-employee-salary!))
```

Он создает новый тип данных СОТРУДНИК employee с полями имени, возраста и заработной платы. Функции доступа создаются для каждого поля, но нет функции модификатора для имени (цель заключается в том, чтобы оно устнавливалось только при создании объекта employee) Объекты этого типа могут быть созданы и использованы, например:

```
<employee> ⇒ #<record-type <employee>>

(define fred (make-employee "Fred" 45 20000.00))

(employee? fred)        ⇒ #t
(employee-age fred)     ⇒ 45
(set-employee-salary! fred 25000.00)  ;; pay rise
```

Функции, созданные методом `define-record-type` являются обычными определениями верхнего уровня. Их можно переопределять или устанавливать по желанию, экспортировать из модуля и т.д.

Определение записей не на верхнем уровне.

Спецификация SRFI-9 явно запрещает определения записей в контексте отличном от верхнего уровня, например внутри тела лямбда`lambda` блока или внутри блока *let*. Однако рализация Guile не соблюдает это ограничение.

Пользовательская Печать

Вы можете использовать `set-record-type-printer!` для настройки поведения печати по умолчанию для записей. Это расширение Guile и не является частью SRFI-9. Оно расположено в модуле `(srfi srfi-9 gnu)`.

`set-record-type-printer!` *type proc* [Scheme Syntax]

Где *type* соответствует первому аргументу `define-record-type`, а *proc* процедура, принимающая два аргумента, запись(record) для печати и выходной порт.

Этот пример печатает имя сотрудника в скобках, например `[Fred]`.

```
(set-record-type-printer! <employee>
  (lambda (record port)
    (write-char #\[ port)
    (display (employee-name record) port)
    (write-char #\] port)))
```

Functional "Setters"

При написании кода в функциональном стиле желательно никогда не изменять содержимое записей. Для такого кода простой способ возврата новых экземпляров записей желательно выполнять на основе сущетствующих.

Модуль `(srfi srfi-9 gnu)` расширяет SRFI-9 возможностью возврата новой записи экземпляра на основе существующей, с только одним или несколькими значениями полей, измененными с помощью функции — *functional setters*. Во-первых, тип `define-immutable-record-type` работает как `define-record-type`, за исключением того что эти поля являются не изменяемыми, а функции изменния определяются как функциональные functional setters.

`define-immutable-record-type` *type* [Scheme Syntax]
(*constructor fieldname ...*)
predicate
(*fieldname accessor* [*modifier*]) ...

Определяет *type* как новый тип записи, подобно `define-record-type`. Однако, запись type станоиться не изменяемой *immutable* (записи не могут быть изменены даже с помощью функции `struct-set!`), и любой модификатор *modifier* определеяется как функциональный установщик(functional setter) —процедура которая возвращает новый экземпляр записи с указанным измененым значением поля и оставляет исходную запись неизменной.(См пример ниже.)

Кроме того, общие макросы `set-field` и `set-fields` могут применяться к любой записи SRFI-9.

`set-field` *record* (*field sub-fields ...*) *value* [Scheme Syntax]
Возвращает новую запись типа *record*, поля которой равны соответствующим полям записи *record*, кроме поля указанного *field*.

field must be the name of the getter corresponding to the field of *record* being "set". Subsequent *sub-fields* must be record getters designating sub-fields within that field value to be set (see example below.)

`set-fields` *record* ((*field sub-fields ...*) *value*) ... [Scheme Syntax]
Like `set-field`, but can be used to set more than one field at a time. This expands to code that is more efficient than a series of single `set-field` calls.

Чтобы проилюстрировать использование функциональных установщиков(functional setters) давайте предположим эти два определения типа записей:

```
(define-record-type <address>
  (address street city country)
  address?
  (street  address-street)
  (city    address-city)
  (country address-country))

(define-immutable-record-type <person>
  (person age email address)
  person?
  (age     person-age set-person-age)
  (email   person-email set-person-email)
  (address person-address set-person-address))
```

Во-первых, обратите внимание, что определение типа записи `<person>` вводит именнованные функциональные установщики. Они могут быть использованы следующим образом:

```
(define fsf-address
  (address "Franklin Street" "Boston" "USA"))

(define rms
```

```
  (person 30 "rms@gnu.org" fsf-address))

(and (equal? (set-person-age rms 60)
             (person 60 "rms@gnu.org" fsf-address))
     (= (person-age rms) 30))
⇒ #t
```

Здесь, исходная запись `<person>`, к которой привязана *rms*, остается неизменной.

Теперь предположим, что мы хотим изменить как улицу, так и возраст *rms*. Это может быть достигнуто использованием `set-fields`:

```
(set-fields rms
  ((person-age) 60)
  ((person-address address-street) "Temple Place"))
⇒ #<<person> age: 60 email: "rms@gnu.org"
  address: #<<address> street: "Temple Place" city: "Boston" country: "USA">>
```

Обратите внимание, как вышеизложенное изменило два поля *rms*, включая поле улицы `street` в его адресе `address`. Также обратите внимание, что `set-fields` работает одинаково для типов, определенных с помощью `define-record-type`.

6.6.17 Записи(Records)

Тип Запись *record type* это первый объект класса представляющего пользовательский тип данных. Запись *record* это экземпляр типа записи.

Обратите внимание, что во многих отношениях этот интерфейс слишком низкоуровневый для повседневного использования. Большинство примениней записи лучше обслживаются записями SRFI-9. См. Раздел 6.6.16 [SRFI-9 Records], страница 231.

`record?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a record of any type and `#f` otherwise.

Note that `record?` may be true of any Scheme value; there is no promise that records are disjoint with other Scheme types.

`make-record-type` *type-name field-names* [*print*] [Scheme Procedure]

Create and return a new *record-type descriptor*.

type-name is a string naming the type. Currently it's only used in the printed representation of records, and in diagnostics. *field-names* is a list of symbols naming the fields of a record of the type. Duplicates are not allowed among these symbols.

```
(make-record-type "employee" '(name age salary))
```

The optional *print* argument is a function used by `display`, `write`, etc, for printing a record of the new type. It's called as `(`*`print`* `record port)` and should look at *record* and write to *port*.

`record-constructor` *rtd* [*field-names*] [Scheme Procedure]

Return a procedure for constructing new members of the type represented by *rtd*. The returned procedure accepts exactly as many arguments as there are symbols in the given list, *field-names*; these are used, in order, as the initial values of those fields in a new record, which is returned by the constructor procedure. The values of any fields not named in that list are unspecified. The *field-names* argument defaults to the list

of field names in the call to `make-record-type` that created the type represented by *rtd*; if the *field-names* argument is provided, it is an error if it contains any duplicates or any symbols not in the default list.

`record-predicate` *rtd* [Scheme Procedure]
Return a procedure for testing membership in the type represented by *rtd*. The returned procedure accepts exactly one argument and returns a true value if the argument is a member of the indicated record type; it returns a false value otherwise.

`record-accessor` *rtd field-name* [Scheme Procedure]
Return a procedure for reading the value of a particular field of a member of the type represented by *rtd*. The returned procedure accepts exactly one argument which must be a record of the appropriate type; it returns the current value of the field named by the symbol *field-name* in that record. The symbol *field-name* must be a member of the list of field-names in the call to `make-record-type` that created the type represented by *rtd*.

`record-modifier` *rtd field-name* [Scheme Procedure]
Return a procedure for writing the value of a particular field of a member of the type represented by *rtd*. The returned procedure accepts exactly two arguments: first, a record of the appropriate type, and second, an arbitrary Scheme value; it modifies the field named by the symbol *field-name* in that record to contain the given value. The returned value of the modifier procedure is unspecified. The symbol *field-name* must be a member of the list of field-names in the call to `make-record-type` that created the type represented by *rtd*.

`record-type-descriptor` *record* [Scheme Procedure]
Return a record-type descriptor representing the type of the given record. That is, for example, if the returned descriptor were passed to `record-predicate`, the resulting predicate would return a true value when passed the given record. Note that it is not necessarily the case that the returned descriptor is the one that was passed to `record-constructor` in the call that created the constructor procedure that created the given record.

`record-type-name` *rtd* [Scheme Procedure]
Return the type-name associated with the type represented by rtd. The returned value is `eqv?` to the *type-name* argument given in the call to `make-record-type` that created the type represented by *rtd*.

`record-type-fields` *rtd* [Scheme Procedure]
Return a list of the symbols naming the fields in members of the type represented by *rtd*. The returned value is `equal?` to the field-names argument given in the call to `make-record-type` that created the type represented by *rtd*.

6.6.18 Структуры(Structures)

Структура*structure* - это первый класс, тип данных, который содержит значения Scheme или слова Си в полях, пронумерованных от нуля и выше. *vtable* - это структура, представляющая тип структуры, предоставляющей типы полей и разрешений доступа и дополнительной функции печати для `write` и т.д.

Структуры(Structures) представляют более низкий уровень абстракций по сравнению с записями (см. Раздел 6.6.17 [Records], страница 234). Как правило, когда вам нужно представлять структурированные данные, вы просто хотите использовать записи. Но иногда вам необходимо реализовать новые виды абстракций структурированных данных, и для этой цели структуры полезны. Действительно, записи в Guile реализованы со структурами.

6.6.18.1 Vtables

vtable это тип структуры, определяющий ее схему, и другую информацию. Фактически vtable сама является структурой, но в здесь не нужно беспокоиться об этом. (см. Раздел 6.6.18.3 [Vtable Contents], страница 238.)

`make-vtable` *fields* [*print*] [Scheme Procedure]

Create a new vtable.

fields is a string describing the fields in the structures to be created. Each field is represented by two characters, a type letter and a permissions letter, for example `"pw"`. The types are as follows.

- `p` – a Scheme value. “p” stands for “protected” meaning it’s protected against garbage collection.
- `u` – an arbitrary word of data (an `scm_t_bits`). At the Scheme level it’s read and written as an unsigned integer. “u” stands for “unboxed”, as it’s stored as a raw value without additional type annotations.

The second letter for each field is a permission code,

- `w` – writable, the field can be read and written.
- `r` – read-only, the field can be read but not written.
-

Here are some examples.

```
(make-vtable "pw")      ;; one writable field
(make-vtable "prpw")    ;; one read-only and one writable
(make-vtable "pwuwuw")  ;; one scheme and two unboxed
```

The optional *print* argument is a function called by `display` and `write` (etc) to give a printed representation of a structure created from this vtable. It’s called `(`*`print`* `struct port)` and should look at *struct* and write to *port*. The default print merely gives a form like ‘`#<struct ADDR:ADDR>`’ with a pair of machine addresses.

The following print function for example shows the two fields of its structure.

```
(make-vtable "prpw"
             (lambda (struct port)
               (format port "#<~a and ~a>"
                       (struct-ref struct 0)
                       (struct-ref struct 1))))
```

6.6.18.2 Structure Basics

This section describes the basic procedures for working with structures. `make-struct/no-tail` creates a structure, and `struct-ref` and `struct-set!` access its fields.

`make-struct/no-tail` *vtable init ...* [Scheme Procedure]

Create a new structure, with layout per the given *vtable* (см. Раздел 6.6.18.1 [Vtables], страница 236).

The optional *init...* arguments are initial values for the fields of the structure. This is the only way to put values in read-only fields. If there are fewer *init* arguments than fields then the defaults are `#f` for a Scheme field (type `p`) or 0 for an unboxed field (type `u`).

The name is a bit strange, we admit. The reason for it is that Guile used to have a `make-struct` that took an additional argument; while we deprecate that old interface, `make-struct/no-tail` is the new name for this functionality.

For example,

```
(define v (make-vtable "prpwpw"))
(define s (make-struct/no-tail v 123 "abc" 456))
(struct-ref s 0) ⇒ 123
(struct-ref s 1) ⇒ "abc"
```

`SCM scm_make_struct` (*SCM vtable, SCM tail_size, SCM init_list*) [C Function]

`SCM scm_c_make_struct` (*SCM vtable, SCM tail_size, SCM init, ...*) [C Function]

`SCM scm_c_make_structv` (*SCM vtable, SCM tail_size, size_t n_inits, scm_t_bits init*[]) [C Function]

There are a few ways to make structures from C. `scm_make_struct` takes a list, `scm_c_make_struct` takes variable arguments terminated with SCM_UNDEFINED, and `scm_c_make_structv` takes a packed array.

For all of these, *tail_size* should be zero (as a SCM value).

`struct?` *obj* [Scheme Procedure]

`scm_struct_p` (*obj*) [C Function]

Return `#t` if *obj* is a structure, or `#f` if not.

`struct-ref` *struct n* [Scheme Procedure]

`scm_struct_ref` (*struct, n*) [C Function]

Return the contents of field number *n* in *struct*. The first field is number 0.

An error is thrown if *n* is out of range.

`struct-set!` *struct n value* [Scheme Procedure]

`scm_struct_set_x` (*struct, n, value*) [C Function]

Set field number *n* in *struct* to *value*. The first field is number 0.

An error is thrown if *n* is out of range, or if the field cannot be written because it's `r` read-only.

Unboxed fields (those with type `u`) need to be accessed with special procedures.

`struct-ref/unboxed` *struct n* [Scheme Procedure]

`struct-set!/unboxed` *struct n value* [Scheme Procedure]

`scm_struct_ref_unboxed` (*struct, n*) [C Function]

`scm_struct_set_x_unboxed` (*struct, n, value*) [C Function]

Like `struct-ref` and `struct-set!`, except that these may only be used on unboxed fields. `struct-ref/unboxed` will always return a positive integer. Likewise,

`struct-set!/unboxed` takes an unsigned integer as the *value* argument, and will signal an error otherwise.

`struct-vtable` *struct* [Scheme Procedure]
`scm_struct_vtable` (*struct*) [C Function]
Return the vtable that describes *struct*.

The vtable is effectively the type of the structure. See Раздел 6.6.18.3 [Vtable Contents], страница 238, for more on vtables.

6.6.18.3 Vtable Contents

A vtable is itself a structure. It has a specific set of fields describing various aspects of its *instances*: the structures created from a vtable. Some of the fields are internal to Guile, some of them are part of the public interface, and there may be additional fields added on by the user.

Every vtable has a field for the layout of their instances, a field for the procedure used to print its instances, and a field for the name of the vtable itself. Access to the layout and printer is exposed directly via field indexes. Access to the vtable name is exposed via accessor procedures.

`vtable-index-layout` [Scheme Variable]
`scm_vtable_index_layout` [C Macro]
The field number of the layout specification in a vtable. The layout specification is a symbol like `pwpw` formed from the fields string passed to `make-vtable`, or created by `make-struct-layout` (см. Раздел 6.6.18.4 [Meta-Vtables], страница 239).

```
(define v (make-vtable "pwpw" 0))
(struct-ref v vtable-index-layout) ⇒ pwpw
```

This field is read-only, since the layout of structures using a vtable cannot be changed.

`vtable-index-printer` [Scheme Variable]
`scm_vtable_index_printer` [C Macro]
The field number of the printer function. This field contains `#f` if the default print function should be used.

```
(define (my-print-func struct port)
  ...)
(define v (make-vtable "pwpw" my-print-func))
(struct-ref v vtable-index-printer) ⇒ my-print-func
```

This field is writable, allowing the print function to be changed dynamically.

`struct-vtable-name` *vtable* [Scheme Procedure]
`set-struct-vtable-name!` *vtable name* [Scheme Procedure]
`scm_struct_vtable_name` (*vtable*) [C Function]
`scm_set_struct_vtable_name_x` (*vtable, name*) [C Function]
Get or set the name of *vtable*. *name* is a symbol and is used in the default print function when printing structures created from *vtable*.

```
(define v (make-vtable "pw"))
(set-struct-vtable-name! v 'my-name)
```

```
(define s (make-struct v 0))
(display s) ⊣ #<my-name b7ab3ae0:b7ab3730>
```

6.6.18.4 Meta-Vtables

As a structure, a vtable also has a vtable, which is also a structure. Structures, their vtables, the vtables of the vtables, and so on form a tree of structures. Making a new structure adds a leaf to the tree, and if that structure is a vtable, it may be used to create other leaves.

If you traverse up the tree of vtables, via calling `struct-vtable`, eventually you reach a root which is the vtable of itself:

```
scheme@(guile-user)> (current-module)
$1 = #<directory (guile-user) 221b090>
scheme@(guile-user)> (struct-vtable $1)
$2 = #<record-type module>
scheme@(guile-user)> (struct-vtable $2)
$3 = #<<standard-vtable> 12c30a0>
scheme@(guile-user)> (struct-vtable $3)
$4 = #<<standard-vtable> 12c3fa0>
scheme@(guile-user)> (struct-vtable $4)
$5 = #<<standard-vtable> 12c3fa0>
scheme@(guile-user)> <standard-vtable>
$6 = #<<standard-vtable> 12c3fa0>
```

In this example, we can say that `$1` is an instance of `$2`, `$2` is an instance of `$3`, `$3` is an instance of `$4`, and `$4`, strangely enough, is an instance of itself. The value bound to `$4` in this console session also bound to `<standard-vtable>` in the default environment.

`<standard-vtable>` [Scheme Variable]

A meta-vtable, useful for making new vtables.

All of these values are structures. All but `$1` are vtables. As `$2` is an instance of `$3`, and `$3` is a vtable, we can say that `$3` is a *meta-vtable*: a vtable that can create vtables.

With this definition, we can specify more precisely what a vtable is: a vtable is a structure made from a meta-vtable. Making a structure from a meta-vtable runs some special checks to ensure that the first field of the structure is a valid layout. Additionally, if these checks see that the layout of the child vtable contains all the required fields of a vtable, in the correct order, then the child vtable will also be a meta-table, inheriting a magical bit from the parent.

`struct-vtable?` *obj* [Scheme Procedure]
`scm_struct_vtable_p` (*obj*) [C Function]

Return `#t` if *obj* is a vtable structure: an instance of a meta-vtable.

`<standard-vtable>` is a root of the vtable tree. (Normally there is only one root in a given Guile process, but due to some legacy interfaces there may be more than one.)

The set of required fields of a vtable is the set of fields in the `<standard-vtable>`, and is bound to `standard-vtable-fields` in the default environment. It is possible to create

a meta-vtable that with additional fields in its layout, which can be used to create vtables with additional data:

```
scheme@(guile-user)> (struct-ref $3 vtable-index-layout)
$6 = pruhsruhpwphuhuhprprpw
scheme@(guile-user)> (struct-ref $4 vtable-index-layout)
$7 = pruhsruhpwphuhuh
scheme@(guile-user)> standard-vtable-fields
$8 = "pruhsruhpwphuhuh"
scheme@(guile-user)> (struct-ref $2 vtable-offset-user)
$9 = module
```

In this continuation of our earlier example, `$2` is a vtable that has extra fields, because its vtable, `$3`, was made from a meta-vtable with an extended layout. `vtable-offset-user` is a convenient definition that indicates the number of fields in `standard-vtable-fields`.

`standard-vtable-fields` [Scheme Variable]
: A string containing the ordered set of fields that a vtable must have.

`vtable-offset-user` [Scheme Variable]
: The first index in a vtable that is available for a user.

`make-struct-layout` *fields* [Scheme Procedure]
`scm_make_struct_layout` (*fields*) [C Function]
: Return a structure layout symbol, from a *fields* string. *fields* is as described under `make-vtable` (см. Раздел 6.6.18.1 [Vtables], страница 236). An invalid *fields* string is an error.

With these definitions, one can define `make-vtable` in this way:

```
(define* (make-vtable fields #:optional printer)
  (make-struct/no-tail <standard-vtable>
    (make-struct-layout fields)
    printer))
```

6.6.18.5 Vtable Example

Let us bring these points together with an example. Consider a simple object system with single inheritance. Objects will be normal structures, and classes will be vtables with three extra class fields: the name of the class, the parent class, and the list of fields.

So, first we need a meta-vtable that allocates instances with these extra class fields.

```
(define <class>
  (make-vtable
   (string-append standard-vtable-fields "pwpwpw")
   (lambda (x port)
     (format port "<<class> ~a>" (class-name x)))))

(define (class? x)
  (and (struct? x)
       (eq? (struct-vtable x) <class>)))
```

To make a structure with a specific meta-vtable, we will use `make-struct/no-tail`, passing it the computed instance layout and printer, as with `make-vtable`, and additionally the extra three class fields.

```
(define (make-class name parent fields)
  (let* ((fields (compute-fields parent fields))
         (layout (compute-layout fields)))
    (make-struct/no-tail <class>
      layout
      (lambda (x port)
        (print-instance x port))
      name
      parent
      fields)))
```

Instances will store their associated data in slots in the structure: as many slots as there are fields. The `compute-layout` procedure below can compute a layout, and `field-index` returns the slot corresponding to a field.

```
(define-syntax-rule (define-accessor name n)
  (define (name obj)
    (struct-ref obj n)))

;; Accessors for classes
(define-accessor class-name (+ vtable-offset-user 0))
(define-accessor class-parent (+ vtable-offset-user 1))
(define-accessor class-fields (+ vtable-offset-user 2))

(define (compute-fields parent fields)
  (if parent
      (append (class-fields parent) fields)
      fields))

(define (compute-layout fields)
  (make-struct-layout
   (string-concatenate (make-list (length fields) "pw"))))

(define (field-index class field)
  (list-index (class-fields class) field))

(define (print-instance x port)
  (format port "<~a" (class-name (struct-vtable x)))
  (for-each (lambda (field idx)
              (format port " ~a: ~a" field (struct-ref x idx)))
            (class-fields (struct-vtable x))
            (iota (length (class-fields (struct-vtable x)))))
  (format port ">"))
```

So, at this point we can actually make a few classes:

```
(define-syntax-rule (define-class name parent field ...)
```

```
    (define name (make-class 'name parent '(field ...))))

(define-class <surface> #f
  width height)

(define-class <window> <surface>
  x y)
```

And finally, make an instance:

```
(make-struct/no-tail <window> 400 300 10 20)
⇒ <<window> width: 400 height: 300 x: 10 y: 20>
```

And that's that. Note that there are many possible optimizations and feature enhancements that can be made to this object system, and the included GOOPS system does make most of them. For more simple use cases, the records facility is usually sufficient. But sometimes you need to make new kinds of data abstractions, and for that purpose, structs are here.

6.6.19 Тип Словарь(Dictionary Types)

Объект Словарь *dictionary* - это структура данных, используемая для индексирования информации способом определяемым пользователем. В стандартном Scheme, основными агрегирующими типами данных являются списки и вектора. Списки не индексируются вообще, а векторы индексируются только по числу (например `(vector-ref foo 5)`). Часто вам будет полезно проиндексировать ваши данные по другому типу, например, в библиотеке вы можете захотеть найти книгу по имени ее автора. Использование Словаря поможет вам организовать информацию таким образом.

Ассоциативный список *association list* (или кратко *alist*) представляет собой список пар ключ-значение. Каждая пара представляет собой единую величину или объект; `car` для пары - дает ключ, который используется для идентификации объекта, а `cdr` дает значение объекта.

Хеш-таблица *hash table* также позволяет вам индексировать объекты с помощью произвольных ключей, но таким образом, чтобы сделать поиск любого объекта чрезвычайно быстрым. Хрошо разработанная хеш-система делает хэш таблицы почти такими же быстрыми, как и обычные массивы или вектры ссылок.

Alists популярны среди программистов Lisp, потому что они используют только примитивные оперции языка:(lists, *car*, *cdr* и примитивы равенства). Никаких изменений в ядре языка. Поэтому, благодаря встроенным средствам управления списками Scheme очень удобно обрабатывать данные хранящиеся в ассоциативном списке Также, alist отличается высокой переносимостью и могут быть реализованы даже в самых минимальных системах Lisp.

Однако, alist неэффективен, особенно для хранения большого количества данных. Поскольку мы(разработчки) хотим, чтобы Guile был полезен для больших программных систем, а также для небольших, Guile предоставляет богатый набор инструментов для использования ассоциативных списков или хеш-таблиц.

6.6.20 Ассоциативные списки(Association Lists)

Ассоциативный список представляет собой обычную структуру данных, которая часто используется для реализации баз данных со структурой ключ-значение. Она состоит из списка полей, в которых каждая запись представляет собой пару. Ключ *key* каждой записи, содержиться в начале пары `car`, а значение *value* в конце каждой записи `cdr`.

```
ASSOCIATION LIST ::=  '( (KEY1 . VALUE1)
                         (KEY2 . VALUE2)
                         (KEY3 . VALUE3)
                         ...
                       )
```

Ассоциативный список также исвестен, как *alists*.

Структура списка ассоциаций - всего лишь один пример бесконечного числа возможных структур, которые могут быть построены с использованием пар и списков. Таким образом, ключами и значениями в списке ассоциаций можно манипулировать с использованием общих процедур управления списками: `cons`, `car`, `cdr`, `set-car!`, `set-cdr!` и так далее. Однако, поскольку списки ассоциаций настолько полезны, Guile также предоставляет конкретные процедуры для манипулирования ими.

6.6.20.1 Сравнение ключей (Alist Key Equality)

Все связанные с Guile процедуры работы с ассоциативными списками, представленны в трех вариантах, в зависимости от уровня равенства, необходимого для принятия решения о том, существует ли соответствующий ключ в списке ассоциаций совпадающий с ключем, который используется вызовом процедуры для идентификации требуемой записи.

- Процедуры с *assq* в их имени используется `eq?` для определения равенства с ключем.
- Процедуры с *assv* в их имени используют `eqv?` для определения равенстваа с ключем.
- Процедуры с *assoc* в их имени используется `equal?` для определения равенства с ключем.

`acons` - исключение, поскольку она используется для создания списков ассоциаций, которые не требуют что бы ключи записей были уникальны.

6.6.20.2 Добавление или Установка записей AList(Adding or Setting Alist Entries)

`acons` добавляет новую запись в списко ассоциаций и возвращает объединенный список ассоциаций. Объединенный список ассоциаций формируется путем вставки новой записи в начало alist указанного в вызове процедуры `acons`. Таким образом, указанный alist не изменяется, но его содержимое становиться разделяемым окончанием объединенного списка ассоциаций alist который возвращает `acons`.

При более общем использовании `acons`, переменная содержащая исходный список ассоциаций изменяется при создании общего alist:

```
(set! address-list (acons name address address-list))
```

В таких случаях не имеет значения, что старые и новые значения `address-list` разделены в их контексте, поскольку старое значение, как правило, больше не доступно, по независимой ссылке.

Обратите внимание, что `acons` добавляет указанную запись независимо от того, что alist уже может содержать запись с таким же ключем. Таким образом `acons` идеально подходит для создания alists, где нет требования уникальности ключей.

```
(set! task-list (acons 3 "pay gas bill" '()))
task-list
⇒
((3 . "pay gas bill"))

(set! task-list (acons 3 "tidy bedroom" task-list))
task-list
⇒
((3 . "tidy bedroom") (3 . "pay gas bill"))
```

`assq-set!`, `assv-set!` и `assoc-set!` используются для добавления или замены записи в ассоциативном списке в котором есть требование ункиальности ключей. Если список указанный как ассоциативный уже содержит запись, ключ которой совпадает с тем, который указан в вызове процедуры, существующая запись заменяется новой. Иначе новая запись заноситься в начало старого списка ассоциаций для создания объединенного alist. Во всех случаях, эти процедуры возвращают объединенный alist.

`assq-set!` и друзья *могут* разрушить структуру старого списка ассоциаций таким образом, чтобы существующая переменная была правильно обновлена без необходимости вызывать `set!` для возвращаемого значения:

```
address-list
⇒
(("mary" . "34 Elm Road") ("james" . "16 Bow Street"))

(assoc-set! address-list "james" "1a London Road")
⇒
(("mary" . "34 Elm Road") ("james" . "1a London Road"))

address-list
⇒
(("mary" . "34 Elm Road") ("james" . "1a London Road"))
```

Или они не могут:

```
(assoc-set! address-list "bob" "11 Newington Avenue")
⇒
(("bob" . "11 Newington Avenue") ("mary" . "34 Elm Road")
 ("james" . "1a London Road"))

address-list
⇒
(("mary" . "34 Elm Road") ("james" . "1a London Road"))
```

Единственный безопасный способ обновить переменную список ассоциаций при добавлении или замене записей, вызвать `set!` к переменной для возвращаемого значения:

```
(set! address-list
      (assoc-set! address-list "bob" "11 Newington Avenue"))
address-list
⇒
(("bob" . "11 Newington Avenue") ("mary" . "34 Elm Road")
 ("james" . "1a London Road"))
```

Из-за этого небольшого неудобства вам может быть удобнее использовать хеш-таблицы для хранения данных словаря. Если ваше приложение не будет изменять содержимое alist очень часто это может не иметь большого значения для вас.

Если вам необходимо сохранить старое значение списка ассоциаций в форме, независимой от списка, который будет результатом модификации `acons`, `assq-set!`, `assv-set!` или `assoc-set!`, используйте `list-copy`, чтобы скопировать старый список ассоциаций, прежде чем изменять его.

`acons` *key value alist* [Scheme Procedure]
`scm_acons` (*key, value, alist*) [C Function]

Add a new key-value pair to *alist*. A new pair is created whose car is *key* and whose cdr is *value*, and the pair is consed onto *alist*, and the new list is returned. This function is *not* destructive; *alist* is not modified.

`assq-set!` *alist key val* [Scheme Procedure]
`assv-set!` *alist key value* [Scheme Procedure]
`assoc-set!` *alist key value* [Scheme Procedure]
`scm_assq_set_x` (*alist, key, val*) [C Function]
`scm_assv_set_x` (*alist, key, val*) [C Function]
`scm_assoc_set_x` (*alist, key, val*) [C Function]

Reassociate *key* in *alist* with *value*: find any existing *alist* entry for *key* and associate it with the new *value*. If *alist* does not contain an entry for *key*, add a new one. Return the (possibly new) alist.

These functions do not attempt to verify the structure of *alist*, and so may cause unusual results if passed an object that is not an association list.

6.6.20.3 Получение записей Alist (Retrieving Alist Entries)

`assq`, `assv` и `assoc` ищут запись в alist для заданного ключа(key), и возвращают пару (***key*** **.** ***value***). `assq-ref`, `assv-ref` и `assoc-ref` делают аналогичный поиск, но возвращают только значение *value*.

`assq` *key alist* [Scheme Procedure]
`assv` *key alist* [Scheme Procedure]
`assoc` *key alist* [Scheme Procedure]
`scm_assq` (*key, alist*) [C Function]
`scm_assv` (*key, alist*) [C Function]
`scm_assoc` (*key, alist*) [C Function]

Return the first entry in *alist* with the given *key*. The return is the pair `(KEY . VALUE)` from *alist*. If there's no matching entry the return is `#f`.

`assq` compares keys with `eq?`, `assv` uses `eqv?` and `assoc` uses `equal?`. See also SRFI-1 which has an extended `assoc` (Раздел 7.5.3.9 [SRFI-1 Association Lists], страница 617).

`assq-ref` *alist key* [Scheme Procedure]
`assv-ref` *alist key* [Scheme Procedure]
`assoc-ref` *alist key* [Scheme Procedure]
`scm_assq_ref` (*alist, key*) [C Function]
`scm_assv_ref` (*alist, key*) [C Function]
`scm_assoc_ref` (*alist, key*) [C Function]

Return the value from the first entry in *alist* with the given *key*, or `#f` if there's no such entry.

`assq-ref` compares keys with `eq?`, `assv-ref` uses `eqv?` and `assoc-ref` uses `equal?`.

Notice these functions have the *key* argument last, like other `-ref` functions, but this is opposite to what `assq` etc above use.

When the return is `#f` it can be either *key* not found, or an entry which happens to have value `#f` in the `cdr`. Use `assq` etc above if you need to differentiate these cases.

6.6.20.4 Удаление записей Alist (Removing Alist Entries)

Чтобы удалить элемент из списка ассоциаций, ключ которого соответствует указанному ключу, ипсользуйте `assq-remove!`, `assv-remove!` или `assoc-remove!` (в зависимости, как обычно, от уровня требуемого равенства между указанным ключем и ключами из списка ассоциаций).

Как и с `assq-set!` и друзьями, указываемый alist может или не может быть разрушающе изменен, и единственный безопасный способ обновить переменную, содержащую alist использовать `set!` чтобы установить значение, которое возрващает `assq-remove!` и друзья.

```
address-list
⇒
(("bob" . "11 Newington Avenue") ("mary" . "34 Elm Road")
 ("james" . "1a London Road"))

(set! address-list (assoc-remove! address-list "mary"))
address-list
⇒
(("bob" . "11 Newington Avenue") ("james" . "1a London Road"))
```

Note that, when `assq/v/oc-remove!` is used to modify an association list that has been constructed only using the corresponding `assq/v/oc-set!`, there can be at most one matching entry in the alist, so the question of multiple entries being removed in one go does not arise. If `assq/v/oc-remove!` is applied to an association list that has been constructed using `acons`, or an `assq/v/oc-set!` with a different level of equality, or any mixture of these, it removes only the first matching entry from the alist, even if the alist might contain further matching entries. For example:

```
(define address-list '())
(set! address-list (assq-set! address-list "mary" "11 Elm Street"))
```

```
(set! address-list (assq-set! address-list "mary" "57 Pine Drive"))
address-list
⇒
(("mary" . "57 Pine Drive") ("mary" . "11 Elm Street"))

(set! address-list (assoc-remove! address-list "mary"))
address-list
⇒
(("mary" . "11 Elm Street"))
```

In this example, the two instances of the string "mary" are not the same when compared using `eq?`, so the two `assq-set!` calls add two distinct entries to `address-list`. When compared using `equal?`, both "mary"s in `address-list` are the same as the "mary" in the `assoc-remove!` call, but `assoc-remove!` stops after removing the first matching entry that it finds, and so one of the "mary" entries is left in place.

`assq-remove!` *alist key* [Scheme Procedure]
`assv-remove!` *alist key* [Scheme Procedure]
`assoc-remove!` *alist key* [Scheme Procedure]
`scm_assq_remove_x` (*alist, key*) [C Function]
`scm_assv_remove_x` (*alist, key*) [C Function]
`scm_assoc_remove_x` (*alist, key*) [C Function]

Delete the first entry in *alist* associated with *key*, and return the resulting alist.

6.6.20.5 Пренебрегающие ошибками функции Alist

`sloppy-assq`, `sloppy-assv` и `sloppy-assoc` ведут себя также как соответствующие не пренебрежительные`sloppy-` функцкии, за исключением того, что они возвращают `#f`, если указанный им список ассоциаций не является корректным, где не-`sloppy-` версии функций выбрасывают сигнал ошибки.

В частности, существуют два условия, при которых не пренебрегабщие`sloppy-` функции сигнализируют об ошибке, которую пренебрежительные `sloppy-` процедуры обрабатывают возвращая `#f`. Во-первых, если указанный alist в целом не является правильным списком:

```
(assoc "mary" '((1 . 2) ("key" . "door") . "open sesame"))
⇒
ERROR: In procedure assoc in expression (assoc "mary" (quote #)):
ERROR: Wrong type argument in position 2 (expecting
   association list): ((1 . 2) ("key" . "door") . "open sesame")

(sloppy-assoc "mary" '((1 . 2) ("key" . "door") . "open sesame"))
⇒
#f
```

Во-вторых, если одна из записей в указанном alist не является парой:

```
(assoc 2 '((1 . 1) 2 (3 . 9)))
⇒
ERROR: In procedure assoc in expression (assoc 2 (quote #)):
ERROR: Wrong type argument in position 2 (expecting
```

```
   association list): ((1 . 1) 2 (3 . 9))

(sloppy-assoc 2 '((1 . 1) 2 (3 . 9)))
⇒
#f
```

Unless you are explicitly working with badly formed association lists, it is much safer to use the non-`sloppy-` procedures, because they help to highlight coding and data errors that the `sloppy-` versions would silently cover up.

`sloppy-assq` *key alist* [Scheme Procedure]
`scm_sloppy_assq` (*key*, *alist*) [C Function]
Behaves like `assq` but does not do any error checking. Recommended only for use in Guile internals.

`sloppy-assv` *key alist* [Scheme Procedure]
`scm_sloppy_assv` (*key*, *alist*) [C Function]
Behaves like `assv` but does not do any error checking. Recommended only for use in Guile internals.

`sloppy-assoc` *key alist* [Scheme Procedure]
`scm_sloppy_assoc` (*key*, *alist*) [C Function]
Behaves like `assoc` but does not do any error checking. Recommended only for use in Guile internals.

6.6.20.6 Пример Alist

Вот более длинный пример того, как списки ассоциаций могут использоваться на практике.

```
(define capitals '(("New York" . "Albany")
                   ("Oregon"   . "Salem")
                   ("Florida"  . "Miami")))

;; What's the capital of Oregon?
(assoc "Oregon" capitals)       ⇒ ("Oregon" . "Salem")
(assoc-ref capitals "Oregon")   ⇒ "Salem"

;; We left out South Dakota.
(set! capitals
      (assoc-set! capitals "South Dakota" "Pierre"))
capitals
⇒ (("South Dakota" . "Pierre")
    ("New York" . "Albany")
    ("Oregon" . "Salem")
    ("Florida" . "Miami"))

;; And we got Florida wrong.
(set! capitals
      (assoc-set! capitals "Florida" "Tallahassee"))
```

```
capitals
⇒ (("South Dakota" . "Pierre")
    ("New York" . "Albany")
    ("Oregon" . "Salem")
    ("Florida" . "Tallahassee"))

;; After Oregon secedes, we can remove it.
(set! capitals
      (assoc-remove! capitals "Oregon"))
capitals
⇒ (("South Dakota" . "Pierre")
    ("New York" . "Albany")
    ("Florida" . "Tallahassee"))
```

6.6.21 Базирующийся на VList Хеш-Список или “VHashes”

Модуль `(ice-9 vlist)` предоставляет реализацию хеш списка на основе *VList* (см. Раздел 6.6.14 [VLists], страница 228). Базирующийся на VList хеш список, или *vhashes*, является типом неизменяемого словаря, аналогичный списку ассоциаций, который сопоставляет ключу*keys* - значение*values*. Однако, в отличие от списка ассоциаций, доступ к значению с учетом его ключа, как правило, является операцией с постоянным временем выполнения.

Программный интерфейс VHash `(ice-9 vlist)` в основном такой же, как у списка ассоциаций, из SRFI-1, с именами процедур предваряемыми `vhash-` вместо `alist-` (см. Раздел 7.5.3.9 [SRFI-1 Association Lists], страница 617).

Кроме того, vhashes можно манипулировать с помощью операций VList:

```
(vlist-head (vhash-consq 'a 1 vlist-null))
⇒ (a . 1)

(define vh1 (vhash-consq 'b 2 (vhash-consq 'a 1 vlist-null)))
(define vh2 (vhash-consq 'c 3 (vlist-tail vh1)))

(vhash-assq 'a vh2)
⇒ (a . 1)
(vhash-assq 'b vh2)
⇒ #f
(vhash-assq 'c vh2)
⇒ (c . 3)
(vlist->list vh2)
⇒ ((c . 3) (a . 1))
```

However, keep in mind that procedures that construct new VLists (`vlist-map`, `vlist-filter`, etc.) return raw VLists, not vhashes:

```
(define vh (alist->vhash '((a . 1) (b . 2) (c . 3)) hashq))
(vhash-assq 'a vh)
⇒ (a . 1)

(define vl
```

```
  ;; This will create a raw vlist.
  (vlist-filter (lambda (key+value) (odd? (cdr key+value))) vh))
(vhash-assq 'a vl)
⇒ ERROR: Wrong type argument in position 2

(vlist->list vl)
⇒ ((a . 1) (c . 3))
```

`vhash?` *obj* [Scheme Procedure]

Return true if *obj* is a vhash.

`vhash-cons` *key value vhash* [*hash-proc*] [Scheme Procedure]
`vhash-consq` *key value vhash* [Scheme Procedure]
`vhash-consv` *key value vhash* [Scheme Procedure]

Return a new hash list based on *vhash* where *key* is associated with *value*, using *hash-proc* to compute the hash of *key*. *vhash* must be either `vlist-null` or a vhash returned by a previous call to `vhash-cons`. *hash-proc* defaults to `hash` (см. Раздел 6.6.22.2 [Hash Table Reference], страница 253). With `vhash-consq`, the `hashq` hash function is used; with `vhash-consv` the `hashv` hash function is used.

All `vhash-cons` calls made to construct a vhash should use the same *hash-proc*. Failing to do that, the result is undefined.

`vhash-assoc` *key vhash* [*equal?* [*hash-proc*]] [Scheme Procedure]
`vhash-assq` *key vhash* [Scheme Procedure]
`vhash-assv` *key vhash* [Scheme Procedure]

Return the first key/value pair from *vhash* whose key is equal to *key* according to the *equal?* equality predicate (which defaults to `equal?`), and using *hash-proc* (which defaults to `hash`) to compute the hash of *key*. The second form uses `eq?` as the equality predicate and `hashq` as the hash function; the last form uses `eqv?` and `hashv`.

Note that it is important to consistently use the same hash function for *hash-proc* as was passed to `vhash-cons`. Failing to do that, the result is unpredictable.

`vhash-delete` *key vhash* [*equal?* [*hash-proc*]] [Scheme Procedure]
`vhash-delq` *key vhash* [Scheme Procedure]
`vhash-delv` *key vhash* [Scheme Procedure]

Remove all associations from *vhash* with *key*, comparing keys with *equal?* (which defaults to `equal?`), and computing the hash of *key* using *hash-proc* (which defaults to `hash`). The second form uses `eq?` as the equality predicate and `hashq` as the hash function; the last one uses `eqv?` and `hashv`.

Again the choice of *hash-proc* must be consistent with previous calls to `vhash-cons`.

`vhash-fold` *proc init vhash* [Scheme Procedure]
`vhash-fold-right` *proc init vhash* [Scheme Procedure]

Fold over the key/value elements of *vhash* in the given direction, with each call to *proc* having the form `(`*`proc`* `key value result)`, where *result* is the result of the previous call to *proc* and *init* the value of *result* for the first call to *proc*.

`vhash-fold*` *proc init key vhash* [*equal?* [*hash*]] [Scheme Procedure]
`vhash-foldq*` *proc init key vhash* [Scheme Procedure]
`vhash-foldv*` *proc init key vhash* [Scheme Procedure]

Fold over all the values associated with *key* in *vhash*, with each call to *proc* having the form `(proc value result)`, where *result* is the result of the previous call to *proc* and *init* the value of *result* for the first call to *proc*.

Keys in *vhash* are hashed using *hash* are compared using *equal?*. The second form uses `eq?` as the equality predicate and `hashq` as the hash function; the third one uses `eqv?` and `hashv`.

Example:

```
(define vh
  (alist->vhash '((a . 1) (a . 2) (z . 0) (a . 3))))

(vhash-fold* cons '() 'a vh)
⇒ (3 2 1)

(vhash-fold* cons '() 'z vh)
⇒ (0)
```

`alist->vhash` *alist* [*hash-proc*] [Scheme Procedure]

Return the vhash corresponding to *alist*, an association list, using *hash-proc* to compute key hashes. When omitted, *hash-proc* defaults to `hash`.

6.6.22 Хеш Таблица(Hash Tables)

Хеш-таблицы это словари, которые пердлагают функциональность аналогичную спискам ассоциций: предоставлять сопоставление от ключей к значениям. Разница в том, что спики ассоциаций требует время линейно зависящее от количества элементов в списке, тогда как хеш-таблицы обычно могут искать за постоянное время. Недостатком является то, что хеш-таблицы требуеют немного больше памяти, и что вы не можете испльзовать обычные процедуры работы со списками (см. Раздел 6.6.9 [Lists], страница 192) для работы с ними.

6.6.22.1 Пример Хеш-Таблицы

Для демонстрационных целей в этом разделе приведены несколько примеров использования некоторых процедур работы с хеш-таблицами, а также некоторое объяснение того, что они делают.

Сначала мы начинаем с создания новой хеш-таблицы с 31-им слотом, и заполняем ее двумя парами ключ/значение.

```
(define h (make-hash-table 31))

;; This is an opaque object
h
⇒
#<hash-table 0/31>

;; Inserting into a hash table can be done with hashq-set!
```

```
(hashq-set! h 'foo "bar")
⇒
"bar"

(hashq-set! h 'braz "zonk")
⇒
"zonk"

;; Or with hash-create-handle!
(hashq-create-handle! h 'frob #f)
⇒
(frob . #f)
```

Вы можете получить значение для данного ключа с помощью процедуры `hashq-ref`, но проблема с этой процедурой заключается в том, что вы не можете надежно определить, существует ли ключ в таблице. Причина в том, что процедура возвращает `#f` если ключа нет в таблице, но она будет возвращать тоже значение, если ключ находиться в таблице, а значение просто равно `#f`, что вы можете увидеть в следующих примерах.

```
(hashq-ref h 'foo)
⇒
"bar"

(hashq-ref h 'frob)
⇒
#f

(hashq-ref h 'not-there)
⇒
#f
```

Зачастую лучше использовать процедуру `hashq-get-handle`, которая делает различие между двумя случаями. Как и `assq`, эта процедура возвращает пару ключ/значение(key/value) при успешном завершении, и `#f` если ключ не найден.

```
(hashq-get-handle h 'foo)
⇒
(foo . "bar")

(hashq-get-handle h 'not-there)
⇒
#f
```

Интересные результаты могут быть вычислены с помощью `hash-fold` работающей с каждым элементом хеш-таблицы. В этом примере будет подсчитано общее количество элементов.

```
(hash-fold (lambda (key value seed) (+ 1 seed)) 0 h)
⇒
3
```

То же самое можно сделать с помощью процедуры `hash-count`, которая также может подсчитать количество элементов, соответствующих определенному предикату. Например, подсчитаем количество элементов со строковыми значениями:

```
(hash-count (lambda (key value) (string? value)) h)
⇒
2
```

Подсчет всех элементов - простая задача, использующая `const`:

```
(hash-count (const #t) h)
⇒
3
```

6.6.22.2 Справочник по Хеш-Таблицам

Подобно функциям списка ассоциаций, функции хеш-таблицы делятся на несколько разновидностей, согласно теста равенства, испльзуемого для сравнения ключей. Обычные хеш-`hash-` функции используют `equal?`, функции с префиксом `hashq-` используют `eq?`, с префиксом `hashv-` испльзуют `eqv?`, и функции `hashx-` испльзуют приложение предоставляющее для тест.

Простой вызов `make-hash-table` создает хеш-таблицу, подходящую для использования с любым набором функций, но очень важно, чтобы только один набор использовался с данной таблице, или результаты будут непредсказуемыми.

Хеш-таблицы реализуются как вектор, индексированный хеш-значением, сформированным из ключа, с ассоциативным списком пар ключ/значение для каждой ячейки в случае конфликта значений хеша ключа. Прямой доступ к парам в этих спиках обеспечивается с помощью `-handle-` функций.

Когда количество записей в хэш-таблице превышает пороговое значение, вектор увеличивается, и записи перезаписываются, чтобы списки ячеек не становились слишком длинными и замедлся доступ. Когда количество записей становиться ниже порогового значения, вектор сокращается, чтобы сэкономить место.

Для расширенных функций `hashx-`, приложение предоставляет хеш-*hash* функцию, вычисляющую целочисленный индекс, например `hashq` и т.д., а также функцию поиска по ключу подобную *assoc* для alist или как `assq` и т.д. (см. Раздел 6.6.20.3 [Retrieving Alist Entries], страница 245). Вот пример таких функций внедрения нечувствительных к регистру хэширования строковых ключей.

```
(use-modules (srfi srfi-1)
             (srfi srfi-13))

(define (my-hash str size)
  (remainder (string-hash-ci str) size))
(define (my-assoc str alist)
  (find (lambda (pair) (string-ci=? str (car pair))) alist))

(define my-table (make-hash-table))
(hashx-set! my-hash my-assoc my-table "foo" 123)

(hashx-ref my-hash my-assoc my-table "FOO")
```

⇒ 123

В `hashx-` цель функции *hash* состоит в том, чтобы рапределять ключи по вектору, поэтому списки ячеек вектора не становятся длинными. Но фактические значения произвольны, если они находятся в диапазоне от 0 до *size* − 1. Полезные функции для формирования хеш-значения, в дополнение к `hashq` и т.д, включают `symbol-hash` (см. Раздел 6.6.6.2 [Symbol Keys], страница 176), `string-hash` и `string-hash-ci` (см. Раздел 6.6.5.7 [String Comparison], страница 158), и `char-set-hash` (см. Раздел 6.6.4.1 [Character Set Predicates/Comparison], страница 143).

`make-hash-table` [*size*] [Scheme Procedure]

Create a new hash table object, with an optional minimum vector *size*.

Когда задан размер *size*, вектор таблицы будет расти и автоматически сокращаться, как описано выше, но размер будет минимум *size*. Если приложение, знает как много записей будет удерживать таблица, тогда оно может размер, чтобы избежать повторного рехеширования когда будут добавляться записи.

`alist->hash-table` *alist* [Scheme Procedure]
`alist->hashq-table` *alist* [Scheme Procedure]
`alist->hashv-table` *alist* [Scheme Procedure]
`alist->hashx-table` *hash assoc alist* [Scheme Procedure]

Convert *alist* into a hash table. When keys are repeated in *alist*, the leftmost association takes precedence.

```
(use-modules (ice-9 hash-table))
(alist->hash-table '((foo . 1) (bar . 2)))
```

When converting to an extended hash table, custom *hash* and *assoc* procedures must be provided.

```
(alist->hashx-table hash assoc '((foo . 1) (bar . 2)))
```

`hash-table?` *obj* [Scheme Procedure]
`scm_hash_table_p` (*obj*) [C Function]

Return `#t` if *obj* is a abstract hash table object.

`hash-clear!` *table* [Scheme Procedure]
`scm_hash_clear_x` (*table*) [C Function]

Remove all items from *table* (without triggering a resize).

`hash-ref` *table key* [*dflt*] [Scheme Procedure]
`hashq-ref` *table key* [*dflt*] [Scheme Procedure]
`hashv-ref` *table key* [*dflt*] [Scheme Procedure]
`hashx-ref` *hash assoc table key* [*dflt*] [Scheme Procedure]
`scm_hash_ref` (*table, key, dflt*) [C Function]
`scm_hashq_ref` (*table, key, dflt*) [C Function]
`scm_hashv_ref` (*table, key, dflt*) [C Function]
`scm_hashx_ref` (*hash, assoc, table, key, dflt*) [C Function]

Lookup *key* in the given hash *table*, and return the associated value. If *key* is not found, return *dflt*, or `#f` if *dflt* is not given.

`hash-set!` *table key val* [Scheme Procedure]
`hashq-set!` *table key val* [Scheme Procedure]
`hashv-set!` *table key val* [Scheme Procedure]
`hashx-set!` *hash assoc table key val* [Scheme Procedure]
`scm_hash_set_x` (*table, key, val*) [C Function]
`scm_hashq_set_x` (*table, key, val*) [C Function]
`scm_hashv_set_x` (*table, key, val*) [C Function]
`scm_hashx_set_x` (*hash, assoc, table, key, val*) [C Function]

Associate *val* with *key* in the given hash *table*. If *key* is already present then it's associated value is changed. If it's not present then a new entry is created.

`hash-remove!` *table key* [Scheme Procedure]
`hashq-remove!` *table key* [Scheme Procedure]
`hashv-remove!` *table key* [Scheme Procedure]
`hashx-remove!` *hash assoc table key* [Scheme Procedure]
`scm_hash_remove_x` (*table, key*) [C Function]
`scm_hashq_remove_x` (*table, key*) [C Function]
`scm_hashv_remove_x` (*table, key*) [C Function]
`scm_hashx_remove_x` (*hash, assoc, table, key*) [C Function]

Remove any association for *key* in the given hash *table*. If *key* is not in *table* then nothing is done.

`hash` *key size* [Scheme Procedure]
`hashq` *key size* [Scheme Procedure]
`hashv` *key size* [Scheme Procedure]
`scm_hash` (*key, size*) [C Function]
`scm_hashq` (*key, size*) [C Function]
`scm_hashv` (*key, size*) [C Function]

Return a hash value for *key*. This is a number in the range 0 to *size* − 1, which is suitable for use in a hash table of the given *size*.

Note that `hashq` and `hashv` may use internal addresses of objects, so if an object is garbage collected and re-created it can have a different hash value, even when the two are notionally `eq?`. For instance with symbols,

```
(hashq 'something 123)   ⇒ 19
(gc)
(hashq 'something 123)   ⇒ 62
```

In normal use this is not a problem, since an object entered into a hash table won't be garbage collected until removed. It's only if hashing calculations are somehow separated from normal references that its lifetime needs to be considered.

`hash-get-handle` *table key* [Scheme Procedure]
`hashq-get-handle` *table key* [Scheme Procedure]
`hashv-get-handle` *table key* [Scheme Procedure]
`hashx-get-handle` *hash assoc table key* [Scheme Procedure]
`scm_hash_get_handle` (*table, key*) [C Function]
`scm_hashq_get_handle` (*table, key*) [C Function]
`scm_hashv_get_handle` (*table, key*) [C Function]

`scm_hashx_get_handle` (*hash, assoc, table, key*) [C Function]
Return the (*key . value*) pair for *key* in the given hash *table*, or `#f` if *key* is not in *table*.

`hash-create-handle!` *table key init* [Scheme Procedure]
`hashq-create-handle!` *table key init* [Scheme Procedure]
`hashv-create-handle!` *table key init* [Scheme Procedure]
`hashx-create-handle!` *hash assoc table key init* [Scheme Procedure]
`scm_hash_create_handle_x` (*table, key, init*) [C Function]
`scm_hashq_create_handle_x` (*table, key, init*) [C Function]
`scm_hashv_create_handle_x` (*table, key, init*) [C Function]
`scm_hashx_create_handle_x` (*hash, assoc, table, key, init*) [C Function]
Return the (*key . value*) pair for *key* in the given hash *table*. If *key* is not in *table* then create an entry for it with *init* as the value, and return that pair.

`hash-map->list` *proc table* [Scheme Procedure]
`hash-for-each` *proc table* [Scheme Procedure]
`scm_hash_map_to_list` (*proc, table*) [C Function]
`scm_hash_for_each` (*proc, table*) [C Function]
Apply *proc* to the entries in the given hash *table*. Each call is (*proc key value*). `hash-map->list` returns a list of the results from these calls, `hash-for-each` discards the results and returns an unspecified value.

Calls are made over the table entries in an unspecified order, and for `hash-map->list` the order of the values in the returned list is unspecified. Results will be unpredictable if *table* is modified while iterating.

For example the following returns a new alist comprising all the entries from `mytable`, in no particular order.

```
(hash-map->list cons mytable)
```

`hash-for-each-handle` *proc table* [Scheme Procedure]
`scm_hash_for_each_handle` (*proc, table*) [C Function]
Apply *proc* to the entries in the given hash *table*. Each call is (*proc handle*), where *handle* is a (*key . value*) pair. Return an unspecified value.

`hash-for-each-handle` differs from `hash-for-each` only in the argument list of *proc*.

`hash-fold` *proc init table* [Scheme Procedure]
`scm_hash_fold` (*proc, init, table*) [C Function]
Accumulate a result by applying *proc* to the elements of the given hash *table*. Each call is (*proc key value prior-result*), where *key* and *value* are from the *table* and *prior-result* is the return from the previous *proc* call. For the first call, *prior-result* is the given *init* value.

Calls are made over the table entries in an unspecified order. Results will be unpredictable if *table* is modified while `hash-fold` is running.

For example, the following returns a count of how many keys in `mytable` are strings.

```
(hash-fold (lambda (key value prior)
             (if (string? key) (1+ prior) prior))
           0 mytable)
```

`hash-count` *pred table* [Scheme Procedure]
`scm_hash_count` (*pred, table*) [C Function]
Return the number of elements in the given hash *table* that cause `(`***pred key value***`)` to return true. To quickly determine the total number of elements, use `(const #t)` for *pred*.

6.6.23 Другие Типы

Процедуры описаны в их собственном разделе. См. Раздел 6.9 [Procedures], страница 262.

Переменные документируются как часть описания модульной системы Guile: См. Раздел 6.20.7 [Variables], страница 440.

См. Раздел 6.22 [Scheduling], страница 463, для обсуждения потоков, мьютексов и т.д.

Порты описаны в разделе I/O: см. Раздел 6.14 [Input and Output], страница 352.

Регулярные выражения описаны в их собственном разделе: см. Раздел 6.15 [Regular Expressions], страница 379.

Существует целый ряд дополнительных типов данных, задокументированных в этом руководстве, если вы думаете что здесь отсутствует ссылка, сообщите об ошибке.

6.7 Внешние Объекты

В этой главе содержижться справочная информация, касающаяся определения и работы с внешниии объектами(foreign object). См. См. Раздел 5.5 [Defining New Foreign Object Types], страница 79, для обучения создания и работы с внешними объектами.

`scm_t_struct_finalize` [C Type]
Этот тип функции возвращает `void` и принимает один `SCM` аргумент.

`SCM scm_make_foreign_object_type` (*SCM name, SCM slots,* [C Function]
scm_t_struct_finalize finalizer)
Создает новый тип внешнего объекта. *name* это символ, именующий этот тип. *slots* это список символов, каждый из которых именует поле в типе внешнего объекта. *finalizer* указывает финализатор, и может быть be `NULL`.

Мы рекомендуем, по возможности избегать финализаторов. См. Раздел 5.5.4 [Foreign Object Memory Management], страница 82. Финализаторы должны быть асинхронно-безопасными и потоко-безопасными. Еще раз, см. Раздел 5.5.4 [Foreign Object Memory Management], страница 82. Если вы внедряете Guile в приложение, которое не является потоко-безопасным, и вы определяете типы внешних объектов котрым требуется финализация, вы можете отключить автоматическую финализацию и организвать вызов `scm_manually_run_finalizers ()`.

`int scm_set_automatic_finalization_enabled` (*int enabled_p*) [C Function]
Включение или отключение автоматической финализации. По умолчанию, Guile организует вызов финализатора объекта автоматически, в отдельном потоке если это возможно. Передача нулевого значения для *enabled_p* отключает автоматическую финализацию для Guile в целом. Если вы отключите автоматическую фиализацию, вам придется вызывать переодически `scm_run_finalizers ()`.

В отличии от большинства других функций Guile, вы можете вызывать `scm_set_automatic_finalization_enabled` до того как Guile был инициализирован.

Возвращает предыдущий стату автоматической финализации.

`int scm_run_finalizers` (*void*) [C Function]

Вызывает любые ожидающие завершения финализаторы. Возвращает количество фигнализаторов, которые были вызваны. Эта функция должна вызываться, когда автоматическая финализация отключена, хотя она может вызываться и когда она включена.

`void scm_assert_foreign_object_type` (*SCM type, SCM val*) [C Function]

Когда *val* является внешним объектом заданного типа *type*, ничего не делает. Иначе, сообщает об ошибке.

`SCM scm_make_foreign_object_0` (*SCM type*) [C Function]
`SCM scm_make_foreign_object_1` (*SCM type, void *val0*) [C Function]
`SCM scm_make_foreign_object_2` (*SCM type, void *val0, void *val1*) [C Function]
`SCM scm_make_foreign_object_3` (*SCM type, void *val0, void *val1, void *val2*) [C Function]
`SCM scm_make_foreign_object_n` (*SCM type, size_t n, void *vals*[]) [C Function]

Создает новый внешний объект с типом *type* и инициализирует первые *n* полей указанными значениями, в зависимости от ситуации.

Число полей для объектов данного типа фиксируется при создании типа. Это ошибка если инициализаторов больше, чем полей в значении. Это отлично - дать меньше инициализаторов, чем необходимо; это удобно, когда некоторые поля являются не являются указателями, и их легче инициализировать с помощью установщиков(setters) описанных ниже.

`void* scm_foreign_object_ref` (*SCM obj, size_t n*); [C Function]
`scm_t_bits scm_foreign_object_unsigned_ref` (*SCM obj, size_t n*); [C Function]
`scm_t_signed_bits scm_foreign_object_signed_ref` (*SCM obj, size_t n*); [C Function]

Возвращает значение *n*го поля во внешнем объекте *obj*. Хранилище для хранения поля имеют ту же ширину что и значение `scm_t_bits` которое по меньшей мере равно ширине указателя. Различные варианты выбора указателя переносимым способом.

`void scm_foreign_object_set_x` (*SCM obj, size_t n, void *val*); [C Function]
`void scm_foreign_object_unsigned_set_x` (*SCM obj, size_t n, scm_t_bits val*); [C Function]
`void scm_foreign_object_signed_set_x` (*SCM obj, size_t n, scm_t_signed_bits val*); [C Function]

Устанавливает значение the *n*-го поля во внешнем объекте *obj* значением *val*, после возможного преобразование в значение `scm_t_bits`, если необходимо.

Можно также получить доступ к внешним объектам из Scheme. См. Раздел 5.5.5 [Foreign Objects and Scheme], страница 86, для некоторых примеров.

```
(use-modules (system foreign-object))
```

`make-foreign-object-type` *name slots* [*#:finalizer=#f*] [Scheme Procedure]
Создает новый тип внешнего объекта. Смотри документацию выше для `scm_make_foreign_object_type`; эти функции в точности эквивавлентны, за исключением того что финализатор подсоединяется к экземплярам (внутренняя деталь).

Получаемое значение это GOOPS класс. См. Глава 8 [GOOPS], страница 787, для получения дополнительной информации о классах в Guile.

`define-foreign-object-type` *name constructor* (*slot ...*) [Scheme Syntax]
[*#:finalizer=#f*]
Удобный макрос для определения типа, используя `make-foreign-object-type`, и связывае его с *name*. Констурктор будет связан с *constructor*, и функции получения значений(getters) будут связаны для каждого слота *slot...*.

6.8 Smobs

smob это "маленький объект(small object)". Прежде чем были введены внешние объекты в Guile 2.0.12 (см. Раздел 6.7 [Foreign Objects], страница 257), smobs были предпочтительным способом для Си кода для определения новых объектов Scheme. За исключением так называемых "применимых(applicable) SMOBs" обсуждаемых ниже, smobs теперь являются устаревшим интерфейсом и могут быть исключены из использования. См. ⟨undefined⟩ [Deprecation], страница ⟨undefined⟩. Новый код должен использовать интерфейс внешних объектов.

Этот раздел содержит справочную информацию касающуюся определения и работы со smobs. Похожее на учебник введение в smobs, можно уведеть в "Defining New Types (Smobs)" в предыдущей версии данного руководства.

`scm_t_bits scm_make_smob_type` (*const char *name, size_t size*) [Функция]
Эта функция добавляет новый тип smob, с именем *name*, с размером экземпляра *size*, в систему. Возвращаемое значение - это тег(метка) который используется для создания экземпляра данного типа.

Если *size* равно 0, функция по умолчнанию *free* ничего не делает.

Если *size* не равно 0, функция по умолчанию *free* освобождает блок памяти на который указывает `SCM_SMOB_DATA` с помощью `scm_gc_free`. Параметром *what* в вызове `scm_gc_free` будет *name*.

Значения по умолчанию предоставляются для функций *mark*, *free*, *print*, и *equalp*. Если вы хотите настроить любую из этих функций, вызов `scm_make_smob_type` должен немедленно сопровождаться вызовом одной или нескольких функций `scm_set_smob_mark`, `scm_set_smob_free`, `scm_set_smob_print`, и/или `scm_set_smob_equalp`.

`void scm_set_smob_free` (*scm_t_bits tc, size_t* (**free*) (*SCM obj*)) [C Function]
Эта функция устанавливает процедуру освобождения smob (иногда называемую финализатором a *finalizer*) для типае smob указываемого тегом *tc*. *tc* это тег, возвращаемый процедурой `scm_make_smob_type`.

Процедура *free* должна освобождать все ресурсы, которые напрямую связаны с экземпляром smob объекта *obj*. Она должна предполагать, что все значения `SCM`

на которые он ссылается, уже были осовбождены и таким образом, являются недействительными.

Она также не должна вызывать никакую функцию или макрос libguile, кроме `scm_gc_free`, `SCM_SMOB_FLAGS`, `SCM_SMOB_DATA`, `SCM_SMOB_DATA_2`, и `SCM_SMOB_DATA_3`.

процедура *free* должна возвращать 0.

Обратите внимание, что определение процедуры осовбождения не требуется, если ресурсы связанные с *obj* состоят только из памяти выделенной с помощью `scm_gc_malloc` или `scm_gc_malloc_pointerless`, потому что эта память автоматически возвращается сборщиком мусора, когда она больше не нужна (см. Раздел 6.19.2 [Memory Blocks], страница 426).

Функции освобождения Smob должны быть потоко-безопасными. См. Раздел 5.5.4 [Foreign Object Memory Management], страница 82, для обсуждения финализаторов и параллелелизма. Если вы встраиваете Guile в приложение которое не является потокобезопасным, и вы определяете smob типы которые нуждаются в финализации, вы можете отключить автоматическую финализацию, и организовать вызов `scm_manually_run_finalizers ()` самостоятельно. См. Раздел 6.7 [Foreign Objects], страница 257.

`void scm_set_smob_mark` (*scm_t_bits tc, SCM* (**mark*) (*SCM obj*)) [C Function]

Эта процедура устанавливает процедуру маркировки(отметки) smob для типа smob, указываемого тегом *tc*. *tc* это тег возвращаемый `scm_make_smob_type`.

Определение процедуры маркировки почти всегда НЕПРАВИЛЬНО! Это излишне, гораздо предпочтительнее выделять данные для smob с помощью функций `scm_gc_malloc` и `scm_gc_malloc_pointerless`, и позволять Сборщику мусора(GC) отслеживать указатели автоматически.

Любые процедуры маркировки, которые вы видите в настоящее время, почти навенряка датируются временем Guile 1.8, перед переходом на сборщик мусора Boehm-Demers-Weiser. Такие реализации smob следует изменить, чтобы просто использовать `scm_gc_malloc` и друзей, и забыть про свою функцию маркировки.

Если вы решили сохранить функцию маркировки, обратите внимание, что она вызвается для объектов, которые находятся в списке освобождения(free list). Пожалуйста почитайте комментарии от BDW GC's заголовка `gc/gc_mark.h`.

Процедура *mark* должна вызывать `scm_gc_mark` для каждого значения `SCM` которое напрямую ссылается на ыьщи экземпляр *obj*. Одно из этих значений `SCM` может быть возвращено из процедуры и Guile вызовет `scm_gc_mark` для него. Это можно использовать для того чтобы избежать глубокой рекурсиии для экземпляров smob которые формируют список.

Она не должна вызывать никаких функций или макросов libguile, кроме `scm_gc_mark`, `SCM_SMOB_FLAGS`, `SCM_SMOB_DATA`, `SCM_SMOB_DATA_2`, и `SCM_SMOB_DATA_3`.

`void scm_set_smob_print` (*scm_t_bits tc, int* (**print*) (*SCM obj, SCM port, scm_print_state* pstate*)) [C Function]

Эта функция устанавливает процедуру печати smob для данного типа smob определяемого тегом *tc*. *tc* это тег возвращаемый процедурой регистрации типа `scm_make_smob_type`.

Процедура печати *print* должна вывести текстовое представление экземпляра smob *obj* в порт *port*, используя инофрмацию в *pstate*.

Текстовое представление должно иметь форму `#<name ...>`. Это гарантирует, что команда чтения `read` не будет интерпретировать его как какое либо другое значение Scheme.

Часто лучше игнорировать *pstate* и просто печатать в порь *port* с помощью `scm_display`, `scm_write`, `scm_simple_format`, и `scm_puts`.

`void scm_set_smob_equalp` (*scm_t_bits tc, SCM* (**equalp*) (*SCM obj1, SCM obj2*)) [C Function]

Эта функция устанавливает предикат проверки равенства объектов для типа smob, указанного тегом *tc*. *tc* это тег возвращаемый `scm_make_smob_type`.

Процедура *equalp* должна возвращать `SCM_BOOL_T` когда объект *obj1* равен `equal?` объекту *obj2*. Иначе она должна вернуть `SCM_BOOL_F`. Оба объекта *obj1* и *obj2* являются экземплярами типа smob с тегом *tc*.

`void scm_assert_smob_type` (*scm_t_bits tag, SCM val*) [C Function]

Когда *val* является типом smob с указанным тегом *tag*, ничего не делает. Иначе, сигнализирует об ошибке.

`int SCM_SMOB_PREDICATE` (*scm_t_bits tag, SCM exp*) [C Macro]

Возращает истину если *exp* является экземпляром smob типа, указываемого тегом *tag*, или ложь в противном случае. Выражение *exp* может быть вычислено больше одного раза, поэтому оно не должно содержать никаких сторонних эффектов(типа вывода на дисплей).

`SCM scm_new_smob` (*scm_t_bits tag, void *data*) [C Function]

`SCM scm_new_double_smob` (*scm_t_bits tag, void *data, void *data2, void *data3*) [C Function]

Создает новый объект smob типа указываемого тэгом *tag* и данными smob - *data*, *data2*, и *data3*, соответственно.

Тэг *tag* это то что было возвращено `scm_make_smob_type`. Начальные значения *data*, *data2*, и *data3* имеют тип `scm_t_bits`; когда вы хотите использовать их как значения `SCM`, эти значения должны быть сначала преобразованы в `scm_t_bits` с помощью `SCM_UNPACK`.

Флаги экземпляра smob начинаются с нуля.

`scm_t_bits SCM_SMOB_FLAGS` (*SCM obj*) [C Macro]

Возвращает дополнительные 16 бит объекта smob *obj*. Для этих битов не определено никакого значения, вы можете их свободно использовать.

`scm_t_bits SCM_SET_SMOB_FLAGS` (*SCM obj, scm_t_bits flags*) [C Macro]

Устанавливает дополнительные 16 бит smob объекта *obj* в *flags*. Нет предопределенного назначения для этих бит, вы можете свободно их использовать.

`scm_t_bits SCM_SMOB_DATA` (*SCM obj*) [C Macro]

`scm_t_bits SCM_SMOB_DATA_2` (*SCM obj*) [C Macro]

`scm_t_bits SCM_SMOB_DATA_3 (SCM obj)` [C Macro]
Возвращает первое (второе, третье) непосредственное слово в smob объекте *obj* как значение `scm_t_bits`. Если слово содержит значение `SCM`, используйте вместо них `SCM_SMOB_OBJECT` (и т.д.).

`void SCM_SET_SMOB_DATA (SCM obj, scm_t_bits val)` [C Macro]
`void SCM_SET_SMOB_DATA_2 (SCM obj, scm_t_bits val)` [C Macro]
`void SCM_SET_SMOB_DATA_3 (SCM obj, scm_t_bits val)` [C Macro]
Устанавливает первое(второе, третье) непосредственное слово в smob объекте *obj* в значение *val*. Когда слово должно быть установлено значение `SCM`, используйте вместо них `SCM_SMOB_SET_OBJECT` (и т.д.).

`SCM SCM_SMOB_OBJECT (SCM obj)` [C Macro]
`SCM SCM_SMOB_OBJECT_2 (SCM obj)` [C Macro]
`SCM SCM_SMOB_OBJECT_3 (SCM obj)` [C Macro]
Возвращает первое (второе, третье) непосредственное слово smob объекта *obj* как значение `SCM`. Когда слово содержит значение `scm_t_bits` используйте вместо них `SCM_SMOB_DATA` (и т.д.).

`void SCM_SET_SMOB_OBJECT (SCM obj, SCM val)` [C Macro]
`void SCM_SET_SMOB_OBJECT_2 (SCM obj, SCM val)` [C Macro]
`void SCM_SET_SMOB_OBJECT_3 (SCM obj, SCM val)` [C Macro]
Устанавливает первое (второе, третье) непосредственное слово smob объекта *obj* в значение *val*. Когда слово должно быть установлено значением `scm_t_bits`, используйте вместо них `SCM_SMOB_SET_DATA` (и т.д.)

`SCM * SCM_SMOB_OBJECT_LOC (SCM obj)` [C Macro]
`SCM * SCM_SMOB_OBJECT_2_LOC (SCM obj)` [C Macro]
`SCM * SCM_SMOB_OBJECT_3_LOC (SCM obj)` [C Macro]
Возвращает указатель на первое (второе, третье) непосредственное слово в smob объекте *obj*. Заметьте что это указатель на `SCM`. Если вам нужно работать со значением `scm_t_bits`, используйте `SCM_PACK` и `SCM_UNPACK`, в зависимости от ситуации.

`SCM scm_markcdr (SCM x)` [Функция]
Отмечает ссылку в smob *x*, предполагая что первое слово данных в *x* содержит обычный объект Scheme, а *x* не сссылается ни на какие другие объекты. Эта функция просто возвращает первое слово данных *x*.

6.9 Процедуры

6.9.1 Lambda: Базовое создание Процедуры

Выражение `lambda` вычисляется создавая процедуру. Окружение, которое действует когда вычисляется `lambda` выражение, привязывается к недавно созданной процедуре, это отсылка к замыканию(*closure*) (см. Раздел 3.4 [About Closure], страница 27).

Когда процедура, созданная `lambda` вызывается с некоторыми фактическими аргументами, окружение заключенное в процедуру(привязанное к ней), рассширяется

связыванием переменных, названных в списке формальных аргументов, дополнительным пространством, где и сохраняются фактические аргументы. Затем тело `lambda` выражения последовательно вычисляется. Результат последнего выражения в теле процедуры является результатом вызова процедуры.

В следующих примерах показано, как процедуры могут быть созданы с использованием `lambda`, и что вы можете сделать с этими процедурами.

```
(lambda (x) (+ x x))         ⇒ a procedure
((lambda (x) (+ x x)) 4)     ⇒ 8
```

Тот факт, что среда, действующая при создании процедуры заключена в процедуре показан в следующем примере:

```
(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(define x 5)
(add4 6)                     ⇒ 10
```

`lambda` *formals body* [syntax]

formals должен быть формальным списком аргументов, как описано в следующей таблице.

(*variable1* ...)
: Процедура принимает фиксированное количество аргументов; когда процерура вызвается, аргументы будут сохранены во вновь созданном месте для формальных переменных.

variable
: Процедура принимает любое количество аргументов; когда вызывается процедура последовательность фактических аргументов преобразуется в список и хранится во вновь созданном месте для формальной переменной.

(*variable1 ... variablen . variablen+1*)
: Если точка предшествует последней переменной, про процедура принимает *n* или более переменных, где *n* это число формальных аргументов до точки. До точки должен быть хотя бы один аргумент. Первые *n* фактических аргументов будут сохранены в недавно выделенных местах для первых *n* формальных аргументов и последовательность оставшихся фактических аргументов преобразуется в список и сохраняется в месте для последнего формального аргумента. Если имеется ровно *n* фактических аргументов, в месте последнего формального аргумента храниться пустой список.

Спимок *variable* или *variablen+1* всегда создается и процедура при необходимости может изменить его. Это так, даже если процедура применяется с помощью `apply`, необходимая часть аргумента спика будет скопирована. См. (см. Раздел 6.18.4 [Procedures for On the Fly Evaluation], страница 409).

body — это последовательность выражений Scheme, которые вычисляются по порядку когда процедура вызывается.

6.9.2 Примитивные Процедуры

Процедуры написанные на Си могут быть зарегистрированы для использования из Scheme, если они принимают только аргументы типа `SCM` и возвращают значения `SCM`. `scm_c_define_gsubr`, вероятно, будет самым полезным механизмом, объединяющим процесс регистрации (`scm_c_make_gsubr`) и определения (`scm_define`).

`SCM scm_c_make_gsubr` (*const char *name, int req, int opt, int rst, fcn*) [Функция]
Регистрирует процедуру Си *fcn* как "subr" — примитивную подпрограмму, которая может быть вызвана из Scheme. Она будет связана с данным именем *name*, но без привязки к окружению в котором она создается. Аргументы *req*, *opt* и *rst* указазывают количество требуемых, необязательных и "последний(rest)" аргументов соответственно. Общее количество этих аргумегнтов дожно соответствовать фактическому числу аргументов в *fcn*, но не может превышать 10. Количсество аргументов rest должно быть 0 или 1. `scm_c_make_gsubr` возвращает значение типа `SCM`, который является "дескриптором(handle)" процедуры.

`SCM scm_c_define_gsubr` (*const char *name, int req, int opt, int rst, fcn*) [Функция]
Регистрирует процедуру Си *fcn*, как и для `scm_c_make_gsubr` выше, и дополнительно создает привязку верхенго уровня Scheme для процедуры в "текущем окружении(current environment)" используя `scm_define`. `scm_c_define_gsubr` возвращает дескриптор процедуры так же, как и `scm_c_make_gsubr`, который обычно не требуется.

6.9.3 Скомпилированные Процедуры

Стратегия вычисления, приведенная в Раздел 6.9.1 [Lambda], страница 262, описывает как процедуры интерпретируются(*interpreted*). Интерпретация работает непосредственно на расширенном исходном коде Scheme, рекурсивно вызывая вычислитель для получения значения вложенных выражений.

Однако большинство процедур скомпилировано. Это означает, что Guile сделал некоторую преварительную работу с процедурой, чтобы определить, что ей нужно будет делать каждй раз, когда процедура запускается. Скомпилированные процедуры выполняются быстрее, чем интерпретируемые процедуры.

Загрузка файлов - это нормальный способ создания скомпилированных процедур. Если Guile видит, что файл не скомпилирован, или что его скомпилированный файл устарел, он попытается скомпилировать файл когда он загружен, и сохранить результат на диск. Процедуры также могут быть скомпилированы во время выполнения. См. Раздел 6.18 [Read/Load/Eval/Compile], страница 404, для получения дополнительной информации о компиляции во время выполнения.

Скомпилированные процедуры, также известные как программы(*programs*), отвечают всем процедурам, котоыре оперируют процедурами. Кроме того, есть несколько дополнительных процедур доступа(accessors) низкоуровневых подробностей о программах.

Большинство людей не должны использовать процедуры, описанные в этом разделе, но это хорошо для их документирования. Сначала вам нужно включить соответствующий модуль:

```
(use-modules (system vm program))
```

`program?` *obj* [Scheme Procedure]
`scm_program_p` (*obj*) [C Function]
Возвращает `#t` если *obj* скомпилированная процедура, или `#f` если нет.

`program-code` *program* [Scheme Procedure]
`scm_program_code` (*program*) [C Function]
Возвращает адрес точки входа в программу, как целое число. Этот адрес в основном полезен для процедур отладчика: `(system vm debug)`.

`program-num-free-variable` *program* [Scheme Procedure]
`scm_program_num_free_variables` (*program*) [C Function]
Возвращает количество свободных переменных, захваченых этой программой.

`program-free-variable-ref` *program n* [Scheme Procedure]
`scm_program_free_variable-ref` (*program, n*) [C Function]
`program-free-variable-set!` *program n val* [Scheme Procedure]
`scm_program_free_variable_set_x` (*program, n, val*) [C Function]
Аксессоры для свободных переменных программы. Некоторые из захваченных значений фактически в переменных “boxes”. См. Раздел 9.3.4 [Variables and the VM], страница 847, для получения дополнительной информации.

Пользователи не должны изменять возвращаемое значение, если не считаю, что они действительно умны.

`program-bindings` *program* [Scheme Procedure]
`make-binding` *name boxed? index start end* [Scheme Procedure]
`binding:name` *binding* [Scheme Procedure]
`binding:boxed?` *binding* [Scheme Procedure]
`binding:index` *binding* [Scheme Procedure]
`binding:start` *binding* [Scheme Procedure]
`binding:end` *binding* [Scheme Procedure]
Анотация связанных переменных для программ, вместе со своими аксессорами.

Связанные перменные объявляют именами и расширяют блок локальных переменных. Лучший путь, чтобы посмотреть, что это такое, поиграть с ними в REPL. См. Раздел 9.3.2 [VM Concepts], страница 845, для получения дополнительной информации.

Обратите внимание, что информация о связанных переменных храниться в программе как часть(thunk) метаданных, поэтому включение его в генерируемый объектный код не налагает штрафа на производительность во время выполнения.

`program-sources` *program* [Scheme Procedure]
`source:addr` *source* [Scheme Procedure]
`source:line` *source* [Scheme Procedure]
`source:column` *source* [Scheme Procedure]
`source:file` *source* [Scheme Procedure]
Анотация местоположения исходного кода, вместе со своими аксессорами.

Информация о расположении исходного кода распространяется через компилятор и заканчивается сериализацией в метаданные программы. Эта информация

подкрепляется указателем смещения инструкции внутри объектного кода программы. В частности, это `ip` *just following* последующих инструкций, так что обратная трассировка может найти местоположение источника кода, который выполняется.

`program-arities` *program* [Scheme Procedure]
`scm_program_arities` (*program*) [C Function]
`program-arity` *program ip* [Scheme Procedure]
`arity:start` *arity* [Scheme Procedure]
`arity:end` *arity* [Scheme Procedure]
`arity:nreq` *arity* [Scheme Procedure]
`arity:nopt` *arity* [Scheme Procedure]
`arity:rest?` *arity* [Scheme Procedure]
`arity:kw` *arity* [Scheme Procedure]
`arity:allow-other-keys?` *arity* [Scheme Procedure]

Аксессоры для представления“арности(arity)” программы.

Обычный случай состоит в том, что процедура имеет одну арность. Например, `(lambda (x) x)`, принимает один требуемый аргумент, и все. Можно получить доступ к этому числу требуемых аргументов через `(arity:nreq (program-arities (lambda (x) x)))`. По аналогии, `arity:nopt` получает количество необязательных аргументов, и `arity:rest?` возвращает истиное значение, если процедура имеет аргумент rest.

`arity:kw` возвращает список пар `(`*`kw`* `.` *`idx`*`)`, если в процедуре есть аргументы ключевые слова. *idx* ссылается на локальную переменную *idx*th; См. Раздел 9.3.4 [Variables and the VM], страница 847, для получения дополнительной информации. Наконец `arity:allow-other-keys?` возвращает истинное значение если разрешены другие ключи. См. Раздел 6.9.4 [Optional Arguments], страница 267, для получения дополнительной информации.

Так что насчет `arity:start` и `arity:end`? Они возвращают диапазон байтов в байт-коде программы для которого арность действительна. Понимаете, процедура может на самом деле может импть больше чем одну арность. Вопрос только в том “что такое арность процедуры” действительно имеет смысл в определенных точках в программе, ограниченных этими значениями `arity:start` и `arity:end`.

`program-arguments-alist` *program* [*ip*] [Scheme Procedure]

Возвращает ассоциативный список, описывающий аргументы которые принимает программа(*program*) или `#f` если информация не может быть получена.

Ключами alist которые в настоящий момент определены являются: ‘required’, ‘optional’, ‘keyword’, ‘allow-other-keys?’, и ‘rest’. Например:

```
(program-arguments-alist
 (lambda* (a b #:optional c #:key (d 1) #:rest e)
   #t)) ⇒
((required . (a b))
 (optional . (c))
 (keyword . ((#:d . 4)))
```

```
(allow-other-keys? . #f)
(rest . d))
```

`program-lambda-list` *program* [*ip*] [Scheme Procedure]

Возвращает представление аргументов программы(*program*) как лямбда список, или `#f` если эта информация не доступна.

Например:

```
(program-lambda-list
 (lambda* (a b #:optional c #:key (d 1) #:rest e)
   #t)) ⇒
```

6.9.4 Необязательные Аргументы

Процедуры Scheme, определенные в R5RS, могут обрабатывать фиксированное количество фактических аргументов, или фиксированное количество фактических аргументов, за которыми следует произвольно много дополнительных аргументов. Написание процедур переменной арности(arity) может быть полезным, но, к сожалению, синтаксические средства для обработки списков аргументов различной длины немного не удобны. Возможно присвоение имен фиксированному числу аргументов, но остальные(не обязательные) аргументы могут упоминаться только как список значений. см.(см. Раздел 6.9.1 [Lambda], страница 262).

По этой причине, Guile предоставляет расширение `lambda`, `lambda*`, которое позволяет пользователю определять процедуры с необзательными аргументами и ключевыми словами. Кроме того, виртуальная машина Guile имеет низкоуровневую поддержку для передачи необязательных аргументов и ключевых слов. Вызов процедур с необязательными аргументами и ключевыми словами можно сделать дешевым, без выделения списка (rest) остаточных аргументов.

6.9.4.1 lambda* и define*.

`lambda*` подобен `lambda`, за исключением некоторых расширений допускающих необязательные аргументы и ключевые слова.

`lambda*` ([*var*...] [library syntax]
[*#:optional vardef*...]
[*#:key vardef*... [*#:allow-other-keys*]]
[*#:rest var* | . *var*])
body1 body2 ...

Создает процедуру, которая принимает необзательные аргументы и/или ключевые слова указанные в `#:optional` и `#:key`. Например,

```
(lambda* (a b #:optional c d . e) '())
```

это процедура с фиксированными аргументами *a* и *b*, необязательными аргументами *c* и *d*, и остальными аргументами доступными через *e*. Если необязательные аргументы опущены в вызове, их переменные связаны с занчением `#f`.

Аналогично, `define*` это синтаксический сахар для определения процедур с использованием `lambda*`.

`lambda*`так же может выполнять процедуры с аргументами ключевыми словами. Например, процедура определяется следующим образом:

```
(define* (sir-yes-sir #:key action how-high)
  (list action how-high))
```

может быть вызвана как `(sir-yes-sir #:action 'jump)`, `(sir-yes-sir #:how-high 13)`, `(sir-yes-sir #:action 'lay-down #:how-high 0)`, или просто just `(sir-yes-sir)`. Какие бы аргументы не указывались, ключевые слова привязаны к значениям(если не заданы то к значению `#f`).

Необязательные аргументы и ключевые слова могут также иметь значение по уполчанию, если их нет в вызове, предоставив дву-элементный список переменных и выражений. Например в

```
(define* (frob foo #:optional (bar 42) #:key (baz 73))
  (list foo bar baz))
```

foo это фиксированный аргумент, *bar* это не обязательный аргумент с значением по умолчанию 42, и baz это аргумент ключевое слово со значением по умолчанию 73. Выражения по умолчанию не вычисляются, если они не нужны, и до тех пор пока процедура не будет вызвана.

Обычно это ошибка, если в вызове есть ключевые слова, отличные от тех, которые указаны в `#:key`, но добавление `#:allow-other-keys` к определению(после объявлений аргументов ключевых слов) будут игнорироваться неизвестные ключевые слова.

Если вызов имеет ключевое слово, заданное дважды, используется последнее значение. Например,

```
(define* (flips #:key (heads 0) (tails 0))
  (display (list heads tails)))

(flips #:heads 37 #:tails 42 #:heads 99)
⊣ (99 42)
```

`#:rest` являтся синонимом для синтаксиса точечного аргумента останова. Списки аргументов `(a . b)` и `(a #:rest b)` эквивалентны во всех отношениях. Это предусмотрено для большего сходства с DSSSL, MIT-Scheme и Kawa в числе других, а также для переходящих из других диалектов Lisp.

Когда `#:key` используется вместе с аргументом rest, параметры ключевые слова в вызове все они остаются в списке rest. Это тоже самое, что и Common Lisp. Например,

```
((lambda* (#:key (x 0) #:allow-other-keys #:rest r)
   (display r))
 #:x 123 #:y 456)
⊣ (#:x 123 #:y 456)
```

`#:optional` и `#:key` последовательно устанавливает свои привязки слева на право. Это означает, что выражения по умолчанию могут ссылаться на предыдущие параметры, например.

```
(lambda* (start #:optional (end (+ 10 start)))
  (do ((i start (1+ i)))
```

```
    ((> i end))
  (display i)))
```

Исключением является правило останова слева на право. Если есть rest аргумент, он связывается после необязательных аргументов, но до аргументов ключевых слов.

6.9.4.2 (ice-9 optargs)

До Guile 2.0, `lambda*` и `define*` были реализованы с использованием макросов, которые обрабатывали список остальных аргументов. Это не было оптимальным, поскольку вызов процедуры с дополнительными аргументами при каждом вызове размещал список rest. Guile 2.0 улучшил эту ситуацию привнеся дополнительные аргументы и ключевые слова в ядро Guile.

Однако есть случаи. когда у вас есть список и вы хотите его разобрать для выделения необязательных аргументов и ключевых слов. Guile предоставляет в модуле `(ice-9 optargs)` некоторые макросы, чтобы помочь в решении этой задачи.

Синтаксис `let-optional` и `let-optional*` предназначен для разбора списка аргументов rest и присвоения имен различным элементам списка. `let-optional` связывает все переменные одновременно, а `let-optional*` связывает их последовательно, согласуясь с `let` и `let*` (см. Раздел 6.12.2 [Local Bindings], страница 314).

`let-optional` *rest-arg* (*binding ...*) *body1 body2 ...* [library syntax]
`let-optional*` *rest-arg* (*binding ...*) *body1 body2 ...* [library syntax]

Эти два макроса предоставляют вам интерфейс необязательных аргументов который является очень Схемным(*Schemey*) и не вводит никакого синтаксиса. Они совместимы с макросами scsh с теми же именами, но немного расширенне. Каждая из (*binding*) может быть одной из форм *var* или (*var default-value*). *rest-arg* должен быть последним аргументом процедуры которая использует эту форму. Элементы в *rest-arg* последовательно связываются к данным именам переменных. Когда *rest-arg* заканчивается, остальные переменные связываются либо со значением по умолчанию или с `#f`, если значение по умолчанию не указано. *rest-arg* остается связанным связанным с тем, что останется от *rest-arg*.

после привязки переменных вычисляются выражения *body1 body2 ...* в порядке их следования.

Аналогично, `let-keywords` и `let-keywords*` извлекают значения из списка аргументов ключевых слов, привязывая локальные переменные к этим значениям или значениям по умолчанию.

`let-keywords` *args allow-other-keys?* (*binding ...*) *body1 body2 ...* [library syntax]
`let-keywords*` *args allow-other-keys?* (*binding ...*) *body1 body2 ...* [library syntax]

args вычисляется и должен быть доступен как список формы `(#:keyword1 value1 #:keyword2 value2 ...)`. *binding* переменные и выражения по умолчанию, с переменными устанавливаемыми(по именам) из значений ключевых слов. *body1 body2 ...* формы вычисляемые и значение последнего выражения является результатом. An example will make the syntax clearest,

```
(define args '(#:xyzzy "hello" #:foo "world"))
```

```
(let-keywords args #t
      ((foo  "default for foo")
       (bar  (string-append "default" "for" "bar")))
  (display foo)
  (display ", ")
  (display bar))
⊣ world, defaultforbar
```

Связываение для `foo` приходит от ключевого слова `#:foo` в `args`. Но связваение для `bar` является значение по умолчанию в `let-keywords`, поскольку в args нет ключа `#:bar`.

allow-other-keys? вычисляет и управляет разрешением на использование неизвестных ключевых слов в списке аргументов *args*. Когда она истинно неизветные ключи игнорируются (такие как, #:xyzzy в примере), когда `#f` генерируется ошибка для неизвестного ключевого слова.

`(ice-9 optargs)` также предоставляет еще несколько синтксического сахара `define*`, которые не так полезны с современным кодированием Guile, но все же поддерживаются: `define*-public` это `lambda*` версия `define-public`; `defmacro*` и `defmacro*-public` используются для определения макросов с улучшеной возможностью обработки списка аргументов. `-public` версии не только определяют процедуры и макросы, но также экспортировать их из текущего модуля.

`define*-public` *formals body1 body2 ...* [library syntax]
это смесь `define*` и `define-public`.

`defmacro*` *name formals body1 body2 ...* [library syntax]
`defmacro*-public` *name formals body1 body2 ...* [library syntax]
Это тоже , как `defmacro` и `defmacro-public` за исключением того, что они берут `lambda*`-стиль расширения списка параметров, где, `#:optional`, `#:key`, `#:allow-other-keys` и `#:rest` разрешены с использованием обычной семантики. Вот пример макроса с необязательным аргументом:

```
(defmacro* transmogrify (a #:optional b)
  (a 1))
```

6.9.5 Case-lambda

R5RS определяет rest аргументы как действительно полезные и очень общие, но они часть не самые подходящие или эффективные средства для выполнения работы. Например, `lambda*` намного лучшее решение проблемы с необязательными аргументами, чем `lambda` с аргументами rest.

Аналогично, `case-lambda` хорошо работает, когда вы хотите, чтобы одна процедура выполняла двойную(или тройную, или ...) нагрузку, без штрафа на создание списка остаточных(rest) аргументов.

Например:

```
(define (make-accum n)
  (case-lambda
    (() n)
```

```
    ((m) (set! n (+ n m)) n)))

(define a (make-accum 20))
(a) ⇒ 20
(a 10) ⇒ 30
(a) ⇒ 30
```

Значение, возвращаемое формой `case-lambda`, это процедура, которая соответствует числу фактических аргументов, а не формальных, в различных вариантах использования вызова. Первое совпадение, соответствующего значения из фактического списка параметров связывается с именем переменной в предложениях и теле вычисляемого предложения. Если никакая позиция не соответствует, выдается сообщение об ошибке.

Синткасис формы `case-lambda` определяется следующей EBNF граматикой. Формальные(*Formals*) означает формальный список аргументов подобный используемому с `lambda` (см. Раздел 6.9.1 [Lambda], страница 262).

```
<case-lambda>
   --> (case-lambda <case-lambda-clause>*)
   --> (case-lambda <docstring> <case-lambda-clause>*)
<case-lambda-clause>
   --> (<formals> <definition-or-command>*)
<formals>
   --> (<identifier>*)
     | (<identifier>* . <identifier>)
     | <identifier>
```

Остаточный(Rest) список может быть полезен дла использования с `case-lambda`:

```
(define plus
  (case-lambda
    "Return the sum of all arguments."
    (() 0)
    ((a) a)
    ((a b) (+ a b))
    ((a b . rest) (apply plus (+ a b) rest))))
(plus 1 2 3) ⇒ 6
```

Кроме того, для полноты. Guile определяет `case-lambda*` также как `case-lambda`, за исключением предложения `lambda*`. Предложение `case-lambda*` соответствует если аргументы заполняют требуемые аргументы, но не более необязательный и/или остаточных аргументов.

Аргументы ключевые слова возможно использовать с `case-lambda*` также, но оне не вносят вклад в поведение вычисляющее “совпадение”, и их взаимодействие с требуемыми, необязательными и остаточными аргументами может быть удивительным.

Для целей `case-lambda*` (и `case-lambda`, в качестве особого случая), предложение соответствует(*matches*) если у него достаточно требуемых аргументов, и не слишком много позиционных аргументов. Необходимыми аргументами являются любые аргументы перед аргументами `#:optional`, `#:key`, и `#:rest`. Позиционными(*Positional*) аргументами являются необходимые аргументы, а также необязательные аргументы.

В отсутствии аргументов `#:key` или `#:rest`, легко увидеть, как может быть слишком моного позиционных аргументов: вы передаете 5 аргументов функции, которая принимает только 4 аргумента, включая необязательные аргументы. Если есть аргумент `#:rest`, никогда не может быть слишком много позиционных аргументов: любое приложение с достаточным числом необходимых аргументов для предложения будет соответствовать этому предложению, даже если есть аргументы `#:key`.

В противном случае для приложений к предложению с аргументами `#:key` (и без аргумента `#:rest`), предложение будет соответствовать только там, где достаточно аргументов, и если следующий аргумент после связывание требуемых и необзяательных аргументов, если они есть, является ключевым словом. Для соображения эффективности, Guile в настоящее время не может включать аргументы ключевые слова в алгоритм поиска соответствия. Предложения проверяют совпадения только с позиционными аргументами, и не сравнивают доступные ключевые слова с доступным набором аргументов ключевых слов, который имеет функция.

Ниже приводятся некоторые примеры.

```
(define f
  (case-lambda*
    ((a #:optional b) 'clause-1)
    ((a #:optional b #:key c) 'clause-2)
    ((a #:key d) 'clause-3)
    ((#:key e #:rest f) 'clause-4)))

(f) ⇒ clause-4
(f 1) ⇒ clause-1
(f) ⇒ clause-4
(f #:e 10) clause-1
(f 1 #:foo) clause-1
(f 1 #:c 2) clause-2
(f #:a #:b #:c #:d #:e) clause-4

;; clause-2 will match anything that clause-3 would match.
(f 1 #:d 2) ⇒ error: bad keyword args in clause 2
```

Не забывайте, что предложения сопоставляются по порядку, и первое предложение соответствия будет принято. Это может привести к тому, что кючевое слово привязывается к требуемому аргументу, как в случае `f #:e 10`.

6.9.6 Функции более высокого порядка

В качестве функционального языка программирования, Scheme позволяет определять функции более высокого порядка(*higher-order functions*), т.е. функции, которые принимают функци как аргументы и/или возвращают функции. Утилиты для получения процедур из других процедур и приводятся ниже.

`const` *value* [Scheme Procedure]

Возвращает процедуру, которая принимает любое количество аргументов и возвращает *value*.

```
(procedure? (const 3))        ⇒ #t
```

```
((const 'hello))              ⇒ hello
((const 'hello) 'world)       ⇒ hello
```

`negate` *proc* [Scheme Procedure]

Возвращает процедуру с той же арностью, что и *proc*, которая возвращает отритцание результата *proc*, т.е `not` *proc*.

```
(procedure? (negate number?)) ⇒ #t
((negate odd?) 2)             ⇒ #t
((negate real?) 'dream)       ⇒ #t
((negate string-prefix?) "GNU" "GNU Guile")
                              ⇒ #f
(filter (negate number?) '(a 2 "b"))
                              ⇒ (a "b")
```

`compose` *proc1 proc2 ...* [Scheme Procedure]

Составляет *proc1* с процедурами *proc2 ...* , так что аргумент *proc* применяется сначала последней процедурой, затем результат передается в предшестующю процедуру и последней обрабатывает результат *proc1*, и возвращает полученную процедуру. Данные процедуры должны быть совместимой арности.

```
(procedure? (compose 1+ 1-)) ⇒ #t
((compose sqrt 1+ 1+) 2)     ⇒ 2.0
((compose 1+ sqrt) 3)        ⇒ 2.73205080756888
(eq? (compose 1+) 1+)        ⇒ #t

((compose zip unzip2) '((1 2) (a b)))
                             ⇒ ((1 2) (a b))
```

`identity` *x* [Scheme Procedure]

Возвращает X.

`and=>` *value proc* [Scheme Procedure]

Когда *value* это `#f`, возвращается `#f`. Иначе, возвращается (*proc value*).

6.9.7 Свойства Процедур и Мета-Информация

В дополнение к информации, которая строго необходима для запуска, процедуры могут иметь другую связанню с ними информацию. Например, имя процедуры - это информация не для процедуры, а о процедуре. Эту мета-информацию можно получить доступ через интерфейс свойств процедур.

Первая группа процедур в этом мета-интерфейсе является предикатом для проверки того, является ли объект Scheme процедурой или специальной процедурой, соответственно. `procedure?` наиболее общий предикат, который возвращает `#t` для любой процедуры.

`procedure?` *obj* [Scheme Procedure]
`scm_procedure_p` (*obj*) [C Function]

Возвращает `#t` если *obj* это процедура.

`thunk?` *obj* [Scheme Procedure]
`scm_thunk_p` (*obj*) [C Function]
Возвращает `#t` если *obj* это thunk—процедура, которая не принимает аргументов.

Свойства процедуры — это общие свойства, связанные с процедурами. Это могут быть название процедуры или другая соответствующая информация, например подсказки отладки.

`procedure-name` *proc* [Scheme Procedure]
`scm_procedure_name` (*proc*) [C Function]
Возвращает имя процедуры *proc*

`procedure-source` *proc* [Scheme Procedure]
`scm_procedure_source` (*proc*) [C Function]
Возвращает исходный код процедуры *proc*. Возвращает `#f` если исходный код недоступен.

`procedure-properties` *proc* [Scheme Procedure]
`scm_procedure_properties` (*proc*) [C Function]
Возвращает свойства связанные с процедурой *proc*, как ассоциативный список.

`procedure-property` *proc key* [Scheme Procedure]
`scm_procedure_property` (*proc, key*) [C Function]
Возвращает свойство процедуры *proc* с именем *key*.

`set-procedure-properties!` *proc alist* [Scheme Procedure]
`scm_set_procedure_properties_x` (*proc, alist*) [C Function]
Устанавливает список свойств *proc* указанным ассоциативным списком *alist*.

`set-procedure-property!` *proc key value* [Scheme Procedure]
`scm_set_procedure_property_x` (*proc, key, value*) [C Function]
В списке свойств *proc*, устанавливает свойство с именем *key* в значение *value*.

Документацию для процедуры можно получить с помощью процедуры `procedure-documentation`.

`procedure-documentation` *proc* [Scheme Procedure]
`scm_procedure_documentation` (*proc*) [C Function]
Возвращает строку документации, связанную с `proc`. По соглашению, если процедура содержит более одного выражения, а первое выражение является строковой константой, предполагается, что эта строка содержит документацию для этой процедуры.

6.9.8 Процедуры с установщиками

Процедура с установщиком(*procedure with setter*) это особый вид процедуры, которая обычно ведет себя как любая процедура доступа, которая является процедурой обращающейся к структуре данных. Разница состоит в том, что эта процедура имеет так называемый установщик(*setter*), который является процедурой для сохранения чего-либо в структуру данных.

Процедуры с установщиком обрабатываются специально, когда процедура появляется в специальной форме `set!` (REFFIXME). Как это работает, лучше всего показано на примере.

Предположим, что у нас есть процедура, называемая `foo-ref`, которая принимает два аргумента, значение типа `foo` и целое число. Процедура возвращает значение сохраненное в указанном индексе объекта `foo`. Пусть `f` переменная содержащая такую структуру данных `foo`.[4]

```
(foo-ref f 0)       ⇒ bar
(foo-ref f 1)       ⇒ braz
```

Также предположим, что соответствующая процедура установщик `foo-set!` существует.

```
(foo-set! f 0 'bla)
(foo-ref f 0)       ⇒ bla
```

Теперь мы можем создать новую процедуру, называемую `foo`, которая является процедурой с установщиком, вызывав `make-procedure-with-setter` с процедурами доступа и устновки `foo-ref` и `foo-set!`. Назовем эту новую процедуру `foo`.

```
(define foo (make-procedure-with-setter foo-ref foo-set!))
```

Теперь `foo` может использоваться для чтения из структуры данных, хранящейся в `f`, или для записи в эту структуру.

```
(set! (foo f 0) 'dum)
(foo f 0)           ⇒ dum
```

`make-procedure-with-setter` *procedure setter* [Scheme Procedure]
`scm_make_procedure_with_setter` (*procedure, setter*) [C Function]
Создает новую процедуру, которая ведет себя как процедура(*procedure*), но с соответствующим установщиком *setter*.

`procedure-with-setter?` *obj* [Scheme Procedure]
`scm_procedure_with_setter_p` (*obj*) [C Function]
Возвращает `#t` если *obj* это процедура связана с процедурой устнановщика.

`procedure` *proc* [Scheme Procedure]
`scm_procedure` (*proc*) [C Function]
Возвращает процедуру *proc*, которая должна быть применяемой структурой.

`setter` *proc* [Scheme Procedure]
Возвращает установщик *proc*, который должне быть либо процедурой с установщиком, либо оператором структуры.

[4] Рабочее определение должно быть таким:

```
(define foo-ref vector-ref)
(define foo-set! vector-set!)
(define f (make-vector 2 #f))
```

6.9.9 Встраиваемые Процедуры

Вы можете определить Встраиваемую Процедуру(*inlinable procedure*) используя `define-inlinable`, а не `define`. Встраиваемая процедура ведет себя так же, как и в обычная процедура, но прямые вызовы приводят к тому что тело процедуры вставляется в вызывающую процедуру.

Имейте в виду, что начиная с версии 2.0.3 у Guile есть частичный оценщик, который может вставить тело внутренних процедур, если сочтет это целесообразным.

```
scheme@(guile-user)> ,optimize (define (foo x)
                                 (define (bar) (+ x 3))
                                 (* (bar) 2))
$1 = (define foo
       (lambda (#{x 94}#) (* (+ #{x 94}# 3) 2)))
```

Частичный оценщик не устанавиливает привязки верхнего уровня, однако, бывают ситуации в которых вам может быть интересно будет использовать `define-inlinable`.

Процедуры, определяемые с помощью `define-inlinable` всегда *всегда* встраиваются, во всех местах где они вызываются на прямую. Это устраняет накладные расходы функций за счет увеличения расзмера кода. Дополнительно, вызывающий не будет использовать новое определение, если встроенная процедура переопределяется. Невозможно трассировать встроенные процедуры или установить в них точку останова (см. Раздел 6.26.4 [Traps], страница 507). По этим причинам, вы должны делать встроенную процедуру, когда это явно улучшает производительность в критических местах.

В целом, только небольшие процедуры следует рассматривать для встраивания, поскольку большие встраиваемые процедуры приведут к увеличению размера кода. Кроме того, устранение издержек вызова редко имеет значение для крупных процедур.

`define-inlinable` (*name parameter ...*) *body1 body2 ...* [Scheme Syntax]
Определяет процедуру с именем *name* с параметрами *parameter* и телами *body1*, *body2*, ...

6.10 Макросы

В лучшем случае, программирование на Lisp это итеративный процесс создания соответствующего языка к имеющейся проблеме, и затем решение этой проблемы на этом языке. Определение новой процедуры является частью этого, но Lisp также позволяет пользователю расширять свой синтаксис используя широко известные Макросы(*macros*).

Макросы - это синтаксические расширения, которые вызывают трансформацию выражений, в которых они появляются каким либо образом, перед(*before*) их вычислением. В выражениях, которые предназначены для преобразования макросов, идентификатор, который именует соответствующий макрос, должен быть первым элементом, например:

```
(macro-name macro-args ...)
```

Расширение макросов это отдельная фаза вычислений, запускаемый до интерпретации кода или компиляции. Макрос это программа работающая над программами, переводящая встроенный язык в базовый язык Scheme[5]).

6.10.1 Определение Макросов

Макрос является связующим звеном между ключевыми словами и синтаксическим преобразованием. Поскольку трудно обсуждать `define-syntax` без обсуждения формата преобразования, рассмотрим следующий пример определения макроса:

```
(define-syntax when
  (syntax-rules ()
    ((when condition exp ...)
     (if condition
         (begin exp ...)))))

(when #t
  (display "hey ho\n")
  (display "let's go\n"))
⊣ hey ho
⊣ let's go
```

В этом примере, `when` связывается охватывающией формой `define-syntax`. Синтаксические преобразователи более подробно рассматриваются в Раздел 6.10.2 [Syntax Rules], страница 278, и Раздел 6.10.3 [Syntax Case], страница 283.

`define-syntax` *keyword transformer* [Syntax]

Связывает *keyword* к синтаксическому преобразованию, полученному путем вычисления *transformer*.

После определения макроса, дополнительные экземпляры *keyword* в исходном коде Scheme будут вызывать синтаксический преобразователь определенный в *transformer*.

Можно также устанавливать локальные синтаксические привязки используя `let-syntax`.

`let-syntax` ((*keyword transformer*) ...) *exp1 exp2 ...* [Syntax]

Связывает каждое *keyword* с соответствующим преобразователем(*transformer*) при рассширении *exp1 exp2 ...*.

Связывание `let-syntax` существует только во время рассширения(локально).

```
(let-syntax ((unless
              (syntax-rules ()
                ((unless condition exp ...)
                 (if (not condition)
                     (begin exp ...))))))
  (unless #t
    (primitive-exit 1))
```

[5] В наши дни такие встроенные языки часто упоминаются как встроенные специфичные языки(*embedded domain-specific languages*, or EDSLs.

```
  "rock rock rock")
⇒ "rock rock rock"
```

Форма `define-syntax` действительна в любом месте, где может появиться определение: на верхнем уровне или локально. Также как локальное `define` расширяется до экземпляра `letrec`, и локально заданное `define-syntax` расширяется до `letrec-syntax`.

`letrec-syntax` ((*keyword transformer*) ...) *exp1 exp2 ...* [Syntax]

Связывает каждое *keyword* с соответствующим преобразователем(*transformer*) пока расширяются выражения *exp1 exp2 ...*.

В духе `letrec` и `let`, расширение создаваемое трансформатором синтаксиса(*transformer*) может выполняться упоминанием *keyword* связванное в том же *letrec-syntax*.

```
(letrec-syntax ((my-or
                 (syntax-rules ()
                   ((my-or)
                    #t)
                   ((my-or exp)
                    exp)
                   ((my-or exp rest ...)
                    (let ((t exp))
                      (if t
                          t
                          (my-or rest ...)))))))
  (my-or #f "rockaway beach"))
⇒ "rockaway beach"
```

6.10.2 Макрос Syntax-rules

Макрос `syntax-rules` - очень простой, синтаксический преобразователь управляемый шаблоном, с достойной красотой Scheme.

`syntax-rules` *literals* (*pattern template*) ... [Syntax]

Создает синтксисчекий преобразователь, который будет переписывать выражение с использованием правил указанных в предложениях *pattern* и *template*.

Макрос `syntax-rules` состоит из трех частей: литералов (если они есть), образцов(patterns) и столькоих же шаблонов(templates).

Когда расширитель синтаксиса видит вызов макроса `syntax-rules`, он вычисляет выражение для образцов(patterns), в порядке следования, и перезаписывает выражения используя шаблон(template) для первого совпадающего образца. Если нет совпавших образцов, сигнализируется о синтаксической ошибке.

6.10.2.1 Образцы(Patterns)

Мы уже видели некоторые примеры образцов в предыдущем разделе: `(unless condition exp ...)`, `(my-or exp)`, и т.д. Образец структурирован как выражение, которому он должен соответствовать. Он также может иметь вложенную структуру,

подобно `(let ((var val) ...) exp exp* ...)`. Вообще говоря, образцы состоят из списков, неподходящих списков, векторов, идентификаторов, и исходных фактов. Пользователи могут сопоставлять последовательность образцов используя многоточия (`...`).

Идентификаторы в образце называются литералами(*literals*), если они присутствуют в списке литералов `syntax-rules`, и переменными образца(*pattern variables*) если нет. При создании вывода макроса, макрорасширитель заменяет экземпляры переменных в образце на вычисленные подвыражения шаблона.

```
(define-syntax kwote
  (syntax-rules ()
    ((kwote exp)
     (quote exp))))
(kwote (foo . bar))
⇒ (foo . bar)
```

Список неправильных образцов соответствует остаточным(rest) аргументам:

```
(define-syntax let1
  (syntax-rules ()
    ((_ (var val) . exps)
     (let ((var val)) . exps))))
```

Однако это определение `let1`, вероятно, не то, что вы ожидали, так как хвостовой образец *exps* будет соответствовать не спискам, например `(let1 (foo 'bar) . baz)`. Поэтому часто вместо использования списков неправильных как образцов, лучше использовать образцы многоточия. За экземпляром переменной образца в шаблоне должно следовать многоточие.

```
(define-syntax let1
  (syntax-rules ()
    ((_ (var val) exp ...)
     (let ((var val)) exp ...))))
```

Этот `let1` вероятно, все еще не делает того, что мы хотим получить, потому что тело сопоставляемых последовательностей является пустым выражением, таким как `(let1 (foo 'bar))`. В этом случае нам нужно проверят что мы имеем тело хотябы с одним выражением. Общей идиомой для этого является имя переменной многоточечного образца со звездочкой:

```
(define-syntax let1
  (syntax-rules ()
    ((_ (var val) exp exp* ...)
     (let ((var val)) exp exp* ...))))
```

Вектор образцов соответствует вектору, содержимое которого соответствует образцам включающим многоточия и хвостовые образцы.

```
(define-syntax letv
  (syntax-rules ()
    ((_ #((var val) ...) exp exp* ...)
     (let ((var val) ...) exp exp* ...))))
(letv #((foo 'bar)) foo)
⇒ bar
```

Литералы используются для соответствия определенным данным в выражении, например испльзования `=>` и `else` в выражении `cond`.

```
(define-syntax cond1
  (syntax-rules (=> else)
    ((cond1 test => fun)
     (let ((exp test))
       (if exp (fun exp) #f)))
    ((cond1 test exp exp* ...)
     (if test (begin exp exp* ...)))
    ((cond1 else exp exp* ...)
     (begin exp exp* ...))))

(define (square x) (* x x))
(cond1 10 => square)
⇒ 100
(let ((=> #t))
  (cond1 10 => square))
⇒ #<procedure square (x)>
```

Литерал соответствует входному выражению, если входное выраженине является идентификатором тем же именем что и литерал, и оба являются несвязанными[6].

Если образец не является списком, вектором, или идентификатором, он проверяется на совпадение как литерал с помощью `equal?`.

```
(define-syntax define-matcher-macro
  (syntax-rules ()
    ((_ name lit)
     (define-syntax name
       (syntax-rules ()
        ((_ lit) #t)
        ((_ else) #f))))))

(define-matcher-macro is-literal-foo? "foo")

(is-literal-foo? "foo")
⇒ #t
(is-literal-foo? "bar")
⇒ #f
(let ((foo "foo"))
  (is-literal-foo? foo))
⇒ #f
```

В последнем примере показано, что сопоставление происходит во время развертывания макроса(на этапе компиляции), а не во время выполнения.

Макросы синтаксических правил(Syntax-rules) всегда используются как (*macro* . *args*), а *macro* всегда будет символом. Соответственно, образец в `syntax-rules` дол-

[6] Языковые законники, вероятно, видят здесь необходимость использования `literal-identifier=?` в первую очередь, чем `free-identifier=?`, и вероятно будут правы. Поправки принимаются.

жен быть списком (правильным или не правильным), и первый образец в этом списке должен быть идентификатором. Кстати, это может быть любой идентификатор – он не обязательно должен быть именем макроса. Таким обрзамо следующие три примера эквивалентны:

```
(define-syntax when
  (syntax-rules ()
    ((when c e ...)
     (if c (begin e ...)))))

(define-syntax when
  (syntax-rules ()
    ((_ c e ...)
     (if c (begin e ...)))))

(define-syntax when
  (syntax-rules ()
    ((something-else-entirely c e ...)
     (if c (begin e ...)))))
```

Для ясности используйте один из первых двух вариантов. Также обратите внимание, что поскольку переменная образца всегда будет соответствовать самому макросу (например `cond1`), она фактически остается не связанной в шаблоне.

6.10.2.2 Гигиена

Макросы синтакисических правил(`syntax-rules`) имеют магическое свойство: они сохраняют ссылочную прозрачность. Когда вы читаете определение макроса, любые свободные привязки в этом макросе разрешаются относительно определения макроса; и когда вы читаете экземпляр макроса, все свободные привязки в этом выражении разрешаются относительно выражения.

Это свойство иногда называют гигиеной(*hygiene*), и оно помогает в чистоте кода В вашем макроопределении, вы можете свободно вводить временные переменные, не беспокоясь о непреднамеренном введении привязок в макрорасширение.

Рассмотрим определение `my-or` из предыдущего раздела:

```
(define-syntax my-or
  (syntax-rules ()
    ((my-or)
     #t)
    ((my-or exp)
     exp)
    ((my-or exp rest ...)
     (let ((t exp))
       (if t
           t
           (my-or rest ...))))))
```

Наивное расширение `(let ((t #t)) (my-or #f t))` даст:

```
(let ((t #t))
```

```
  (let ((t #f))
    (if t t t)))
⇒ #f
```

Это явно не то, что мы хотим. Так или иначе `t` в определении отличается от `t` в месте использования; и это действительно такое различие, которое поддерживается расширителем синтаксиса, при расширении макросов с гигиеной.

Это обсуждение в основном актуально в контексте традиционных макросов Lisp (см. Раздел 6.10.5 [Defmacros], страница 292), которые не сохраняют ссылочную прозрачность. Гигиена добавляет выразительной силы Scheme.

6.10.2.3 Стенография(сокращение ввода)

Часто пишут простые макросы с одним предложением `syntax-rules`. Существует удобное сокращение для этой идиомы, в форме `define-syntax-rule`.

`define-syntax-rule` (*keyword . pattern*) [*docstring*] *template* [Syntax]
Определяет ключевое слово(*keyword*) как новый макрос `syntax-rules` с одним предложением.

Вставка этой формы, делает наш пример `when` значительно короче:

```
(define-syntax-rule (when c e ...)
  (if c (begin e ...)))
```

6.10.2.4 Сообщения о синтаксических ошибках в Макросах

`syntax-error` *message* [*arg ...*] [Syntax]
Сообщение об ошибке во время макро-расширения *message* должно быть строковым литералом, а необязательные операнды *arg* могут быть произвольными выражениями, предоставляющими дополнительную информацию

`syntax-error` предназначен для использования в шабалонах `syntax-rules` Например:

```
(define-syntax simple-let
  (syntax-rules ()
    ((_ (head ... ((x . y) val) . tail)
        body1 body2 ...)
     (syntax-error
      "expected an identifier but got"
      (x . y)))
    ((_ ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))))
```

6.10.2.5 Указание пользовательского идентификатора многоточия

При написании макросов, которые генерируют определения макросов, удобно использовать разные идентификаторы многоточия на каждом уровне. Guile позволяет указать желаемый идентификатор многточия как первый операнд в `syntax-rules`, как указано в SRFI-46 и R7RS. Например:

```
(define-syntax define-quotation-macros
  (syntax-rules ()
    ((_ (macro-name head-symbol) ...)
     (begin (define-syntax macro-name
              (syntax-rules ::: ()
                ((_ x :::)
                 (quote (head-symbol x :::)))))
            ...))))
(define-quotation-macros (quote-a a) (quote-b b) (quote-c c))
(quote-a 1 2 3) ⇒ (a 1 2 3)
```

6.10.2.6 Дальнейшая информация

Для формального определения `syntax-rules` и языка шаблонов, см. См. Раздел "Macros" в *Revised(5) Report on the Algorithmic Language Scheme*.

Макрос `syntax-rules` простой и чистый, но имеют ограничения. Они делают не поддающиеся выражени сообщения об ошибках: образец либо совпадает, либо нет. Их способность генерировать код ограничивается расширением шаблонов; часто нужно определять ряд вспомогательных макросов, чтобы получить реальную работу. Иногда хочется предоставить привязку к лексическому контексту сгенерированного кода; это не возможно с `syntax-rules`. Кроме того, они не могут программно генерировать идентификаторы.

Решение всех этих проблем использовать `syntax-case` если вам нужны его функции. Но если по какой либо причине вы придерживаетесь `syntax-rules`, вам может понравиться работа Joe Marshall `syntax-rules` Primer for the Merely Eccentric (`http://sites.google.com/site/evalapply/eccentric.txt`).

6.10.3 Поддержка для системы `syntax-case`

Макрос `syntax-case` это процедурный синтаксический преобразователь, обладающий мощью достойной Scheme.

`syntax-case` *syntax literals* (*pattern* [*guard*] *exp*) . . . [Syntax]

Сопоставляет синтаксический объект *syntax* с данными образцами, в порядке следования. Если *pattern* совпадает, возвращает результат вычисления соответствующего выражения *exp*.

Сравните следующие определения `when`:

```
(define-syntax when
  (syntax-rules ()
    ((_ test e e* ...)
     (if test (begin e e* ...)))))

(define-syntax when
  (lambda (x)
    (syntax-case x ()
      ((_ test e e* ...)
       #'(if test (begin e e* ...))))))
```

Очевидно, что определение `syntax-case` аналогично `syntax-rules` и одинаково ясно, что есть несколько различий. Определение `syntax-case` обернуто в лямбда(`lambda`), функцию от одного аргумента; этот аргумент передается вызову `syntax-case`; и "возвращаемое значение" макроса имеет префикс `#'`.

Все эти различия связаны с тем, что `syntax-case` не определяет преобразователь синтаксиса самостоятельно – вместо этого выражение `syntax-case` обеспечивает способ разрушить синтаксический объект (*syntax object*), и перестроить синтаксический объект при выводе.

Таким образом, обертка `lambda` это просто проходная деталь реализации, чтобы преобразователь синтаксиса просто функция преобразовывающая синтаксис в синтаксис. Это не должно удивлять, учитывая что мы уже описали макросы как "программы, которые пишут программы". `syntax-case` это простой способ разобрать и скомпоновать текст программы, и допустимый преобразователь синтаксиса должен быть обернут в процедуру.

В отличии от традиционных макросов Лисп (см. Раздел 6.10.5 [Defmacros], страница 292), макросы `syntax-case` преобразуют синтаксический объект, а не необработанные формы Scheme. Напомним наивное расширение `my-or` заданное в предыдущем разделе:

```
(let ((t #t))
  (my-or #f t))
;; naive expansion:
(let ((t #t))
  (let ((t #f))
    (if t t t)))
```

В необработанных формах Scheme просто не хватает информации чтобы отличить первые два экземпляра `t` в форме `(if t t t)` из/от третьего `t`. Поэтому вместо предоставления идентификаторов в качестве символов, расширитель синтаксиса представляет идентификаторы как синтаксические объекты, присоединяя к ним информацию которая необходима синтаксическим объектам для поддержания ссылочной прозрачности.

`syntax` *form* [Syntax]

создает синтаксический объект обернутый в форму *form* в текущем лексическом контексет.

Синтаксические объекты обычно создаются внутри процесса расширения, но это возможность создать их вне расширения синтаксиса.

```
(syntax (foo bar baz))
⇒ #<some representation of that syntax>
```

Однако более часто и полезно, создавать синтаксические объекты при посторении вывода из выражения `syntax-case`.

```
(define-syntax add1
  (lambda (x)
    (syntax-case x ()
      ((_ exp)
       (syntax (+ exp 1))))))
```

Не обязательно, чтобы выражение `syntax-case` возвращало синтаксический объект, потому что выражения `syntax-case` могут использоваться вспомогательными функциями или использоваться иным образом вне самого расширения синтаксиса. Однако процедура преобразования синтаксиса должна возрвращать синтаксический объект, поэтому большинство применений `syntax-case` в конечном итоге возвращает синтаксические объекты.

Здесь в этом случае форма, которая построила возвращаемое значение, была `(syntax (+ exp 1))`. Интересная вещь заключается в том, что в выражении `syntax` любое появление образца переменной подставляется в результирущий синтаксический объект, перенося с собой все соответствующие метаданные из исходного выражения, такого как лексический идентификатор и местоположение источника.

В самом деле, переменная шаблона может ссылаться только внутри формы `syntax`. Синтаксический расширитель вызовет ошибку при определении `add1` если он найдет *exp* ссылающееся вовне `syntax` формы.

Поскольку `syntax` часто появляется в макро-тяжелом коде, он имеет специальный макрос чтения: `#'`. `#'foo` преобразуется читателем в `(syntax foo)`, также как `'foo` преобразуется в `(quote foo)`.

Язык шаблонов используемый `syntax-case` это удобный язык используемый в `syntax-rules`. Учитывая это, Guile фактически определеят `syntax-rules` в терминах `syntax-case`:

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      ((_ (k ...) ((keyword . pattern) template) ...)
       #'(lambda (x)
           (syntax-case x (k ...)
             ((dummy . pattern) #'template)
             ...))))))
```

И вот что.

6.10.3.1 Почему syntax-case?

Данные нами до сих пор примеры можно было точно также выразить `syntax-rules`, и только что показали, что `syntax-case` является более подробным, что верно. Но есть разница: `syntax-case` создает процедурный(*procedural*) макрос, давая полную мощь Scheme макро-расширителю. Этому есть много практических применений.

Общее желание состоит в том, чтобы иметь возможность сопоставлять форму только в том случае, если она является идентификатором. Это невозможно с `syntax-rules`, с учетом форм сопоставления данных. Но с `syntax-case` это легко:

`identifier?` *syntax-object* [Scheme Procedure]

Возвращает `#t` если *syntax-object* это идентификатор, или `#f` если нет.

```
;; relying on previous add1 definition
(define-syntax add1!
  (lambda (x)
    (syntax-case x ()
```

```
        ((_ var) (identifier? #'var)
         #'(set! var (add1 var))))))

(define foo 0)
(add1! foo)
foo ⇒ 1
(add1! "not-an-identifier") ⇒ error
```

С `syntax-rules`, ошибка для `(add1! "not-an-identifier")` была бы чем то похожим на “invalid `set!`”. С `syntax-case`, он может сказать что то вроде “invalid `add1!`”, потому что мы прикрепляем защитное предложение (*guard clause* к образцу: `(identifier? #'var)`. Это становиться более важным с более сложными макросами. Нижно использовать `identifier?`, потому что расширитель, идентифицирует более чем пустой символ.

Обратите внимание, что даже в защищающем предложении, мы ссылаемся на переменную *var* образца в форме `syntax`, через `#'var`.

Еще одно общее желание – ввести привязки в лексический контекст выражений вывода. Один пример будет в так называемых “anaphoric макросах”, таких как `aif`. Anaphoric макросы связывают некоторое выражение с известным идентификатором, часто он(`it`), внутри своих тел. Например, в `(aif (foo) (bar it))`, `it` будет связан с результатом `(foo)`.

Для начала следует упомянуть решение, которое не работает:

```
;; doesn't work
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       #'(let ((it test))
           (if it then else))))))
```

Причина, по которой это не работает, заключается в том, что по умолчанию расширитель сохраняет ссылочную прозрачность; выражения *then* и *else* не будут иметь доступ к привязке `it`.

Но они смогут ее получить, если мы введем привязку через `datum->syntax`.

`datum->syntax` *template-id datum* [Scheme Procedure]
: Создает синтаксический объект, который оборачивает *datum* в лексическом контексте, соответствующим идентификатору *template-id*.

Для полноты, следует отметить, что метаданные можно удалить из синтаксического объекта, вернув необработанные данные(datum) Scheme:

`syntax->datum` *syntax-object* [Scheme Procedure]
: Убирает метаданные из *syntax-object*, возвращая его содержимое в виде необработанных данных(datum) Scheme.

В этом случае мы хотим пердставить `it` в контексте всего выражения, поэтому мы можем создать синтаксический объект как `(datum->syntax x 'it)`, где `x` это целое выражение, как прошедшее процедуру преоборазованиня.

Вот еще одно решение которое не работает:

```
;; doesn't work either
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       (let ((it (datum->syntax x 'it)))
         #'(let ((it test))
             (if it then else)))))))
```

Причина, по которой это не работает, состоит в том, что на самом деле здесь работают две среды – среда переменных образца, связываемых `syntax-case`, и среда лексических переменных, связываемых в обычной Scheme. Внешняя форма let устанавливает привязки в среде лексических переменных, но внутренняя форма let находиться внутри синтаксической формы, где будут заменены только переменные образца. Здесь нам нужно ввести кусочек лексической среды(окружения) в среду переменных образца, и мы можем сделать это используя сам `syntax-case`:

```
;; works, but is obtuse
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       ;; invoking syntax-case on the generated
       ;; syntax object to expose it to `syntax'
       (syntax-case (datum->syntax x 'it) ()
         (it
           #'(let ((it test))
               (if it then else))))))))

(aif (getuid) (display it) (display "none")) (newline)
⊣ 500
```

Однако есть более простые способы написать это. `with-syntax` часто удобно:

`with-syntax` ((*pat val*) ...) *exp* ... [Syntax]

Связывает образцы *pat* с их соответствующими значениями *val*, в лексическом контексте *exp* ....

```
;; better
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       (with-syntax ((it (datum->syntax x 'it)))
         #'(let ((it test))
             (if it then else)))))))
```

Как вы можете себе представить, `with-syntax` определяется в терминах `syntax-case`. Но даже это может быть неприятным если вы старый хакер макросов

на Lispe, привыкший к созданию с помощью `quasiquote`. Проблема в том, что `with-syntax` создает разделение между точкой определения значения и точкой его подстановки.

Так что для случаев, в которых стиль `quasiquote` имеет больше смысла, `syntax-case` также определяет `quasisyntax`, и связанные с ним `unsyntax` и `unsyntax-splicing`, сокращенно читаемые как `#`, `#,`, и `#,@`, соответственно.

Например, чтобы определить макрос, который вставляет метку времени компиляции в искходный файл, можно написать:

```
(define-syntax display-compile-timestamp
  (lambda (x)
    (syntax-case x ()
      ((_)
       #`(begin
           (display "The compile timestamp was: ")
           (display #,(current-time))
           (newline))))))
```

Читатели, интересующиеся дополнительной информацией о макросах `syntax-case` должны смотреть R. Kent Dybvig's отличную книгу *The Scheme Programming Language*, редакции 3 или 4, в главе о синтаксисе. Dybvig был основным автором системы `syntax-case`. Сама книга доступна в интернете `http://scheme.com/tspl4/`. (ищите русский перевод)

6.10.3.2 Пользовательские Идентификаторы Многоточий для макросов syntax-case

При написании процедурных макросов, которые генерируют определения макросов, удобно использовать различные идентификаторы многоточий на каждом уровне. Guile поддерживает это для процедурных макросов используюзуя специальную форму `with-ellipsis`:

`with-ellipsis` *ellipsis body ...* [Syntax]

ellipsis должен быть идентификатором. Вычисляет *body* в специальном лексическом окружении(среде), такой что все макро обрацзы и шаблоны внутри тела *body* будут использовать *ellipsis* как идентификатор многоточия вместо обычных трех точек (`...`).

Например:

```
(define-syntax define-quotation-macros
  (lambda (x)
    (syntax-case x ()
      ((_ (macro-name head-symbol) ...)
       #'(begin (define-syntax macro-name
                  (lambda (x)
                    (with-ellipsis :::
                      (syntax-case x ()
                        ((_ x :::)
                         #'(quote (head-symbol x :::)))))))))
```

```
                ...)))))
(define-quotation-macros (quote-a a) (quote-b b) (quote-c c))
(quote-a 1 2 3) ⇒ (a 1 2 3)
```

Обратите внимание, что `with-ellipsis` не влияет на идентификатор многоточия генерируемого кода, разве что `with-ellipsis` будет включен в сгенерированный код.

6.10.4 Вспомогательные функции Синтаксического Преобразования

Как отмечалось в предыдущем разделе, синтаксический расширитель Guile работет с синтаксическими объектами. Процедурные макросы потребляют и создают синтаксические объекты. В этом разделе описаны некоторые из вспомогательных помощников, которые процедурные макросы могут использовать для сравнения, генерации и запросов объектов этого типа данных.

`bound-identifier=?` *a b* [Scheme Procedure]
: Возвращает `#t` если синтаксический объект *a* и *b* ссылаются на один и тот же лексически связанный идентификатор, или `#f` в противном случае.

`free-identifier=?` *a b* [Scheme Procedure]
: Возвращает `#t` если синтаксические объекты *a* и *b* ссылаются на один и тот же свободный идентификатор, или `#f` в противном случае.

`generate-temporaries` *ls* [Scheme Procedure]
: Возвращает список временных идентификаторов такой же длины какой длины список *ls*.

`syntax-source` *x* [Scheme Procedure]
: Возвращает исходные свойства, которые соответствуют синтаксическому объекту *x*. См. Раздел 6.26.2 [Source Properties], страница 498, для дальнейшей информации.

А теперь уделим немного времения для исповеди. Синтаксический расширитель Guile берет свое начало в коде из Chez Scheme: версия расширителя в Chez Scheme которая была сделана переносимой в другую систему Scheme. Еще в середине 1990x, некоторые системы Scheme даже не имели возможности определять новые абстрактные типы данных. По этой причине, переносимый расширитель из Chez Scheme который унаследовала Guile использовал тегированные вектора в качестве синтаксических объектов: вектора, первый элемент которых был символ, `syntax-object`.

На момент написания статьи это был 2017 и Guile все еще поддерживает эту стратегию. Это работало долго, потому что никто не помещал буквальный(литеральныЙ) вектор в положение оператора:

```
(#(syntax-object ...) 1 2 3)
```

Но такое положение дел было ошибкой. Поскольку синтаксические объекты являлись просто векторами, это дает возможность любому коду Scheme подделать объект синтаксиса, которая может привести к нарушению границы абстракции. В не можете построить средство для песочницы, которая ограничивает набор привязок в области, когда всегда можно избежать этого ограничения, просто вычислив специальный вектор. Для устранения этой проблемы, Guile 2.2.1 наконец, переносит представление

синтаксических объектов в виде отдельного типа с отдельным конструктором, который недоступен для пользовательского кода.

Тем не менее, Guile по прежнему должен поддерживать “устаревшие” синтаксические объекты, потому что это может быть файл скомпилированный с Guile 2.2.0, который встраивает синтаксические объекты векторного типа. Расширитель обрабатывает специальные тегированные вектора, так как синтаксические объекты теперь контролируются с помощью параметра `allow-legacy-syntax-objects?`:

`allow-legacy-syntax-objects?` [Scheme Procedure]
Параметр который указывает, должен ли расширитель поддерживать устаревшие синтаксические объекты, как описано выше. По причинам стабильности ABI по умолчанию используется значение `#t`. Используйте `parameterize`, чтобы связать его с `#f`. См. Раздел 6.13.12 [Parameters], страница 347.

Guile также предлагает еще несколько экспериментальных интерфейсов в отдельном модуле. Как было в случае большим адронным коллайдером, нашим ведущим думателям(macrologists) неясно, добавление этих интерфейсов приведет к потрясению или уничтожению Guile посредством создания сингулярности. Мы сохраним их функциональность в серии 2.0, но мы оставляем за собой право изменять их в будущих стабильных сериях, большей чем обычно степени.

```
(use-modules (system syntax))
```

`syntax-module` *id* [Scheme Procedure]
Возвращает имя модуля, источника содержащего идентификатор *id*.

`syntax-local-binding` *id* **[***#:resolve-syntax-parameters?=#t***]** [Scheme Procedure]
Разыскивает(разрешает) идентификатор *id*, синтаксического объекта, в текущей лексической среде и возвращает два значения, тип привязки и значение привязки. Тип привязки является символом, который может быть одним из следующих:

`lexical` Лексически связанная переменная. Значение является уникальным токеном (в смысле `eq?`) идетифицирующим эту привязку.

`macro` Синтаксический преобразователь, локальный или глобальный. Значением является процедурой преобразования.

`syntax-parameter`
Синтаксический параметр (см. Раздел 6.10.7 [Syntax Parameters], страница 295). По умолчанию, `syntax-local-binding` будет разрешать синтаксический параметр, так что это значение ну будет возвращено. Передайте `#:resolve-syntax-parameters? #f` чтобы указать, что вас интересуют синтаксические параметры. Значение является процедурой преобразования по умолчанию, как в `macro`.

`pattern-variable`
Переменная образца, связанная через `syntax-case`. Значением является непрозрачный объект, внутренний для расширителя.

`ellipsis` Внутренняя привязка, связанная через `with-ellipsis`. Значением является (anti-marked) локальный идентификатор многоточия.

`displaced-lexical`
: Лексическая переменная, вышедшая из области видимости. Это может произойти, если плохо написанный процедурный макрос сохраняет синтаксический объект, а затем пытается разместить его в контексте, в котором он не связан. Значение равно `#f`.

`global`
: Глобальная привязка. Значение представляет собой пару, чья голова является символом, и чей хвост - это имя модуля, в котором разрешается символ.

`other`
: Некоторые другие привязки, такие как `lambda` или другие привязки в ядре. Значение равно `#f`.

Это очень низкоуровневая процедура с ограниченным использованием. Один случай, когда она полезна это построение абстракций, которые связывают вспомогательную информацию с макросами:

```
(define aux-property (make-object-property))
(define-syntax-rule (with-aux aux value)
  (let ((trans value))
    (set! (aux-property trans) aux)
    trans))
(define-syntax retrieve-aux
  (lambda (x)
    (syntax-case x ()
      ((x id)
       (call-with-values (lambda () (syntax-local-binding #'id))
         (lambda (type val)
           (with-syntax ((aux (datum->syntax #'here
                                             (and (eq? type 'macro)
                                                  (aux-property val)))))
             #''aux)))))))
(define-syntax foo
  (with-aux 'bar
    (syntax-rules () ((_) 'foo))))
(foo)
⇒ foo
(retrieve-aux foo)
⇒ bar
```

`syntax-local-binding` должна вызываться в пределах динамического пространства синтаксического преобразователя; чтобы вызвать его в противном случае сигнализировать об ошибке.

`syntax-locally-bound-identifiers` *id* [Scheme Procedure]

Возвращает список идентификаторов, которые были видны лексически при создании идентификатора *id*, в порядке от самого внешнего, до самого внутреннего.

Эта процедура предназначена для использования в специализированных процедурных макросах, чтобы обеспечить макрос набором связанных идентификаторов, на которые может ссылаться макрос.

В качестве технической детали реализации, идентификаторы возвращаемые `syntax-locally-bound-identifiers` будут помечены, как синтаксические объекты, заданные в качестве входных данных в макрос. Это сигнализирует макро рассширителю, что эти привязки присутствуют в первоисточнике(исходнике) и не должны быть гигиенически переименованы, как было бы в случае с другими введенными идентификаторами. См. обсуждение гигиены в разделе 12.1 R6RS, для получения дополнительной информации о маркировке.

```
(define (local-lexicals id)
  (filter (lambda (x)
            (eq? (syntax-local-binding x) 'lexical))
          (syntax-locally-bound-identifiers id)))
(define-syntax lexicals
  (lambda (x)
    (syntax-case x ()
      ((lexicals) #'(lexicals lexicals))
      ((lexicals scope)
       (with-syntax (((id ...) (local-lexicals #'scope)))
         #'(list (cons 'id id) ...))))))

(let* ((x 10) (x 20)) (lexicals))
⇒ ((x . 10) (x . 20))
```

6.10.5 Определение макросов в стиле Lisp

Традиционный способ определения макросов в Lisp очень похож на определение процедур. Основные отличия заключаются в том, что тело определения макроса должно возвращать список, который описывает преобразованное выражение, и который определен как помеченное макроопределения (а не определение процедуры) с использованием отличного ключевого слова определения: в Lisp, `defmacro` а не `defun`, и в Scheme, `define-macro` а не `define`.

Guile поддерживает этот стиль определения макросов, используя оба, и `defmacro` и `define-macro`. Единственная разница между ними заключается в том, как сгруппированы имя макроса и аргументы в определении:

```
(defmacro name (args ...) body ...)
```

такой же как

```
(define-macro (name args ...) body ...)
```

Разница аналогична соответствующей разнице между `defun` в Lisp и `define` в Scheme.

Прочитав предыдущий раздел `syntax-case`, вероятно вам ясно, что Guile реализуеts defmacros в терминах `syntax-case`, применяя преобразователь выражений между вызовами `syntax->datum` и `datum->syntax`. Эта реализация приводит нас к проблемам с defmacros, состоящих в том, что он не сохраняет ссылочную прозрачность. Можно, осторожно, чтобы не вводить привязки в расширенный код, через либеральное использование `gensym`, но здесь не обойтись без ссылочной прозрачности для свободных привязок в самом макросе.

Даже такой простой макрос как `when` из предыдущего разделаЮ было бы трудно понять:

```
(define-macro (when cond exp . rest)
  `(if ,cond
       (begin ,exp . ,rest)))

(when #f (display "Launching missiles!\n"))
⇒ #f

(let ((if list))
  (when #f (display "Launching missiles!\n")))
⊣ Launching missiles!
⇒ (#f #<unspecified>)
```

Перспектива Guile заключается в том, что defmacros отлично работает, но современные макросы должны быть написаны с помощью `syntax-rules` или `syntax-case`. Есть еще много примененений defmacros в самом Guil, но мы будем постепенно сворачивать их. Кончено, мы не будем убирать `defmacro` или `define-macro`, так как есть много кода, который использует их.

6.10.6 Идентификатор Макросов

Когда расширитель синтаксиса видит форму, в которой первый элемент является макросом, вся форма передается преобразователю синтаксиса макроса. Можно представить это как:

```
(define-syntax foo foo-transformer)
(foo arg...)
;; expands via
(foo-transformer #'(foo arg...))
```

Если, с другой стороны, макрос упоминается в какой либо другой части формы, перобразователь синтаксиса вызывается только со ссылкой на макрос, а не на всю форму.

```
(define-syntax foo foo-transformer)
foo
;; expands via
(foo-transformer #'foo)
```

Это позволяет программно заменять голые ссылки на идентификаторы с помощью макроса. `syntax-rules` предоставлят несколько синтаксисов для более легкого осуществления этого преобразования.

`identifier-syntax` *exp* [Syntax]

Возвращает макро преобразователь, который заменит вхождение макроса на *exp*.

Например, если вы импортируете внешний код написанный в терминах `fx+`, fixnum оператор сложения, но в Guile нет `fx+`, вы можете использовать следующее для замены `fx+` на `+`:

```
(define-syntax fx+ (identifier-syntax +))
```

Так же есть специальная поддержка распознавания идентификаторов в левой части выражения `set!`, как в следующем примере:

```
(define-syntax foo foo-transformer)
(set! foo val)
;; expands via
(foo-transformer #'(set! foo val))
;; if foo-transformer is a "variable transformer"
```

Как видно из примера, процедура преобразования должна быть явно помечена как преобразователь переменной("variable transformer"), так как большинство макросов не пишуться для распознавания в форме положения оператора.

`make-variable-transformer` *transformer* [Scheme Procedure]

Помечает процедуру *transformer* как преобразователь переменной("variable transformer"). На практике это означает, что при привязке к синтаксическому ключевому слову, он может обнаружить ссылки на это ключевое слово в левой части `set!`.

```
(define bar 10)
(define-syntax bar-alias
  (make-variable-transformer
   (lambda (x)
     (syntax-case x (set!)
       ((set! var val) #'(set! bar val))
       ((var arg ...) #'(bar arg ...))
       (var (identifier? #'var) #'bar)))))

bar-alias ⇒ 10
(set! bar-alias 20)
bar ⇒ 20
(set! bar 30)
bar-alias ⇒ 30
```

Имеется расширение identifier-syntax которое позволяет ему обрабатывать так же `set!`:

`identifier-syntax` (*var exp1*) ((*set! var val*) *exp2*) [Syntax]

Создает преобразователь переменной. Первое предложение используется для ссылок на переменную в положении оператора или операнда, а второе - для появления переменной в левой стороне оператора присваивания.

Например, предыдущий пример `bar-alias` можно выразить более кратко, подобно этому:

```
(define-syntax bar-alias
  (identifier-syntax
    (var bar)
    ((set! var val) (set! bar val))))
```

Как и прежде, шаблоны в формах `identifier-syntax` не нуждаются в обертывании в `#'` синтаксических форм.

6.10.7 Синтаксические Параметры

Синтаксические параметры[7] представляют собой механизм для повторного связывания определения макроса в пределах динамического расширения макроса. Это обеспечивает удобное решение для одного из самых распространенных типов негигиенических макросов: те, которые вводят нигигиеническое связывание каждый раз когда макрос используется. Например включение формы `lambda` с ключевым словом `return`, или макросов класса, которые вводят специальную привязку `self`.

С синтаксическими параметрами, вместо того, чтобы негигиенично вводить каждый раз привязу, вместо этого мы создаем одну привязку для ключевого слова, которую затем можем настроить позже, когда захотим, чтобы ключевое слово имело другое значение. Поскольку новые привязки не вводяться, гигиена сохраняется. Это похоже на механизмы динамического связывания, которые мы имеем во время выполнения (см. Раздел 7.5.27 [SRFI-39], страница 653), за исключением того, что динамическое связывание происходит только во время расширения макроса. Код после расширения макроса остается лексически ограниченным(охваченным).

`define-syntax-parameter` *keyword transformer* [Syntax]

Привязывает *keyword* к значению, полученному путем выполнения преобразователя *transformer*. *transformer* обеспечивает расширение по умолчанию для синтаксического параметра, а при отсутствии `syntax-parameterize`, функционально эквивалентен `define-syntax`. Обычно вы просто хотите, что бы перобразователь *transformer* выдал синтаксическую ошибку, указвающую что *keyword* предполагается использовать вместе с другим макросом, например:

```
(define-syntax-parameter return
  (lambda (stx)
    (syntax-violation 'return "return used outside of a lambda^" stx)))
```

`syntax-parameterize` ((*keyword transformer*) ...) *exp* ... [Syntax]

Корректирует *keyword* ... для использования значения полученного путем вычисления преобразователя *transformer* ..., при расширении формы *exp* .... Каждое *keyword* должно быть связано с синтаксическим параметром. `syntax-parameterize` отличается от `let-syntax`, тем что привязка не затеняется, а скорее корректируется, и поэтому использует keyword при расширении *exp* ... использует новый преобразователь. Это несколько похоже на то, как `parameterize` корректирует значения регулярных параметров, вместо того чтобы создавать новые привязки.

```
(define-syntax lambda^
  (syntax-rules ()
    [(lambda^ argument-list body body* ...)
     (lambda argument-list
       (call-with-current-continuation
        (lambda (escape)
          ;; В теле мы настраиваем ключевое слово 'return' так чтобы вызов
          ;; 'return' заменялся бы вызовом escape прекращения(побега)
```

[7] Описанные в публикации *Сохранение чистоты с использованием синтаксических параметров(Keeping it Clean with Syntax Parameters)* by Barzilay, Culpepper and Flatt.

```
            ;; продолжения.
            (syntax-parameterize ([return (syntax-rules ()
                                            [(return vals (... ...))
                                             (escape vals (... ...))])])
              body body* ...))))]))

;; Теперь мы можем написать функции, которые возвращаются досрочно. Здесь, резуль
;; немедленный возврат если он видит 0 элементов.
(define product
  (lambda^ (list)
           (fold (lambda (n o)
                   (if (zero? n)
                       (return 0)
                       (* n o)))
                 1
                 list)))
```

6.10.8 Eval-when

Так как макросам `syntax-case` доступна вся сила Scheme, они представляют проблему, связанную со временем: когда макрос запускается, какие части программы доступны для использования макросом?

Ответ на этот вопрос по умолчанию, заключается в том, что при импортеа модуля (через `define-module` или `use-modules`), этот модуль будет загружен во время расширения, так же как и во время выполнения. Кроме того, синтаксичсеские определения верхнего уровня во время компиляции элемента(файла) созданные с помощью `define-syntax`, также вычисляются во время расширения в том порядке, в котором они появляются в компилируемом элементе(файле).

Но если синтаксическое определение должно вызвать нормальную процедуру во время расширения, вполне может потребоваться специальная декларация, указывающая на необходимость доступности процедуры во время расширения.

Например, следующий код будет работать в REPL, не не в файле:

```
;; не правильное
(use-modules (srfi srfi-19))
(define (date) (date->string (current-date)))
(define-syntax %date (identifier-syntax (date)))
(define *compilation-date* %date)
```

Это работает в REPL, потому что выражения вычисляются(обрабатываются) по порядку, но если они будут помещены в файл, выражения расширяются один за другим, но не вычисляются до того как скомпилированный файл будет загружен.

Исправление состоит в использовании `eval-when`.

```
;; правильно: исползуя eval-when
(use-modules (srfi srfi-19))
(eval-when (expand load eval)
  (define (date) (date->string (current-date))))
(define-syntax %date (identifier-syntax (date)))
```

```
(define *compilation-date* %date)
```

`eval-when` *conditions exp...* [Syntax]

Вычисляет *exp...* при данных условиях *conditions*. Действительные условия включают в себя:

`expand` Вычисление во время макро расширения, не зависимо от того скомпилирован он или нет.

`load` Вычисление на этапе вычисления скомпилированного кода, т.е. при загрузке скомпилированного модуля или запуске скомпилированного кода в REPL.

`eval` Вычисление во время фазы вычисления не скомпилированного кода.

`compile` Вычисление во время макро расширения, но только при компиляции.

Другими словами, при использовании примитивного исполнителя(evaluator), выражение `eval-when` с `expand` выполняются во время расширения макроса, а те, которые имеют `eval` запускаются во время этапа вычисления.

При использовании компилятора, выражение `eval-when` с `expand` или `compile` запускаются во время макро расширения, а те у кого есть `load` также запускаются во время этапаа вычисления.

Если вы сомневаетесь, исползуйте три условия `(expand load eval)`, как в примере выше. Другие способы использования `eval-when` могут привести к анулированию гарантий или отравлению вашего кота.

6.10.9 Расширение Макроса

Обычно, макросы расширяются по мере необходимости их использования.Расширение макросов неотемлемая часть `eval` и `compile`. Пользователи также могут расширять макросы в запросе REPL через команду REPL `expand`; См. Раздел 4.4.4.4 [Compile Commands], страница 54.

Макросы также могут быть расширены программно, через `macroexpand`, не детали получаются ужасными, по двум причинам.

Первая трудность состоит в том, что результатом макрорасширения является не Scheme: а Tree-IL, высокоуровневый промежуточный язык Guile. См. Раздел 9.4.3 [Tree-IL], страница 873. Как “гигиенические макросы” могут создавать идентификаторы, которые отличаются друг от друга, но имеют одинаковое имя, выходной формат должен быть способен представлять различия между переменными и именами. Еще раз См. Раздел 9.4.3 [Tree-IL], страница 873, для получаения детальной информации. Самое простое - просто запустить `tree-il->scheme` для результата макрорасширения:

```
(macroexpand '(+ 1 2))
⇒
#<tree-il (call (toplevel +) (const 1) (const 2))>

(use-modules (language tree-il))
(tree-il->scheme (macroexpand '(+ 1 2)))
⇒
```

```
(+ 1 2)
```

Вторая трудность заключена в `eval-when`. В качестве примера, что означает для macro-expand определение макроса?

```
(macroexpand '(define-syntax qux (identifier-syntax 'bar)))
⇒
?
```

Ответ зависит от того, кто является расширителем макроса и почему. Вы определяете макрос в текущей окружающей среде? Остаточное определение макроса? И то и другое? Ни? По умолчанию это расширение в режиме “eval”, что означает, что предложения `eval-when` будет действовать только когда `eval` (или `expand`) удовлетворяют набору условий. Макрос верхнего уровня будет вычислен `eval` в среде верхнего уровня.

Таким образом `(macroexpand` *foo*`)` эквивалентно `(macroexpand` *foo* `'e '(eval))`. вторым аргументом является режим (`'e` for “eval”) и второй параметр eval-syntax-expanders-when (только `eval` в этой установке по умолчанию).

Но если вы компилируете определение макроса, возможн, вы захотите восстановить само определение макроса. В этом случае вы передаете `'c` в качестве второго параметра в `macroexpand`. Но, вероятно, вы хотите, чтобы определение макроса присутствовало во врмея компиляции, так что вы передаете `'(compile load eval)` как параметр *esew*. На самом деле `(compile` *foo* `#:to 'tree-il)` полностью эквивалентен `(macroexpand` *foo* `'c '(compile load eval))`; См. Раздел 9.4.2 [The Scheme Compiler], страница 872.

Это ужасный интерфейс; мы знаем. Макрорасширитель(macroexpander) несколько сложнее в этом отношении, поэтому, если вы не создаете утилиту макро-расширения, мы предлагаем избегать его вызова на прямую.

6.10.10 Гигиена и Верхний Уровень

Рассмотрим следующий макрос.

```
(define-syntax-rule (defconst name val)
  (begin
    (define t val)
    (define-syntax-rule (name) t)))
```

Если мы используем его, чтобы сделать пару привязок:

```
(defconst foo 42)
(defconst bar 37)
```

Его расширение будет выглядеть примерно так:

```
(begin
  (define t 42)
  (define-syntax-rule (foo) t))
(begin
  (define t 37)
  (define-syntax-rule (bar) t))
```

Поскольку два `t` привязки были введены макросом, они должны быть введены гигиенически(hygienically) – и действительно, они находяться внутри лексического

контура (т.е. `let` или какой-то другой лексической области охвата). Ссылка `t` в `foo` отличается от ссылке в `bar`.

На верхнем уровне все сложнее. До Guile 2.2, исползование `defconst` на верхнем уровне не будет вводить новую привязку для `t`. Это было сделано в соответствии с гибкой интерпретацией стандарта Scheme, в котором предполагалось, что все возможные привязки осуществляются на верхнем уровне, и в котором мы просто используем преимущества `define` верхнего уровня для существующей привязки она эквивалентна `set!`. Но это не веская причина.

Решение состоит в том, чтобы создать новые имена для всех привязок, предоставленных макросами, а не только привязками в лексических контурах, но и привязкам введенным на верхнем уровне.

Однако, очевидно стратегия простого присвоения случайных имен идентификаторам верхнего уровня создает проблему для отдельной компиляции. Рассмотрим без ограничения общности `defconst` из `foo` в модуле `a`, который вводит новое имя верхнего уровня `t-1`. Если мы тогда скомпилируем модуль `b` который исползжует `foo`, теперь в модуле `b` есть ссылка на `t-1`. Если модуль `a` затем вновь расширяется, по любой причине, например при простой перекомпиляции, введенный `t` получает новое имя; скажем, `t-2`. Теперь модуль `b` ломается, потому что в модуле `a` нет больше привязки с именем `t-1`.

Если введенные идентификаторы верхнего уровня экранируются(“escape”) модулем, то в любом случае они образуют часть двоичного интерфейса (ABI) модуля. С инженерной точки зрения не допустимо случайное изменение ABI. (Это также создает практические проблемы в выполнении условий перекомпиляции лицензии Lesser GPL, для таких модулей.) По этой причине многие люди предпочитают никогда не использовать макросы, вводящие идентификаторы на верхнем уровне, вместо этого заставляя эти макросы получать имена для вводимых ими идентификаторов как часть их аргументов, или чтобы строить их программно и использовать `datum->syntax`. Но этот подход требует всеведения в отношении реализации всех макросов, которые можно использовать, и также ограничивает выразительную силу макросов Scheme.

Идеального решения этой проблемы не существует. Guile делает здесь ужасную вещь. Когда дело идет к введению идентификатора верхнего уровня, Guile дает идентификатору псевдо-чистое имя: имя это зависит от хеша исходного выражения, в котором встречается имя. В этом случае введенные определения расширяются как:

```
(begin
  (define t-1dc5e42de7c1050c 42)
  (define-syntax-rule (foo) t-1dc5e42de7c1050c))
(begin
  (define t-10cb8ce9fdddd6e9 37)
  (define-syntax-rule (bar) t-10cb8ce9fdddd6e9))
```

Однако обратите внимание, что поскольку хеш зависит исключительно от выражения, вводящего определение, у нас также есть:

```
(defconst baz 42)
⇒ (begin
    (define t-1dc5e42de7c1050c 42)
    (define-syntax-rule (baz) t-1dc5e42de7c1050c))
```

Обратите внимание, что введенная привязка имеет тоже имя! Это потому, что исходное выражение, `(define t 42)`, было тоже самое. Возможно, вы никогда не увидите ошибку в этой области, но важно понимать компоненты интерфейса модуля, и что интерфейс может включать в себя макро-введенные идентификаторы.

6.10.11 Внутренние Макросы

`make-syntax-transformer` *name type binding* [Scheme Procedure]
Строит объект синтаксического преобразования. Это часть низкоуровневой поддержки в Guile syntax-case.

`macro?` *obj* [Scheme Procedure]
`scm_macro_p` (*obj*) [C Function]
Возвращает `#t` если *obj* является синтаксическим преобразователем, или `#f` otherwise.

Обратите внимание, что немного сложно получить макрос как объект первого класса(первоклассный); простое именование его (такие как `case`) приведет к синтаксической ошибке. Но можно получить эти объекты используя `module-ref`:

```
(macro? (module-ref (current-module) 'case))
⇒ #t
```

`macro-type` *m* [Scheme Procedure]
`scm_macro_type` (*m*) [C Function]
Возвращает тип *type* который был задан, при создании *m*, с помощью преобразователя синтаксиса `make-syntax-transformer`.

`macro-name` *m* [Scheme Procedure]
`scm_macro_name` (*m*) [C Function]
Возвращает имя макроса *m*.

`macro-binding` *m* [Scheme Procedure]
`scm_macro_binding` (*m*) [C Function]
Возвращает привязку макроса *m*.

`macro-transformer` *m* [Scheme Procedure]
`scm_macro_transformer` (*m*) [C Function]
Возвращает преобразователь макроса *m*. Возвращает процедуру, для которой можно запросить документацию(docstring). Вот и вся причина поо которой этот раздел задокументирован. На самом деле это часть результата `macro-binding`.

6.11 Общие Утилиты(Служебные Функции)

Эта глава содержит информацию о процедурах, которые н привязаны к конкретному типу данных. Из-за широкого спектра их приложений, они собраны в главу Утилиты (*utility*).

6.11.1 Равенство

В Scheme есть три типа основных предикатов равенства, описанных ниже. Такие же виды сравнений возникают в других функциях, таких как `memq` и ее друзьях (см. Раздел 6.6.9.7 [List Searching], страница 197).

Для всех трех проверок, объекты разных типов никогда не бывают равными. Так, например экземпляр списка и вектора не равны (not `equal?`), даже если их содержимое одинаково. Точные и не точные числа также рассматриваются как различные типы, и следовательно они не равны, даже если их значения одинаковы.

`eq?` просто проверяет является ли объект тем же самым (по сути, это сравнение указателей). Это быстро, и может использоваться при поиске определенного объекта или при работе с символами или ключевыми словами (которые всегда являются уникальными объектами).

`eqv?` расширяет `eq?` и смотрит на значение чисел и символьных знаков. Это может быть использовано для экземплякров подобно `=` (см. Раздел 6.6.2.8 [Comparison], страница 126) но без ошибки, если операнд не является числом.

`equal?` идет дальше, он просматривает (рекурсивно) содержимое списков, векторов и т.д. Это хорошо, например, для экземпляров списков, которые были прочитаны или вычислены в разных местах и являются теми же самыми, только не состоят из одних и тех же пар. Такие списки выглядят одинаково(когда их печатают) и `equal?` будет считать их равными.

`eq?` *x y* [Scheme Procedure]
`scm_eq_p` (*x*, *y*) [C Function]

Возвращает `#t` если *x* и *y* один и тот же объект, за исключением цифр и символьных знаков. Например,

```
(define x (vector 1 2 3))
(define y (vector 1 2 3))

(eq? x x)  ⇒ #t
(eq? x y)  ⇒ #f
```

Числа и символьные знаки не равны никаким другим объектам, но проблема в том, что они не обязательно равны `eq?` самим себе. Это так, даже когда приходит число прямо из переменной.

```
(let ((n (+ 2 3)))
  (eq? n n))       ⇒ *unspecified*
```

Вообще `eqv?` описанное ниже, следует использовать при сравнении чисел или символьных знаков. `=` (см. Раздел 6.6.2.8 [Comparison], страница 126) or `char=?` (см. Раздел 6.6.3 [Characters], страница 139) тоже может быть спользовано.

Стоит отметить, что конец списка `()`, `#t`, `#f`, символ данного имени, и ключевое слово данного имени, являются уникальными объектами. Для каждого экземпляра это один и тот же объект. Не зависимо от того, как `()` возникает в программе, это один и тот же объект и его можно сравнивать используя `eq?`,

```
(define x (cdr '(123)))
(define y (cdr '(456)))
```

```
(eq? x y) ⇒ #t

(define x (string->symbol "foo"))
(eq? x 'foo) ⇒ #t
```

`int scm_is_eq` (*SCM x, SCM y*) [C Function]

Возвращает `1` когда *x* и *y* являются эквивалентными в смысле `eq?`, иначе возвращает `0`.

Оператор `==` не должен использоваться для значений `SCM`, `SCM` это тип данных си который нельзя непосредственно сравнивать исползуя `==` (см. Раздел 6.3 [The SCM Type], страница 108).

`eqv?` *x y* [Scheme Procedure]
`scm_eqv_p` (*x, y*) [C Function]

Возвращает `#t` если *x* и *y* это один и тот же объект или для символьных знаков и чисел имеющих одно и тоже значение.

На объекты, за исключением символьных знаков и чисел, `eqv?` действует так же как `eq?` выше, сравнение даст истину если *x* и *y* являются одним и темже объектом.

Если *x* и *y* являются числами или символьными знаками, `eqv?` сравнивает их тип и значение. Точные числа не будут равны `eqv?` не точным числам(даже если их значения одинаковы).

```
(eqv? 3 (+ 1 2)) ⇒ #t
(eqv? 1 1.0)     ⇒ #f
```

`equal?` *x y* [Scheme Procedure]
`scm_equal_p` (*x, y*) [C Function]

возвращает `#t` если *x* и *y* одинакового типа и их содержимое или значение являются равным.

Для пар, строк, векторов, массивов или структур, `equal?` сравнивает их содержимое, и делает так используя `equal?` рекурсивно, поэтому может быть пройдена структура произвольной глубины.

```
(equal? (list 1 2 3) (list 1 2 3))   ⇒ #t
(equal? (list 1 2 3) (vector 1 2 3)) ⇒ #f
```

Для других объектов, `equal?` сравнивает как `eqv?` выше, что означает что символьные знакии числа сравниваются по типу и значению (как в `eqv?`, точные и неточные числа не равны - not `equal?`, даже если их значения одинаковы).

```
(equal? 3 (+ 1 2)) ⇒ #t
(equal? 1 1.0)     ⇒ #f
```

Хеш-таблицы в настоящее время сравниваются только в соответствии с `eq?`, поэтому длв разные таблицы не равны(not `equal?`), даже если их содержимое одно и тоже.

`equal?` не поддерживает циклические структуры данных, он может зациклиться, если попросить сравинить два циклическвих списка или аналогичные структуры.

Типы объектов GOOPS (см. Глава 8 [GOOPS], страница 787), включая типы внешних объектов (см. Раздел 5.5 [Defining New Foreign Object Types], страница 79), могут иметь специальную реализацию `equal?` для сравнения двух значений одного типа. Если `equal?` вызван для двух объектов GOOPS одинакового типа, `equal?` отправит вызов на обобщенную функцию. Это позволяет приложению просматривать содержимое или контролировать то, что считается равным `equal?` для двух объектов одинакового типа. Если такого обработчика нет, по умолчанию просто сравнивается как `eq?`.

6.11.2 Object Properties

ІЧасто полезно связать часть дополнительной информации с объектом Scheme даже когда объект не имеет выделенного слота, в котором дополнительная информация может быть сохранена. Свойства объекта позволяют вам сделать это.

В Guile представлении свойства объекта является процедурой с установщиком (см. Раздел 6.9.8 [Procedures with Setters], страница 274) которую можно использовать с обобщенной формой `set!` (REFFIXME) чтобы установить и получить свойство для любого объекта Scheme. Итак, установка свойств выглядит так:

```
(set! (my-property obj1) value-for-obj1)
(set! (my-property obj2) value-for-obj2)
```

и получение значения одного и того же свойства выглядит следующим образом:

```
(my-property obj1)
⇒
value-for-obj1

(my-property obj2)
⇒
value-for-obj2
```

Чтобы создать свойство объекта в первую очередь используйте процедуру `make-object-property`:

```
(define my-property (make-object-property))
```

`make-object-property` [Scheme Procedure]

Создает и возвращает свойство объекта. Свойство объекта является процедурой с установщиком(procedure-with-setter) котороую можно вызвать двумя способами. `(set! (`*`property obj`*`) `*`val`*`)` устанавлилвает свойство *property* объекта *obj* в значение *val*. `(`*`property obj`*`)` возвращает текущее значение свойства *property* объекта *obj* .

Одно свойство объекта созданное с помощью `make-object-property` может связать отдельное свойство значения со всеми значениями Scheme которые различаются по `eq?` (исключая числовые значения).

Внутренне, свойства объекта реализуются с использованием хеш-таблицы слабых ключей. Это означает, что пока значение Scheme связанное со значениями свойств защищены от сборки мусора, значение его свойства также защищены. Когда значение Scheme собирается сборщиком мусора, его запись в таблице свойств удаляется и поэтому значение экс-свойства больше не защищено таблицей.

Guile также реализует более традиционный списковый интерфейс поддерживающий свойства, в котором каждый объект имеет список пар ключ - значение, связанный с ним. Свойства в этом списке обозначены символами. Это устаревший интерфейс; Вы должны использовать слабые хеш-таблицы или свойства объектов вместо него.

`object-properties` *obj* [Scheme Procedure]
`scm_object_properties` (*obj*) [C Function]
Возвращает список свойств *obj*.

`set-object-properties!` *obj alist* [Scheme Procedure]
`scm_set_object_properties_x` (*obj, alist*) [C Function]
Устанавливает список свойств объекта *obj* в *alist*.

`object-property` *obj key* [Scheme Procedure]
`scm_object_property` (*obj, key*) [C Function]
Возвращает свойство объекта *obj* с именем *key*.

`set-object-property!` *obj key value* [Scheme Procedure]
`scm_set_object_property_x` (*obj, key, value*) [C Function]
В списке свойств объекта *obj* устанавливает свойство с именем *key* в значение *value*.

6.11.3 Сортировка

Сортировка очень важна в компьютерных программах. Таким образом, Guile поставляется с несколькими встроенными процедурами сортировки. Как всегда, процедуры заканчивающиеся знаком `!` имеют побочные эффекты, это означает что они могут изменять свои параметры для получения результатов.

Первая группа процедур может использоваться для объединения двух списков (которые должны быть уже отсортированы самостоятельно) и создает отсортированные списки содержащие все элементы входных списков.

`merge` *alist blist less* [Scheme Procedure]
`scm_merge` (*alist, blist, less*) [C Function]
Объединяет два уже отсортированных списка в один. Предоставляются два списка *alist* и *blist*, такие как `(sorted? alist less?)` и `(sorted? blist less?)`, возвращается новый список состоящий из элементов *alist* и *blist* чередующихся так чтобы остаться отсортированными `(sorted? (merge alist blist less?) less?)`. Примечание: функция не принимает векторы!

`merge!` *alist blist less* [Scheme Procedure]
`scm_merge_x` (*alist, blist, less*) [C Function]
Принимает два уже отсортированных списка *alist* и *blist*, такие как `(sorted? alist less?)` и `(sorted? blist less?)`, возвращается новый список состоящий из элементов *alist* и *blist* чередующихся так чтобы остаться отсортированными `(sorted? (merge alist blist less?) less?)`. Это деструктивный вариант слияния. Примечание: функция не принимает векторы!

Следующие процедуры могут работать с последовательностями, которые являются векторами или списками. Согласно предоставленным аргументам, онги возвращают

отсортированные вектора или списки, соответственно. Первая из следующих процедур определяет, отсортирована ли последовательность, другая сортирует заданную последовательность. Варианты с именами, начинающимися с `stable-`, отличаются тем, что они поддерживают специальное свойство входных последовательностей: если два или более элемента являются уже соответствующими предикату сравнения, они остаются в том же порядке, в котором они появились на входе.

`sorted?` *items less* [Scheme Procedure]
`scm_sorted_p` (*items, less*) [C Function]
Возвращает `#t` если *items* является списком или вектором таким, что для каждого элемента *x* и следующего элемента *y* из *items*, функция `(less y x)` возвращает `#f`. В противном случае возвращает `#f`.

`sort` *items less* [Scheme Procedure]
`scm_sort` (*items, less*) [C Function]
Сортирует элементы последовательности *items*, которые могут быть списком или вектором. Процедура *less* исползуется для сравнения элементов последовательности. Это не стабильная сортировка.

`sort!` *items less* [Scheme Procedure]
`scm_sort_x` (*items, less*) [C Function]
Сортирует элементы последовательности *items*, которые могут быть списком или вектором. Процедура *less* исползуется для сравнения элементов последовательности. Это деструктивная сортировка, это означает, что входная последовательность модифицируется для получения отсортированного результата. Это не стабильная сортировка.

`stable-sort` *items less* [Scheme Procedure]
`scm_stable_sort` (*items, less*) [C Function]
Сортирует последовательность *items*, которая может быть списком или вектором. *less* используется для сравнения элементов последовательности. Это стабильная сортировка.

`stable-sort!` *items less* [Scheme Procedure]
`scm_stable_sort_x` (*items, less*) [C Function]
Сортирует последовательность *items*, которая может быть списком или вектором. *less* используется для сравнения элементов последовательности. Это деструктивная сортировка, это означает, что входная последовательность модифицируется для получения отсортированного результата. Это стабильная сортировка.

Процедуры в последней группе принимают в качестве входных данных только списки или векторы, на что укзаывают их имена.

`sort-list` *items less* [Scheme Procedure]
`scm_sort_list` (*items, less*) [C Function]
Сортирует список *items*, используя *less* для сравнения элементов списка. Это стабильная сортировка.

`sort-list!` *items less* [Scheme Procedure]
`scm_sort_list_x` (*items, less*) [C Function]
Сортирует список *items*, используя *less* для сравнения элементов списка. Это деструктивная сортировка, это означает, что входной список модифицируется для получения отсортированного результата. Это стабильная сортировка.

`restricted-vector-sort!` *vec less startpos endpos* [Scheme Procedure]
`scm_restricted_vector_sort_x` (*vec, less, startpos, endpos*) [C Function]
Сортирует вектор *vec*, исползуя *less* для сравнения элементов вектора. *startpos* (включая) и *endpos* (исключая) разграничивают диапазон вектора, который сортируется. Возвращаемое значение не определено.

6.11.4 Копирования Глубоких Структур

Процедуры копирования списков (см. Раздел 6.6.9 [Lists], страница 192) дают только поверхнустную копию входного списка, и в настоящее время Guile даже не содержит процедур для копирования векторов. `copy-tree` может быть использована для глубинного копирования, так как она не только копирует верхнуюю структуру списка, но также копирует любые пары во входных списках.

`copy-tree` *obj* [Scheme Procedure]
`scm_copy_tree` (*obj*) [C Function]
Рекурсивно копирует дерево данных, которое связано с *obj*, и возвращает новую структуру данных. `copy-tree` рекурсивно просматривает содержимое как пар, таки векторов (так как обе ячейки cons и векторные ячейки могут указывать на произвольные объекты), и останавливает повторение при попадании в любой другой объект.

6.11.5 Общее преобразование в Строки

При отладке программ Scheme, а также для обеспечения удобного для пользователя интерфейса, процедура препобразования произвольного объекта Scheme в формат строки очень полезна. Преобразование из/в строки конечно может быть выполнено с помощью специальных процедур, когда тип данных объекта предназначенного для преобразования известен, но этой процедурой очень часто гораздо удобнее пользоваться.

`object->string` преобразует объект, используя процедуру печати для записи строки в порт, и возвращает результирующую строку. Преобразование в объект обратно из строки возможно только в том случае, если тип объекта имеет синтаксис чтения, и синтаксис чтения сохраняется процедурой печати.

`object->string` *obj* [*printer*] [Scheme Procedure]
`scm_object_to_string` (*obj, printer*) [C Function]
Возвращает строку Scheme, полученную при печати *obj*. Функция печати может быть указана необязательным вторым аргументом *printer* (по умолчанию: `write`).

6.11.6 Хуки(К(Х)рюки, ;-) Крючки, Зацепки)

Хук это список процедур, которые должны вызваться в четко определенные моменты времени. Как правило, приложение предоставляет хук *h* и обещате своим пользо-

вателям, что оно вызовет все процедуры из h в определенный момент времени при выполнении приложения. Добавляя свою сообственную процедуру к h, пользователь приложения может подключиться или даже повлиять на ход выполнения приложения.

Сам Guile предоставляет несколько таких хуков для отладки и настройки: они перечислены в подразделе ниже.

Когда приложение впервые создает хук, оно должно знать, сколько аргументов будет передаваться процедурам из хука, когда хук будет запускаться. Выбранное количество аргументов (которых может и не быть) объявляется при создании хука, и все процедуры, которые добавляются к этому хуку должны быть способны принимать такое количество аргументов.

Хук создается с помощью `make-hook`. Процедура может быть добавлена или удалена из хука использованием `add-hook!` или `remove-hook!`, и все процедуры хука можно удалить используя `reset-hook!`. Когда приложению необходимо запустить хук, оно делает это используя `run-hook`.

6.11.6.1 Пример Использования Хука

Использование хуков показано на нескольких примерах в этом разделе. Сначала мы определяем хук арности(размерности) 2 — т.е. процедуры хранящиеся в хуке должны будут принимать два аргумента.

```
(define hook (make-hook 2))
hook
⇒ #<hook 2 40286c90>
```

Теперь мы готовы добавить несколько процедур к вновь созданному хуку с помощью `add-hook!`. В следующем примере добавлены две процедуры, которые печатают разные сообщения и делают разные вещи со своими аргументами.

```
(add-hook! hook (lambda (x y)
                  (display "Foo: ")
                  (display (+ x y))
                  (newline)))
(add-hook! hook (lambda (x y)
                  (display "Bar: ")
                  (display (* x y))
                  (newline)))
```

Как только процедуры были добавлены, мы можем вызвать хук используя `run-hook`.

```
(run-hook hook 3 4)
⊣ Bar: 12
⊣ Foo: 7
```

Обратите внимание, что процедуры вызываются в обратном порядке, в котором они были добавлены. Это потому, что стандартным поведением функции `add-hook!` является добавление процедуры в начало(*front*) списка процедур хука. Вы можете заставить `add-hook!` добавить вашу процедуру в конец (*end*) списка, предоставив третий аргумент равный `#t` во втором вызове `add-hook!`.

```
(add-hook! hook (lambda (x y)
```

```
                    (display "Foo: ")
                    (display (+ x y))
                    (newline)))
(add-hook! hook (lambda (x y)
                    (display "Bar: ")
                    (display (* x y))
                    (newline))
                    #t)             ; <- Change here!

(run-hook hook 3 4)
⊣ Foo: 7
⊣ Bar: 12
```

6.11.6.2 Ссылка на Хук

Когда вы создаете хук с помощью `make-hook`, вы можете указать арность (размерность) процедур, которые можно добавить к хуку. Если арность не указана в качестве аргумента `make-hook`, по умолчанию она будет равно нулю. Все процедуры данного хука должны быть одинаковой арности, и когда процедуры вызываются с использованием `run-hook`, количество передаваемых аргументов должно соответствовать арности указанной во время создания хука.

Порядок в котором процедуры добавляются к хуку имеет значение. Если третий параметр `add-hook!` пропущен или равен `#f`, процедура добавляется перед процедурами которые уже были добавлены к хуку, в противном случае процедура добавляется в конец. Процедуры всегда вызываются с начала и до конца списка, когда они вызываются через `run-hook`.

Порядок процедур в списке, возвращаемый `hook->list` соответствует порядку в котором эти процедуры будут вызваны, если хук будет запущен с использованием `run-hook`.

Обратите внимание, что функции Си в следующих записях предназначены для обработки хуков уровня схемы(*Scheme-level*) на Си. Существуют также хуки уровня Си(*C-level*) которые имеют собственный интерфейс (см. Раздел 6.11.6.3 [C Hooks], страница 309).

`make-hook` [*n_args*] [Scheme Procedure]
`scm_make_hook` (*n_args*) [C Function]
Создает хук для хранения процедур размерности(арности) *n_args*. *n_args* по умолчанию равен нулю. Возвращаемое значение является объектом хуком, который будет использоваться с другими процедурами хука.

`hook?` *x* [Scheme Procedure]
`scm_hook_p` (*x*) [C Function]
Возвращает `#t` если *x* это хук, `#f` иначе.

`hook-empty?` *hook* [Scheme Procedure]
`scm_hook_empty_p` (*hook*) [C Function]
Возвращает `#t` если *hook* это пустой хук, `#f` иначе.

`add-hook!` *hook proc* [*append_p*] [Scheme Procedure]
`scm_add_hook_x` (*hook, proc, append_p*) [C Function]
Добавляет процедуру *proc* к хуку *hook*. Процедура добавляется в конец если *append_p* равен истине, иначе она добавляется в начало. Возвращаемое значение этой процедурой не указано.

`remove-hook!` *hook proc* [Scheme Procedure]
`scm_remove_hook_x` (*hook, proc*) [C Function]
Удаляет процедуру *proc* из хука *hook*. Возвращаемое значение этой процедурой не указано.

`reset-hook!` *hook* [Scheme Procedure]
`scm_reset_hook_x` (*hook*) [C Function]
Удаляет все процедуры из хука *hook*. Возвращаемое этой процедурой значение не указано.

`hook->list` *hook* [Scheme Procedure]
`scm_hook_to_list` (*hook*) [C Function]
Преобразует список процедур хука *hook* в список.

`run-hook` *hook arg . . .* [Scheme Procedure]
`scm_run_hook` (*hook, args*) [C Function]
Применяет все процедуры из хука *hook* к аргументам *arg . . . .* Порядок применения процедур от первого к последнему. Возвращаемое этой процедурой значение не указано.

Если , в коде Сие, вы уверены что есть объект хук и правильно сформированный список аргументов для хука, вы можете также использовать `scm_c_run_hook`, который идентичен to `scm_run_hook` но не проверяет типы.

`void scm_c_run_hook` (*SCM hook, SCM args*) [C Function]
То же, что и `scm_run_hook` но без проверки типа для подтверждения того что *hook* на самом деле является хук объектом и этот *args* представляет собой правильно сформированный список соответствующей арности хука.

Для Си кода `SCM_HOOKP` более быстрая альтернатива `scm_hook_p`:

`int SCM_HOOKP` (*x*) [C Macro]
Возвращает 1 если *x* это хук уровня Схемы(Scheme-level), иначе 0.

6.11.6.3 Хуки для Си кода.

Уже описанные хуки предназначены для исполнения процедур на уровне Схемы (Scheme-level). В дополнении к ним, библиотека Guile предоставляет назависимый набор интерфейсов для создания и манипулирования хуками, которые предназначены для заполнения функциями, реализоованными на Си.

Первоначальная мотивация здесь заключается в том, чтобы обеспечить совего рода хук, который можно было бы безопасно использовать в разные моменты во время сборки мусора. Хуки уровня Схемы(Scheme-level) для этого не подходят, цель их запуска сама по себе может потребовать выделения памяти, что приведет к вызову

сборщика мусора рекурсивно . . . Тем не менее, это также тот случай, когда эти хуки легче работают с уровнем Схемы (Scheme-level), если вы хотите зарегистрировать только функции Си. Так что если это в основном то, что должен делать ваш код, вы можете предпочесть использовать этот интерфейс.

Для создания Си хука, вы должны выделить хранилище для структуры типа `scm_t_c_hook` и инициализировать ее используя `scm_c_hook_init`.

`scm_t_c_hook` [C Type]
Тип данных для Си хука. Внутреннее устройство этого типа должно рассматриваться кака не прозрачное, не видимое.

`scm_t_c_hook_type` [C Enum]
Перечисление возможных типов хуков, как то:

`SCM_C_HOOK_NORMAL`
Тип хука, для которого всегда будут вызываться все зарегистрированные функции.

`SCM_C_HOOK_OR`
Тип хука, для которого будет вызываться последовательность зарегистрированных функций только до тех пор, пока одна из них не вернет Си истину(т.е не нулевое значение).

`SCM_C_HOOK_AND`
Тип хука, для которого будет вызываться последовательность зарегистрированных функций только до тех пор, пока одна из них не вернет Си ложь(т.е ноль)

`void scm_c_hook_init` (*scm_t_c_hook *hook, void *hook_data, scm_t_c_hook_type type*) [C Function]
Инициализирует Си хук в памяти указываемой *hook*. *type* должен быть одним из значений перечисления `scm_t_c_hook_type`, и управляет тем как функции хука будут вызываться. *hook_data* это параметр замыкания который будет передан всем зарегистрированным функциям хука, когда они вызываются.

Чтобы добавить или удалить Си функцию из Си хука, используйте `scm_c_hook_add` или `scm_c_hook_remove`. Функция хука должна ожидать три параметра `void *`, соответственно:

hook_data Данные замыкания хука, которые были указаны во время инициализации хука `scm_c_hook_init`.

func_data Данные замыкания функции, которые были указаны, когда эта функция была зарегистрирована с хуком с помощью `scm_c_hook_add`.

data Данные замыкания вызова указанные в вызове `scm_c_hook_run`, который запускает хук.

`scm_t_c_hook_function` [C Type]
Тип функции для Си функций хук: получает три параметра `void *` и возвращает результат `void *`.

`void scm_c_hook_add` (*scm_t_c_hook *hook, scm_t_c_hook_function func, void *func_data, int appendp*) [C Function]

Добавляет функцию *func*, с данными замыкания функции *func_data*, в Си хук *hook*. Новая функция добавляется в список функций хука если *appendp* не равен нулю, в противном случае предваряется списком.

`void scm_c_hook_remove` (*scm_t_c_hook *hook, scm_t_c_hook_function func, void *func_data*) [C Function]

Удаляет функцию *func*, с данными замыкания функции *func_data*, из Си хука *hook*. `scm_c_hook_remove` проверяет оба *func* и *func_data* что позволяет для одной и той же функции *func* быть зарегистрированной во множестве мест с различными данными замыкания.

Наконец, чтобы вызвать Си хук. вызовите функцию `scm_c_hook_run` указав хук и данные замыкания вызова для его работы:

`void * scm_c_hook_run` (*scm_t_c_hook *hook, void *data*) [C Function]

Запуск Си хука *hook* вызовет замыкание данных *data*. С учетом вариантов для типа хука `SCM_C_HOOK_OR` и `SCM_C_HOOK_AND`, `scm_c_hook_run` вызывая зарегистрированные в хуке *hook* функции, передавая им данные замыкания хука, данные замыкания каждой функции, и данные замыкания вызова.

`scm_c_hook_run` возвращаемое значени является возвращенным значением последней вызванной функции.

6.11.6.4 Хуки для Сборщика Мусора (GC)

Всякий раз, когда Guile выполняет сборку мусора, он вызывает следующие хуки в указанном порядке.

`scm_before_gc_c_hook` [C Hook]

Вызывает Си хук в самом начале сборки мусора, после установки `scm_gc_running_p` в 1, но перед входом GC в критическую секцию.

Если сборка мусора заблокирована, потому что `scm_block_gc` не равен нулю, GC завершает работу раньше, вскоре после вызова этого хука, больше никаких хуков не будет вызвано.

`scm_before_mark_c_hook` [C Hook]

Хук Си вызывается перед началом фазы пометки сборщика мусора, после тогок как поток GC вошел в критическую секцию.

`scm_before_sweep_c_hook` [C Hook]

Си хук вызывается перед началом фазы оочистки сборщиком мусора. Это тоже же, что и конец фазы маркировки, так что ничего не происходит между фазой маркировки и очистки.

`scm_after_sweep_c_hook` [C Hook]

Си хук вызывается после завершения фазы очистки сборщиком мусора, но пока поток GC все еще находиться в критической секции.

`scm_after_gc_c_hook` [C Hook]

Хук Си вызывается в самом конце сборки мусора, после того как поток GC покинул свою критическую секцию.

`after-gc-hook` [Scheme Hook]

Хук Scheme с арностью 0. Этот хук запускается асинхронно (см. Раздел 6.22.3 [Asyncs], страница 466) вскоре после завершения GC и любых других событий, которые были отложены во время процесса сборки мусора. (Также доступно из Си с именем `scm_after_gc_hook`.)

Все перечисленные здесь хуки имеют тип `SCM_C_HOOK_NORMAL`, и инициализируются с данными замыкания хука NULL, и вызываются `scm_c_hook_run` с данными замыкания вызова NULL.

Хук Scheme `after-gc-hook` особенно полезен в сочетании с защитниками (guardians) (см. Раздел 6.19.4 [Guardians], страница 430). Как правило, если вы используете защитника, вы хотите вызвать хранителя после сборки мусора, что бы увидеть есть ли какие либо объекты, добавленные в защитника были собраны. Добавляя чанк(thunk), который выполняет вызов к `after-gc-hook`, вы можете убедиться, что ваш защитнику проверяется после каждого цикла сборки мусора.

6.11.6.5 Хуки в Guile REPL

Документации по ним нет, но выполнив `grep hook *` в директории guile/2.2/system/repl можно обнаружить следующие хуки:

```
command.scm:                (run-hook before-print-hook x)
debug.scm:                  (run-hook before-print-hook val)
describe.scm:               (cons <hook>      "a hook")
error-handling.scm:         (run-hook before-error-hook)
error-handling.scm:         (run-hook after-error-hook)
repl.scm:                   (run-hook before-eval-hook exp)
```

которые можно попытаться использовать.

6.12 Definitions and Variable Bindings

Scheme поддерживает определение переменных в разных контекстах. Переменные могут быть определены на верхнем уровне, чтобы они были видны во всей программе и переменные могут быть определены локально в процедурах и выражениях. Это важно для модульности и абстракции данных.

6.12.1 Top Level Variable Definitions

На верхнем уровне программы (т.е не вложенной в другое выражене), определение имеет форму:

```
(define a value)
```

определение переменной вызывает `a` и устанавливает значение *value*.

Если переменная уже существует в текущем модуле, поскольку она уже создана в предыдущем выражении с тем же именем, ее значение просто изменяется на новое значение. В этом случае указанная выше форма полностью эквивалентна

```
(set! a value)
```

Эта эквивалентность означает, что `define` может использоваться взаимозаменяемо с `set!` для изменения значений переменных на верхнем уровне REPL или исходном файле Scheme. Это полезно во время интерактивной разработки при перезагрузке файла Scheme который вы изменили, поскольку это позволяет определить выражения в этом файле, так что бы они корректно работали как при загрузке файла в первый, так и при последующих загрузках

Заметьте однако, что `define` и `set!` не всегда эквивалентны. На пример, `set!` не доступна если именнованная переменная еще не существует.И эти два выражения могут вести себя иначе, в случае когда импортируемые переменные видны из другого модуля.

`define` *name value* [Scheme Syntax]

Создадим переменную верхнего уровня с именем *name* и значением *value*. Если имя переменной уже существует, то значение переменной просто изменится. ВозвращаемоеThe значекние выражения `define` не определено.

В C API эквивалентом `define` выступают `scm_define` и `scm_c_define`, которые отличаются друг от друга определением имени переменной, в первом это `SCM` символ, а во втором Си строка оканчивающаяся нулем.

`scm_define` (*sym, value*) [C Function]
`scm_c_define` (*const char *name, value*) [C Function]

Си эквивалент `define`, с именем переменной определенным как *sym* символ, или как *name*, заканчивающаяся нулем Си строка. Оба варианта возвращают новый или существующий объект переменной.

`define` (когда это происходит на верхнем уровне), `scm_define` и `scm_c_define` все создают или устанавливают значение переменной среды верхнего уровня текущего модуля. Если еще не было переменной с узазанным именем, принадлежащей к текущему модулю, но аналогично именованная переменная из другого модуля была видна через импорт, то недавно созданная переменная в текущем модуле перекроет импортированную переменную, так что импортированная переменная больше не будет видна.

Внимание: Определения в Scheme связываний внутри локальных конструкций (см. Раздел 6.12.2 [Local Bindings], страница 314) действуют по разному см. (см. Раздел 6.12.3 [Internal Definitions], страница 315).

Многие люди используют стиль разработки добавляя и изменяя определения во время выполнения программы. Встраивая их в свою программу, но не презапуская ее. (Вы можете сделать это используя команду `reload-module`, команду `reload` REPL, процедуру `load`, или просто вставляя код в REPL.) Если вы один из этих людей, вы обранужите что иногда есть некоторые переменные, которые вы не хотите переопределять все время. Для этого используется `define-once`.

`define-once` *name value* [Scheme Syntax]

Создает переменную верхнего уровня с именем *name* и значением *value*, но только если имя *name* еще не используется в текущем модуле.

Старые лисперы, вероятно знают `define-once` под лисп именем, `defvar`.

6.12.2 Связывание Локальных Переменных(Local Variable Bindings)

В отличии от определений на верхнем уровне, которые создают привзяки, которые видны всему коду в модуле, также можно определять переменные, которые видны только в четко определенной части программы. Обычно эта часть программы будет процедурой или подвыражением процедуры.

С помощью конструкций для локального связывания (`let`, `let*`, `letrec`, и `letrec*`), язык Scheme имеет получает блочную структуру, как и большинство других языков программирования со времен Algol 60. Читатели, знакомые с такими языками как C или Java уже должны быть знакомы с этой концепцией, но у семейства выражений `let` есть несколько свойств, которые стоит знать.

Самой базовой конструкцией создания локальных привязок является `let`.

`let` *bindings body* [syntax]

bindings has the form

```
((variable1 init1) ...)
```

это ноль или больше двух-элементных списков, каждый из которых состоит из переменной и произвольного выражения. Все имена переменных *variable* должны быть разными.

Все выражения `let` вычисляются следующим образом.

- Вычисляются все выражения *init*.
- Выделяется новое хранилище для переменных *variables*.
- Значения выражений *init* сохраняюся в переменных.
- Выражения в теле(*body*) вычислляются по порядку, и значение последнего выражения возвращается как значение выражения `let`.

Выражения *init* не могут ссылаться ни на одну из локальных переменных *variables*.

Другие конструкции создания привязок - это вариации на одну и туже тему: создания новых значений, связывания их с переменными и вычисления тела в этом новом расширенном лексическом контексте.

`let*` *bindings body* [syntax]

Аналогично `let`, но привязка переменных выполняется последовательно, так что все выражения *init* могут использовать переменные определенные левее в списке определения привязок.

Выражение `let*` всегда может быть выражено вложенными выражениями `let`.

```
(let* ((a 1) (b a))
   b)
≡
(let ((a 1))
  (let ((b a))
    b))
```

`letrec` *bindings body* [syntax]

Подобно `let`, но можно ссылаться на переменные *variable* из лямбда выражений созданных в любом из *inits*. То есть, процедуры созданные в выражении *init* могут рекурсивно ссылаться на определенные переменные.

```
(letrec ((even? (lambda (n)
                  (if (zero? n)
                      #t
                      (odd? (- n 1)))))
         (odd? (lambda (n)
                  (if (zero? n)
                      #f
                      (even? (- n 1))))))
  (even? 88))
⇒
#t
```

Обратите внимание, что хотя выражения *init* могут ссылаться на новвые переменные, они не могут получить доступ к их значениям. Например, созданная выше функция `even?` создает замыкание (см. Раздел 3.4 [About Closure], страница 27) ссылающеся на переменную `odd?`. Но `odd?` не может быть вызвана(а значит и вызвана `even?`) до тех пор пока выполнение не перейдет в тело(body).

`letrec*` *bindings body* [syntax]

Аналогично `letrec`, за исключением того, что выражения *init* связываются со своими переменными по порядку.

Таким образом `letrec*` ослабляет ограничения letrec, так как более поздние выражения *init* могут ссылаться на значения ранее связанных переменных. variables.

```
(letrec ((a 42)
         (b (+ a 10)))  ;; Illegal access
  (* a b))
;; The behavior of the expression above is unspecified

(letrec* ((a 42)
          (b (+ a 10)))
  (* a b))
⇒ 2184
```

Существует так же альтернативная форма формы `let`, которая исползуется для выражения итерации. Из-за использования в качестве циклической конструкции, эта форма именованного let ((the *named let*)) задокументирована в разделе рассказывающем об итерациях (см. Раздел 6.13.4 [while do], страница 321)

6.12.3 Внтуренние определения

Форма `define`, которая появляется внутри тела `lambda`, `let`, `let*`, `letrec`, `letrec*` или эквивалентного выражения называется внутренним определением(*internal definition*). Внутреннее определение отличается от определения верхнего уровня (см.

Раздел 6.12.1 [Top Level], страница 312), поскольку это определение видно только внутри тела вмещающей формы. Давайте рассмотрим следующий пример.

```
(let ((frumble "froz"))
   (define banana (lambda () (apple 'peach)))
   (define apple (lambda (x) x))
   (banana))
⇒
peach
```

Здесь вмещающая форма - это `let`, поэтому `define` в теле `let` являются внутренними определениями. Поскольку область действия внутренних определений является ВСЕ(**complete**) тело выражения `let`, `lambda`-выражение, котолрое связывается с переменной `banana` может ссылаться на переменную `apple`, хотя его определение появляется лексически после(*after*) определения `banana`. Это происходит потому, что последовательность внутреннего определения действует так, как будто это выражение `letrec*`.

```
(let ()
  (define a 1)
  (define b 2)
  (+ a b))
```

этквивалентно

```
(let ()
  (letrec* ((a 1) (b 2))
    (+ a b)))
```

Внутренние определения допускаются только в начале тела включающего выражения. Они не могут быть смешаны с другими выражениями.

Другое примечательное отличие от определений верхнего уровня состоит в том, что в пределах одной группы внутренних определений определения всех имен переменных должны быть разными. Это означает, что на верхнем уровне второе определение для данной переменной действует как `set!`, а для внутренних определений с повторной привязкой выбрасывается исключение.

Историческое замечание, раньше внутренние привязки расширялись в терминах `letrec`, а не `letrec*`. Такая ситуация была для отчета R5RS и более ранних. Тем не менее в R6RS, было признано, что последовательное определение является интуитивным расширением, как в следующем случае:

```
(let ()
  (define a 1)
  (define b (+ a a))
  (+ a b))
```

Guile решил следовать R6RS в этом отношении, и теперь расширяет внутренние определения используя `letrec*`.

6.12.4 Запрос о связанных переменных

Guile предоставляет процедуру для проверки того, связан ли символ в среде верхнего уровня.

`defined?` *sym* [*module*] [Scheme Procedure]
`scm_defined_p` (*sym, module*) [C Function]

Возвращает `#t` если *sym* определен в модуле *module* или текущем модуле, когда *module* не указан; в противном случае возвращается `#f`.

6.12.5 Связываение нескольких возвращаемых значений.

`define-values` *formals expression* [Syntax]

Выражение *expression* вычисляется, и переменные *formals* связываются с возвращаемыми значениями , также как formals в `lambda` выражении, сопоставляются с аргументами при вызове процедуры.

```
(define-values (q r) (floor/ 10 3))
(list q r) ⇒ (3 1)

(define-values (x . y) (values 1 2 3))
x ⇒ 1
y ⇒ (2 3)

(define-values x (values 1 2 3))
x ⇒ (1 2 3)
```

6.13 Управление потоком выполнения Программы

См. Раздел 5.4.3 [Control Flow], страница 73, для обсуждения того, как более общий контроль потока управления Scheme влияет на Си код.

6.13.1 Последовательности(Sequencing) и Соедениения(Splicing)

В качестве выражения, синтаксис `begin` исползуется для вычисления последовательности подвыражений в установленном порядке. Ниже рассмотрим условное выражение:

```
(if (> x 0)
    (begin (display "greater") (newline)))
```

Если test истинен, мы отобразим “greater” для текущего порта вывода, а затем отобразим новую строку. Мы используем `begin` для формирования составного выражения из этой последовательности подвыражений.

`begin` *expr* ... [syntax]

Выражение(я) вычисляются в порядке слева на право и значение последнего выражения возвращается как значение выражения `begin`. Этот тип выражения используется когда выражения перед последним выражением выражением вычисляются из за их "сторонних эффектов".

Синтаксис `begin` играет другую роль в контектсе определений (см. Раздел 6.12.3 [Internal Definitions], страница 315). Форма `begin` в контексте определений объединяет(*splices*) свои подчиненные формы в одном месте. Например, рассмотрим следующую процедуру.:

```
(define (make-seal)
```

```
(define-sealant seal open)
(values seal open))
```

Давайте предположим, что существует макрос `define-sealant` который расширяется до некоторого определения обернутого в `begin`, подобно этому:

```
(define (make-seal)
  (begin
    (define seal-tag
      (list 'seal))
    (define (seal x)
      (cons seal-tag x))
    (define (sealed? x)
      (and (pair? x) (eq? (car x) seal-tag)))
    (define (open x)
      (if (sealed? x)
          (cdr x)
          (error "Expected a sealed value:" x))))
  (values seal open))
```

Здесь, посокльку `begin` находиться в контексте определения, его подчиненные формы объединены(*spliced*) внутри `begin`. Это позволяет определениям, созданным макросом быть видимыми для последующих выражений, формы `values`.

Это прекрасный момент, но соединение(splicing) и последовательность (sequencing) отличаются. Может иметь смысл соединять нулевые формы, потому что может иметь смысл иметь ноль внутренних определений перед выражениями в процедуре или лексической форме связывания. Однако не имеет смысла иметь последовательность нулевых выражений, потому что в этом случае не было бы понятно, какое значение должна иметь последовательность, поскольку в последовательности нулевых выражений не может быть последнего значения. Последовательность нулевых выражений является ошибкой.

В некотором смысле было бы более элегантно устранить объединение(splicing) из языка Scheme, и без макросов (см. Раздел 6.10 [Macros], страница 276), это было бы хорошей идеей. Но полезно иметь возможность писать макросы, которые расширяются до множественных определений, как в `define-sealant` выше, поэтому Scheme злоупотребляет формой `begin` для этих двух задач.

6.13.2 Простое Условное Вычисление

Guile предоставляет три синтаксические конструкции для условных вычислений. `if` это обычное выражение if-then-else (с необязательной ветвью else), `cond` является условным выражением с множеством ветвей и `case` выбирающее ветвь, если ключевое выражение имеет значение из набора константных значений в ветвях.

`if` *test consequent* [*alternate*] [syntax]

Все аргументы могут быть произвольными выражениями. Сначала, вычисляется *test*. Если он возвращает истинное значение, вычисляется выражение *consequent*, а *alternate* игнорируется. Если *test* вычисляется в значение `#f`, вычисляется только *alternate*. Значения вычисляемой ветви (*consequent* или *alternate*) возвращаются как значения выражения `if`.

Когда *alternate* пропущена и *test* вычисляется в `#f`, значение выражения не определено.

Когда вы идете на написание `if` без альтернативы (однорукое условие *one-armed* `if`), вас не волнует, возвращаемое значение(значения) возвращаемые этим выражением. Таким образом, вас больше интересует эффект(*effect*) от вычисления последующего(consequent) выражения. (По соглашени, мы используем слово оператор(*statement*) для ссылки на выражение которое вычисляется ради эффекта, а не ради значения).

В таком случае, считается более ясным выразить эти намерения с помощью специальных форм `when` и `unless`. В качестве дополнительного бонуса, эти формы принимают множественные операторы(statements) для вычисления, которые неявно обернуты в форму `begin`.

`when` *test statement1 statement2 ...* [Scheme Syntax]
`unless` *test statement1 statement2 ...* [Scheme Syntax]

Фактическое определение этих форм во многом является их наиболее ясной документацией:

```
(define-syntax-rule (when test stmt stmt* ...)
  (if test (begin stmt stmt* ...)))

(define-syntax-rule (unless test stmt stmt* ...)
  (if (not test) (begin stmt stmt* ...)))
```

Можно сказать, `when` вычисляет последующие операторы по порядку,если значение *test* истинно. `unless` наоборот: вычисляет операторы если занчение *test* ложно.

`cond` *clause1 clause2 . . .* [syntax]

Каждое предложение `cond`-clause должно выглядеть так:

(*test expression ...*)

где *test* и *expression* произвольные вывражения, или так

(*test => expression*)

где *expression* должно вычислять как процедуру.

test предложений вычисляются по порядку, и как только один из них вычислит значение истина, вычисляется соответствующие выражения *expression* по порядку, а значение последнего выражения возвращается как значение всего выражения `cond`. Для передложения типа `=>`, вычисляется выражение *expression* и результирующая процедура применяется к значению *test*. Результат этого приенения процедуры явлляется результатом всего выражения `cond`.

Одно дополнительное предложение `cond`-clause доступно как расширение стандартной Scheme:

(*test guard => expression*)

где *guard* и *expression* должны вычисляться как процедуры. Для предложения этого типа, *test* может возвращать множественные значекния и `cond` игнорирует его логическое состояни; вместо этого, `cond` вычисляет *guard* применяя его к результирующим значение(я) *test*, как если бы *guard* была потребителем

(*consumer*) аргументов в `call-with-values`. Если результат вызова процедуры возвратит истинное значение, вычисляется выражение *expression* и результирующая процедура пременяется к значению(ям) выражения *test*, также как вызывалась процедура *guard*.

Тестом *test* последнего предложения *clause* может быть символ `else`. Затем, если ни один из предшествующих тестов *test* не равен истине, вычисляются выражения *expression* следующие за `else` и получаемый результат будет результатом выражения `cond`.

`case` *key clause1 clause2 ...* [syntax]

key может быть выражением, а *clause* должны иметь вид

```
((datum1 ...) expr1 expr2 ...)
```

или

```
((datum1 ...) => expression)
```

и последнее *clause* может иметь форму

```
(else expr1 expr2 ...)
```

или

```
(else => expression)
```

Все *datum* должны быть разными. Сначала вычисляется *key*. Результат этого вычисления сравнивается со всеми *datum* значениями используя `eqv?`. Когда сравнение заканчивается успехом, выражения следующее за *datum* вычисляются слева на право, результирующее значение последнего выражения является результатом всего выражения `case`.

Если *key* не соответствует ни одному *datum* и существует предложение `else`, вычисляются следующие за `else` выражения. Если такого предложения нет, результат выражения не определен.

Для типа предложений `=>`, вычисляется *expression* и результирующая процедура применяется к занчению *key*. Результат применения этой процедуры является результатом всего выражения `case`.

6.13.3 Условное вычисление последовательности выражений

`and` и `or` вычисляют все свои аргументы в порядке аналогичном `begin`, но вычисление останавливается как только одно из выражений вычисляется как ложное или истинное, соответственно.

`and` *expr ...* [syntax]

Вычисляет выражения формы *expr* слева на право и останавливает вычисления как только одно из выражений вычисляется в `#f` остальные выражения не вычисляются; Значение последнего вычисленного выражения возвращается Если никакое выражение не вычисляется как `#f`, возвращается значение последнего выражения.

Если используется без выражений, возвращает `#t`.

`or` *expr ...* [syntax]

Вычисляет вывражения формы *expr*s слева на право и останавливает вычисления как только выражение возвращает истинное значение (т.е. значение отличное от `#f`); остальные выражения не вычисляются. Значение последнего вычисленного выражения возвращается. Если все выражения вычисляются в `#f`, возвращается `#f`.

Если используется без выражений, возвращает `#f`.

6.13.4 Механизмы Итерации

Scheme имеет несколько механизмов итерации, главным образом потому, что итерация в программах Scheme обычно выражается с помощью рекурсии. Тем не менее, R5RS определяет конструкцию для программирования циклов, называемый `do`. Кроме того, Guile имеет явный синтаксис цикла, называемый `while`.

`do` ((*variable init* [*step*]) ...) (*test expr ...*) *body ...* [syntax]

Связывает переменные *variable*s и вычисляет тело *body* до тех пор пока *test* не вернет истину. Возвращает значение выполнения последнего выражения *expr* следующих после *test*, если они есть. Простой пример проилюстрирует основную форму,

```
(do ((i 1 (1+ i)))
    ((> i 4))
  (display i))
⊣ 1234
```

Или с двумя переменными и возвращаемым окончательным значением,

```
(do ((i 1 (1+ i))
     (p 3 (* 3 p)))
    ((> i 4)
     p)
  (format #t "3**~s is ~s\n" i p))
⊣
3**1 is 3
3**2 is 9
3**3 is 27
3**4 is 81
⇒
243
```

Привязки переменных *variable* устанавливаются так же как `let`, в котором все выражения вычисляются и все привязки выполяются. При итерации, необязательные выражения *step* выполняются с предыдущими привязками в области видимости, при этом создаются новые привязки.

Выражение *test* является условием завершения. Цикл останавливается когда значение выражения *test* становиться равным истине. Оно вычисляется каждый раз перед выполнением *body*, если в первый раз оно вычисляется как истинное, *body* не вычисляется вообще.

Не обязательные выражения *expr* после *test* выполняются в конце цикла, с последними достпуными привязками переменных *variable*. Последнее *expr* дает

возвращаемое значение, или если нет выражений *expr* возвращаемое значение неопределено.

Каждая итерация устанавливает привязки к новым местам для переменных *variable*, подобно новому `let` для каждой итереации. это делается и для переменных *variable* не имеющих выражений *step*. Следующий пример это илюстрирует, показывая как новыая `i` захыватывается `lambda` на каждой итерации (см. Раздел 3.4 [The Concept of Closure], страница 27).

```
(define lst '())
(do ((i 1 (1+ i)))
    ((> i 4))
  (set! lst (cons (lambda () i) lst)))
(map (lambda (proc) (proc)) lst)
⇒
(4 3 2 1)
```

`while` *cond body ...* [syntax]

Запускает цикл выполнения форм *body* пока выражение *cond* равно истине. *cond* проверяется в начале каждой итерации, так что если оно станет `#f` в самом начале, то *body* не будет выполняться вообще.

С `while`, предоставляются две дополнительные привязки, их можно использовать как из *cond* так и из *body*.

`break` *break-arg ...* [Scheme Procedure]

Производит выход из формы `while`.

`continue` [Scheme Procedure]

Отменяет текущую итерацию, возвращается в начало и снова выполняет проверку *cond* и т.д.

Если цикл завершается нормально, когда *cond* вычисляется в `#f`, тогда выражение the `while` в целом вычисляется как `#f`. Если оно завершается вызовом `break` с каким либо количеством аргументв, эти аргументы возвращаются из выражения `while`, как множественные значения. В противном случае, если он завершается вызовом `break` без аргументов, он возвращает значение `#t`.

```
(while #f (error "not reached")) ⇒ #f
(while #t (break)) ⇒ #t
(while #t (break 1 2 3)) ⇒ 1 2 3
```

Каждая форма `while` получает свои собственные процедуры `break` и `continue`, работающие на этом `while`. Это означает, что когда циклы вложены, внешний `break` может быть использован для выхода из всех внутренних циклов. Например,

```
(while (test1)
  (let ((outer-break break))
    (while (test2)
      (if (something)
        (outer-break #f))
      ...)))
```

Обратите внимание, что каждая процедура `break` и `continue` может использоваться только в динамическом пространстве своего `while`. Вне `while` его поведение не определено.

Другим очень распростарненным способом итерации в программах Scheme является использование так называемого Именованного let(*named let*).

Именованный let это вариант `let`, который создает процедуру и вызывает ее за один шаг. Так как вновь создаваемая процедура, именованного let более мощная процедура чем `do`– она может быть использована не только для итерации, но и для произвольной рекурсии.

`let` *variable bindings body* [syntax]

Для определения *bindings* смотри документацию по `let` (см. Раздел 6.12.2 [Local Bindings], страница 314).

Именованный `let` работает следующим образом:

- Создается новая процедура, которая принимает столько аргументов, сколько содержиться в *bindings* и связываются локально (используя `let`) к *variable*. Имена формальных аргументов новой процедуры это имена переменных *variables*.
- Выражения *body* вставляются во вновь созданную процедуру.
- Процедура вызывается с выражениями *init* в качестве формальных аргументов.

В следующем примере реализован цикл который повоторяется (рекурсивно) 1000 раз.

```
(let lp ((x 1000))
  (if (positive? x)
      (lp (- x 1))
      x))
⇒
0
```

6.13.5 Запросы(Prompts)

Запросы являются барьерами потока управления между различными частями программы. Таким же образом пользователь видит запрос(prompt) командной оболочки(shell) (например, приглашение/запрос Bash) как барьер между операционной системой и его программой, запросы Scheme позволяют программисту Scheme обрабатывать части программы, как если бы они работали в разных операционных системах.

Мы используем это окольное объяснение, потому что, если вы не наркоман функционального программирования, то наверное не слышали термин “разграниченное, составное продолжение”. Это ХОРОШО; это относительно новая тема, но очень полезная для понимания.

6.13.5.1 Примитивы Запросов(Prompt)

Примитивы Guile разделяют управляющие операторы на `call-with-prompt` и `abort-to-prompt`.

`call-with-prompt` *tag thunk handler* [Scheme Procedure]
Устанавливает запрос(prompt), и вызывает чанк(*thunk*) в этом запросе(prompt).

Во время динамического распространения вызова чанка *thunk*, в динамическом контексте запроса(prompt) будет присутствовать метка с именем *tag*, такая что, если пользователь вызовет `abort-to-prompt` (смотри ниже) с этой меткой(tag), управление перейдет обратно в запрос(prompt), и будет запущен обработчик *handler*.

handler должен быть процедурой. Первым аргументом обработчика *handler* будет состояние вычислений начавшихся когда был вызван чанк *thunk* и закончившихся когда был вызван `abort-to-prompt`. Остальные аргументы для обработчика *handler* это аргументы передаваемые в `abort-to-prompt`.

`make-prompt-tag` [*stem*] [Scheme Procedure]
Создает новый тег запроса(prompt). Тег запроса(prompt tag) это просто уникальный объект. В настоящее время тег запроса это чистая пара. Это может измениться в будущих версиях Guile.

`default-prompt-tag` [Scheme Procedure]
Возвращает тег запроса(prompt) по умолчанию. Наличие отличительного тега у запроса по умолчанию, позволяет создавать некоторые полезные запросы и идиомы прерываний, обсуждаемые в следующем разделе. Обратите внимание, что `default-prompt-tag` на самом деле является параметром, и может быть динамически перепривязан использованием `parameterize`. См. Раздел 6.13.12 [Parameters], страница 347.

`abort-to-prompt` *tag val1 val2 ...* [Scheme Procedure]
Раскручивает динамический и управляющий контекст ближайшего запроса с именем *tag*, а также передает предоставляемые значения.

Си программисты могут увидеть в вызовах `call-with-prompt` и `abort-to-prompt` причудливые на вид `setjmp` и `longjmp`, соответственно. Запросы действительно весьма полезны в качестве не локального механизма выхода. Guile конструкции `catch` и `throw` реализованы в терминах запросов(prompts). Запросы(Prompts) более удобны, чем `longjmp`, так как они имеют возможность передавать несколько значений цели перехода.

Кроме того, в отличии от `longjmp`, обработчик(handler) запроса(prompt) получает полное состояние процесса, который был прерван, как первый аргумент обработчика(prompt handler). Это состояние является продолжением(*continuation*) вычисления, заключенного в запрос(prompt). Это и есть разделенное продолжение (*delimited continuation*), потому что это не всё продолжение программы; скорее , это просто вычисление начатое вызовом `call-with-prompt`.

Продолжение является процедурой, и его можно восстановить просто вызвав его, с любым количеством значений. Здесь все становиться интересным и сложным. Кроме того описываемые как разделенные, продолжения, ограниченные запросами(prompts), также являются составными(*composable*), потому что вызов сохраненного запроса (prompt-saved) продолжения составляет это продолжение с текущим.

Представьте что вы сохранили продолжение с помощью call-with-prompt:

```
(define cont
  (call-with-prompt
   ;; tag
   'foo
   ;; thunk
   (lambda ()
     (+ 34 (abort-to-prompt 'foo)))
   ;; handler
   (lambda (k) k)))
```

В результате вы получает продолжение добавляющее 34. Это как если бы вы написали:

```
(define cont
  (lambda (x)
    (+ 34 x)))
```

Итак, если мы вызовем `cont` с одним числовым значением, мы получим это число увеличенное на 34:

```
(cont 8)
⇒ 42
(* 2 (cont 8))
⇒ 84
```

Последний пример иллюстрирует то, что мы имеем в виду, когда говорим, составляется с текущим продолжением ("composes with the current continuation"). Мы имеем в виду, что есть текущее продолжение — некоторое оставшееся вычисление, такое как `(lambda (x) (* x 2))` – и что вызов сохраненного продолжения не стирает текущее продолжение, он составляет сохраненное продолжение с текущим.

Мы обсуждаем этот вопрос здесь, потому что традиционные продолжения Scheme, какие обсуждаются в следующем разделе, не являются соединяемыми и на самом деле менее выразительны, чем продолжения захваченные с помощью запросов(prompts). Но тут есть место для них обоих.

Прежде чем двигаться дальше, мы должны упомянуть, что если обработчик запроса является `lambda` выражением, и не ссылается на свой первый аргумент, прерывание этого запроса(prompt) не вызовет создания продолжения. Это может быть важным фактором эффективности, который следует помнить.

Одним из примеров, где эта оптимизация имеет значение являются escape продолжения (*escape continuations*). Escape-продолжения, являются разделенными продолжениями, единственное использование которых – создать нелокальный выход, т.е убежать(escape) из текущего продолжения. Обычное использование escape-продолжений это выбрасывание исключений. (см. Раздел 6.13.8 [Exceptions], страница 332).

Приведенные ниже конструкции представляют собой синтаксический сахар над запросами(propts), упрощающий использование escape-продолжений.

`call-with-escape-continuation` *proc* [Scheme Procedure]
`call/ec` *proc* [Scheme Procedure]

Вызывает *proc* с escape-продолжением.

В приведенном ниже примере, возвращаемое *return* продолжение используется для выхода(побега) из продолжения вызова `fold`.

```
(use-modules (ice-9 control)
             (srfi srfi-1))

(define (prefix x lst)
  ;; Return all the elements before the first occurrence
  ;; of X in LST.
  (call/ec
    (lambda (return)
      (fold (lambda (element prefix)
              (if (equal? element x)
                  (return (reverse prefix))  ; escape `fold'
                  (cons element prefix)))
            '()
            lst))))

(prefix 'a '(0 1 2 a 3 4 5))
⇒ (0 1 2)
```

`let-escape-continuation` *k body ...* [Scheme Syntax]
`let/ec` *k body ...* [Scheme Syntax]

Связывает *k* внутри *body* с escape-продолжением.

Это эквивалентно `(call/ec (lambda (`*k*`) `*body*` ...))`.

Кроме того, есть еще один вспомогательный примитив, экспортируемый `(ice-9 control)`, поэтому загрузите этот модуль для использования `suspendable-continuation?`:

```
(use-modules (ice-9 control))
```

`suspendable-continuation?` *tag* [Scheme Procedure]

Возвращает `#t` если вызов `abort-to-prompt` с тегом запроса (prompt tag) *tag* приведет к созданию разделенного продолжения, которое может быть возобновлено позже.

Почти все продолжения имеют это свойство. Исключением является случай, когда между `call-with-prompt` и вызов `abort-to-prompt` повторяются через Си по нескольким причинам, `abort-to-prompt` будет успешным, но любая попытка возобновить продолжение(вызвав его) потерпит неудачу. Это потому, что составление сохраненного продолжения с текущим продолжением сопряжено с перемещением кадров стека, которые были сохранены из старого стека на (возможно) новую позицию нового стека, и Guile может это сделать только для кадров стека созданных для кода Scheme, а не кадров стека созданных Си компилятором. Это немного грубовато, но если вы будете придерживаться Scheme, у вас не будет проблем.

Если запрос с данным тегом не найден, эта процедура просто возвращает `#f`.

6.13.5.2 Сдвиг(Shift), Переустановка(Reset), и все такое.

Существует целый зоопарк операторов управления с разделителями, и как представляется он не ограничивается этим набором, Guile реализует их поддержку в отдельном модуле:

```
(use-modules (ice-9 control))
```

Во первых, у нас есть полезное сокращение для оператора `call-with-prompt`.

`%` *expr* [Scheme Syntax]
`%` *expr handler* [Scheme Syntax]
`%` *tag expr handler* [Scheme Syntax]

Вычисляет выражение *expr* в запросе(prompt), опционально указывается тег(tag) и обработчик(handler). Если тега нет, по умолчанию используется тег подсказки по умолчанию.

Если обработчик не указан, устанавливается обработчик по умолчанию. Обработчик по умолчанию принимает процедуру одного аргумента, которая будет вызываться на захваченном продолжении, в пределах запроса.

Иногда проще показать код, как в этом случае:

```
(define (default-prompt-handler k proc)
  (% (default-prompt-tag)
     (proc k)
     default-prompt-handler))
```

Символьный знак `%` выбран потому, что он выглядит как запрос.

Аналогично, существует сокращение для `abort-to-prompt`, которое принимает метку запоса по умолчанию(default prompt tag):

`abort` *val1 val2 . . .* [Scheme Procedure]

Прерывает запрос по умолчанию с меткой, передавая обработчику *val1 val2 . . .*.

Как упоминалось ранее, `(ice-9 control)` также предоставляет другие операторы управления разделениями. Этот раздел является немного техническим, и начинающим пользователям работающим с разделенными продолжениями, вероятно стоит вернуться к нему после некоторой практики работы с `%`.

Вы все еще здесь? Итак, когда реализуется операто управления разделениями, подобный `call-with-prompt`, нужно принять два решения. Первое, работает ли обработчик внутри или снаружи запроса? Наличие обработчика запускаемого в запросе позволяет превать внутри и вернуться к тому же обработчику запроса, что часто бывает полезно. Однако это предотвращает хвостовые вызовы от обработчика, поэтому он менее общий.

Аналогично, вызывает ли захваченное продолжение восстановление запроса(prompt)? Снова у нас есть компромисс между удобством и правильными хвостовыми вызовами.

Эти решения фиксируются в операторе Феллайзена(Felleisen) *F*. Если ни одно продолжение не имеет обработчиков неявно добавляется запрос, оператор известный как –*F*–. Этот случай для Guile `call-with-prompt` и `abort-to-prompt`.

Если оба, продолжение и обработчик неявно добавляют запрос, то оператор будет +*F*+. `shift` и `reset` являются такими операторами.

`reset` *body1 body2 ...* [Scheme Syntax]

Устанавливают запрос, и вычисляют *body1 body2 ...* в этом запросе.

Обработчик запроса предназначенный для работы с `shift`, описан ниже.

`shift` *cont body1 body2 ...* [Scheme Syntax]

Прерывает до ближайшего `reset`, и вычисляет *body1 body2 ...* в контексте, в котором захваченное продолжение связано с *cont*.

Как упоминалось выше, вместе взятые выражения the *body1 body2 ...* и вызовы *cont* неявно устанавливают запрос(prompt).

Заинтересованным читателям предлагается ознакомиться с замечательным сайтом Олега Киселёва по адресу `http://okmij.org/ftp/`, для получения дополнительной информации об этих операторах.

6.13.6 Продолжения

Продолжение("continuation") это код, который будет выполняться когда функция или выражение завершиться(возвратит управление). Например, рассмотрим

```
(define (foo)
  (display "hello\n")
  (display (bar)) (newline)
  (exit))
```

Продолжение из вызова `bar` содержит `display` возвращенного значения, и вызовы функций `newline` и `exit`. Это может быть выражено как функция одного аргумента.

```
(lambda (r)
  (display r) (newline)
  (exit))
```

В Scheme, продолжения представлены в виде специальных процедур, подобнгых этой. Специальное свойство заключается в том, что когда вызывается продолжение, оно оставляет текущее местоположение программы и переходит непосредственно к тому, которое представлено продолжением.

Продолжение похоже на динамическую метку, фиксирующую во время выполнения точку выполнения программы, включая все вложенные вызовы, которые привели к ней(или скорее код, который будет выполнен, когда эти вызовы завершаться).

Продолжения создаются следующими функциями.

`call-with-current-continuation` *proc* [Scheme Procedure]
`call/cc` *proc* [Scheme Procedure]

Захватывает текущее продолжение и вызывает процедуру (***proc cont***) с ним. Возвращаемое значение является значением возвращаемым *proc*, или когда (***cont value***) вызывается позже, возвращается переданное значение *value*.

Обысно *cont* должен вызываться с одним аргументом, но когда местоположение возобновляется ожидается множественное значение (см. Раздел 6.13.7 [Multiple Values], страница 330) затем они должны быть переданы в виде нескольких аргументов, для экземпляра продолжения (***cont x y z***).

cont может использоваться только с той же стороны от барьера продолжения, с какой оно было создано (см. Раздел 6.13.14 [Continuation Barriers], страница 351), и в много-поточных программах, только из той же ветки, в которой оно было создано.

Вызов *proc* не является частью захваченного продолжения, оно выполняется только тогда, когда продолжение создается. Часто программа захочет хранить продолжение *cont* где то, для использования в дальнейшем; это можно сделать в процедуре *proc*.

Слово `call` в имени `call-with-current-continuation` относиться к способу вызова процедуры *proc* передавая ей вновь созданное продолжения. Это не связано с тем, как используется вызов позже, чтобы вызывать продолжение.

`call/cc` это псевноним для `call-with-current-continuation`. Он для общего использования, так как последнее имя довольно длинное.

Вот простой пример,

```
(define kont #f)
(format #t "the return is ~a\n"
        (call/cc (lambda (k)
                   (set! kont k)
                   1)))
⇒ the return is 1

(kont 2)
⇒ the return is 2
```

`call/cc` захватывает продолжение, в котором возвращаемое значение будет отображаться с помощью `format`. Функция `lambda` сохраняет данное продолжение в `kont` и делает начальный возврат значения равного `1` который и отображается. Более поздний вызов `kont` возобновляет захваченную точку, но на этот раз возвращает `2`, которое и отображается.

Когда Guile запускается интерактивно, вызов такого `format` неявно возвращается назад в цикл REPL(read-eval-print loop). `call/cc` перехватывает это как и любой другой возврат, вот почему интерактивно `kont` выполнит возврат, чтобы читать ввод дальше.

Си программисты могут заметить, что `call/cc` подобен `setjmp` в том, как он записывает во время выполнения точку выполнения программы. А вызов продолжения подобен `longjmp` в том смысле, что он покидает текущее местоположение и переходит к записанному. Как и `longjmp`, значение передаваемое в продолжение является значением возвращаемым `call/cc` при возобновлении там. Однако `longjmp` может идти только вверх, по стеку программ, но механизм продолжений может идти куда угодно.

Когда вызывается продолжение, `call/cc` и последующий код эффективно “возвращаются(returns)” повторно. Может показаться странным, что функция возвращается больше раз, чем она была вызвана. Вместо этого может помочь мысль о том, что она тайно повторно перевходит и затем поток программы следует как обычно.

`dynamic-wind` (см. Раздел 6.13.10 [Dynamic Wind], страница 339) может использоваться для обеспечения кода установки и кода очистк, запускаемого когда локус программы возобновляется или отменяется через механизм продолжения.

Продолжения являются мощным механизмом, и могут использоваться для реализации практически любого вида структур управления, таких как циклы, сопрограммы или обработка исключений.

Однако реализация продолжений в Guile не так эффективна, какой она могла бы быть, поскольку Guile предназначен для взаимодействия с программами написанными на других языках, таких как Си, которые не знают о продолжениях. В основном продолжения захватываются копированием блока стека, и возобновляются копированием блока обратно.

По этой причине, продолжения захваченные с помощью `call/cc` должны использоваться только когда нет другого простого способа достижения желаемого результата, или когда элегантность механизма продолжения перевешивает необходимость в быстродействии.

Побеги(Escapes) из всех циклов или вложенных функций обычно лучше всего обрабатываются с помощью запросов(prompts) (см. Раздел 6.13.5 [Prompts], страница 323). Сопрограммы (Coroutines) могут быть эффективно реализованы с помощью взаимодействующих потоков(cooperating threads) (поток содержит полный программный стек, но не копирует его, как это делает продолжение).

6.13.7 Возврат и Прием Множества значений.

Scheme позволяет процедуре возвращать более одного значения вызывающему. Это совсем не похоже на другие языки, которые позволяют возвращать только одно значение. Возврат нескольких значений отличается от возврата списка (или пары, или вектора) значений вызывающей стороне, поскольку концептуально возвращается не один(not *one*) составной объект, а несколько различных значений.

Примитивными процедурами для обработки нескольких значений являются `values` и `call-with-values`. `values` используется для возврата нескольких значений из процедуры. Просто поместите вызов `values` с нулем или большим количеством аргументов в хвостовую позицию тела процедуры. `call-with-values` объединяет возврат процедуры возвращающей множество значений с процедурой которая принимает эти значения в качестве параметров.

`values` *arg ...* [Scheme Procedure]
`scm_values` (*args*) [C Function]

Поставляет все свои аргументы в продолжение. За исключением продолжений созданных процедурой `call-with-values`, все продолжения принимают одно значение. Эффект от перадачи нуля значений или более чем одного значения в продолжения которые не были созданы `call-with-values` не определен.

Для `scm_values`, *args* это список аргументов, а возвращаемое значение это объект множества значений, которые мог вернуть вызывающий. В текущей реализаци этот объект разделяет структуру с *args*, поэтому *args* не должны быть изменены в последствии.

`SCM scm_c_values` (*SCM *base, size_t n*) [C Function]

`scm_c_values` яляется альтернативой `scm_values`. Она создает новый объект значений, и копирует в него *n* значений начиная с *base*.

В настоящее время он создает список и передает его в `scm_values`, но в будущем мы сможем использовать более эффективное представление

`size_t scm_c_nvalues` (*SCM obj*) [C Function]

Если *obj* является объектом с несколькими значениями, возвращает количество значений, которые он содержит. Иначе возвращает 1.

`SCM scm_c_value_ref` (*SCM obj, size_t idx*) [C Function]

Возвращает значение в позиции указанной *idx* объекта *obj*. Обратите внимание, что объект *obj* обычно будет объектом с несколькими значениями, но это не обязательно. Любой другой объект представляет собой одиночной значение(он сам), и обрабатывается соответствующим образом.

`call-with-values` *producer consumer* [Scheme Procedure]

Вызывает аргумент *producer* без значений и продолжение, которое при передаче нескольких значений, вызывает процедуру *consumer* с этими значениями в качестве аргументов. Продолжение для вызвова *consumer* является продолжением вызвова `call-with-values`.

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
⇒ 5

(call-with-values * -)
⇒ -1
```

В дополнение к основным процедурам, описанным выше, в Guile есть модуль который экспортирует синтаксис с именем `receive`, который гораздо удобнее. Он есть в `(ice-9 receive)` и соответствует SRFI-8 (см. Раздел 7.5.7 [SRFI-8], страница 630).

```
(use-modules (ice-9 receive))
```

`receive` *formals expr body . . .* [library syntax]

Вычисляет выражение *expr*, и связывает результирующие значения (ноль или больее) с результирующими значениями(ноль или больше) в *formals*. *formals* это список символов, такие как список аргументов в `lambda` (см. Раздел 6.9.1 [Lambda], страница 262). После связывания переменных в *body . . .* вычисляются по порядку, возвращая значениев последнем выражении.

Например, получение результатов из `partition` в SRFI-1 (см. Раздел 7.5.3 [SRFI-1], страница 607),

```
(receive (odds evens)
    (partition odd? '(7 4 2 8 3))
  (display odds)
  (display " and ")
  (display evens))
⊣ (7 3) and (4 2 8)
```

6.13.8 Исключения

Общим требованием в приложениях является желание выпрыгивать не локально (*non-locally*) из глубины вычислений обратно, скажем, в основной цикл обработки(main processing loop). Обычно, место, то есть цель перехода находиться где то в стеке вызывающих процедур, которые вызвали процедуру, которая хочет выпрыгнуть назад. Например, типичная логика для обработки нажатий клавиш приложения может выглядеть примерно так:

```
main-loop:
  read the next key press and call dispatch-key

dispatch-key:
  lookup the key in a keymap and call an appropriate procedure,
  say find-file

find-file:
  interactively read the required file name, then call
  find-specified-file

find-specified-file:
  check whether file exists; if not, jump back to main-loop
  ...
```

Переход к основному циклу to `main-loop` может быть достигнут путем возврата через стек одной процедуры за раз, используя возвращаемое значение каждой процедуры, чтобы указать состояние ошибки, но Guile (как и большинство современных языков программирования) предоставляет дополнительный механизм, называемый обработкой исключений(*exception handling*), который может быть использован для реализации таких переходов гораздо удобнее.

6.13.8.1 Терминология Исключений

Существует несколько вариантов терминологии для работы с нелокальными переходами(non-local jumps). Это полезно знать о них и понимать, что все они относятся к одному и тому же основному механизму.

- На самом деле совершение не локального перехода, можно называть вызовом исключения(*raising an exception*), вызовом сигнала(*raising a signal*), выбросом исключения (*throwing an exception*) или выполнением длинного перехода *doing a long jump*. Когда переход указывает на состояние ошибки, люди могут говорит о сигнале(*signalling*), вызове(*raising*) или выбросе(*throwing*) *an error*.
- Обработка перехода(прыжка) к своей цели может упоминаться как ловля(*catching*) или обработка(*handling*) исключения(*exception*), сигнала(*signal*) или в случае состояния ошибки, ошибкой(*error*).

Там где используется сигнал(*signal*) и сигнализация(*signalling*), требуется особая осторожность, чтобы избежать риска путаницы с сигналами POSIX.

Это руководство предпочитает говорить о создании и отлове исключений, так как эта терминология соответствует соответствующим примитивам Guile.

Механизм исключений описанный в этом разделе, имеет связи с разделенными продолжениями(*delimited continuations*) (см. Раздел 6.13.5 [Prompts], страница 323). В частности, выбрасывание исключения сродни вызову выхода/побега из продолжения(*escape continuation*) (см. Раздел 6.13.5.1 [Prompt Primitives], страница 323).

6.13.8.2 Ловля/Перехват Исключений(Catching Exceptions)

`catch` используется для установки цели для возможного не локального прыжка/перехода. Аргументами выражения `catch` являются ключ(*key*), который ограничивает набор исключений, к которым применяется этот `catch`, чанк(thunk) который определяет исполняемый код и одина или две процедуры обработчики (*handler*), которые говорят, что делать, если при выполнении кода возникает исключение. Если выполнение чанка(thunk) происходит нормально (*normally*), что означает, без исключений, процедуры обработчики не вызываются вовсе.

Когда исключение выбрасывается с использованием функции `throw`(бросать), первый аргумент `throw` является символом, указывающим на тип исключения. Например, Guile выбрасывает исключения используя символ `numerical-overflow` для указания ошибок числового переполнения, таких как деление на ноль:

```
(/ 1 0)
⇒
ABORT: (numerical-overflow)
```

Аргумент *key* в выражении `catch` соответствует этому символу. *key* может быть конкретным символом, таким как `numerical-overflow`, в этом случае `catch` применяется к конкретным исключениям этого типа; или это может быть `#t`, что означает, что `catch` применяется ко всем исключениям, независимо от их типа.

Второй аргумент выражения `catch` должен быть чанком(thunk) (т.е. процедурой которая не принимает ни одного аргумента), который указывает обычнй исполняемый код. `catch` активен когда исполняется этот чанк(thunk), включая любой код вызываемый непосредственно или косвенно из тела чанка. Вычисление выражения `catch` активирует этот catch и затем вызывает этот чанк(thunk).

Третий аргумент выражения `catch` это процедура обработчик. Если возникает исключение, вызывается эта процедура с точно теми же аргументами, которые указываюстя в `throw`. Следовательно, процедура обработчика должна быть разработана, что бы принимать такое число аргументов, которое соответствует всем выражениям `throw`, которые могут быть перехвачены данным `catch`.

Четвертый, необязательный аргумент выражения `catch` это другая процедура обработчик, называемая обработчик предварительной раскрутки(*pre-unwind*). Он отличается от третьего аргумента тем, что если выбрасывается исключение, он вызывается перед(*before*) обработчиком из третьего аргумента, точно в динамическом контексте выражения, которое выбросило исключение `throw`. Это означает, что оно полезно для захвата или отображения стека в точке вызова `throw`, или для изучения других аспектов динамического контекста, таких как значения флюидов(fluid values), до того, как контекст будет перемота обратно, к контексту господствующего `catch`.

`catch` *key thunk handler* [*pre-unwind-handler*] [Scheme Procedure]

`scm_catch_with_pre_unwind_handler` (*key, thunk, handler, pre_unwind_handler*) [C Function]

`scm_catch` (*key, thunk, handler*) [C Function]

Вызывает чанк *thunk* в динамическом контексте обработчика *handler* исключения соответствующего ключу *key*. Если чанк(thunk) выбросит исключение с ключем *key*, вызовется обработчик *handler*, вот так:

```
(handler key args ...)
```

key это символ или `#t`.

thunk не принимает аргументов. Если *thunk* возвращается нормально, его возвращаемое значене возвращается выражением `catch`.

Обработчик вызывается за пределами области охвата `catch`. Если *handler* снова выбрасывает исключение с тем же ключем(key), далее вызывается новый обработчик стоящий по цепочке выше.

Если ключ это `#t`, тогда выбрасывание любого(*any*) символа будет соответствовать этому вызову для `catch`.

Если задан *pre-unwind-handler* и чанк(*thunk*) выбрасывает исключение которое соответствует *key*, Guile вызывает *pre-unwind-handler* до того, как произойдет разматывание(unwinding) динамического состояния и вызов основного обработчика *handler*. *pre-unwind-handler* должен быть процедурой с той же сигнатурой что и *handler*, который имеет сигнатуру `(lambda (key . args))`. Он обычно используется для сохранения стека в точке, где произошло исключение, но может также запрашивать другие части динамического состояния в этой точке, такие как значения флюидов(fluid values).

pre-unwind-handler может выйти как нормально, так и не локально. Если он выходит нормально, Guile раскручивает стек и динамический контекст и затем вызывает обычный обработчк(третий аргумент). Если он выходит не локально, этот выход определяется продолжением.

Если процедура обработчика должна соответствовать различным выражениям `throw` с различным числом аргументов, вы должны написать его так:

```
(lambda (key . args)
  ...)
```

Аргумент *key* гарантированно присутствует всегда, потому что выбрасывание исключения(`throw`) без ключа *key* не действительна. Количество и интерпретация аргументов *args* варьируется от одного типа исключений к другому, но должна быть указана к документации для каждого типа исключений.

Обратите внимание, что после вызова обычного обработчика (после разматывания стека(post-unwind)), перехват прерываний(catch), который вызвал процедуру обработки исключений, больше не активен. Поэтому, если обработчик исключений сам выбросит исключение, это исключение может быть перехвачено только другими активными выражениями catch находящимися в стеке выше, если они есть.

`SCM scm_c_catch` (*SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data, scm_t_catch_handler pre_unwind_handler, void *pre_unwind_handler_data*) [C Function]

`SCM scm_internal_catch` (*SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data*) [C Function]

Вышеупомянутые `scm_catch_with_pre_unwind_handler` и `scm_catch` принимают Scheme процеруды в качестве аргументов тела и обработчика исключений. `scm_c_catch` и `scm_internal_catch` являются их эквивалентами, принимающими Си функции.

body вызывается как ***body*** (***body_data***) с перехватом исключений соответствующего *tag* типа. Если ислючение поймано, вызываются *pre_unwind_handler* и *handler* как ***handler*** (***handler_data, key, args***). *key* и *args* являются `SCM` ключем(key) и списком аргументов из `throw`.

body и *handler* должны иметь следующие прототипы. `scm_t_catch_body` и `scm_t_catch_handler` это указатель определения типа для них.

```
SCM body (void *data);
SCM handler (void *data, SCM key, SCM args);
```

параметры *body_data* и *handler_data* передаются в соответствующие вызовы так что приложение может передавать дополнительную информацию для этих функций.

Если data состоит из объекта `SCM`, следует позаботиться о том, чтобы он небыл собран сборщиком мусора, пока он еще требуется. Если `SCM` это локальная Си переменная, один из способов защитить ее - передать указатель на эту переменную в качестве параметра data, поскольку Си компилятор будет знать ее значение, оно будет храниться в стеке. Другой способ — использовать `scm_remember_upto_here_1` (см. Раздел 5.5.4 [Foreign Object Memory Management], страница 82).

6.13.8.3 Обработчики Исключений

Иногда полезно перехватить исключение, которое выдается перед разматыванием стека. Это может быть необходимо, чтобы очистить некоторые связанные состояния, распечатки трассы вызовов или например, передать информацию об исключении в отладчик. Процедура `with-throw-handler` предоставляет способ сделать это.

`with-throw-handler` *key thunk handler* [Scheme Procedure]

`scm_with_throw_handler` (*key, thunk, handler*) [C Function]

Добавляет обработчик *handler* в динамический контекст в качестве обработчика исключения для ключа *key*, затем вызывает чанк(*thunk*).

Он ведет себя также как `catch`, за исключением того, что не разматывает стек до того как вызовет обработчик *handler*. Если процедура *handler* возвращается нормально, Guile повторно выбрасывает тоже самое исключение для следующего ближайшего catch или throw обработчика. *handler* может выходить не локально, конечно, выйти не локально можно через явный выброс исключения или вызов продолжения.

Обычно обработчик *handler* используется для отображения трассы вызовов стека, в точке где произошел соответствующий выброс исключения(`throw`), или чтобы сохранить эту информацию для возможного отображения позже.

Не разматывать стек(Not unwinding) означает, что выбрасывается исключение, которое обрабатывается с помощью обработчика throw, эквивалентно вызову встроенного встроенного(inline) обработчика для каждого `throw`, и затем пропуском окружающих `with-throw-handler`. Другими словами,

```
(with-throw-handler 'key
  (lambda () ... (throw 'key args ...) ...)
  handler)
```

в основом эквивалентно

```
((lambda () ... (handler 'key args ...) ...))
```

В частности, динамический контекст при вызове обработчика *handler* это это место где вызывается `throw`. Примеры, не совсем эквиваленты, поскольку тело `with-throw-handler` не находиться в хвостовой позиции по отношению к `with-throw-handler`, и если обработчик *handler* завершиться нормально, Guile организует повторный выброс (rethrow) ошибки, но надеюсь, цель этого ясна. (For an introduction to what is meant by dynamic context, См. Раздел 6.13.10 [Dynamic Wind], страница 339.)

`SCM scm_c_with_throw_handler` (*SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data, int lazy_catch_p*) [C Function]

Приведенные выше `scm_with_throw_handler` получают аргументами процедуры Scheme в качестве тела((thunk) и обработчика. `scm_c_with_throw_handler` является эквивалентом, принимающим Си функции. См. `scm_c_catch` (см. Раздел 6.13.8.2 [Catch], страница 333) для описания параметров, однако поведение коенчно соответствует `with-throw-handler`.

Если чанк(*thunk*) выбрасывает исключение, Guile обрабатывает это исключение, вызывая самый внутренний обработчик `catch` или throw ключ которого совпадает с ключем исключения. Когда самый внутренний обработчик исключения найден, Guile вызывает указанную процедуру обработчика используя `(apply` ***handler*** `key args)`. Процедура обработчика может вернуться нормально, либо выйти не локально. Если она завершается нормально, Guile передает исключение следующему самому внутреннему обработчику `catch` или throw. Если он выходит не локально, этот выход определяется продолжением.

Поведение обработчика исключений throw очень похоже на поведение необязательного обработчика pre-unwind в выражении `catch`. В частности, процедура обработчика throw вызывается в том же динамическом конетесте, так же как и в выражении the `throw`, так и обработчике pre-unwind. `with-throw-handler` можно рассматривать как половину `-catch`: он делает все, что и `catch` до тех пор пока `catch` не начинает разматывать стек и динамический контекст, но затем он возвращается к следующему внутреннему обработчику `catch` или throw.

Также обратите внимание, что поскольку динамический контекст не разматывается, если обработчик `with-throw-handler` выбрасывает исключение с ключем, который

не соответствует ключу *key* выражения `with-throw-handler`, новый бросок(исключение) может быть обработано обработчиком `catch` или throw которые БЛИЖЕ(*closer*) выбросившему исключение коду, чем первый обработчик `with-throw-handler`.

Вот пример, демонстрирующий это поведение:

```
(catch 'a
  (lambda ()
    (with-throw-handler 'b
      (lambda ()
        (catch 'a
          (lambda ()
            (throw 'b))
          inner-handler))
      (lambda (key . args)
        (throw 'a))))
  outer-handler)
```

Этот код вызывает внутренний обработчик `inner-handler` и затем продолжает с продолжения внутреннего `catch`.

6.13.8.4 Выброс/Генерация Исключений

Примитив `throw` используется для выброса исключения. Один аргумент *key*, является обязательным, и должен быть символом; он указывает тип исключения которое выбрасывается. Следующий аргумент *args*, `throw` принимает произвольное число дополнительных аргументов, значение которых зависит от типа исключения. Документация для каждого возможного типа исключения должна описывать дополнительные аргументы, которые ожидаются для такого рода исключения.

`throw` *key arg ...* [Scheme Procedure]
`scm_throw` (*key, args*) [C Function]

Вызывает catch форму соответствующую ключу *key*, передавая аргументы *arg ...* в обработчик *handler*.

key это симовл. Он будет соответствовать перехватывающим формам(catch) с тем же самым символом или `#t`.

Если обработчик вообще отсутствует, Guile напечатает ошибку и затем завершит работу.

Когда генерируется исключение, оно будет перехвачено самым внутренним обработчиком `catch` или throw который применим к данному типу исключения; другими словами, чей ключ *key* является `#t` или тем же символом что и используемый в выражении `throw`. Как только Guile определил соответствующий обработчик `catch` или throw, он обрабатывает исключение, применяя соответсвтующие процедуры обработчики к аргументам из `throw`.

Если для сгенерированного ислючения нет подходящего обработчика `catch` или throw, Guile печатает ошибку в текущий порт ошибки, указывающую на неперехваченное исключение, и затем завершается. На практике, это довольно трудно наблюдать, поскольку Guile при интерактивном использовании устанавливает обработчик верхнего уровня `catch`, который перехватывает все исключения и печатает соответствующее

сообщение об ошибке без(*without*) выхода. Например, это то, что происходит если вы пытаетесь выбросить не обрабатываемое исключение в стандартном Guile REPL; обратите внимание, что цикл команд Guile продолжается после сообщения об ошибке:

```
guile> (throw 'badex)
<unnamed port>:3:1: In procedure gsubr-apply ...
<unnamed port>:3:1: unhandled-exception: badex
ABORT: (misc-error)
guile>
```

Поведение по умолчанию для необработанного исключения можно наблюдать, вычислив выражение `throw` из командной строки shell:

```
$ guile -c "(begin (throw 'badex) (display \"here\\n\"))"
guile: uncaught throw to badex: ()
$
```

То что Guile выходит сразу после неизвестного исключения, демонстрируется отсутствием любого вывода от выражения `display`, поскольку Guile никогда не достигает точки вычисления этого выражения.

6.13.8.5 Как Guile Реализует Исключения

В обычной Scheme система исключений реализуется использованием `call-with-current-continuation`. Продолжения (см. Раздел 6.13.6 [Continuations], страница 328) являются мощной концепцией, которую может использовать любой другой механизм управления— в том числе `catch` и `throw` — могут быть реализованы в терминах продолжений.

Однако Guile не реализует функции `catch` и `throw` подобным образом. Почему нет? Потому что Guile специально разработан для легкой интеграции с приложениями написанными на Си. В смешанной среде Scheme/Си, концепция продолжений(*continuation*) должна логически включать “что произойдет следующим(what happens next)” в Си части приложения, а также в части Scheme, и получается, что единственный разумный способ реализации таких продолжений - это сохранять и восстанавливать полный Си стек.

Таким образом, реализация через `call-with-current-continuation` в Guile является копированием стека. Это позволяет ей хорошо взаимодействовать с обычным Си кодом, но означает, что создание и вызов продолжений замедляется на время которое нужно для копирования Си стека.

Более целенаправленный механизм, предоставляемый `catch` и `throw` не требует сохранения и востановления Си стека, потому что `throw` всегда прыгает в место ВЫШЕ по стеку, относительно кода выполнившего `throw`. Поэтому Guile реализует примитивы `catch` и `throw` независимо от `call-with-current-continuation` способом, который использует преимущества обычных исключений движущихся только вверх (*upwards only*).

6.13.9 Процедуры для Сообщения об Ошибках

Guile предоставляет набор удобных процедур для сообщения об ошибках, которые реализованы поверх только что описанных примитивов исключений.

`error` *msg arg ...* [Scheme Procedure]

Выбрасывает ошибку с кодом `misc-error` и сообщением, созданным с помощью отображения *msg* и записи *arg ...*.

`scm-error` *key subr message args data* [Scheme Procedure]
`scm_error_scm` (*key, subr, message, args, data*) [C Function]

Выбрасывает ошибку с кодом *key*. *subr* может быть строкой с именем связанной с ошибкой процедуры, или `#f`. *message* это строка сообщения об ошибке, возможно содержащая эскейп-коды `~S` и `~A`. Когда сообщается об ошибке, они заменяются форматированными соотвующими членами *args*: `~A` (был `%s` в более старых версиях Guile) форматом используемым для `display` и `~S` (было `%S`) форматом используемым для `write`. *data* это список или `#f` в зависимости от ключа *key*: если *key* это `system-error` тогда это должен быть список, содержащий значение Unix `errno`; Если *key* это `signal` тогда это должен быть список содержащий номер сигнала Unix; Если *key* это `out-of-range`, `wrong-type-arg`, или `keyword-argument-error`, это список содержащий плохое значение; иначе это обычно `#f`.

`strerror` *err* [Scheme Procedure]
`scm_strerror` (*err*) [C Function]

Возвращает сообщение об ошибке Unix соответствующее *err*, целочисленное значение `errno`.

Когда вызывается `setlocale` (см. Раздел 7.2.13 [Locales], страница 571), сообщение будет на языке и в кодировке(charset) соответсвующей `LC_MESSAGES`. (Это делается библиотекой Си.)

`false-if-exception` *expr* [syntax]

Возвращает результат вычисления своего аргумента; однако если возникает исключение, то вместо него возвращается `#f`.

6.13.10 Динамический Ветер(Dynamic Wind)

Для кода Scheme, основной процедурой реагирования на нелокальные входы и выходы в/из динамического контекста является `dynamic-wind`. Си код может использовать `scm_internal_dynamic_wind`, но так как Си код не позволяет удобно строить анонимные процедуры которые замыкаются относительно лексических переменных, это будет, ну, неудобно.

Поэтому, Guile предлагает функции `scm_dynwind_begin` и `scm_dynwind_end` для разделения динамического динамического пространства(extent). В этом динамическом пространстве, который называется динамический контекст(*dynwind context*), вы можете выполнять различные действия(*dynwind actions*) которые управляют тем, что происходит когда происходит вход в динамический контекс, или выход из него. Например, вы можете зарегистрировать процедуру очистки с помощью `scm_dynwind_unwind_handler` которая будет выполняться, когда управление покидает контекст. Есть еще несколько других специалзированных действий динамического ветра(dynwind), например, чтобы временно заблокировать асинхронное выполнение или временно изменить текущий порт вывода. Они описаны в другом месте этого руководтсва.

Вот пример, который показывает, как предотвратить утечки памяти.

```
/* Suppose there is a function called FOO in some library that you
   would like to make available to Scheme code (or to C code that
   follows the Scheme conventions).

   FOO takes two C strings and returns a new string.  When an error has
   occurred in FOO, it returns NULL.
*/

char *foo (char *s1, char *s2);

/* SCM_FOO interfaces the C function FOO to the Scheme way of life.
   It takes care to free up all temporary strings in the case of
   non-local exits.
 */

SCM
scm_foo (SCM s1, SCM s2)
{
  char *c_s1, *c_s2, *c_res;

  scm_dynwind_begin (0);

  c_s1 = scm_to_locale_string (s1);

  /* Call 'free (c_s1)' when the dynwind context is left.
  */
  scm_dynwind_unwind_handler (free, c_s1, SCM_F_WIND_EXPLICITLY);

  c_s2 = scm_to_locale_string (s2);

  /* Same as above, but more concisely.
  */
  scm_dynwind_free (c_s2);

  c_res = foo (c_s1, c_s2);
  if (c_res == NULL)
    scm_memory_error ("foo");

  scm_dynwind_end ();

  return scm_take_locale_string (res);
}
```

`dynamic-wind` *in_guard thunk out_guard* [Scheme Procedure]
`scm_dynamic_wind` (*in_guard, thunk, out_guard*) [C Function]

Все три аргумента должны быть процедурами без аргументов. сначала вызывается *in_guard*, потом *thunk*, затем *out_guard*.

Если в любое время, во время выполнения чанка *thunk*, в динамическом пространства выражения `dynamic-wind` произойдет нелокальный выход, вызывается *out_guard*. Если произойдет повторный вход в динамическое пространство описываемое динамическим ветром(dynamic-wind) вызывается *in_guard*. Таким образом *in_guard* и *out_guard* могут быть вызваны любое количество раз.

```
(define x 'normal-binding)
⇒ x
(define a-cont
  (call-with-current-continuation
   (lambda (escape)
     (let ((old-x x))
       (dynamic-wind
           ;; in-guard:
           ;;
           (lambda () (set! x 'special-binding))

           ;; thunk
           ;;
           (lambda () (display x) (newline)
                      (call-with-current-continuation escape)
                      (display x) (newline)
                      x)

           ;; out-guard:
           ;;
           (lambda () (set! x old-x)))))))
;; Prints:
special-binding
;; Evaluates to:
⇒ a-cont
x
⇒ normal-binding
(a-cont #f)
;; Prints:
special-binding
;; Evaluates to:
⇒ a-cont  ;; the value of the (define a-cont...)
x
⇒ normal-binding
a-cont
⇒ special-binding
```

`scm_t_dynwind_flags` [C Type]

Это перечисление имеет несколько флагов, которые изменяют поведение `scm_dynwind_begin`. Флаги перечислены в следующей таблице.

`SCM_F_DYNWIND_REWINDABLE`

Динамический контекст становиться "перематываемым"(*rewindable*). Это означает что в него можно повторно войти нелокально(через вызов продолжения). По умолчанию в этот контекст динамического ветра(dynwind) нельзя повторно войти не локально.

`void scm_dynwind_begin (`*scm_t_dynwind_flags flags*`)` [C Function]

Функция `scm_dynwind_begin` запускает новый динамический контекст и делает его 'текущим'.

Аргумент *flags* определяет поведение контекста по умолчанию. Обычно используется 0. Это приведет к созданию контекста, в который нельзя повторно войти с захваченным продолжением. Когда вы будете готовы обработать повторные входы, включите повторные входы `SCM_F_DYNWIND_REWINDABLE` в *flags*.

Быть готовым к повторному входу означает, что эффекты обработчика unwind могут быть отменены при повторном входе. В приведенном выше примере мы хотели предотвратить утечку памяти при нелокальном выходе и поэтому зарегистрировали обработчик раскрутки(unwind) которые освобождает память. Но как только память освобождена, мы не можем вернуться обратно, т.е войти в продолжение. Таким образом возвращение не может быть разрешено.

Следствием этого является то, что продолжения становяться менее полезными, нельзя войти в захваченные контексты, но вам не нужно слишком сильно беспокоиться об этом.

Контекст завершается либо неявно, когда происходит нелокальный выход, либо явно вызовом `scm_dynwind_end`. Вы должны убедиться, что контекст динамического ветра(dynwind) действительно закончен должным образом. Если вам не удасться вызвать `scm_dynwind_end` для каждого `scm_dynwind_begin`, поведение будет неопеделенным.

`void scm_dynwind_end ()` [C Function]

Явно завершает текущий динамический контекст и делает текущим предыдущий.

`scm_t_wind_flags` [C Type]

Это перечисление нескольких флагов, которые изменяют поведение `scm_dynwind_unwind_handler` и `scm_dynwind_rewind_handler`. Флаги перечислены в следующей таблице.

`SCM_F_WIND_EXPLICITLY`

Зарегистрированное действие так же выполняется, когда в/из контекста динамического ветра(dynwind) осуществляется локальный вход или выход.

`void scm_dynwind_unwind_handler` (*void (*func)(void *), void *data, scm_t_wind_flags flags*) [C Function]

`void scm_dynwind_unwind_handler_with_scm` (*void (*func)(SCM), SCM data, scm_t_wind_flags flags*) [C Function]

Организация вызова для *func* с аргументами *data*, когда текущий контекст завершается не явно. Если *flags* содержит `SCM_F_WIND_EXPLICITLY`, *func* также вызывается, когда контекст завершается явно с помощью `scm_dynwind_end`.

Функция `scm_dynwind_unwind_handler_with_scm` обеспечивает защиту *data* от сборщика мусора.

`void scm_dynwind_rewind_handler` (*void (*func)(void *), void *data, scm_t_wind_flags flags*) [C Function]

`void scm_dynwind_rewind_handler_with_scm` (*void (*func)(SCM), SCM data, scm_t_wind_flags flags*) [C Function]

Организация вызова для функции *func* вызываемой с аргументом *data*, когда текущий контекст стартует заново путем перемотки стека. когда флаги *flags* содержат `SCM_F_WIND_EXPLICITLY`, функция *func* также вызывается еще и немедленно.

Функция `scm_dynwind_rewind_handler_with_scm` обеспечивает защиту данных *data* от сборщика мусора.

`void scm_dynwind_free` (*void *mem*) [C Function]

Организует автоматическое освобождение для *mem* при каждом выходе из текущего контекста, нормального или не-локального. `scm_dynwind_free (mem)` ялвяется эквивалентнцм сокращением для `scm_dynwind_unwind_handler (free, mem, SCM_F_WIND_EXPLICITLY)`.

6.13.11 Флюиды/Жидкие и изменичивые(Fluids) и Динамические состояния(Dynamic States)

Флюид(*fluid*) это переменная, значение которой связано с динамическим пространством вызванной функции. Так же как операционная система запускает процесс с предоставлением ему установленных портов текущего ввода и вывода (или файловых дискрипторов), в Guile вы можете организовать вызов функции во время связывания флюида(fluid/переменно) с особым значением. Эта связь между флюидом и значением будет существовать во время динамического пространства вызова функции.

Поэтому флюиды являются строительными блоками для реализации переменных с динамической сферой действия. Переменные с динамической сферой действия полезны, когда вы хотите установить значение переменной значением в течении некоторого динамического пространства при исполнении вашей программы и заставить ее вернуться к своему исходному значению, поток управления выходит за пределы этого динамического пространствва. Смотри описание к `with-fluids` ниже, для более детального ознакомления. Эта связь между флюидами, значениями и динамическим пространством устойчива к множественным входам(например, когда захваченное продолжение вызывается более чем один раз) и ранним выходам (например при выбрасывании/возникновении исключений).

Guile использует флюиды для реализации параметров (см. Раздел 6.13.12 [Parameters], страница 347). Обычно вы просто хотите использовать параметры

напрямую. Однако, может быть полезным узнать что такое флюиды и как они работают, вот о чем этот раздел.

Текущий набор ассоциаций флюид-значение может быть зафиксирован в объекте динамического состояния(*dynamic state* object). Динамическое пространство(dynamic extent) это просто моментальный снимок текущий ассоциаций флюид-значение(fluid-value). Пользователи Guile могут захватывать(фиксировать) текущее динамическое состояние с помощью `current-dynamic-state` и восстанавливать его позже через `with-dynamic-state` или пдобных процедур. Это средство особенно полезно, когда реализуются абстракции подобные легковесным потокам(lightweight thread).

Новые флюиды создаются с помощью `make-fluid` и `fluid?` используется для проверки, является ли объект на самом деле флюидом. К значениям хранящися во флюиде можно получать доступ с помощью `fluid-ref` и `fluid-set!`.

См. Раздел 6.22.2 [Thread Local Variables], страница 465, для получения дополнительных сведений о флюидах, потоках и динамических состояниях.

`make-fluid` [*dflt*] [Scheme Procedure]
`scm_make_fluid` () [C Function]
`scm_make_fluid_with_default` (*dflt*) [C Function]

Возвращает вновь созданный флюид, начальное значение которого равно *dflt*, или `#f` если *dflt* не задано. Флюиды это объекты которые могут одно значение для одного динамического состояния. То есть, изменения этого значения видны только коду, который выполняется с тем же динамическим состоянием, что и модифицирующий код. Когда создается новое динамическое состояние, оно наследует значения от своего родителя. Поскольку каждый поток, обычно, выполняется со своим собственным динамическим состоянием, вы может использовать флюиды, для хранения локальных значений потока.

`make-unbound-fluid` [Scheme Procedure]
`scm_make_unbound_fluid` () [C Function]

Возвращает новый флюид, который изначально не связан (вместо того, чтобы быть связанным с каким то определенным значением).

`fluid?` *obj* [Scheme Procedure]
`scm_fluid_p` (*obj*) [C Function]

Возвращает `#t` если *obj* это флюид, иначе возвращает `#f`.

`fluid-ref` *fluid* [Scheme Procedure]
`scm_fluid_ref` (*fluid*) [C Function]

Возвращает значение связанное с флюидом *fluid* в текущем динамическвом корне. Если *fluid* не был установлен, возвращается значение по умолчанию. Вызов `fluid-ref` для несвязанного флюида вызывает ошибку времени выполнения.

`fluid-set!` *fluid value* [Scheme Procedure]
`scm_fluid_set_x` (*fluid, value*) [C Function]

Устанавливает значение связанное с флюидом *fluid* в текущем динамическом корне.

`fluid-ref*` *fluid depth* [Scheme Procedure]
`scm_fluid_ref_star` (*fluid, depth*) [C Function]
Возвращает старое значение присвоенное флюиду *fluid* имеющее глубину(*depth*) в текущем потоке. Если глубина(*depth*) равна или превышает количество значений, которые были присвоены *fluid*, возвращает значение по умолчанию для флюида. `(fluid-ref* f 0)` эквивалентно коду `(fluid-ref f)`.

`fluid-ref*` полезен, когда вы хотите поддерживать стеко-подобную структуру во флюиде, например, такую как стек текущих обработчиков исключений. Использование `fluid-ref*` вместо явного стека позволяет любому частичному продолжению захватывать используя `call-with-prompt` только привязки созданные только в пределах запроса, вместо всего продолжения. См. Раздел 6.13.5 [Prompts], страница 323, для получения дополнительной информации о разделенных продолжениях.

`fluid-unset!` *fluid* [Scheme Procedure]
`scm_fluid_unset_x` (*fluid*) [C Function]
Отсоединяет данный флюид от любого значения, делая его несвязанным.

`fluid-bound?` *fluid* [Scheme Procedure]
`scm_fluid_bound_p` (*fluid*) [C Function]
Возвращает `#t` если данный флюид связан со значением, иначе `#f`.

`with-fluids*` временно изменяет значение одного или нескольких флюидов, так что процедура и каждая вызванная ей процедура обращаются к заданным значениям. После возврата процедуры, старые значения восстанавливаются.

`with-fluid*` *fluid value thunk* [Scheme Procedure]
`scm_with_fluid` (*fluid, value, thunk*) [C Function]
Устанавливает флюиду *fluid* временное значение *value* и вызывает чанк *thunk*. *thunk* должен быть процедурой без аргументов.

`with-fluids*` *fluids values thunk* [Scheme Procedure]
`scm_with_fluids` (*fluids, values, thunk*) [C Function]
Устанавливает флюидам *fluids* временные значения *values* и вызывает чанк *thunk*. *fluids* должен быть списком флюидов и *values* должен иметь тоже количество значений. Каждая подстановка делается в указанном порядке. *thunk* должен быть процедурой без аргументов. Он вызывается внутри динамического ветра(`dynamic-wind`) и флюиды устанавливаются/восстанавиливаются когда управление входит или покидает динамическое пространство.

`with-fluids` ((*fluid value*) ...) *body1 body2 ...* [Scheme Macro]
Выполняет тело *body1 body2 ...* в то время как каждый флюид *fluid* устанавиливается соответствующим значением *value*. Как *fluid*, так и значение *value* вычисляются и *fluid* должен выдавать флюид. Тело выполняется внутри динамического ветра(`dynamic-wind`) и флюиды устанавливаются/восстанавливаются когда управление входит или покидает установленный динамическое пространство.

`SCM scm_c_with_fluids` (*SCM fluids, SCM vals, SCM* (**cproc*)(*void **), *void *data*) [C Function]

`SCM scm_c_with_fluid` (*SCM fluid, SCM val, SCM* (**cproc*)(*void **), *void *data*) [C Function]

Функция `scm_c_with_fluids` похожа на `scm_with_fluids` за исключением того, что она принимает Си функцию для вызова, вместо чанка Scheme.

Функция `scm_c_with_fluid` аналогична, но позволяет устанавливать один флюид вместо списка.

`void scm_dynwind_fluid` (*SCM fluid, SCM val*) [C Function]

Эта функция должна использоваться внутри пары вызовов `scm_dynwind_begin` и `scm_dynwind_end` (см. Раздел 6.13.10 [Dynamic Wind], страница 339). Во время контекста динамического ветра(dynwind), флюид *fluid* устанавливается значением *val*.

Точнее, значение флюида заменяется на 'сохраненное/резервированное' значение, всякий раз когда в контекст динамического ветра(dynwind) входит поток управления и покидает его. Резервное значение инициализируется значением аргумента *val*.

`dynamic-state?` *obj* [Scheme Procedure]

`scm_dynamic_state_p` (*obj*) [C Function]

Возвращает `#t` если *obj* это объект динамического состояния(dynamic state object); иначе `#f`.

`int scm_is_dynamic_state` (*SCM obj*) [C Procedure]

Возвращает не ноль, если *obj* это объект динамического состояния; иначе возвращает 0.

`current-dynamic-state` [Scheme Procedure]

`scm_current_dynamic_state` () [C Function]

Возвращает снимок текущих связей флюид-значение(fluid-value) в виде нового объекта динамического состояния.

`set-current-dynamic-state` *state* [Scheme Procedure]

`scm_set_current_dynamic_state` (*state*) [C Function]

Восстанавливает сохранненые ассоциации флюид-значение(fluid-value), заменяя текущие ассоциации флюид-значение(fluid-value). Возвращает текущие ассоциации fluid-value в виде объекта динамического состояния, как в `current-dynamic-state`.

`with-dynamic-state` *state proc* [Scheme Procedure]

`scm_with_dynamic_state` (*state, proc*) [C Function]

Вызывает *proc* делая привязки флюидов из *state* текущими, запоминая текущие привязки флюидов. Когда управление покидает вызов *proc*, восстанавливает сохраненные привязки, сохраняя вместо этого привязки флюидов внутри вызова. Если управление обратно входит в *proc*, восстанавливает эти сохраненные привязки, запоминая текущие привязки и так далее.

`void scm_dynwind_current_dynamic_state` (*SCM state*) [C Procedure]
Устанавливает текущим динамическое состояние *state* для текущего контекста динамического ветра(dynwind). Подобно `with-dynamic-state`, но в с точки зрения Си интерфейса API Guile “dynwind”.

`void * scm_c_with_dynamic_state` (*SCM state, void *(*func)(void *), void *data*) [C Procedure]
Как `scm_with_dynamic_state`, но вызывая *func* с данными *data*.

6.13.12 Параметры(Parameters)

Параметры являются средством Guile для динамически связанных переменных.

На самом нижнем уровне, объект параметр это процедура. Вызывая ее без аргументов возвращает свое значение. Вызов ее с одним аргументом устанавливает ее значение.

```
(define my-param (make-parameter 123))
(my-param) ⇒ 123
(my-param 456)
(my-param) ⇒ 456
```

Специальная форма `parameterize` устанавливает новое местоположение для параметров, т.е новые местоположения, имеющие эффект в пределах динамического пространства параметризированного(`parameterize`) тела. Выход востанавливает предыдущие местположения. Повторный вход (через сохраненное продолжение) снова будет использовать новые местопоолжения.

```
(parameterize ((my-param 789))
  (my-param)) ⇒ 789
(my-param) ⇒ 456
```

Параметры похожи на динамически связанные переменные в других диалектах Лиспа. Они позволяют приложениям устанавливать параметры(как следует из названия) только для выполнения определенной части кода, восстанавливая их когда он завершиться. Примером таких параметров могут быть чувствительность к регистру во время поиска, или подсказка для пользовательского ввода.

Глобальные переменные не так хороши, как объекты параметров для такого рода вещей. Изменения в них видны всем потокам, но в Guile местоположение объекта параметра устанавливается для каждого потока, тем самым действительно ограничивая эффект от параметризации(`parameterize`) только его динамическим выполнением.

Передача аргументов в функции является потоко-защищенной, но это становиться утомительным когда есть больше чем несколько, или когда они должны пройти через несколько уровней вызова перед тем, как дойти до точки где они вызвать эффект. Введение нового параметра в существующий код, часто гораздо проще с использованием объекта параметра, чем добавлением аргументов.

`make-parameter` *init* [*converter*] [Scheme Procedure]
Возвращает новый объект параметр, с начальным значением *init*.

Если задан преобразователь *converter*, то для каждой установки значения выполняется вызов (*converter* val), возвращаемое им значение сохраняется. Такой вызов также выполняется для начального значения *init*.

converter позволяет проверять значения, или вводить их в каноническую форму. Например,

```
(define my-param (make-parameter 123
                   (lambda (val)
                     (if (not (number? val))
                         (error "must be a number"))
                     (inexact->exact val))))
(my-param 0.75)
(my-param) ⇒ 3/4
```

`parameterize` ((*param value*) ...) *body1 body2* ... [library syntax]

Устанавливает новую динамическую область с заданными параметрами *param*, привязанными к новым местоположениям и установленными заданными значениями *value*s. Выражения *body1 body2* ... вычисляются в этой среде. Значение возвращаемое последним body формы возвращается.

Каждый параметр *param* является выражением, которое вычисляется для получения объекта параметра. Часто это будет просто имя переменной, содержащей объект, но это может быть что угодно что вычисляет параметр.

Выражения *param* и выражения *value* все вычисляются перед установкой новой динамической привязки, и они вычисляются в неопределенном порядке.

Например,

```
(define prompt (make-parameter "Type something: "))
(define (get-input)
  (display (prompt))
  ...)

(parameterize ((prompt "Type a number: "))
  (get-input)
  ...)
```

Объекты параметры реализуются с использованием флюидов (см. Раздел 6.13.11 [Fluids and Dynamic States], страница 343), поэтому каждое динамическое состояние имеет свои собственные местоположения. Это включает отдельные местопоолжения за пределами любой паорametризованной(`parameterize`) формы. Когда параметр создан он получает отдельное начальное местположение в каждом динамическом состоянии, все они инициализируются заданным значением *init*.

Новый код, вероятно, должен просто использовать параметры вместо флюидов, поскольку интерфейс работы с ними лучше. Но для переноса старого кода или обеспечения иного взаимодействия, Guile предоставляет процедуру `fluid->parameter`:

`fluid->parameter` *fluid* [*conv*] [Scheme Procedure]

Создает параметр, который оборачивает флюид.

Значение параметра будет тем же самым, что и значение флюида. Если параметр пересвязывается в некотором динамическом пространстве, возможно через `parameterize`, новое значение будет выполнять необязательную процедуру *conv*, как и для любого параметра. Обратите внимание, что в отличии от `make-parameter`, *conv* не применяется к начальному значению.

Как упоминалось выше, потому что каждый поток обычно имеет отдельное динамическое состояние, каждый поток имеет имеет и свои собственные местоположения для объектов параметров, и изменения в одном потоке не видно никакому другому. Когда создается новое динамическое состояние или поток, значения параметров в исходном контексте, копируются в новое местоположение.

Параметры Guile соответствуют SRFI-39 (см. Раздел 7.5.27 [SRFI-39], страница 653).

6.13.13 Как обрабатывать Ошибки

Обработка ошибок основана на `catch` и `throw`. Ошибки всегда выбрасываются с ключем *key* и четырьмя аргументами:

- *key*: символ, который указывает тип ошибки. Символы используемые в libguile перечислены ниже.
- *subr*: имя процедуры, из которой выдается ошибка, или `#f`.
- *message*: это строка (возможно, зависящая от языка и системы) описывающая ошибку. Токены `~A` и `~S` могут быть встроены в это сообщение: они будут заменены на элементы списка *args* когда сообщение будет распечатано. `~A` указывает на аргумент печатаемый с использованием `display`, тогда как `~S` указывает на аргумент печатаемый с использованием `write`. *message* также может быть `#f`, чтобы его можно было извлечь из *key* обработчиком ошибок (может быть полезно, если *key* может быть выброшен как из Си, так и из Scheme).
- *args*: список аргументов, которые будут использоваться для расширения токенов `~A` и `~S` в *message*. Может также быть `#f` если аргументы не требуются.
- *rest*: список любых дополнительных необходимых объектов. например, когда ключ key это `'system-error`, он содержит значение Си errno. Также может быть `#f` если никакие дополнительные объекты не требуется.

В дополнении к `catch` и `throw`, доступны следующие возможности Scheme:

`display-error` *frame port subr message args rest* [Scheme Procedure]
`scm_display_error` (*frame*, *port*, *subr*, *message*, *args*, *rest*) [C Function]

Выводит сообщение об ошибке в порт вывода *port*. *frame* это кадр в котором произошла ошибка, *subr* это имя процедуры в которой произошла ошибка и *message* это фактическое сообщение об ошибке, которое может содержать инструкции по форматированию. Они позволяют форматированно выводить аргументы из списка *args*. *rest* игнорируется.

Ниже приведены ключи ошибок, определенные в libguile и ситуации, в которых они используются:

- `error-signal`: выбрасывается после получение необработанного фатального сигнала, такого как SIGSEGV, SIGBUS, SIGFPE и т.п. Аргумент *rest* в функции

throw содержит код номера сигнала (в настоящее время это не тоже самое, что обычный номер сигнала Unix).

- `system-error`: выбрасывается после того как операционная система указала на состояние ошибки. Аргумент *rest* при вызове throw содержит значение errno.
- `numerical-overflow`: числовое переполнение.
- `out-of-range`: аргументы процедуры не попадают в допустимую область.
- `wrong-type-arg`: аргумент процедуры имеет неправильный тип.
- `wrong-number-of-args`: процедура была вызвана с неправильным числом аргументов.
- `memory-allocation-error`: ошибка выделения памяти.
- `stack-overflow`: ошибка переполнения стека.
- `regular-expression-syntax`: ошибки генерируемые библиотекой регулярных выражений.
- `misc-error`: другие ошибки.

6.13.13.1 Поддержка Си

В следующих Си функциях, параметры *SUBR* и *MESSAGE* могут иметь значение `NULL`, что бы получить эффект как от `#f` описанный выше.

`SCM scm_error` (*SCM key, char *subr, char *message, SCM args, SCM rest*) [C Function]

Выбрасывает(генерирует) ошибку, для каждого `scm-error` (см. Раздел 6.13.9 [Error Reporting], страница 338).

`void scm_syserror` (*char *subr*) [C Function]

`void scm_syserror_msg` (*char *subr, char *message, SCM args*) [C Function]

Генерирует ошибку с ключем `system-error` и указывает `errno` в аргументе *rest*. Для `scm_syserror` сообщение генерируется с использованием `strerror`.

Следует позаботиться о том, чтобы любой код между ошибочной операцией и вызовом этой процедуры не изменил `errno`.

`void scm_num_overflow` (*char *subr*) [C Function]

`void scm_out_of_range` (*char *subr, SCM bad_value*) [C Function]

`void scm_wrong_num_args` (*SCM proc*) [C Function]

`void scm_wrong_type_arg` (*char *subr, int argnum, SCM bad_value*) [C Function]

`void scm_wrong_type_arg_msg` (*char *subr, int argnum, SCM bad_value, const char *expected*) [C Function]

`void scm_memory_error` (*char *subr*) [C Function]

`void scm_misc_error` (*const char *subr, const char *message, SCM args*) [C Function]

Выбрасывает ошибку с различными ключами описанными выше.

В `scm_wrong_num_args`, *proc* должен быть символом Scheme, который является именем неправильно вызванной процедуры. Другие подпрограммы принимают имя вызванной процедуры в виде Си строки.

В `scm_wrong_type_arg_msg`, *expected* ожидается Си строка описывающая тип аргумента, который ожидался.

В `scm_misc_error`, *message* это строка сообщения об ошибке, возможно, содержащая простой формат `simple-format` экранирования (см. Раздел 6.14.5 [Simple Output], страница 359), и соответствующие аргументы в списке *args*.

6.13.13.2 Сигнализация об Ошибках в Типе

Каждая функция, видимая на уровне Scheme должна активно проверять типы своих аргументов, чтобы избежать неверной интерпретации значения, и возможно выдавать ошибку сегментации. Guile предоставляет некоторые макросы чтобы делать это просто.

`void SCM_ASSERT (`*int* ***test****, SCM* ***obj****, unsigned int* ***position****, const char* ***subr****)* [Macro]

`void SCM_ASSERT_TYPE (`*int* ***test****, SCM* ***obj****, unsigned int* ***position****, const char* ***subr****, const char* ***expected****)* [Macro]

Если *test* равен нулю, сигнализирует об ошибке "неверный тип аргумента(wrong type argument)", свяазнной с программой с именем *subr*, оработающей со значением *obj*, котое является аргументом в позиции *position subr*.

В `SCM_ASSERT_TYPE`, *expected* это Си строка описывающая тип аргумента, который ожидается.

`int SCM_ARG1` [Macro]
`int SCM_ARG2` [Macro]
`int SCM_ARG3` [Macro]
`int SCM_ARG4` [Macro]
`int SCM_ARG5` [Macro]
`int SCM_ARG6` [Macro]
`int SCM_ARG7` [Macro]

Одно из приведенных выше значений может использоваться для *position*, чтобы указать номер аргумента в *subr* который проверяется. Кроме того, положительное целое число может быть испольовано, что позволяет проверять аргументы после семи. Однако для параметра с номером выше семи, предпочтительно использовать `SCM_ARGN` вместо соответствующего необработанного числа, поскольку оно облегчает понимание кода.

`int SCM_ARGn` [Macro]

Передача нулевого значения или `SCM_ARGn` для позиции *position* позволяет оставить ее неопределенной, с неправильным типом аргумента. Опять же `SCM_ARGn` должен быть предпочтительнее постаоянного необработанного нуля.

6.13.14 Барьеры Продолжений

Не локальный поток управления, вызыванный продолжениями, иногда может быть нежелательным. Вы можете использовать `with-continuation-barrier` для возведения барьера, который не смогут проходить продолжения.

`with-continuation-barrier` *proc* [Scheme Procedure]
`scm_with_continuation_barrier (`*proc*`)` [C Function]

Вызывает *proc* и возвращает его результат. Не позволяйте вызываемым продолжениям, которые покидают или входят в динамическое пространство(экстент)

вызывать `with-continuation-barrier`. Такая попытка приведет к сообщению об ошибке.

Выбросы/вызовы(Throws) (такие как ошибки), которые не перехвачены внутри *proc* ловяться `with-continuation-barrier`. В этом случае, печатается короткое сообщение в текущий порт ошибки и возвращается `#f`.

Таким образом, `with-continuation-barrier` возвращается ровноо один раз.

`void * scm_c_with_continuation_barrier` (*void *(*func) (void *), void *data*) [C Function]

Как `scm_with_continuation_barrier` но вызывает *func* для *data*. Когда ловит ошибку, возвращает `NULL`.

6.14 Ввод и Вывод

6.14.1 Порты(Ports)

Порты, это способ, которым Guile выполняет ввод и вывод. Guile может читать в символах или байтах из порта ввода(*input port*), или записывать их в порт вывода(*output port*). Некоторые порты поддерживают оба интерфейса.

В Guile реализованы несколько различных типов портов. Файловые порты обеспечивают ввод и вывод поверх файлов, как вы можете представить. Например, мы можем отобразить строку в файл, так:

```
(let ((port (open-output-file "foo.txt")))
  (display "Hello, world!\n" port)
  (close-port port))
```

Существуют также строковые порты, для получения ввода из строки или сброса вывода в строку; байтвекторные(bytevector) порты, для того же самого, но используя байтвектор(bytevector) как источника или приемника данных; и программные порты, для организации вызова функций Scheme для обеспечения ввода или обработки вывода. См. Раздел 6.14.10 [Port Types], страница 365.

Порты должны быть закрыты(*closed*) когда они не нужны, вызовом `close-port`, как в примере выше. Это обеспечит успешную запись любого ожидающего вывода на диск, в случае файлового порта, или иным образом в любое изменяемое хранилище поддерживаемое портом. Любая ошибка, которая возникает при записи этих буфферезированных данных, также будет возникать незамедлительно при закрытии порта(`close-port`), но не позднее, когда порт закрывается с помощью сборщика мусора. См. Раздел 6.14.6 [Buffering], страница 360, для получения дополнительной информации о буферизированном выводе.

Закрытие порта также освобождает любой драгоценный ресурс, который может иметь файл. Обычно в Scheme программисту не нужно очищать свои структуры данных (см. Раздел 6.19 [Memory Management], страница 425), но большинство систем имеют жесткие ограничения на количество открытых файлов, как для отдельных процессов, так и для всей системы. Программа, которая использует много файлов, должна стараться не выходить за эти пределы. То же самое относиться к аналогичным системным ресурсам, таким как сокеты(socket) и каналы/трубы(pipe).

Действительно, по этим причинам приведенный выше пример на самый идеоматичный способ использования портов. В более общем случае получать порты надо через процедуры подобные `call-with-output-file`, которые обрабатывают `close-port` автоматически:

```
(call-with-output-file "foo.txt"
  (lambda (port)
    (display "Hello, world!\n" port)))
```

Наконец, все порты имеют соответсвтующие входные и выходные буферы, в зависимости от ситуации. Буферизация это общая стратегия для ограничения накладных расходов на небольшие операции чтения и записи: без буферизации, каждый символьный знак извлекаемый из файла, будет включать как минимум один вызов ядра, и, возможно, больше в зависимости от символа и кодировки. Вместо этого, Guile будет пакетно читать и записывать во внутренние буферы. Однако, иногда вы хотите сделать вывод в порт и получить отображение немедленно. См. См. Раздел 6.14.6 [Buffering], страница 360, для получения дополнительной информации о управлении буферизацией порта.

`port?` *x* [Scheme Procedure]
`scm_port_p` (*x*) [C Function]
Return a boolean indicating whether *x* is a port.

`input-port?` *x* [Scheme Procedure]
`scm_input_port_p` (*x*) [C Function]
Return `#t` if *x* is an input port, otherwise return `#f`. Any object satisfying this predicate also satisfies `port?`.

`output-port?` *x* [Scheme Procedure]
`scm_output_port_p` (*x*) [C Function]
Return `#t` if *x* is an output port, otherwise return `#f`. Any object satisfying this predicate also satisfies `port?`.

`close-port` *port* [Scheme Procedure]
`scm_close_port` (*port*) [C Function]
Close the specified port object. Return `#t` if it successfully closes a port or `#f` if it was already closed. An exception may be raised if an error occurs, for example when flushing buffered output. См. Раздел 6.14.6 [Buffering], страница 360, for more on buffered output. См. Раздел 7.2.2 [Ports and File Descriptors], страница 521, for a procedure which can close file descriptors.

`port-closed?` *port* [Scheme Procedure]
`scm_port_closed_p` (*port*) [C Function]
Return `#t` if *port* is closed or `#f` if it is open.

6.14.2 Бинарный Ввод/Вывод(Binary I/O)

Порты Guile по своей природе являются двоичными: на самом низком уровне, они работают с байтами. В этом разделе описываются основные операции двоичного ввода/вывода. См. Раздел 6.14.4 [Textual I/O], страница 357, для ввода и вывода строк и символьных знаков.

Чтобы использовать эти функции, сначала подключите модуль бинарного ввода/вывода:

```
(use-modules (ice-9 binary-ports))
```

Обратите внимание, что хотя название модуля предполагает, что бинарные порты отличаются от портов, это не так: все порты Guile являются и бинарными и текстовыми портами.

`get-u8` *port* [Scheme Procedure]
`scm_get_u8` (*port*) [C Function]
Return an octet read from *port*, an input port, blocking as necessary, or the end-of-file object.

`lookahead-u8` *port* [Scheme Procedure]
`scm_lookahead_u8` (*port*) [C Function]
Like `get-u8` but does not update *port*'s position to point past the octet.

The end-of-file object is unlike any other kind of object: it's not a pair, a symbol, or anything else. To check if a value is the end-of-file object, use the `eof-object?` predicate.

`eof-object?` *x* [Scheme Procedure]
`scm_eof_object_p` (*x*) [C Function]
Return `#t` if *x* is an end-of-file object, or `#f` otherwise.

Note that unlike other procedures in this module, `eof-object?` is defined in the default environment.

`get-bytevector-n` *port count* [Scheme Procedure]
`scm_get_bytevector_n` (*port, count*) [C Function]
Read *count* octets from *port*, blocking as necessary and return a bytevector containing the octets read. If fewer bytes are available, a bytevector smaller than *count* is returned.

`get-bytevector-n!` *port bv start count* [Scheme Procedure]
`scm_get_bytevector_n_x` (*port, bv, start, count*) [C Function]
Read *count* bytes from *port* and store them in *bv* starting at index *start*. Return either the number of bytes actually read or the end-of-file object.

`get-bytevector-some` *port* [Scheme Procedure]
`scm_get_bytevector_some` (*port*) [C Function]
Read from *port*, blocking as necessary, until bytes are available or an end-of-file is reached. Return either the end-of-file object or a new bytevector containing some of the available bytes (at least one), and update the port position to point just past these bytes.

`get-bytevector-all` *port* [Scheme Procedure]
`scm_get_bytevector_all` (*port*) [C Function]
Read from *port*, blocking as necessary, until the end-of-file is reached. Return either a new bytevector containing the data read or the end-of-file object (if no data were available).

`unget-bytevector` *port bv* [*start* [*count*]] [Scheme Procedure]
`scm_unget_bytevector` (*port, bv, start, count*) [C Function]
Place the contents of *bv* in *port*, optionally starting at index *start* and limiting to *count* octets, so that its bytes will be read from left-to-right as the next bytes from *port* during subsequent read operations. If called multiple times, the unread bytes will be read again in last-in first-out order.

To perform binary output on a port, use `put-u8` or `put-bytevector`.

`put-u8` *port octet* [Scheme Procedure]
`scm_put_u8` (*port, octet*) [C Function]
Write *octet*, an integer in the 0–255 range, to *port*, a binary output port.

`put-bytevector` *port bv* [*start* [*count*]] [Scheme Procedure]
`scm_put_bytevector` (*port, bv, start, count*) [C Function]
Write the contents of *bv* to *port*, optionally starting at index *start* and limiting to *count* octets.

6.14.3 Кодировка(Encoding)

Текстовый ввод и вывод через порты Guile располагается поверх бинарных операций. Каждый порт имеет связанную с ним кодировку символов, которая контролирует, как байты читаемые из порта преобразуются в симольные знаки, и как символьные знаки записываемые в порт преобразуются в байты.

`port-encoding` *port* [Scheme Procedure]
`scm_port_encoding` (*port*) [C Function]
Returns, as a string, the character encoding that *port* uses to interpret its input and output.

`set-port-encoding!` *port enc* [Scheme Procedure]
`scm_set_port_encoding_x` (*port, enc*) [C Function]
Sets the character encoding that will be used to interpret I/O to *port*. *enc* is a string containing the name of an encoding. Valid encoding names are those defined by IANA (`http://www.iana.org/assignments/character-sets`), for example `"UTF-8"` or `"ISO-8859-1"`.

Когда порты созданы, им присваивается кодировка. Обычный процесс определения начальной кодировки для порта — получение значения флюида `%default-port-encoding`.

`%default-port-encoding` [Scheme Variable]
A fluid containing name of the encoding to be used by default for newly created ports (см. Раздел 6.13.11 [Fluids and Dynamic States], страница 343). As a special case, the value `#f` is equivalent to `"ISO-8859-1"`.

`%default-port-encoding` по умолчанию использует кодировку, соответствующую текущей локали, если `setlocale` был вызван. См. Раздел 7.2.13 [Locales], страница 571, для получения дополнительной информации о локалях и когда вам может понадобитсья вызов `setlocale`.

Некоторые порты имеют другой способ определения их начальных локалей. Строковые порты, например, по умолчанию используют кодировку UTF-8, чтобы иметь возможность представлять все символы независимо от текущей локали. Файловые порты могут опционально прослушивать свой файл для поиска объявления `coding:`; См. Раздел 6.14.10.1 [File Ports], страница 365. Двоичные порты могут быть инициализированы в кодировке ISO-8859-1 в которой каждая кодовая точка(codepoint) между 0 и 255 соответствует байту с этим значением.

В настоящее время порты работают с не модальными(*non-modal*) кодировками. Большинство кодировок являются не модальными, это означают что преобразование байтов в строку не зависит от ее контекста: одна и таже последовательность байтов всегда возвращает одну и ту же строку. Несколько модальных кодировок находятся в общем использовании, таке как ISO-2022-JP и ISO-2022-KR, и они еще не поддерживаются.

С каждым портом также связана стратегия конвертации, которая определяет, что делать, когда символьный знак Guile не может быть преобразован в закодированное представление символьных знаков порта для вывода. Существует три возможных стратегии: выбросить ошибку, заменить символьный знак с помощью шестнадцатеричного экранирования, или заменить символьный знак замещающим символьным знаком. Стратегия преобразования порта также используется при декодировании символов из входного порта.

`port-conversion-strategy` *port* [Scheme Procedure]
`scm_port_conversion_strategy` (*port*) [C Function]

Returns the behavior of the port when outputting a character that is not representable in the port's current encoding.

If *port* is `#f`, then the current default behavior will be returned. New ports will have this default behavior when they are created.

`set-port-conversion-strategy!` *port sym* [Scheme Procedure]
`scm_set_port_conversion_strategy_x` (*port, sym*) [C Function]

Sets the behavior of Guile when outputting a character that is not representable in the port's current encoding, or when Guile encounters a decoding error when trying to read a character. *sym* can be either `error`, `substitute`, or `escape`.

If *port* is an open port, the conversion error behavior is set for that port. If it is `#f`, it is set as the default behavior for any future ports that get created in this thread.

As with port encodings, there is a fluid which determines the initial conversion strategy for a port.

`%default-port-conversion-strategy` [Scheme Variable]

The fluid that defines the conversion strategy for newly created ports, and also for other conversion routines such as `scm_to_stringn`, `scm_from_stringn`, `string->pointer`, and `pointer->string`.

Its value must be one of the symbols described above, with the same semantics: `error`, `substitute`, or `escape`.

When Guile starts, its value is `substitute`.

Note that `(set-port-conversion-strategy! #f` *sym*`)` is equivalent to `(fluid-set! %default-port-conversion-strategy` *sym*`)`.

As mentioned above, for an output port there are three possible port conversion strategies. The `error` strategy will throw an error when a nonconvertible character is encountered. The `substitute` strategy will replace nonconvertible characters with a question mark (‘`?`’). Finally the `escape` strategy will print nonconvertible characters as a hex escape, using the escaping that is recognized by Guile’s string syntax. Note that if the port’s encoding is a Unicode encoding, like `UTF-8`, then encoding errors are impossible.

For an input port, the `error` strategy will cause Guile to throw an error if it encounters an invalid encoding, such as might happen if you tried to read `ISO-8859-1` as `UTF-8`. The error is thrown before advancing the read position. The `substitute` strategy will replace the bad bytes with a U+FFFD replacement character, in accordance with Unicode recommendations. When reading from an input port, the `escape` strategy is treated as if it were `error`.

6.14.4 Текстовый Ввод/Вывод(Textual I/O)

В этом разделе описываются основные текстовые операции ввода/вывода символьных знаков и строк Guile. См. Раздел 6.14.2 [Binary I/O], страница 353, для ввода и вывода байтов и байтовых векторов. См. Раздел 6.14.3 [Encoding], страница 355, для получения дополнительной информации о том как символьные знаки связаны с байтами. Чтения общих S-выражений из потортов, См. Раздел 6.18.2 [Scheme Read], страница 407. См. Раздел 6.18.3 [Scheme Write], страница 408, для интерфейсво которые записывают общие данные Scheme.

Чтобы использовать данные функции, сначала подключите модуль текстового ввода/вывода:

```
(use-modules (ice-9 textual-ports))
```

Обратите внимание, что хотя название модуля предполагает, что текстовые порты несколько отличаются от портов, это не так: все порты в Guile являются двоичными и текстовыми одновременно..

`get-char` *input-port* [Scheme Procedure]

Reads from *input-port*, blocking as necessary, until a complete character is available from *input-port*, or until an end of file is reached.

If a complete character is available before the next end of file, `get-char` returns that character and updates the input port to point past the character. If an end of file is reached before any character is read, `get-char` returns the end-of-file object.

`lookahead-char` *input-port* [Scheme Procedure]

The `lookahead-char` procedure is like `get-char`, but it does not update *input-port* to point past the character.

In the same way that it’s possible to "unget" a byte or bytes, it’s possible to "unget" the bytes corresponding to an encoded character.

`unget-char` *port char* [Scheme Procedure]

Place character *char* in *port* so that it will be read by the next read operation. If called multiple times, the unread characters will be read again in last-in first-out order.

`unget-string` *port str* [Scheme Procedure]
Place the string *str* in *port* so that its characters will be read from left-to-right as the next characters from *port* during subsequent read operations. If called multiple times, the unread characters will be read again in last-in first-out order.

Reading in a character at a time can be inefficient. If it's possible to perform I/O over multiple characters at a time, via strings, that might be faster.

`get-string-n` *input-port count* [Scheme Procedure]
The `get-string-n` procedure reads from *input-port*, blocking as necessary, until *count* characters are available, or until an end of file is reached. *count* must be an exact, non-negative integer, representing the number of characters to be read.

If *count* characters are available before end of file, `get-string-n` returns a string consisting of those *count* characters. If fewer characters are available before an end of file, but one or more characters can be read, `get-string-n` returns a string containing those characters. In either case, the input port is updated to point just past the characters read. If no characters can be read before an end of file, the end-of-file object is returned.

`get-string-n!` *input-port string start count* [Scheme Procedure]
The `get-string-n!` procedure reads from *input-port* in the same manner as `get-string-n`. *start* and *count* must be exact, non-negative integer objects, with *count* representing the number of characters to be read. *string* must be a string with at least $*start* + *count*$ characters.

If *count* characters are available before an end of file, they are written into *string* starting at index *start*, and *count* is returned. If fewer characters are available before an end of file, but one or more can be read, those characters are written into *string* starting at index *start* and the number of characters actually read is returned as an exact integer object. If no characters can be read before an end of file, the end-of-file object is returned.

`get-string-all` *input-port* [Scheme Procedure]
Reads from *input-port* until an end of file, decoding characters in the same manner as `get-string-n` and `get-string-n!`.

If characters are available before the end of file, a string containing all the characters decoded from that data are returned. If no character precedes the end of file, the end-of-file object is returned.

`get-line` *input-port* [Scheme Procedure]
Reads from *input-port* up to and including the linefeed character or end of file, decoding characters in the same manner as `get-string-n` and `get-string-n!`.

If a linefeed character is read, a string containing all of the text up to (but not including) the linefeed character is returned, and the port is updated to point just past the linefeed character. If an end of file is encountered before any linefeed character is read, but some characters have been read and decoded as characters, a string containing those characters is returned. If an end of file is encountered before any characters are read, the end-of-file object is returned.

Finally, there are just two core procedures to write characters to a port.

`put-char` *port char* [Scheme Procedure]
Writes *char* to the port. The `put-char` procedure returns an unspecified value.

`put-string` *port string* [Scheme Procedure]
`put-string` *port string start* [Scheme Procedure]
`put-string` *port string start count* [Scheme Procedure]
Write the *count* characters of *string* starting at index *start* to the port.

start and *count* must be non-negative exact integer objects. *string* must have a length of at least *start* + *count*. *start* defaults to 0. *count* defaults to (`string` − `lengthstring`) − *start*$.

Calling `put-string` is equivalent in all respects to calling `put-char` on the relevant sequence of characters, except that it will attempt to write multiple characters to the port at a time, even if the port is unbuffered.

The `put-string` procedure returns an unspecified value.

Textual ports have a textual position associated with them: a line and a column. Reading in characters or writing them out advances the line and the column appropriately.

`port-column` *port* [Scheme Procedure]
`port-line` *port* [Scheme Procedure]
`scm_port_column` (*port*) [C Function]
`scm_port_line` (*port*) [C Function]
Return the current column number or line number of *port*.

Port lines and positions are represented as 0-origin integers, which is to say that the the first character of the first line is line 0, column 0. However, when you display a line number, for example in an error message, we recommend you add 1 to get 1-origin integers. This is because lines numbers traditionally start with 1, and that is what non-programmers will find most natural.

`set-port-column!` *port column* [Scheme Procedure]
`set-port-line!` *port line* [Scheme Procedure]
`scm_set_port_column_x` (*port, column*) [C Function]
`scm_set_port_line_x` (*port, line*) [C Function]
Set the current column or line number of *port*.

6.14.5 Простой Текстовый Вывод

Guile экспортирует простую функцию форматированного вывода, `simple-format`. Для более мощного средства форматированого вывода, См. Раздел 7.10 [Formatted Output], страница 728.

`simple-format` *destination message . args* [Scheme Procedure]
`scm_simple_format` (*destination, message, args*) [C Function]
Write *message* to *destination*, defaulting to the current output port. *message* can contain `~A` and `~S` escapes. When printed, the escapes are replaced with corresponding members of *args*: `~A` formats using `display` and `~S` formats using `write`. If *destination* is `#t`, then use the current output port, if *destination* is `#f`, then return a string containing the formatted text. Does not add a trailing newline.

Несколько запутанно, Guile связывает идентификатор `format` с `simple-format` при запуске. После загрузки `(ice-9 format)`, он фактически заменяет привязку `format`, поэтому в зависимости от того, загружаете ли вы модуль `(ice-9 format)`, вы можете пользоваться простой или более мощной версией.

6.14.6 Буфферизация

Every port has associated input and output buffers. You can think of ports as being backed by some mutable store, and that store might be far away. For example, ports backed by file descriptors have to go all the way to the kernel to read and write their data. To avoid this round-trip cost, Guile usually reads in data from the mutable store in chunks, and then services small requests like `get-char` out of that intermediate buffer. Similarly, small writes like `write-char` first go to a buffer, and are sent to the store when the buffer is full (or when port is flushed). Buffered ports speed up your program by reducing the number of round-trips to the mutable store, and they do so in a way that is mostly transparent to the user.

There are two major ways, however, in which buffering affects program semantics. Building correct, performant programs requires understanding these situations.

The first case is in random-access read/write ports (см. Раздел 6.14.7 [Random Access], страница 362). These ports, usually backed by a file, logically operate over the same mutable store when both reading and writing. So, if you read a character, causing the buffer to fill, then write a character, the bytes you filled in your read buffer are now invalid. Every time you switch between reading and writing, Guile has to flush any pending buffer. If this happens frequently, the cost can be high. In that case you should reduce the amount that you buffer, in both directions. Similarly, Guile has to flush buffers before seeking. None of these considerations apply to sockets, which don't logically read from and write to the same mutable store, and are not seekable. Note also that sockets are unbuffered by default. См. Раздел 7.2.11.4 [Network Sockets and Communication], страница 564.

The second case is the more pernicious one. If you write data to a buffered port, it probably doesn't go out to the mutable store directly. (This "probably" introduces some indeterminism in your program: what goes to the store, and when, depends on how full the buffer is. It is something that the user needs to explicitly be aware of.) The data is written to the store later – when the buffer fills up due to another write, or when `force-output` is called, or when `close-port` is called, or when the program exits, or even when the garbage collector runs. The salient point is, *the errors are signalled then too*. Buffered writes defer error detection (and defer the side effects to the mutable store), perhaps indefinitely if the port type does not need to be closed at GC.

One common heuristic that works well for textual ports is to flush output when a newline (`\n`) is written. This *line buffering* mode is on by default for TTY ports. Most other ports are *block buffered*, meaning that once the output buffer reaches the block size, which depends on the port and its configuration, the output is flushed as a block, without regard to what is in the block. Likewise reads are read in at the block size, though if there are fewer bytes available to read, the buffer may not be entirely filled.

Note that binary reads or writes that are larger than the buffer size go directly to the mutable store without passing through the buffers. If your access pattern involves many big reads or writes, buffering might not matter so much to you.

To control the buffering behavior of a port, use `setvbuf`.

`setvbuf` *port mode* [*size*] [Scheme Procedure]
`scm_setvbuf` (*port*, *mode*, *size*) [C Function]
Set the buffering mode for *port*. *mode* can be one of the following symbols:

`none` non-buffered

`line` line buffered

`block` block buffered, using a newly allocated buffer of *size* bytes. If *size* is omitted, a default size will be used.

Another way to set the buffering, for file ports, is to open the file with `0` or `l` as part of the mode string, for unbuffered or line-buffered ports, respectively. См. Раздел 6.14.10.1 [File Ports], страница 365, for more.

Any buffered output data will be written out when the port is closed. To make sure to flush it at specific points in your program, use `force-otput`.

`force-output` [*port*] [Scheme Procedure]
`scm_force_output` (*port*) [C Function]
Flush the specified output port, or the current output port if *port* is omitted. The current output buffer contents, if any, are passed to the underlying port implementation.

The return value is unspecified.

`flush-all-ports` [Scheme Procedure]
`scm_flush_all_ports` () [C Function]
Equivalent to calling `force-output` on all open output ports. The return value is unspecified.

Similarly, sometimes you might want to switch from using Guile's ports to working directly on file descriptors. In that case, for input ports use `drain-input` to get any buffered input from that port.

`drain-input` *port* [Scheme Procedure]
`scm_drain_input` (*port*) [C Function]
This procedure clears a port's input buffers, similar to the way that force-output clears the output buffer. The contents of the buffers are returned as a single string, e.g.,

```
(define p (open-input-file ...))
(drain-input p) => empty string, nothing buffered yet.
(unread-char (read-char p) p)
(drain-input p) => initial chars from p, up to the buffer size.
```

All of these considerations are very similar to those of streams in the C library, although Guile's ports are not built on top of C streams. Still, it is useful to read what other systems do. См. Раздел "Streams" в *The GNU C Library Reference Manual*, for more discussion on C streams.

6.14.7 Random Access

`seek` *fd_port offset whence* [Scheme Procedure]
`scm_seek` (*fd_port, offset, whence*) [C Function]

Sets the current position of *fd_port* to the integer *offset*. For a file port, *offset* is expressed as a number of bytes; for other types of ports, such as string ports, *offset* is an abstract representation of the position within the port's data, not necessarily expressed as a number of bytes. *offset* is interpreted according to the value of *whence*.

One of the following variables should be supplied for *whence*:

`SEEK_SET` [Переменная]
Seek from the beginning of the file.

`SEEK_CUR` [Переменная]
Seek from the current position.

`SEEK_END` [Переменная]
Seek from the end of the file.

If *fd_port* is a file descriptor, the underlying system call is `lseek`. *port* may be a string port.

The value returned is the new position in *fd_port*. This means that the current position of a port can be obtained using:

```
(seek port 0 SEEK_CUR)
```

`ftell` *fd_port* [Scheme Procedure]
`scm_ftell` (*fd_port*) [C Function]

Return an integer representing the current position of *fd_port*, measured from the beginning. Equivalent to:

```
(seek port 0 SEEK_CUR)
```

`truncate-file` *file* [*length*] [Scheme Procedure]
`scm_truncate_file` (*file, length*) [C Function]

Truncate *file* to *length* bytes. *file* can be a filename string, a port object, or an integer file descriptor. The return value is unspecified.

For a port or file descriptor *length* can be omitted, in which case the file is truncated at the current position (per `ftell` above).

On most systems a file can be extended by giving a length greater than the current size, but this is not mandatory in the POSIX standard.

6.14.8 Line Oriented and Delimited Text

The delimited-I/O module can be accessed with:

```
(use-modules (ice-9 rdelim))
```

It can be used to read or write lines of text, or read text delimited by a specified set of characters.

`read-line` [*port*] [*handle-delim*] [Scheme Procedure]

Return a line of text from *port* if specified, otherwise from the value returned by `(current-input-port)`. Under Unix, a line of text is terminated by the first end-of-line character or by end-of-file.

If *handle-delim* is specified, it should be one of the following symbols:

`trim` Discard the terminating delimiter. This is the default, but it will be impossible to tell whether the read terminated with a delimiter or end-of-file.

`concat` Append the terminating delimiter (if any) to the returned string.

`peek` Push the terminating delimiter (if any) back on to the port.

`split` Return a pair containing the string read from the port and the terminating delimiter or end-of-file object.

`read-line!` *buf* [*port*] [Scheme Procedure]

Read a line of text into the supplied string *buf* and return the number of characters added to *buf*. If *buf* is filled, then `#f` is returned. Read from *port* if specified, otherwise from the value returned by `(current-input-port)`.

`read-delimited` *delims* [*port*] [*handle-delim*] [Scheme Procedure]

Read text until one of the characters in the string *delims* is found or end-of-file is reached. Read from *port* if supplied, otherwise from the value returned by `(current-input-port)`. *handle-delim* takes the same values as described for `read-line`.

`read-delimited!` *delims buf* [*port*] [*handle-delim*] [*start*] [*end*] [Scheme Procedure]

Read text into the supplied string *buf*.

If a delimiter was found, return the number of characters written, except if *handle-delim* is `split`, in which case the return value is a pair, as noted above.

As a special case, if *port* was already at end-of-stream, the EOF object is returned. Also, if no characters were written because the buffer was full, `#f` is returned.

It's something of a wacky interface, to be honest.

`%read-delimited!` *delims str gobble* [*port* [*start* [*end*]]] [Scheme Procedure]
`scm_read_delimited_x` (*delims, str, gobble, port, start, end*) [C Function]

Read characters from *port* into *str* until one of the characters in the *delims* string is encountered. If *gobble* is true, discard the delimiter character; otherwise, leave it in the input stream for the next read. If *port* is not specified, use the value of `(current-input-port)`. If *start* or *end* are specified, store data only into the substring of *str* bounded by *start* and *end* (which default to the beginning and end of the string, respectively).

Return a pair consisting of the delimiter that terminated the string and the number of characters read. If reading stopped at the end of file, the delimiter returned is the *eof-object*; if the string was filled without encountering a delimiter, this value is `#f`.

`%read-line` [*port*] [Scheme Procedure]
`scm_read_line` (*port*) [C Function]

Read a newline-terminated line from *port*, allocating storage as necessary. The newline terminator (if any) is removed from the string, and a pair consisting of the line and its delimiter is returned. The delimiter may be either a newline or the *eof-object*; if `%read-line` is called at the end of file, it returns the pair `(#<eof> . #<eof>)`.

6.14.9 Default Ports for Input, Output and Errors

`current-input-port` [Scheme Procedure]
`scm_current_input_port` () [C Function]

Return the current input port. This is the default port used by many input procedures.

Initially this is the *standard input* in Unix and C terminology. When the standard input is a tty the port is unbuffered, otherwise it's fully buffered.

Unbuffered input is good if an application runs an interactive subprocess, since any type-ahead input won't go into Guile's buffer and be unavailable to the subprocess.

Note that Guile buffering is completely separate from the tty "line discipline". In the usual cooked mode on a tty Guile only sees a line of input once the user presses `Return`.

`current-output-port` [Scheme Procedure]
`scm_current_output_port` () [C Function]

Return the current output port. This is the default port used by many output procedures.

Initially this is the *standard output* in Unix and C terminology. When the standard output is a tty this port is unbuffered, otherwise it's fully buffered.

Unbuffered output to a tty is good for ensuring progress output or a prompt is seen. But an application which always prints whole lines could change to line buffered, or an application with a lot of output could go fully buffered and perhaps make explicit `force-output` calls (см. Раздел 6.14.6 [Buffering], страница 360) at selected points.

`current-error-port` [Scheme Procedure]
`scm_current_error_port` () [C Function]

Return the port to which errors and warnings should be sent.

Initially this is the *standard error* in Unix and C terminology. When the standard error is a tty this port is unbuffered, otherwise it's fully buffered.

`set-current-input-port` *port* [Scheme Procedure]
`set-current-output-port` *port* [Scheme Procedure]
`set-current-error-port` *port* [Scheme Procedure]
`scm_set_current_input_port` (*port*) [C Function]
`scm_set_current_output_port` (*port*) [C Function]
`scm_set_current_error_port` (*port*) [C Function]

Change the ports returned by `current-input-port`, `current-output-port` and `current-error-port`, respectively, so that they use the supplied *port* for input or output.

`with-input-from-port` *port thunk* [Scheme Procedure]
`with-output-to-port` *port thunk* [Scheme Procedure]
`with-error-to-port` *port thunk* [Scheme Procedure]
Call *thunk* in a dynamic environment in which `current-input-port`, `current-output-port` or `current-error-port` is rebound to the given *port*.

`void scm_dynwind_current_input_port` (*SCM port*) [C Function]
`void scm_dynwind_current_output_port` (*SCM port*) [C Function]
`void scm_dynwind_current_error_port` (*SCM port*) [C Function]
These functions must be used inside a pair of calls to `scm_dynwind_begin` and `scm_dynwind_end` (см. Раздел 6.13.10 [Dynamic Wind], страница 339). During the dynwind context, the indicated port is set to *port*.

More precisely, the current port is swapped with a 'backup' value whenever the dynwind context is entered or left. The backup value is initialized with the *port* argument.

6.14.10 Types of Port

6.14.10.1 File Ports

The following procedures are used to open file ports. See also Раздел 7.2.2 [Ports and File Descriptors], страница 521, for an interface to the Unix `open` system call.

All file access uses the "LFS" large file support functions when available, so files bigger than 2 Gbytes (2^31 bytes) can be read and written on a 32-bit system.

Most systems have limits on how many files can be open, so it's strongly recommended that file ports be closed explicitly when no longer required (см. Раздел 6.14.1 [Ports], страница 352).

`open-file` *filename mode* [*#:guess-encoding=#f*] [*#:encoding=#f*] [Scheme Procedure]
`scm_open_file_with_encoding` (*filename, mode, guess_encoding, encoding*) [C Function]
`scm_open_file` (*filename, mode*) [C Function]
Open the file whose name is *filename*, and return a port representing that file. The attributes of the port are determined by the *mode* string. The way in which this is interpreted is similar to C stdio. The first character must be one of the following:

'`r`' Open an existing file for input.

'`w`' Open a file for output, creating it if it doesn't already exist or removing its contents if it does.

'`a`' Open a file for output, creating it if it doesn't already exist. All writes to the port will go to the end of the file. The "append mode" can be turned off while the port is in use см. Раздел 7.2.2 [Ports and File Descriptors], страница 521,

The following additional characters can be appended:

'`+`' Open the port for both input and output. E.g., `r+`: open an existing file for both input and output.

‘0’ Create an "unbuffered" port. In this case input and output operations are passed directly to the underlying port implementation without additional buffering. This is likely to slow down I/O operations. The buffering mode can be changed while a port is in use (см. Раздел 6.14.6 [Buffering], страница 360).

‘l’ Add line-buffering to the port. The port output buffer will be automatically flushed whenever a newline character is written.

‘b’ Use binary mode, ensuring that each byte in the file will be read as one Scheme character.

To provide this property, the file will be opened with the 8-bit character encoding "ISO-8859-1", ignoring the default port encoding. См. Раздел 6.14.1 [Ports], страница 352, for more information on port encodings.

Note that while it is possible to read and write binary data as characters or strings, it is usually better to treat bytes as octets, and byte sequences as bytevectors. См. Раздел 6.14.2 [Binary I/O], страница 353, for more.

This option had another historical meaning, for DOS compatibility: in the default (textual) mode, DOS reads a CR-LF sequence as one LF byte. The `b` flag prevents this from happening, adding `O_BINARY` to the underlying `open` call. Still, the flag is generally useful because of its port encoding ramifications.

Unless binary mode is requested, the character encoding of the new port is determined as follows: First, if *guess-encoding* is true, the `file-encoding` procedure is used to guess the encoding of the file (см. Раздел 6.18.8 [Character Encoding of Source Files], страница 416). If *guess-encoding* is false or if `file-encoding` fails, *encoding* is used unless it is also false. As a last resort, the default port encoding is used. См. Раздел 6.14.1 [Ports], страница 352, for more information on port encodings. It is an error to pass a non-false *guess-encoding* or *encoding* if binary mode is requested.

If a file cannot be opened with the access requested, `open-file` throws an exception.

`open-input-file` *filename* [*#:guess-encoding=#f*] [*#:encoding=#f*] [*#:binary=#f*] [Scheme Procedure]

Open *filename* for input. If *binary* is true, open the port in binary mode, otherwise use text mode. *encoding* and *guess-encoding* determine the character encoding as described above for `open-file`. Equivalent to

```
(open-file filename
           (if binary "rb" "r")
           #:guess-encoding guess-encoding
           #:encoding encoding)
```

`open-output-file` *filename* [*#:encoding=#f*] [*#:binary=#f*] [Scheme Procedure]

Open *filename* for output. If *binary* is true, open the port in binary mode, otherwise use text mode. *encoding* specifies the character encoding as described above for `open-file`. Equivalent to

```
(open-file filename
```

```
(if binary "wb" "w")
#:encoding encoding)
```

`call-with-input-file` *filename proc* [*#:guess-encoding=#f*] [Scheme Procedure]
[*#:encoding=#f*] [*#:binary=#f*]
`call-with-output-file` *filename proc* [*#:encoding=#f*] [Scheme Procedure]
[*#:binary=#f*]

Open *filename* for input or output, and call (*proc* `port`) with the resulting port. Return the value returned by *proc*. *filename* is opened as per `open-input-file` or `open-output-file` respectively, and an error is signaled if it cannot be opened.

When *proc* returns, the port is closed. If *proc* does not return (e.g. if it throws an error), then the port might not be closed automatically, though it will be garbage collected in the usual way if not otherwise referenced.

`with-input-from-file` *filename thunk* [*#:guess-encoding=#f*] [Scheme Procedure]
[*#:encoding=#f*] [*#:binary=#f*]
`with-output-to-file` *filename thunk* [*#:encoding=#f*] [Scheme Procedure]
[*#:binary=#f*]
`with-error-to-file` *filename thunk* [*#:encoding=#f*] [Scheme Procedure]
[*#:binary=#f*]

Open *filename* and call (*thunk*) with the new port setup as respectively the `current-input-port`, `current-output-port`, or `current-error-port`. Return the value returned by *thunk*. *filename* is opened as per `open-input-file` or `open-output-file` respectively, and an error is signaled if it cannot be opened.

When *thunk* returns, the port is closed and the previous setting of the respective current port is restored.

The current port setting is managed with `dynamic-wind`, so the previous value is restored no matter how *thunk* exits (eg. an exception), and if *thunk* is re-entered (via a captured continuation) then it's set again to the *filename* port.

The port is closed when *thunk* returns normally, but not when exited via an exception or new continuation. This ensures it's still ready for use if *thunk* is re-entered by a captured continuation. Of course the port is always garbage collected and closed in the usual way when no longer referenced anywhere.

`port-mode` *port* [Scheme Procedure]
`scm_port_mode` (*port*) [C Function]

Return the port modes associated with the open port *port*. These will not necessarily be identical to the modes used when the port was opened, since modes such as `"append"` which are used only during port creation are not retained.

`port-filename` *port* [Scheme Procedure]
`scm_port_filename` (*port*) [C Function]

Return the filename associated with *port*, or `#f` if no filename is associated with the port.

port must be open; `port-filename` cannot be used once the port is closed.

`set-port-filename!` *port filename* [Scheme Procedure]
`scm_set_port_filename_x` (*port*, *filename*) [C Function]
Change the filename associated with *port*, using the current input port if none is specified. Note that this does not change the port's source of data, but only the value that is returned by `port-filename` and reported in diagnostic output.

`file-port?` *obj* [Scheme Procedure]
`scm_file_port_p` (*obj*) [C Function]
Determine whether *obj* is a port that is related to a file.

6.14.10.2 Bytevector Ports

`open-bytevector-input-port` *bv* **[***transcoder***]** [Scheme Procedure]
`scm_open_bytevector_input_port` (*bv*, *transcoder*) [C Function]
Return an input port whose contents are drawn from bytevector *bv* (см. Раздел 6.6.12 [Bytevectors], страница 205).

The *transcoder* argument is currently not supported.

`open-bytevector-output-port` **[***transcoder***]** [Scheme Procedure]
`scm_open_bytevector_output_port` (*transcoder*) [C Function]
Return two values: a binary output port and a procedure. The latter should be called with zero arguments to obtain a bytevector containing the data accumulated by the port, as illustrated below.

```
(call-with-values
  (lambda ()
    (open-bytevector-output-port))
  (lambda (port get-bytevector)
    (display "hello" port)
    (get-bytevector)))

⇒ #vu8(104 101 108 108 111)
```

The *transcoder* argument is currently not supported.

6.14.10.3 String Ports

`call-with-output-string` *proc* [Scheme Procedure]
`scm_call_with_output_string` (*proc*) [C Function]
Calls the one-argument procedure *proc* with a newly created output port. When the function returns, the string composed of the characters written into the port is returned. *proc* should not close the port.

`call-with-input-string` *string proc* [Scheme Procedure]
`scm_call_with_input_string` (*string*, *proc*) [C Function]
Calls the one-argument procedure *proc* with a newly created input port from which *string*'s contents may be read. The value yielded by the *proc* is returned.

`with-output-to-string` *thunk* [Scheme Procedure]
Calls the zero-argument procedure *thunk* with the current output port set temporarily to a new string port. It returns a string composed of the characters written to the current output.

`with-input-from-string` *string thunk* [Scheme Procedure]
Calls the zero-argument procedure *thunk* with the current input port set temporarily to a string port opened on the specified *string*. The value yielded by *thunk* is returned.

`open-input-string` *str* [Scheme Procedure]
`scm_open_input_string` (*str*) [C Function]
Take a string and return an input port that delivers characters from the string. The port can be closed by `close-input-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

`open-output-string` [Scheme Procedure]
`scm_open_output_string` () [C Function]
Return an output port that will accumulate characters for retrieval by `get-output-string`. The port can be closed by the procedure `close-output-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

`get-output-string` *port* [Scheme Procedure]
`scm_get_output_string` (*port*) [C Function]
Given an output port created by `open-output-string`, return a string consisting of the characters that have been output to the port so far.

`get-output-string` must be used before closing *port*, once closed the string cannot be obtained.

With string ports, the port-encoding is treated differently than other types of ports. When string ports are created, they do not inherit a character encoding from the current locale. They are given a default locale that allows them to handle all valid string characters. Typically one should not modify a string port's character encoding away from its default. См. Раздел 6.14.3 [Encoding], страница 355.

6.14.10.4 Custom Ports

Custom ports allow the user to provide input and handle output via user-supplied procedures. Guile currently only provides custom binary ports, not textual ports; for custom textual ports, См. Раздел 6.14.10.5 [Soft Ports], страница 371. We should add the R6RS custom textual port interfaces though. Contributions are appreciated.

`make-custom-binary-input-port` *id read! get-position set-position! close* [Scheme Procedure]
Return a new custom binary input port[8] named *id* (a string) whose input is drained by invoking *read!* and passing it a bytevector, an index where bytes should be written, and the number of bytes to read. The `read!` procedure must return an integer indicating the number of bytes read, or `0` to indicate the end-of-file.

Optionally, if *get-position* is not `#f`, it must be a thunk that will be called when `port-position` is invoked on the custom binary port and should return an integer indicating the position within the underlying data stream; if *get-position* was not supplied, the returned port does not support `port-position`.

[8] This is similar in spirit to Guile's *soft ports* (см. Раздел 6.14.10.5 [Soft Ports], страница 371).

Likewise, if *set-position!* is not `#f`, it should be a one-argument procedure. When `set-port-position!` is invoked on the custom binary input port, *set-position!* is passed an integer indicating the position of the next byte is to read.

Finally, if *close* is not `#f`, it must be a thunk. It is invoked when the custom binary input port is closed.

The returned port is fully buffered by default, but its buffering mode can be changed using `setvbuf` (см. Раздел 6.14.6 [Buffering], страница 360).

Using a custom binary input port, the `open-bytevector-input-port` procedure (см. Раздел 6.14.10.2 [Bytevector Ports], страница 368) could be implemented as follows:

```
(define (open-bytevector-input-port source)
  (define position 0)
  (define length (bytevector-length source))

  (define (read! bv start count)
    (let ((count (min count (- length position))))
      (bytevector-copy! source position
                        bv start count)
      (set! position (+ position count))
      count))

  (define (get-position) position)

  (define (set-position! new-position)
    (set! position new-position))

  (make-custom-binary-input-port "the port" read!
                                 get-position set-position!
                                 #f))

(read (open-bytevector-input-port (string->utf8 "hello")))
⇒ hello
```

`make-custom-binary-output-port` *id write! get-position set-position! close* [Scheme Procedure]

Return a new custom binary output port named *id* (a string) whose output is sunk by invoking *write!* and passing it a bytevector, an index where bytes should be read from this bytevector, and the number of bytes to be "written". The `write!` procedure must return an integer indicating the number of bytes actually written; when it is passed `0` as the number of bytes to write, it should behave as though an end-of-file was sent to the byte sink.

The other arguments are as for `make-custom-binary-input-port`.

`make-custom-binary-input/output-port` *id read! write! get-position set-position! close* [Scheme Procedure]

Return a new custom binary input/output port named *id* (a string). The various arguments are the same as for The other arguments are as for `make-custom-binary-input-port` and `make-custom-binary-output-port`. If buffering is enabled on

the port, as is the case by default, input will be buffered in both directions; См. Раздел 6.14.6 [Buffering], страница 360. If the *set-position!* function is provided and not `#f`, then the port will also be marked as random-access, causing the buffer to be flushed between reads and writes.

6.14.10.5 Soft Ports

A *soft port* is a port based on a vector of procedures capable of accepting or delivering characters. It allows emulation of I/O ports.

`make-soft-port` *pv modes* [Scheme Procedure]

Return a port capable of receiving or delivering characters as specified by the *modes* string (см. Раздел 6.14.10.1 [File Ports], страница 365). *pv* must be a vector of length 5 or 6. Its components are as follows:

0. procedure accepting one character for output
1. procedure accepting a string for output
2. thunk for flushing output
3. thunk for getting one character
4. thunk for closing port (not by garbage collection)
5. (if present and not `#f`) thunk for computing the number of characters that can be read from the port without blocking.

For an output-only port only elements 0, 1, 2, and 4 need be procedures. For an input-only port only elements 3 and 4 need be procedures. Thunks 2 and 4 can instead be `#f` if there is no useful operation for them to perform.

If thunk 3 returns `#f` or an `eof-object` (см. Раздел "Input" в *The Revised^5 Report on Scheme*) it indicates that the port has reached end-of-file. For example:

```
(define stdout (current-output-port))
(define p (make-soft-port
           (vector
            (lambda (c) (write c stdout))
            (lambda (s) (display s stdout))
            (lambda () (display "." stdout))
            (lambda () (char-upcase (read-char)))
            (lambda () (display "@" stdout)))
           "rw"))

(write p p) ⇒ #<input-output: soft 8081e20>
```

6.14.10.6 Void Ports

This kind of port causes any data to be discarded when written to, and always returns the end-of-file object when read from.

`%make-void-port` *mode* [Scheme Procedure]
`scm_sys_make_void_port` (*mode*) [C Function]

Create and return a new void port. A void port acts like `/dev/null`. The *mode* argument specifies the input/output modes for this port: see the documentation for `open-file` in Раздел 6.14.10.1 [File Ports], страница 365.

6.14.11 Venerable Port Interfaces

Over the 25 years or so that Guile has been around, its port system has evolved, adding many useful features. At the same time there have been four major Scheme standards released in those 25 years, which also evolve the common Scheme understanding of what a port interface should be. Alas, it would be too much to ask for all of these evolutionary branches to be consistent. Some of Guile's original interfaces don't mesh with the later Scheme standards, and yet Guile can't just drop old interfaces. Sadly as well, the R6RS and R7RS standards both part from a base of R5RS, but end up in different and somewhat incompatible designs.

Guile's approach is to pick a set of port primitives that make sense together. We document that set of primitives, design our internal interfaces around them, and recommend them to users. As the R6RS I/O system is the most capable standard that Scheme has yet produced in this domain, we mostly recommend that; `(ice-9 binary-ports)` and `(ice-9 textual-ports)` are wholly modelled on `(rnrs io ports)`. Guile does not wholly copy R6RS, however; См. Раздел 7.6.1 [R6RS Incompatibilities], страница 677.

At the same time, we have many venerable port interfaces, lore handed down to us from our hacker ancestors. Most of these interfaces even predate the expectation that Scheme should have modules, so they are present in the default environment. In Guile we support them as well and we have no plans to remove them, but again we don't recommend them for new users.

`char-ready?` [*port*] [Scheme Procedure]

Return `#t` if a character is ready on input *port* and return `#f` otherwise. If `char-ready?` returns `#t` then the next `read-char` operation on *port* is guaranteed not to hang. If *port* is a file port at end of file then `char-ready?` returns `#t`.

`char-ready?` exists to make it possible for a program to accept characters from interactive ports without getting stuck waiting for input. Any input editors associated with such ports must make sure that characters whose existence has been asserted by `char-ready?` cannot be rubbed out. If `char-ready?` were to return `#f` at end of file, a port at end of file would be indistinguishable from an interactive port that has no ready characters.

Note that `char-ready?` only works reliably for terminals and sockets with one-byte encodings. Under the hood it will return `#t` if the port has any input buffered, or if the file descriptor that backs the port polls as readable, indicating that Guile can fetch more bytes from the kernel. However being able to fetch one byte doesn't mean that a full character is available; См. Раздел 6.14.3 [Encoding], страница 355. Also, on many systems it's possible for a file descriptor to poll as readable, but then block when it comes time to read bytes. Note also that on Linux kernels, all file ports backed by files always poll as readable. For non-file ports, this procedure always returns `#t`, except for soft ports, which have a `char-ready?` handler. См. Раздел 6.14.10.5 [Soft Ports], страница 371.

In short, this is a legacy procedure whose semantics are hard to provide. However it is a useful check to see if any input is buffered. См. Раздел 6.14.14 [Non-Blocking I/O], страница 377.

`read-char` [*port*] [Scheme Procedure]
The same as `get-char`, except that *port* defaults to the current input port. См. Раздел 6.14.4 [Textual I/O], страница 357.

`peek-char` [*port*] [Scheme Procedure]
The same as `lookahead-char`, except that *port* defaults to the current input port. См. Раздел 6.14.4 [Textual I/O], страница 357.

`unread-char` *cobj* [*port*] [Scheme Procedure]
The same as `unget-char`, except that *port* defaults to the current input port, and the arguments are swapped. См. Раздел 6.14.4 [Textual I/O], страница 357.

`unread-string` *str port* [Scheme Procedure]
`scm_unread_string` (*str, port*) [C Function]
The same as `unget-string`, except that *port* defaults to the current input port, and the arguments are swapped. См. Раздел 6.14.4 [Textual I/O], страница 357.

`newline` [*port*] [Scheme Procedure]
Send a newline to *port*. If *port* is omitted, send to the current output port. Equivalent to `(put-char port #\newline)`.

`write-char` *chr* [*port*] [Scheme Procedure]
The same as `put-char`, except that *port* defaults to the current input port, and the arguments are swapped. См. Раздел 6.14.4 [Textual I/O], страница 357.

6.14.12 Using Ports from C

Guile's C interfaces provides some niceties for sending and receiving bytes and characters in a way that works better with C.

`size_t scm_c_read` (*SCM port, void *buffer, size_t size*) [C Function]
Read up to *size* bytes from *port* and store them in *buffer*. The return value is the number of bytes actually read, which can be less than *size* if end-of-file has been reached.

Note that as this is a binary input procedure, this function does not update `port-line` and `port-column` (см. Раздел 6.14.4 [Textual I/O], страница 357).

`void scm_c_write` (*SCM port, const void *buffer, size_t size*) [C Function]
Write *size* bytes at *buffer* to *port*.

Note that as this is a binary output procedure, this function does not update `port-line` and `port-column` (см. Раздел 6.14.4 [Textual I/O], страница 357).

`size_t scm_c_read_bytes` (*SCM port, SCM bv, size_t start, size_t count*) [C Function]
`void scm_c_write_bytes` (*SCM port, SCM bv, size_t start, size_t count*) [C Function]
Like `scm_c_read` and `scm_c_write`, but reading into or writing from the bytevector *bv*. *count* indicates the byte index at which to start in the bytevector, and the read or write will continue for *count* bytes.

`void scm_unget_bytes` (*const unsigned char *buf, size_t len, SCM port*) [C Function]
`void scm_unget_byte` (*int c, SCM port*) [C Function]
`void scm_ungetc` (*scm_t_wchar c, SCM port*) [C Function]
Like `unget-bytevector`, `unget-byte`, and `unget-char`, respectively. См. Раздел 6.14.4 [Textual I/O], страница 357.

`void scm_c_put_latin1_chars` (*SCM port, const scm_t_uint8 *buf, size_t len*) [C Function]
`void scm_c_put_utf32_chars` (*SCM port, const scm_t_uint32 *buf, size_t len*); [C Function]
Write a string to *port*. In the first case, the `scm_t_uint8*` buffer is a string in the latin-1 encoding. In the second, the `scm_t_uint32*` buffer is a string in the UTF-32 encoding. These routines will update the port's line and column.

6.14.13 Implementing New Port Types in C

This section describes how to implement a new port type in C. Although ports support many operations, as a data structure they present an opaque interface to the user. To the port implementor, you have two pieces of information to work with: the port type, and the port's "stream". The port type is an opaque pointer allocated when defining your port type. It is your key into the port API, and it helps you identify which ports are actually yours. The "stream" is a pointer you control, and which you set when you create a port. Get a stream from a port using the `SCM_STREAM` macro. Note that your port methods are only ever called with ports of your type.

A port type is created by calling `scm_make_port_type`. Once you have your port type, you can create ports with `scm_c_make_port`, or `scm_c_make_port_with_encoding`.

`scm_t_port_type* scm_make_port_type` (*char *name, size_t* (**read*) (*SCM port, SCM dst, size_t start, size_t count*), *size_t* (**write*) (*SCM port, SCM src, size_t start, size_t count*)) [Функция]
Define a new port type. The *name*, *read* and *write* parameters are initial values for those port type fields, as described below. The other fields are initialized with default values and can be changed later.

`SCM scm_c_make_port_with_encoding` (*scm_t_port_type *type, unsigned long mode_bits, SCM encoding, SCM conversion_strategy, scm_t_bits stream*) [Функция]
`SCM scm_c_make_port` (*scm_t_port_type *type, unsigned long mode_bits, scm_t_bits stream*) [Функция]
Make a port with the given *type*. The *stream* indicates the private data associated with the port, which your port implementation may later retrieve with `SCM_STREAM`. The mode bits should include one or more of the flags `SCM_RDNG` or `SCM_WRTNG`, indicating that the port is an input and/or an output port, respectively. The mode bits may also include `SCM_BUF0` or `SCM_BUFLINE`, indicating that the port should be unbuffered or line-buffered, respectively. The default is that the port will be block-buffered. См. Раздел 6.14.6 [Buffering], страница 360.

As you would imagine, *encoding* and *conversion_strategy* specify the port's initial textual encoding and conversion strategy. Both are symbols. `scm_c_make_port` is the same as `scm_c_make_port_with_encoding`, except it uses the default port encoding and conversion strategy.

The port type has a number of associate procedures and properties which collectively implement the port's behavior. Creating a new port type mostly involves writing these procedures.

`name` A pointer to a NUL terminated string: the name of the port type. This property is initialized via the first argument to `scm_make_port_type`.

`read` A port's `read` implementation fills read buffers. It should copy bytes to the supplied bytevector `dst`, starting at offset `start` and continuing for `count` bytes, returning the number of bytes read.

`write` A port's `write` implementation flushes write buffers to the mutable store. A port's `read` implementation fills read buffers. It should write out bytes from the supplied bytevector `src`, starting at offset `start` and continuing for `count` bytes, and return the number of bytes that were written.

`read_wait_fd`
`write_wait_fd`
If a port's `read` or `write` function returns `(size_t) -1`, that indicates that reading or writing would block. In that case to preserve the illusion of a blocking read or write operation, Guile's C port run-time will `poll` on the file descriptor returned by either the port's `read_wait_fd` or `write_wait_fd` function. Set using

`void scm_set_port_read_wait_fd (`*scm_t_port_type *type, int (*wait_fd) (SCM port)*`)` [Функция]

`void scm_set_port_write_wait_fd (`*scm_t_port_type *type, int (*wait_fd) (SCM port)*`)` [Функция]

Only a port type which implements the `read_wait_fd` or `write_wait_fd` port methods can usefully return `(size_t) -1` from a read or write function. См. Раздел 6.14.14 [Non-Blocking I/O], страница 377, for more on non-blocking I/O in Guile.

`print` Called when `write` is called on the port, to print a port description. For example, for a file port it may produce something like: `#<input: /etc/passwd 3>`. Set using

`void scm_set_port_print (`*scm_t_port_type *type, int (*print) (SCM port, SCM dest_port, scm_print_state *pstate)*`)` [Функция]

The first argument *port* is the port being printed, the second argument *dest_port* is where its description should go.

`close` Called when the port is closed. It should free any resources used by the port. Set using

`void scm_set_port_close (`*scm_t_port_type *type, void (*close) (SCM port)*`)` [Функция]

By default, ports that are garbage collected just go away without closing. If your port type needs to release some external resource like a file descriptor, or needs to make sure that its internal buffers are flushed even if the port is collected while it was open, then mark the port type as needing a close on GC.

`void scm_set_port_needs_close_on_gc (scm_t_port_type *type, int needs_close_p)` [Функция]

`seek` Set the current position of the port. Guile will flush read and/or write buffers before seeking, as appropriate.

`void scm_set_port_seek (scm_t_port_type *type, scm_t_off (*seek) (SCM port, scm_t_off offset, int whence))` [Функция]

`truncate` Truncate the port data to be specified length. Guile will flush buffers before hand, as appropriate. Set using

`void scm_set_port_truncate (scm_t_port_type *type, void (*truncate) (SCM port, scm_t_off length))` [Функция]

`random_access_p`

Determine whether this port is a random-access port.

Seeking on a random-access port with buffered input, or switching to writing after reading, will cause the buffered input to be discarded and Guile will seek the port back the buffered number of bytes. Likewise seeking on a random-access port with buffered output, or switching to reading after writing, will flush pending bytes with a call to the `write` procedure. См. Раздел 6.14.6 [Buffering], страница 360.

Indicate to Guile that your port needs this behavior by returning a nonzero value from your `random_access_p` function. The default implementation of this function returns nonzero if the port type supplies a seek implementation.

`void scm_set_port_random_access_p (scm_t_port_type *type, int (*random_access_p) (SCM port));` [Функция]

`get_natural_buffer_sizes`

Guile will internally attach buffers to ports. An input port always has a read buffer and an output port always has a write buffer. См. Раздел 6.14.6 [Buffering], страница 360. A port buffer consists of a bytevector, along with some cursors into that bytevector denoting where to get and put data.

Port implementations generally don't have to be concerned with buffering: a port type's `read` or `write` function will receive the buffer's bytevector as an argument, along with an offset and a length into that bytevector, and should then either fill or empty that bytevector. However in some cases, port implementations may be able to provide an appropriate default buffer size to Guile.

`void scm_set_port_get_natural_buffer_sizes (scm_t_port_type *type, void (*get_natural_buffer_sizes) (SCM, size_t *read_buf_size, size_t *write_buf_size))` [Функция]

Fill in *read_buf_size* and *write_buf_size* with an appropriate buffer size for this port, if one is known.

File ports implement a `get_natural_buffer_sizes` to let the operating system inform Guile about the appropriate buffer sizes for the particular file opened by the port.

Note that calls to all of these methods can proceed in parallel and concurrently and from any thread up until the point that the port is closed. The call to `close` will happen when no other method is running, and no method will be called after the `close` method is called. If your port implementation needs mutual exclusion to prevent concurrency, it is responsible for locking appropriately.

6.14.14 Non-Blocking I/O

Most ports in Guile are *blocking*: when you try to read a character from a port, Guile will block on the read until a character is ready, or end-of-stream is detected. Likewise whenever Guile goes to write (possibly buffered) data to an output port, Guile will block until all the data is written.

Interacting with ports in blocking mode is very convenient: you can write straightforward, sequential algorithms whose code flow reflects the flow of data. However, blocking I/O has two main limitations.

The first is that it's easy to get into a situation where code is waiting on data. Time spent waiting on data when code could be doing something else is wasteful and prevents your program from reaching its peak throughput. If you implement a web server that sequentially handles requests from clients, it's very easy for the server to end up waiting on a client to finish its HTTP request, or waiting on it to consume the response. The end result is that you are able to serve fewer requests per second than you'd like to serve.

The second limitation is related: a blocking parser over user-controlled input is a denial-of-service vulnerability. Indeed the so-called "slow loris" attack of the early 2010s was just that: an attack on common web servers that drip-fed HTTP requests, one character at a time. All it took was a handful of slow loris connections to occupy an entire web server.

In Guile we would like to preserve the ability to write straightforward blocking networking processes of all kinds, but under the hood to allow those processes to suspend their requests if they would block.

To do this, the first piece is to allow Guile ports to declare themselves as being nonblocking. This is currently supported only for file ports, which also includes sockets, terminals, or any other port that is backed by a file descriptor. To do that, we use an arcane UNIX incantation:

```
(let ((flags (fcntl socket F_GETFL)))
  (fcntl socket F_SETFL (logior O_NONBLOCK flags)))
```

Now the file descriptor is open in non-blocking mode. If Guile tries to read or write from this file and the read or write returns a result indicating that more data can only be had by doing a blocking read or write, Guile will block by polling on the socket's `read-wait-fd` or `write-wait-fd`, to preserve the illusion of a blocking read or write. См. Раздел 6.14.13 [I/O Extensions], страница 374, for more on those internal interfaces.

So far we have just reproduced the status quo: the file descriptor is non-blocking, but the operations on the port do block. To go farther, it would be nice if we could suspend the "thread" using delimited continuations, and only resume the thread once the file descriptor is readable or writable. (См. Раздел 6.13.5 [Prompts], страница 323).

But here we run into a difficulty. The ports code is implemented in C, which means that although we can suspend the computation to some outer prompt, we can't resume it because Guile can't resume delimited continuations that capture the C stack.

To overcome this difficulty we have created a compatible but entirely parallel implementation of port operations. To use this implementation, do the following:

```
(use-modules (ice-9 suspendable-ports))
(install-suspendable-ports!)
```

This will replace the core I/O primitives like `get-char` and `put-bytevector` with new versions that are exactly the same as the ones in the standard library, but with two differences. One is that when a read or a write would block, the suspendable port operations call out the value of the `current-read-waiter` or `current-write-waiter` parameter, as appropriate. См. Раздел 6.13.12 [Parameters], страница 347. The default read and write waiters do the same thing that the C read and write waiters do, which is to poll. User code can parameterize the waiters, though, enabling the computation to suspend and allow the program to process other I/O operations. Because the new suspendable ports implementation is written in Scheme, that suspended computation can resume again later when it is able to make progress. Success!

The other main difference is that because the new ports implementation is written in Scheme, it is slower than C, currently by a factor of 3 or 4, though it depends on many factors. For this reason we have to keep the C implementations as the default ones. One day when Guile's compiler is better, we can close this gap and have only one port operation implementation again.

Note that Guile does not currently include an implementation of the facility to suspend the current thread and schedule other threads in the meantime. Before adding such a thing, we want to make sure that we're providing the right primitives that can be used to build schedulers and other user-space concurrency patterns, and that the patterns that we settle on are the right patterns. In the meantime, have a look at 8sync (`https://gnu.org/software/8sync`) for a prototype of an asynchronous I/O and concurrency facility.

`install-suspendable-ports!` [Scheme Procedure]

Replace the core ports implementation with suspendable ports, as described above. This will mutate the values of the bindings like `get-char`, `put-u8`, and so on in place.

`uninstall-suspendable-ports!` [Scheme Procedure]

Restore the original core ports implementation, un-doing the effect of `install-suspendable-ports!`.

`current-read-waiter` [Scheme Parameter]
`current-write-waiter` [Scheme Parameter]

Parameters whose values are procedures of one argument, called when a suspendable port operation would block on a port while reading or writing, respectively. The default values of these parameters do a blocking `poll` on the port's file descriptor. The procedures are passed the port in question as their one argument.

6.14.15 Handling of Unicode Byte Order Marks

This section documents the finer points of Guile's handling of Unicode byte order marks (BOMs). A byte order mark (U+FEFF) is typically found at the start of a UTF-16 or UTF-32 stream, to allow readers to reliably determine the byte order. Occasionally, a BOM is found at the start of a UTF-8 stream, but this is much less common and not generally recommended.

Guile attempts to handle BOMs automatically, and in accordance with the recommendations of the Unicode Standard, when the port encoding is set to `UTF-8`, `UTF-16`, or `UTF-32`. In brief, Guile automatically writes a BOM at the start of a UTF-16 or UTF-32 stream, and automatically consumes one from the start of a UTF-8, UTF-16, or UTF-32 stream.

As specified in the Unicode Standard, a BOM is only handled specially at the start of a stream, and only if the port encoding is set to `UTF-8`, `UTF-16` or `UTF-32`. If the port encoding is set to `UTF-16BE`, `UTF-16LE`, `UTF-32BE`, or `UTF-32LE`, then BOMs are *not* handled specially, and none of the special handling described in this section applies.

- To ensure that Guile will properly detect the byte order of a UTF-16 or UTF-32 stream, you must perform a textual read before any writes, seeks, or binary I/O. Guile will not attempt to read a BOM unless a read is explicitly requested at the start of the stream.
- If a textual write is performed before the first read, then an arbitrary byte order will be chosen. Currently, big endian is the default on all platforms, but that may change in the future. If you wish to explicitly control the byte order of an output stream, set the port encoding to `UTF-16BE`, `UTF-16LE`, `UTF-32BE`, or `UTF-32LE`, and explicitly write a BOM (`#\xFEFF`) if desired.
- If `set-port-encoding!` is called in the middle of a stream, Guile treats this as a new logical "start of stream" for purposes of BOM handling, and will forget about any BOMs that had previously been seen. Therefore, it may choose a different byte order than had been used previously. This is intended to support multiple logical text streams embedded within a larger binary stream.
- Binary I/O operations are not guaranteed to update Guile's notion of whether the port is at the "start of the stream", nor are they guaranteed to produce or consume BOMs.
- For ports that support seeking (e.g. normal files), the input and output streams are considered linked: if the user reads first, then a BOM will be consumed (if appropriate), but later writes will *not* produce a BOM. Similarly, if the user writes first, then later reads will *not* consume a BOM.
- For ports that are not random access (e.g. pipes, sockets, and terminals), the input and output streams are considered *independent* for purposes of BOM handling: the first read will consume a BOM (if appropriate), and the first write will *also* produce a BOM (if appropriate). However, the input and output streams will always use the same byte order.
- Seeks to the beginning of a file will set the "start of stream" flags. Therefore, a subsequent textual read or write will consume or produce a BOM. However, unlike `set-port-encoding!`, if a byte order had already been chosen for the port, it will remain in effect after a seek, and cannot be changed by the presence of a BOM. Seeks anywhere other than the beginning of a file clear the "start of stream" flags.

6.15 Regular Expressions

Регулярное выражение (или *regexp*) представляет собой шаблон, который описывает целый класс строкч. Полное описание регулярных выражений и их синтаксис выодит за рамки настоящего руководства.

Если ваша система не содержит библиотеки регулярных выражений POSIX, и у вас нет связанной с Guile независимой библиотеки регулярных выражений, такой как Rx, эти функции будут вам не доступны. Вы можете узнать, включает ли ваша Guile поддержку регулярных выражений проверяя, возвращает ли `(provided? 'regex)` значение true.

Следующие функции регулярных выражений и соответствия строк предоставляются модулем `(ice-9 regex)`. Перед использованием описанных функций вы должны загрузить этот модуль, выполнив `(use-modules (ice-9 regex))`.

6.15.1 Regexp Functions

По умолчанию, Guile поддерживает расширенные регулярные выражения POSIX. Это означает, что символы '`(`', '`)`', '`+`' и '`?`' являются специальными, и должны быть экранированы, если вы хотите использовать эти символы буквально и нет поддержки "нежелательных" вариантов '`*`', '`+`' или '`?`'.

Этот интерфейс регулярных выражений был смоделирован после того, как реализован SCSH(Scheme shell). Он должен быть совместим с регулярными выражениями SCSH.

Нулевой байт (`#\nul`) не может использоваться в шаблонах регулярных выражений или входных строках, поскольку базовые функции Си рассматривают его как конец стороки. Если есть нулевой байт, возникнет ошибка.

Внутренние шаблоны и входные строки преобразуются в кодировку текущего языка и затем передаются в подпрограммы регулярных выражений библиотеки Си (см. Раздел "Regular Expressions" в *The GNU C Library Reference Manual*). Возвращенные структуры соответствия всегда указывают на символы в строках, а не на отдельные байты, даже в случае многобайтовых кодировок.

`string-match` *pattern str* [*start*] [Scheme Procedure]

Компилирует строку *pattern* в регулярное выражение и сравнивает его со строкой *str*. Опционально аргумент *start* указывает с какой позиции в строке *str* начинать сопоставление.

`string-match` возвращает *match structure* которая описывает что было сопоставлено с регулярным выражением. См. Раздел 6.15.2 [Match Structures], страница 384. Если *str* не соответствует шаблону *pattern* вообще, `string-match` возвращается `#f`.

Ниже приводятся два примера поиска соотвествий. В первом примере шаблон сопоставляется четырем цифрам в проверяемой строке. Во втором случае шаблону ничего не сопоставляется.

```
(string-match "[0-9][0-9][0-9][0-9]" "blah2002")
⇒ #("blah2002" (4 . 8))

(string-match "[A-Za-z]" "123456")
⇒ #f
```

Каждый раз когда вызывается `string-match`, функция должна скомпилировать свой аргумент *pattern* во внутреннюю структуру регулярного выражения. Это операция занимает много времени, что делает `string-match` не эффективной, если одно и тоже регулярное выражение используется несколько раз (например в цикле).

Для лучшей производительности, вы можете скомпилировать регулярное выражение заранее, а затем сопоставлять входные строки с уже скопилированным регулярным выражением.

`make-regexp` *pat flag...* [Scheme Procedure]
`scm_make_regexp` (*pat, flaglst*) [C Function]

Компилирует регулярное выражение описанное в *pat*, и возвращает скопилированную regexp структуру. Если *pat* не описывает корректное регулярное выражение, `make-regexp` вызывает ошибку с кодом `regular-expression-syntax`.

Аргумент *flag* изменяет поведение компилируемого регулярного выражения. Поддерживаются следующие значения флага:

`regexp/icase` [Переменная]

Считать прописные и строчные буквы одинаковыми при сравнении.

`regexp/newline` [Переменная]

Если в целевой строке появляется значение новой стоки, то операторы '`^`' и '`$`' соотвествуют ей сразу после или непосредственно перед ней, соответственно. Также , операторы '`.`' и '`[^...]`' никогда не будут соответствовать символу новой строки. Цель этого флага в обработке целевой строки в виде буфера, содержащего много строк текста и регулярное выражение как шаблон, который может соответствовать одной из этих строк

`regexp/basic` [Переменная]

Компилировать как базовое ("устаревшее") вместо расширенных ("современных") регулярных выражений, синтаксис которых принимается по умолчанию . Базовые регулярные выражения не считают символы '`|`', '`+`' или '`?`' специальными и требуют, чтобы метасимволы '`{...}`' и '`(...)`' экранировались обратным слешем. (см. Раздел 6.15.3 [Backslash Escapes], страница 386). Есть несколько других отличий между базовыми и расширенными регулярными выражениями, но эти самые значительные.

`regexp/extended` [Переменная]

Компилировать регуляроное выражение как расширенное, а не базовое регулярное выражение. Данное поведение происходит по умолчанию, этот флаг обычно не требуется. Если вызов `make-regexp` включает в себя оба флага `regexp/basic` и `regexp/extended`, тот который стоит последним будет отменять предыдущий.

`regexp-exec` *rx str* [*start* [*flags*]] [Scheme Procedure]
`scm_regexp_exec` (*rx, str, start, flags*) [C Function]

Сопоставляет скомпилированое регулярное выражение *rx* с входной строкой `str`. Если есть необязательный целый аргумент *start* сопоставление начинается с указанной позиции в строке. Возвращается структура, описывающая результа поиска соответсвия или `#f` если совпадение не может быть найдено.

Аргумент *flags* изменяет поведение поиска соответствия. Последовательность значений флага можеть быть объединена операцией `logior` (см. Раздел 6.6.2.13 [Bitwise Operations], страница 134),

`regexp/notbol` [Переменная]
Считает что начальное смещение *start* в строке *str* не является началом строки и не соответствует оператору '`^`'.

Если *rx* был создан с испльзованием параметра `regexp/newline` из опций указанных выше, '`^`' будет по прежнему соответствовать после новой стороки в буфере *str*.

`regexp/noteol` [Переменная]
Считает что конец *str* не является концом стороки и не должен совпадать с оператором '`$`'.

Если *rx* был создан с опцией `regexp/newline` указаных выше, '`$`' по прежнему будет соответствовать перед новой строкой в буфере *str*.

```
;; Regexp to match uppercase letters
(define r (make-regexp "[A-Z]*"))

;; Regexp to match letters, ignoring case
(define ri (make-regexp "[A-Z]*" regexp/icase))

;; Search for bob using regexp r
(match:substring (regexp-exec r "bob"))
⇒ ""                    ; no match

;; Search for bob using regexp ri
(match:substring (regexp-exec ri "Bob"))
⇒ "Bob"                 ; matched case insensitive
```

`regexp?` *obj* [Scheme Procedure]
`scm_regexp_p` (*obj*) [C Function]
Возвращает `#t` если *obj* является скомпилированным регулярным выражением или `#f` если это не так.

`list-matches` *regexp str* [*flags*] [Scheme Procedure]
Возвращает список структур соответствия, которые являются не перекрывающимися совпадениями регулярного выражения *regexp* с *str*. *regexp* может быть либо строкой шаблона, либо скомпилированным регулярным выражением. Аргумент *flags* аналогичен рассказанному выше `regexp-exec` .

```
(map match:substring (list-matches "[a-z]+" "abc 42 def 78"))
⇒ ("abc" "def")
```

`fold-matches` *regexp str init proc* [*flags*] [Scheme Procedure]
Применяет *proc* к неперекрывающимся совпадениям *regexp* с *str*, и выдает результат. *regexp* может быть строкой шаблона или скомпилированным регулярным выражением. Аргумент *flags* соответствует выше описанному `regexp-exec`.

proc вызывается как: (*proc* `match prev`) где *match* это структура соответствия и *prev* это предыдущие возвраты из *proc*. Для первого вызова *prev* используется параметр *init*. `fold-matches` возвращает окончательное значение из *proc*.

Для примера посчитаем количество совпадений

```
(fold-matches "[a-z][0-9]" "abc x1 def y2" 0
              (lambda (match count)
                (1+ count)))
⇒ 2
```

Регулярные выражения обычно используются для поиска шаблонов в одной строке и их замены содержимым другой строки. Для этого удобны следующие функции:

`regexp-substitute` *port match item . . .* [Scheme Procedure]

Пишет в *port* выделенную часть совпадения структуры соответсвтия. *match*. Или если *port* равен `#f` формирует строку из эти частей и возвращает ее.

Каждый элемент *item* указывает часть, которая должна быть записана, и может быть одним из следующих:

- Строка. Строковые аргументы вписываются дословно.
- Целое число. Будет записано найденное соответствие с этим номером (`match:substring`). Если Ноль то это все соотвествие.
- Символ ‘`pre`’. Будт вписана часть совпадающей строки, преджествующая регулярному выражению (`match:prefix`).
- Символ ‘`post`’. Будет вписана часть совпадающей строки стоящая после соответствия регулярному выражению (`match:suffix`).

Для примера, изменяем найденное соответствие и сохраняем текст до и после совпадения.

```
(regexp-substitute #f (string-match "[0-9]+" "number 25 is good")
                   'pre "37" 'post)
⇒ "number 37 is good"
```

Или сопоставление даты формата yyyymmdd такой как ‘20020828’ и реорганизация и перенос полей.

```
(define date-regex
   "([0-9][0-9][0-9][0-9])([0-9][0-9])([0-9][0-9])")
(define s "Date 20020429 12am.")
(regexp-substitute #f (string-match date-regex s)
                   'pre 2 "-" 3 "-" 1 'post " (" 0 ")")
⇒ "Date 04-29-2002 12am. (20020429)"
```

`regexp-substitute/global` *port regexp target item. . .* [Scheme Procedure]

Пишет в *port* выбранные части совпадений регулярного выражения *regexp* с *target*. Если *port* равен `#f` тогда формируется строка из этих частей и возвращается. *regexp* может быть строкой или скомпилированным регулярным выражением.

Работает похоже на `regexp-substitute`, но позволяет проводить глобальные подстановки в целевой строке *target*. Каждый элемент *item* ведет себя так же как в функции `regexp-substitute`, со следующими отличиями:

- Функция. Вызывается как (*item* match) с параметром структурой соответствия регулярному выражению *regexp*, она должна вернуть строку которая и будет записана в порт *port*.
- Символ 'post'. Здесь он ничего не выводит, а вместо этого вызывает рекурсивно regexp-substitute/global для оставшейся части *target*.

  Это необходимо для выполнения глобального поиска и замены на входной строке *target*; без этого regexp-substitute/global делает возврат после одного совпадения и вывода.

Например, чтобы свернуть все последовательности табуляций и пробелов в один дефис(для каждой последовательности)

```
(regexp-substitute/global #f "[ \t]+"  "this   is   the text"
                          'pre "-" 'post)
⇒ "this-is-the-text"
```

Или что бы использовать функцию reverse, что бы переставить буквы в каждом слове.

```
(regexp-substitute/global #f "[a-z]+"  "to do and not-do"
  'pre (lambda (m) (string-reverse (match:substring m))) 'post)
⇒ "ot od dna ton-od"
```

Без символа post, выполняется только одно регулярное совпадение. Например, следующее: для даты из указанного выше regexp-substitute, отдельного строкового вызова string-match.

```
(define date-regex
   "([0-9][0-9][0-9][0-9])([0-9][0-9])([0-9][0-9])")
(define s "Date 20020429 12am.")
(regexp-substitute/global #f date-regex s
                          'pre 2 "-" 3 "-" 1 'post " (" 0 ")")

⇒ "Date 04-29-2002 12am. (20020429)"
```

6.15.2 Match Structures

Структура соответствия (*match structure*) это объек, возвращаемый string-match и regexp-exec. Он описывает какая часть строки, если таковая имеется, соответствует данному регулярному выражению. Структура соответствия включает: ссылку на строку, которая была проверена на сответствие, начальную и конечную позицию совпадения с регулярным выражением, и если регулярное выражение содержит любые подвыражения в скобках, начальную и конечную позицию каждого совпадения с этими подвыражениями.

В каждой из функций соответствия регулярному выражению, описанных ниже, аргумент соответствия должен быть структурой сопоставления, возвращенной предыдущим вызовом string-match или regexp-exec. Большинство этих функций возвращают некоторую информацию об исходной строке, которая была сопоставлена с регулярным выражением; мы будем называть эту строку целевой (*target*) для простоты обращения.

`regexp-match?` *obj* [Scheme Procedure]

Возвращает `#t` если *obj* является структурой соответствия возвращенной предыдущим вызовом `regexp-exec`, если нет `#f`.

`match:substring` *match* [*n*] [Scheme Procedure]

Возвращает часть целевой строки*target* соответствующую номеру подвыражения *n*. Подвыражение 0 (по умолчанию) представляет полное совпадение регулярного выражения. Если регулярное выражение в целом сопоставлено, но число подвыражений не *n* не совпало, возвращается `#f`.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:substring s)
⇒ "2002"

;; match starting at offset 6 in the string
(match:substring
  (string-match "[0-9][0-9][0-9][0-9]" "blah987654" 6))
⇒ "7654"
```

`match:start` *match* [*n*] [Scheme Procedure]

Возвращает начальную позицию совпадения с подвыражением *n*.

В последующем примере результат равен 4, так как совпадение начинается с символа с численным индексом 4:

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:start s)
⇒ 4
```

`match:end` *match* [*n*] [Scheme Procedure]

Возвращает конечную позицию совпадения с подвыражением *n*.

In the following example, the result is 8, since the match runs between characters 4 and 8 (т.е. “2002”)

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:end s)
⇒ 8
```

`match:prefix` *match* [Scheme Procedure]

Возвращет не совпавшую часть целевой строки *target* предшествующую совпадению с регулярным выражением.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:prefix s)
⇒ "blah"
```

`match:suffix` *match* [Scheme Procedure]

Возвращает несовпавшую часть целевой строки *target* следующую за совпадением с регулярным выражением.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:suffix s)
⇒ "foo"
```

`match:count` *match* [Scheme Procedure]

Возвращает количество совпавших подвыражений в скобках из *match*. Обратите внимание, что полное совпадение с регулярным выражением считается также совпавшим подвыражением, а неудачные подвыражения также включены в счет.

`match:string` *match* [Scheme Procedure]

Возвращает исходную целевую строку *target*.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:string s)
⇒ "blah2002foo"
```

6.15.3 Backslash Escapes

Иногда вам нужно, что бы регулярное выражение соответствовало символам типа '`*`' или '`$`'. Например, чтобы проверить, указывает ли конкретная строка на запись меню из узла Info, ее нужно будет сопоставить с регулярным выражением, например: '`^* [^:]*::`'. Однако это не сработает, так как звездочка является метасимволом, она не будет соответствовать симовлу '`*`' в начале строки. В нашем случае мы хотим чтобы первая звездочка была обычным символом.

Вы можете сделать это указав метасимвол с символом обратной косой черты '`\`'. (Это так называемое цитирование метасимвола *quoting*, и известное как *backslash escape*.) Когда Guile видит обратную косую черту в регулярном выражении, он считает, что следующий символ является обычным, не зависимо от того какой особый смысл он обычно имеет. Поэтому мы можем выполнить описанный выше пример, изменив регулярное выражение на '`^\* [^:]*::`'. Последовательность '`\*`' сообщает движку регулярных выражений что соответствие должно быть одной звездочке в обрабатываемой стоке.

Поскольку обратная косая черта сама по себе является метасимволом, вы можете заставить регулярное выражение соответствовать обратной косой черте в обрабатываемой строке, предваряя обратную косую черту самой собой(обратной косой чертой). Например, чтобы найти переменную ссылки в программе TEX, вам может потребоваться найти вхождения строки '`\let\`' следующим за ним любым количеством символов. Регулярное выражение '`\\let\\[A-Za-z]*`' будет делать это: двойная обратная косая черта в регулярном выражении соответствует обычной одиночной косой черте в обрабатываемой строке.

`regexp-quote` *str* [Scheme Procedure]

Предваряет каждый специальный символ в *str* обратной косой чертой и возвращает строковый результат.

Очень важно: Использование обратной косой черты в исходном коде Guile source (как в Emacs Lisp или C) может быть сложным, потому что символ обратной косой черты имеет особое значение для Guile. Например, если Guile встречает последовательность символов '`\n`' в середине строки, в то время как обрабатывается код Scheme, он заменяет эти символы символом новой строки. Аналогичным образом, последовательность символов '`\t`' заменяется символом табуляции. Некоторые из этих

escape последовательностей обрабатываются Guile до того как ваш код будет выполнен. Неопознанные управляющие последовательности игнорируются: если символ '`\*`' появляется в строке он будет переведен в одиночный символ '`*`'.

Эта трансляция явно нежелательна для регулярных выражений, поскольку мы хотим включать обратную косую черту в строку, чтобы экранировать метасимволы в регулярном выражении. Следовательно, чтобы убедиться, что обратная косая черта сохраняется в вашей программе Guile, вы должны использовать две последовательные обратные косые черты.

```
(define Info-menu-entry-pattern (make-regexp "^\\* [^:]*"))
```

Строка в этом примере предварительно обрабатывается клиентом Guile перед выполнением любого кода. Результирующим аргументом `make-regexp` будет строка '`^\* [^:]*`', что нам и надо.

Это также означает, что для того, чтобы написать регулярное выражение, которое соответствует одному символу обратной косой черты, строка регулярного выражения должна включать в себя ЧЕТЫРЕ!!!! обратных косых черты. Каждая последовательная пара обратных косых черт переводиться Guile на одиночную обратную косую черту, и полученая двойная обратная косая черта интерпретируется движком regexp как сопоставление с одним символом обратной косой черты. Следовательно:

```
(define tex-variable-pattern (make-regexp "\\\\let\\\\=[A-Za-z]*"))
```

Причина громоздкости этого синтаксиса является исторической. Обе системы: построения шаблонов регулярных выражений и система обработки строк Unix традиционно использовали обратную косую черту со специальными назначением, описанным выше. Спецификация регулярного выражения POSIX и стандарт ANSI C требуют этой семантики. Попытка отказаться от любой конвенции может вызвать другие проблемы совместимости, возможно более серьезные. Следовательно, без расширения программы Scheme для поддержки различными соглашениями о квотировании(экранировании) (неуправляемое и запутанное расширение, когда оно реализуется на других языках) мы должны придерживаться этого громоздкого escape синтаксиса.

6.16 LALR(1) Разбор(Parsing)

Модуль `(system base lalr)` предоставляет `lalr-scm` LALR(1) генератор парсеров от Dominique Boucher (`http://code.google.com/p/lalr-scm/`). `lalr-scm` использует тот же самый алгоритм как GNU Bison (см. Раздел "Introduction" в *Bison, это Yacc-совместимый генератор разборщиков(Parser)*). Парсеры определяются с помощью использования макроса `lalr-parser`.

`lalr-parser` [*options*] *tokens rules...* [Scheme Syntax]

Создает синтаксический анализатор LALR(1). *tokens* это список символов, представляющих терминальные символы граматики. *rules* это правила продукций граматики.

Каждое правило имеет вид (*non-terminal* (*rhs* ...) : *action* ...), где *non-terminal* это имя правила, *rhs* это правые стороны, т.е. продукционные правила, и *action* это семантическое действие связанное с правилом.

Сгенерированный парсер представляет собой процедуру с двумя аргументами, которая принимает *tokenizer* и *syntax error procedure*. tokenizer(токенайзер) дол-

жен быть чанком(кодом схемы) который возвращает лексические токены, созданные как `make-lexical-token`. Процедура синтаксической ошибки(syntax error procedure) может быть вызвана с, по крайней мере, сообщением об ошибке(строкой) и необязательно, лексическим токеном, который вызвал ошибку.

Пожалуйста обратитесь к документации `lalr-scm` за деталями.

6.17 Разбор(Parsing) PEG

Разбор граматических выражений(Parsing Expression Grammars — PEG) это спообб укзания формальных языков для обработки текстов. Их можно использовать либо для сопоставления (подобно регулярным выраженям), либо для построения парсеров рекурсивного спуска (таких как lex/yacc). Guile использует расширенный синтаксис PEG, который позволяет получить больший контролироль над тем, какая информация сохраняется при разборе.

Wikipedia имеет четкое и краткое введение в PEGs если вы хотите ознакомиться с его синтаксисом(есть также ее перевод на русском, там же): `http://en.wikipedia.org/wiki/Parsing_expression_grammar`.

Модуль `(ice-9 peg)` работает путем компиляциинg PEGs до лямбда выражений. Они могут быть сохранены в переменных во время компиляции с помощью макросов (`define-peg-pattern` и `define-peg-string-patterns`) или вычислелны явно во время выполнения с функциями компиляции (`compile-peg-pattern` и `peg-string-compile`).

Затем их можно использовать либо для разбора (`match-pattern`) либо для поиска (`search-for-pattern`). Для удобства, `search-for-pattern` также принимает образцы литералов в случае, если вы хотите встроить простой поиск (люди часто используют регулярные выражения таким образом).

Остальная часть этой документации, состоит из ссылок на синтаксис, сссылок на API и обучения.

6.17.1 PEG Syntax Reference

Normal PEG Syntax:

`sequence` *a b* [PEG Pattern]

Parses *a*. If this succeeds, continues to parse *b* from the end of the text parsed as *a*. Succeeds if both *a* and *b* succeed.

```
"a b"
```

```
(and a b)
```

`ordered choice` *a b* [PEG Pattern]

Parses *a*. If this fails, backtracks and parses *b*. Succeeds if either *a* or *b* succeeds.

```
"a/b"
```

```
(or a b)
```

`zero or more` *a* [PEG Pattern]

Parses *a* as many times in a row as it can, starting each *a* at the end of the text parsed by the previous *a*. Always succeeds.

```
"a*"
(* a)
```

`one or more` *a* [PEG Pattern]

Parses *a* as many times in a row as it can, starting each *a* at the end of the text parsed by the previous *a*. Succeeds if at least one *a* was parsed.

```
"a+"
(+ a)
```

`optional` *a* [PEG Pattern]

Tries to parse *a*. Succeeds if *a* succeeds.

```
"a?"
(? a)
```

`followed by` *a* [PEG Pattern]

Makes sure it is possible to parse *a*, but does not actually parse it. Succeeds if *a* would succeed.

```
"&a"
(followed-by a)
```

`not followed by` *a* [PEG Pattern]

Makes sure it is impossible to parse *a*, but does not actually parse it. Succeeds if *a* would fail.

```
"!a"
(not-followed-by a)
```

`string literal` *"abc"* [PEG Pattern]

Parses the string "*abc*". Succeeds if that parsing succeeds.

```
"'abc'"
"abc"
```

`any character` [PEG Pattern]

Parses any single character. Succeeds unless there is no more text to be parsed.

```
"."
peg-any
```

`character class` *a b* [PEG Pattern]

Alternative syntax for "Ordered Choice *a b*" if *a* and *b* are characters.

```
"[ab]"
(or "a" "b")
```

`range of characters` *a z* [PEG Pattern]

Parses any character falling between *a* and *z*.

```
"[a-z]"
(range #\a #\z)
```

Example:

```
"(a !b / c &d*) 'e'+"
```

Would be:

```
(and
 (or
  (and a (not-followed-by b))
  (and c (followed-by (* d))))
 (+ "e"))
```

Extended Syntax

There is some extra syntax for S-expressions.

`ignore` *a* [PEG Pattern]
: Ignore the text matching *a*

`capture` *a* [PEG Pattern]
: Capture the text matching *a*.

`peg` *a* [PEG Pattern]
: Embed the PEG pattern *a* using string syntax.

Example:

```
"!a / 'b'"
```

Is equivalent to

```
(or (peg "!a") "b")
```

and

```
(or (not-followed-by a) "b")
```

6.17.2 PEG API Reference

Define Macros

The most straightforward way to define a PEG is by using one of the define macros (both of these macroexpand into `define` expressions). These macros bind parsing functions to variables. These parsing functions may be invoked by `match-pattern` or `search-for-pattern`, which return a PEG match record. Raw data can be retrieved from this record with the PEG match deconstructor functions. More complicated (and perhaps enlightening) examples can be found in the tutorial.

`define-peg-string-patterns` *peg-string* [Scheme Macro]
: Defines all the nonterminals in the PEG *peg-string*. More precisely, `define-peg-string-patterns` takes a superset of PEGs. A normal PEG has a `<-` between the nonterminal and the pattern. `define-peg-string-patterns` uses this symbol to determine what information it should propagate up the parse tree. The normal `<-` propagates the matched text up the parse tree, `<--` propagates the matched text up the parse tree tagged with the name of the nonterminal, and `<` discards that matched text and propagates nothing up the parse tree. Also, nonterminals may consist of any

alphanumeric character or a "-" character (in normal PEGs nonterminals can only be alphabetic).

For example, if we:

```
(define-peg-string-patterns
  "as <- 'a'+
bs <- 'b'+
as-or-bs <- as/bs")
(define-peg-string-patterns
  "as-tag <-- 'a'+
bs-tag <-- 'b'+
as-or-bs-tag <-- as-tag/bs-tag")
```

Then:

```
(match-pattern as-or-bs "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>
(match-pattern as-or-bs-tag "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: (as-or-bs-tag (as-tag aa))>
```

Note that in doing this, we have bound 6 variables at the toplevel (*as*, *bs*, *as-or-bs*, *as-tag*, *bs-tag*, and *as-or-bs-tag*).

`define-peg-pattern` *name capture-type peg-sexp* [Scheme Macro]

Defines a single nonterminal *name*. *capture-type* determines how much information is passed up the parse tree. *peg-sexp* is a PEG in S-expression form.

Possible values for capture-type:

`all` passes the matched text up the parse tree tagged with the name of the nonterminal.

`body` passes the matched text up the parse tree.

`none` passes nothing up the parse tree.

For Example, if we:

```
(define-peg-pattern as body (+ "a"))
(define-peg-pattern bs body (+ "b"))
(define-peg-pattern as-or-bs body (or as bs))
(define-peg-pattern as-tag all (+ "a"))
(define-peg-pattern bs-tag all (+ "b"))
(define-peg-pattern as-or-bs-tag all (or as-tag bs-tag))
```

Then:

```
(match-pattern as-or-bs "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>
(match-pattern as-or-bs-tag "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: (as-or-bs-tag (as-tag aa))>
```

Note that in doing this, we have bound 6 variables at the toplevel (*as*, *bs*, *as-or-bs*, *as-tag*, *bs-tag*, and *as-or-bs-tag*).

Compile Functions

It is sometimes useful to be able to compile anonymous PEG patterns at runtime. These functions let you do that using either syntax.

`peg-string-compile` *peg-string capture-type* [Scheme Procedure]
Compiles the PEG pattern in *peg-string* propagating according to *capture-type* (capture-type can be any of the values from `define-peg-pattern`).

`compile-peg-pattern` *peg-sexp capture-type* [Scheme Procedure]
Compiles the PEG pattern in *peg-sexp* propagating according to *capture-type* (capture-type can be any of the values from `define-peg-pattern`).

The functions return syntax objects, which can be useful if you want to use them in macros. If all you want is to define a new nonterminal, you can do the following:

```
(define exp '(+ "a"))
(define as (compile (compile-peg-pattern exp 'body)))
```

You can use this nonterminal with all of the regular PEG functions:

```
(match-pattern as "aaaaa") ⇒
#<peg start: 0 end: 5 string: bbbbb tree: bbbbb>
```

Parsing & Matching Functions

For our purposes, "parsing" means parsing a string into a tree starting from the first character, while "matching" means searching through the string for a substring. In practice, the only difference between the two functions is that `match-pattern` gives up if it can't find a valid substring starting at index 0 and `search-for-pattern` keeps looking. They are both equally capable of "parsing" and "matching" given those constraints.

`match-pattern` *nonterm string* [Scheme Procedure]
Parses *string* using the PEG stored in *nonterm*. If no match was found, `match-pattern` returns false. If a match was found, a PEG match record is returned.

The `capture-type` argument to `define-peg-pattern` allows you to choose what information to hold on to while parsing. The options are:

`all` tag the matched text with the nonterminal

`body` just the matched text

`none` nothing

```
(define-peg-pattern as all (+ "a"))
(match-pattern as "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: (as aa)>

(define-peg-pattern as body (+ "a"))
(match-pattern as "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>

(define-peg-pattern as none (+ "a"))
```

```
(match-pattern as "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: ()>

(define-peg-pattern bs body (+ "b"))
(match-pattern bs "aabbcc") ⇒
#f
```

`search-for-pattern` *nonterm-or-peg string* [Scheme Macro]

Searches through *string* looking for a matching subexpression. *nonterm-or-peg* can either be a nonterminal or a literal PEG pattern. When a literal PEG pattern is provided, `search-for-pattern` works very similarly to the regular expression searches many hackers are used to. If no match was found, `search-for-pattern` returns false. If a match was found, a PEG match record is returned.

```
(define-peg-pattern as body (+ "a"))
(search-for-pattern as "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>
(search-for-pattern (+ "a") "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>
(search-for-pattern "'a'+" "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>

(define-peg-pattern as all (+ "a"))
(search-for-pattern as "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: (as aa)>

(define-peg-pattern bs body (+ "b"))
(search-for-pattern bs "aabbcc") ⇒
#<peg start: 2 end: 4 string: aabbcc tree: bb>
(search-for-pattern (+ "b") "aabbcc") ⇒
#<peg start: 2 end: 4 string: aabbcc tree: bb>
(search-for-pattern "'b'+" "aabbcc") ⇒
#<peg start: 2 end: 4 string: aabbcc tree: bb>

(define-peg-pattern zs body (+ "z"))
(search-for-pattern zs "aabbcc") ⇒
#f
(search-for-pattern (+ "z") "aabbcc") ⇒
#f
(search-for-pattern "'z'+" "aabbcc") ⇒
#f
```

PEG Match Records

The `match-pattern` and `search-for-pattern` functions both return PEG match records. Actual information can be extracted from these with the following functions.

`peg:string` *match-record* [Scheme Procedure]

Returns the original string that was parsed in the creation of `match-record`.

`peg:start` *match-record* [Scheme Procedure]
Returns the index of the first parsed character in the original string (from `peg:string`). If this is the same as `peg:end`, nothing was parsed.

`peg:end` *match-record* [Scheme Procedure]
Returns one more than the index of the last parsed character in the original string (from `peg:string`). If this is the same as `peg:start`, nothing was parsed.

`peg:substring` *match-record* [Scheme Procedure]
Returns the substring parsed by `match-record`. This is equivalent to `(substring (peg:string match-record) (peg:start match-record) (peg:end match-record))`.

`peg:tree` *match-record* [Scheme Procedure]
Returns the tree parsed by `match-record`.

`peg-record?` *match-record* [Scheme Procedure]
Returns true if `match-record` is a PEG match record, or false otherwise.

Example:

```
(define-peg-pattern bs all (peg "'b'+"))

(search-for-pattern bs "aabbcc") ⇒
#<peg start: 2 end: 4 string: aabbcc tree: (bs bb)>

(let ((pm (search-for-pattern bs "aabbcc")))
   `((string ,(peg:string pm))
     (start ,(peg:start pm))
     (end ,(peg:end pm))
     (substring ,(peg:substring pm))
     (tree ,(peg:tree pm))
     (record? ,(peg-record? pm)))) ⇒
((string "aabbcc")
 (start 2)
 (end 4)
 (substring "bb")
 (tree (bs "bb"))
 (record? #t))
```

Miscellaneous

`context-flatten` *tst lst* [Scheme Procedure]
Takes a predicate *tst* and a list *lst*. Flattens *lst* until all elements are either atoms or satisfy *tst*. If *lst* itself satisfies *tst*, `(list lst)` is returned (this is a flat list whose only element satisfies *tst*).

```
(context-flatten (lambda (x) (and (number? (car x)) (= (car x) 1))) '(2 2 (1 1 (2
(2 2 (1 1 (2 2)) 2 2 (1 1))
(context-flatten (lambda (x) (and (number? (car x)) (= (car x) 1))) '(1 1 (1 1 (2
```

```
((1 1 (1 1 (2 2)) (2 2 (1 1))))
```

If you're wondering why this is here, take a look at the tutorial.

`keyword-flatten` *terms lst* [Scheme Procedure]

A less general form of `context-flatten`. Takes a list of terminal atoms `terms` and flattens *lst* until all elements are either atoms, or lists which have an atom from `terms` as their first element.

```
(keyword-flatten '(a b) '(c a b (a c) (b c) (c (b a) (c a)))) ⇒
(c a b (a c) (b c) c (b a) c a)
```

If you're wondering why this is here, take a look at the tutorial.

6.17.3 PEG Tutorial

Parsing /etc/passwd

This example will show how to parse /etc/passwd using PEGs.

First we define an example /etc/passwd file:

```
(define *etc-passwd*
  "root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
messagebus:x:103:107::/var/run/dbus:/bin/false
")
```

As a first pass at this, we might want to have all the entries in /etc/passwd in a list.

Doing this with string-based PEG syntax would look like this:

```
(define-peg-string-patterns
  "passwd <- entry* !.
entry <-- (! NL .)* NL*
NL < '\n'")
```

A `passwd` file is 0 or more entries (`entry*`) until the end of the file (`!.` (`.` is any character, so `!.` means "not anything")). We want to capture the data in the nonterminal `passwd`, but not tag it with the name, so we use `<-`.

An entry is a series of 0 or more characters that aren't newlines (`(! NL .)*`) followed by 0 or more newlines (`NL*`). We want to tag all the entries with `entry`, so we use `<--`.

A newline is just a literal newline (`'\n'`). We don't want a bunch of newlines cluttering up the output, so we use `<` to throw away the captured data.

Here is the same PEG defined using S-expressions:

```
(define-peg-pattern passwd body (and (* entry) (not-followed-by peg-any)))
(define-peg-pattern entry all (and (* (and (not-followed-by NL) peg-any))
       (* NL)))
(define-peg-pattern NL none "\n")
```

Obviously this is much more verbose. On the other hand, it's more explicit, and thus easier to build automatically. However, there are some tricks that make S-expressions

easier to use in some cases. One is the `ignore` keyword; the string syntax has no way to say "throw away this text" except breaking it out into a separate nonterminal. For instance, to throw away the newlines we had to define `NL`. In the S-expression syntax, we could have simply written `(ignore "\n")`. Also, for the cases where string syntax is really much cleaner, the `peg` keyword can be used to embed string syntax in S-expression syntax. For instance, we could have written:

```
(define-peg-pattern passwd body (peg "entry* !."))
```

However we define it, parsing `*etc-passwd*` with the `passwd` nonterminal yields the same results:

```
(peg:tree (match-pattern passwd *etc-passwd*)) ⇒
((entry "root:x:0:0:root:/root:/bin/bash")
 (entry "daemon:x:1:1:daemon:/usr/sbin:/bin/sh")
 (entry "bin:x:2:2:bin:/bin:/bin/sh")
 (entry "sys:x:3:3:sys:/dev:/bin/sh")
 (entry "nobody:x:65534:65534:nobody:/nonexistent:/bin/sh")
 (entry "messagebus:x:103:107::/var/run/dbus:/bin/false"))
```

However, here is something to be wary of:

```
(peg:tree (match-pattern passwd "one entry")) ⇒
(entry "one entry")
```

By default, the parse trees generated by PEGs are compressed as much as possible without losing information. It may not look like this is what you want at first, but uncompressed parse trees are an enormous headache (there's no easy way to predict how deep particular lists will nest, there are empty lists littered everywhere, etc. etc.). One side-effect of this, however, is that sometimes the compressor is too aggressive. No information is discarded when `((entry "one entry"))` is compressed to `(entry "one entry")`, but in this particular case it probably isn't what we want.

There are two functions for easily dealing with this: `keyword-flatten` and `context-flatten`. The `keyword-flatten` function takes a list of keywords and a list to flatten, then tries to coerce the list such that the first element of all sublists is one of the keywords. The `context-flatten` function is similar, but instead of a list of keywords it takes a predicate that should indicate whether a given sublist is good enough (refer to the API reference for more details).

What we want here is `keyword-flatten`.

```
(keyword-flatten '(entry) (peg:tree (match-pattern passwd *etc-passwd*))) ⇒
((entry "root:x:0:0:root:/root:/bin/bash")
 (entry "daemon:x:1:1:daemon:/usr/sbin:/bin/sh")
 (entry "bin:x:2:2:bin:/bin:/bin/sh")
 (entry "sys:x:3:3:sys:/dev:/bin/sh")
 (entry "nobody:x:65534:65534:nobody:/nonexistent:/bin/sh")
 (entry "messagebus:x:103:107::/var/run/dbus:/bin/false"))
(keyword-flatten '(entry) (peg:tree (match-pattern passwd "one entry"))) ⇒
((entry "one entry"))
```

Of course, this is a somewhat contrived example. In practice we would probably just tag the `passwd` nonterminal to remove the ambiguity (using either the `all` keyword for S-expressions or the `<--` symbol for strings)..

```
(define-peg-pattern tag-passwd all (peg "entry* !."))
(peg:tree (match-pattern tag-passwd *etc-passwd*)) ⇒
(tag-passwd
  (entry "root:x:0:0:root:/root:/bin/bash")
  (entry "daemon:x:1:1:daemon:/usr/sbin:/bin/sh")
  (entry "bin:x:2:2:bin:/bin:/bin/sh")
  (entry "sys:x:3:3:sys:/dev:/bin/sh")
  (entry "nobody:x:65534:65534:nobody:/nonexistent:/bin/sh")
  (entry "messagebus:x:103:107::/var/run/dbus:/bin/false"))
(peg:tree (match-pattern tag-passwd "one entry"))
(tag-passwd
  (entry "one entry"))
```

If you're ever uncertain about the potential results of parsing something, remember the two absolute rules:

1. No parsing information will ever be discarded.
2. There will never be any lists with fewer than 2 elements.

For the purposes of (1), "parsing information" means things tagged with the `any` keyword or the `<--` symbol. Plain strings will be concatenated.

Let's extend this example a bit more and actually pull some useful information out of the passwd file:

```
(define-peg-string-patterns
  "passwd <-- entry* !.
entry <-- login C pass C uid C gid C nameORcomment C homedir C shell NL*
login <-- text
pass <-- text
uid <-- [0-9]*
gid <-- [0-9]*
nameORcomment <-- text
homedir <-- path
shell <-- path
path <-- (SLASH pathELEMENT)*
pathELEMENT <-- (!NL !C  !'/' .)*
text <- (!NL !C  .)*
C < ':'
NL < '\n'
SLASH < '/'")
```

This produces rather pretty parse trees:

```
(passwd
  (entry (login "root")
         (pass "x")
         (uid "0")
         (gid "0")
         (nameORcomment "root")
         (homedir (path (pathELEMENT "root")))
         (shell (path (pathELEMENT "bin") (pathELEMENT "bash")))))
```

```
(entry (login "daemon")
       (pass "x")
       (uid "1")
       (gid "1")
       (nameORcomment "daemon")
       (homedir
         (path (pathELEMENT "usr") (pathELEMENT "sbin")))
       (shell (path (pathELEMENT "bin") (pathELEMENT "sh"))))
(entry (login "bin")
       (pass "x")
       (uid "2")
       (gid "2")
       (nameORcomment "bin")
       (homedir (path (pathELEMENT "bin")))
       (shell (path (pathELEMENT "bin") (pathELEMENT "sh"))))
(entry (login "sys")
       (pass "x")
       (uid "3")
       (gid "3")
       (nameORcomment "sys")
       (homedir (path (pathELEMENT "dev")))
       (shell (path (pathELEMENT "bin") (pathELEMENT "sh"))))
(entry (login "nobody")
       (pass "x")
       (uid "65534")
       (gid "65534")
       (nameORcomment "nobody")
       (homedir (path (pathELEMENT "nonexistent")))
       (shell (path (pathELEMENT "bin") (pathELEMENT "sh"))))
(entry (login "messagebus")
       (pass "x")
       (uid "103")
       (gid "107")
       nameORcomment
       (homedir
         (path (pathELEMENT "var")
               (pathELEMENT "run")
               (pathELEMENT "dbus")))
       (shell (path (pathELEMENT "bin") (pathELEMENT "false")))))
```

Notice that when there's no entry in a field (e.g. `nameORcomment` for messagebus) the symbol is inserted. This is the "don't throw away any information" rule—we succesfully matched a `nameORcomment` of 0 characters (since we used `*` when defining it). This is usually what you want, because it allows you to e.g. use `list-ref` to pull out elements (since they all have known offsets).

If you'd prefer not to have symbols for empty matches, you can replace the `*` with a `+` and add a `?` after the `nameORcomment` in `entry`. Then it will try to parse 1 or more

characters, fail (inserting nothing into the parse tree), but continue because it didn't have to match the nameORcomment to continue.

Embedding Arithmetic Expressions

We can parse simple mathematical expressions with the following PEG:

```
(define-peg-string-patterns
  "expr <- sum
sum <-- (product ('+' / '-') sum) / product
product <-- (value ('*' / '/') product) / value
value <-- number / '(' expr ')'
number <-- [0-9]+")
```

Then:

```
(peg:tree (match-pattern expr "1+1/2*3+(1+1)/2")) ⇒
(sum (product (value (number "1")))
     "+"
     (sum (product
            (value (number "1"))
            "/"
            (product
              (value (number "2"))
              "*"
              (product (value (number "3")))))
          "+"
          (sum (product
                 (value "("
                        (sum (product (value (number "1")))
                             "+"
                             (sum (product (value (number "1")))))
                        ")")
                 "/"
                 (product (value (number "2")))))))
```

There is very little wasted effort in this PEG. The `number` nonterminal has to be tagged because otherwise the numbers might run together with the arithmetic expressions during the string concatenation stage of parse-tree compression (the parser will see "1" followed by "/" and decide to call it "1/"). When in doubt, tag.

It is very easy to turn these parse trees into lisp expressions:

```
(define (parse-sum sum left . rest)
  (if (null? rest)
      (apply parse-product left)
      (list (string->symbol (car rest))
    (apply parse-product left)
    (apply parse-sum (cadr rest)))))

(define (parse-product product left . rest)
  (if (null? rest)
```

```
      (apply parse-value left)
      (list (string->symbol (car rest))
    (apply parse-value left)
    (apply parse-product (cadr rest)))))

(define (parse-value value first . rest)
  (if (null? rest)
      (string->number (cadr first))
      (apply parse-sum (car rest))))

(define parse-expr parse-sum)
```

(Notice all these functions look very similar; for a more complicated PEG, it would be worth abstracting.)

Then:

```
(apply parse-expr (peg:tree (match-pattern expr "1+1/2*3+(1+1)/2"))) ⇒
(+ 1 (+ (/ 1 (* 2 3)) (/ (+ 1 1) 2)))
```

But wait! The associativity is wrong! Where it says `(/ 1 (* 2 3))`, it should say `(* (/ 1 2) 3)`.

It's tempting to try replacing e.g. `"sum <-- (product ('+' / '-') sum) / product"` with `"sum <-- (sum ('+' / '-') product) / product"`, but this is a Bad Idea. PEGs don't support left recursion. To see why, imagine what the parser will do here. When it tries to parse `sum`, it first has to try and parse `sum`. But to do that, it first has to try and parse `sum`. This will continue until the stack gets blown off.

So how does one parse left-associative binary operators with PEGs? Honestly, this is one of their major shortcomings. There's no general-purpose way of doing this, but here the repetition operators are a good choice:

```
(use-modules (srfi srfi-1))

(define-peg-string-patterns
  "expr <- sum
sum <-- (product ('+' / '-'))* product
product <-- (value ('*' / '/'))* value
value <-- number / '(' expr ')'
number <-- [0-9]+")

;; take a deep breath...
(define (make-left-parser next-func)
  (lambda (sum first . rest) ;; general form, comments below assume
    ;; that we're dealing with a sum expression
    (if (null? rest) ;; form (sum (product ...))
      (apply next-func first)
      (if (string? (cadr first));; form (sum ((product ...) "+") (product ...))
  (list (string->symbol (cadr first))
(apply next-func (car first))
(apply next-func (car rest)))
```

```
          ;; form (sum (((product ...) "+") ((product ...) "+")) (product ...))
   (car
    (reduce ;; walk through the list and build a left-associative tree
     (lambda (l r)
       (list (list (cadr r) (car r) (apply next-func (car l)))
     (string->symbol (cadr l))))
     'ignore
     (append ;; make a list of all the products
             ;; the first one should be pre-parsed
      (list (list (apply next-func (caar first))
 (string->symbol (cadar first))))
         (cdr first)
             ;; the last one has to be added in
         (list (append rest '("done")))))))))

(define (parse-value value first . rest)
  (if (null? rest)
      (string->number (cadr first))
      (apply parse-sum (car rest))))
(define parse-product (make-left-parser parse-value))
(define parse-sum (make-left-parser parse-product))
(define parse-expr parse-sum)
```

Then:

```
(apply parse-expr (peg:tree (match-pattern expr "1+1/2*3+(1+1)/2"))) ⇒
(+ (+ 1 (* (/ 1 2) 3)) (/ (+ 1 1) 2))
```

As you can see, this is much uglier (it could be made prettier by using `context-flatten`, but the way it's written above makes it clear how we deal with the three ways the zero-or-more * expression can parse). Fortunately, most of the time we can get away with only using right-associativity.

Simplified Functions

For a more tantalizing example, consider the following grammar that parses (highly) simplified C functions:

```
(define-peg-string-patterns
  "cfunc <-- cSP ctype cSP cname cSP cargs cLB cSP cbody cRB
ctype <-- cidentifier
cname <-- cidentifier
cargs <-- cLP (! (cSP cRP) carg cSP (cCOMMA / cRP) cSP)* cSP
carg <-- cSP ctype cSP cname
cbody <-- cstatement *
cidentifier <- [a-zA-z][a-zA-Z0-9_]*
cstatement <-- (!';'.)*cSC cSP
cSC < ';'
cCOMMA < ','
cLP < '('
cRP < ')'
```

```
cLB < '{'
cRB < '}'
cSP < [ \t\n]*")
```

Then:

```
(match-pattern cfunc "int square(int a) { return a*a;}") ⇒
(32
 (cfunc (ctype "int")
        (cname "square")
        (cargs (carg (ctype "int") (cname "a")))
        (cbody (cstatement "return a*a"))))
```

And:

```
(match-pattern cfunc "int mod(int a, int b) { int c = a/b;return a-b*c; }") ⇒
(52
 (cfunc (ctype "int")
        (cname "mod")
        (cargs (carg (ctype "int") (cname "a"))
               (carg (ctype "int") (cname "b")))
        (cbody (cstatement "int c = a/b")
               (cstatement "return a- b*c"))))
```

By wrapping all the `carg` nonterminals in a `cargs` nonterminal, we were able to remove any ambiguity in the parsing structure and avoid having to call `context-flatten` on the output of `match-pattern`. We used the same trick with the `cstatement` nonterminals, wrapping them in a `cbody` nonterminal.

The whitespace nonterminal `cSP` used here is a (very) useful instantiation of a common pattern for matching syntactically irrelevant information. Since it's tagged with `<` and ends with `*` it won't clutter up the parse trees (all the empty lists will be discarded during the compression step) and it will never cause parsing to fail.

6.17.4 PEG Internals

A PEG parser takes a string as input and attempts to parse it as a given nonterminal. The key idea of the PEG implementation is that every nonterminal is just a function that takes a string as an argument and attempts to parse that string as its nonterminal. The functions always start from the beginning, but a parse is considered successful if there is material left over at the end.

This makes it easy to model different PEG parsing operations. For instance, consider the PEG grammar `"ab"`, which could also be written `(and "a" "b")`. It matches the string "ab". Here's how that might be implemented in the PEG style:

```
(define (match-and-a-b str)
  (match-a str)
  (match-b str))
```

As you can see, the use of functions provides an easy way to model sequencing. In a similar way, one could model `(or a b)` with something like the following:

```
(define (match-or-a-b str)
  (or (match-a str) (match-b str)))
```

Here the semantics of a PEG `or` expression map naturally onto Scheme's `or` operator. This function will attempt to run `(match-a str)`, and return its result if it succeeds. Otherwise it will run `(match-b str)`.

Of course, the code above wouldn't quite work. We need some way for the parsing functions to communicate. The actual interface used is below.

Parsing Function Interface

A parsing function takes three arguments - a string, the length of that string, and the position in that string it should start parsing at. In effect, the parsing functions pass around substrings in pieces - the first argument is a buffer of characters, and the second two give a range within that buffer that the parsing function should look at.

Parsing functions return either #f, if they failed to match their nonterminal, or a list whose first element must be an integer representing the final position in the string they matched and whose cdr can be any other data the function wishes to return, or '() if it doesn't have any more data.

The one caveat is that if the extra data it returns is a list, any adjacent strings in that list will be appended by `match-pattern`. For instance, if a parsing function returns `(13 ("a" "b" "c"))`, `match-pattern` will take `(13 ("abc"))` as its value.

For example, here is a function to match "ab" using the actual interface.

```
(define (match-a-b str len pos)
   (and (<= (+ pos 2) len)
        (string= str "ab" pos (+ pos 2))
        (list (+ pos 2) '())))  ; we return no extra information
```

The above function can be used to match a string by running `(match-pattern match-a-b "ab")`.

Code Generators and Extensible Syntax

PEG expressions, such as those in a `define-peg-pattern` form, are interpreted internally in two steps.

First, any string PEG is expanded into an s-expression PEG by the code in the `(ice-9 peg string-peg)` module.

Then, then s-expression PEG that results is compiled into a parsing function by the `(ice-9 peg codegen)` module. In particular, the function `compile-peg-pattern` is called on the s-expression. It then decides what to do based on the form it is passed.

The PEG syntax can be expanded by providing `compile-peg-pattern` more options for what to do with its forms. The extended syntax will be associated with a symbol, for instance `my-parsing-form`, and will be called on all PEG expressions of the form

```
(my-parsing-form ...)
```

The parsing function should take two arguments. The first will be a syntax object containing a list with all of the arguments to the form (but not the form's name), and the second will be the `capture-type` argument that is passed to `define-peg-pattern`.

New functions can be registered by calling `(add-peg-compiler! symbol function)`, where `symbol` is the symbol that will indicate a form of this type and `function` is the code generating function described above. The function `add-peg-compiler!` is exported from the `(ice-9 peg codegen)` module.

6.18 Reading and Evaluating Scheme Code

This chapter describes Guile functions that are concerned with reading, loading, evaluating, and compiling Scheme code at run time.

6.18.1 Scheme Syntax: Standard and Guile Extensions

6.18.1.1 Expression Syntax

An expression to be evaluated takes one of the following forms.

symbol A symbol is evaluated by dereferencing. A binding of that symbol is sought and the value there used. For example,

```
(define x 123)
x ⇒ 123
```

(*proc args...*)

A parenthesised expression is a function call. *proc* and each argument are evaluated, then the function (which *proc* evaluated to) is called with those arguments.

The order in which *proc* and the arguments are evaluated is unspecified, so be careful when using expressions with side effects.

```
(max 1 2 3) ⇒ 3

(define (get-some-proc)  min)
((get-some-proc) 1 2 3) ⇒ 1
```

The same sort of parenthesised form is used for a macro invocation, but in that case the arguments are not evaluated. See the descriptions of macros for more on this (см. Раздел 6.10 [Macros], страница 276, and см. Раздел 6.10.2 [Syntax Rules], страница 278).

constant Number, string, character and boolean constants evaluate "to themselves", so can appear as literals.

```
123     ⇒ 123
99.9    ⇒ 99.9
"hello" ⇒ "hello"
#\z     ⇒ #\z
#t      ⇒ #t
```

Note that an application must not attempt to modify literal strings, since they may be in read-only memory.

(quote *data*)

'*data* Quoting is used to obtain a literal symbol (instead of a variable reference), a literal list (instead of a function call), or a literal vector. ' is simply a shorthand for a `quote` form. For example,

```
'x                  ⇒ x
'(1 2 3)            ⇒ (1 2 3)
'#(1 (2 3) 4)       ⇒ #(1 (2 3) 4)
(quote x)           ⇒ x
```

```
(quote (1 2 3))       ⇒ (1 2 3)
(quote #(1 (2 3) 4)) ⇒ #(1 (2 3) 4)
```

Note that an application must not attempt to modify literal lists or vectors obtained from a `quote` form, since they may be in read-only memory.

`(quasiquote` *`data`*`)`

`` ``*`data`* Backquote quasi-quotation is like `quote`, but selected sub-expressions are evaluated. This is a convenient way to construct a list or vector structure most of which is constant, but at certain points should have expressions substituted.

The same effect can always be had with suitable `list`, `cons` or `vector` calls, but quasi-quoting is often easier.

`(unquote` *`expr`*`)`

`,`*`expr`* Within the quasiquote *data*, `unquote` or `,` indicates an expression to be evaluated and inserted. The comma syntax `,` is simply a shorthand for an `unquote` form. For example,

```
`(1 2 (* 9 9) 3 4)       ⇒ (1 2 (* 9 9) 3 4)
`(1 2 ,(* 9 9) 3 4)      ⇒ (1 2 81 3 4)
`(1 (unquote (+ 1 1)) 3) ⇒ (1 2 3)
`#(1 ,(/ 12 2))          ⇒ #(1 6)
```

`(unquote-splicing` *`expr`*`)`

`,@`*`expr`* Within the quasiquote *data*, `unquote-splicing` or `,@` indicates an expression to be evaluated and the elements of the returned list inserted. *expr* must evaluate to a list. The "comma-at" syntax `,@` is simply a shorthand for an `unquote-splicing` form.

```
(define x '(2 3))
`(1 ,x 4)                          ⇒ (1 (2 3) 4)
`(1 ,@x 4)                        ⇒ (1 2 3 4)
`(1 (unquote-splicing (map 1+ x))) ⇒ (1 3 4)
`#(9 ,@x 9)                       ⇒ #(9 2 3 9)
```

Notice `,@` differs from plain `,` in the way one level of nesting is stripped. For `,@` the elements of a returned list are inserted, whereas with `,` it would be the list itself inserted.

6.18.1.2 Comments

Comments in Scheme source files are written by starting them with a semicolon character (`;`). The comment then reaches up to the end of the line. Comments can begin at any column, and the may be inserted on the same line as Scheme code.

```
; Comment
;; Comment too
(define x 1)        ; Comment after expression
(let ((y 1))
  ;; Display something.
  (display y)
;;; Comment at left margin.
```

```
(display (+ y 1)))
```

It is common to use a single semicolon for comments following expressions on a line, to use two semicolons for comments which are indented like code, and three semicolons for comments which start at column 0, even if they are inside an indented code block. This convention is used when indenting code in Emacs' Scheme mode.

6.18.1.3 Block Comments

In addition to the standard line comments defined by R5RS, Guile has another comment type for multiline comments, called *block comments*. This type of comment begins with the character sequence `#!` and ends with the characters `!#`.

These comments are compatible with the block comments in the Scheme Shell `scsh` (см. Раздел 7.17 [The Scheme shell (scsh)], страница 756). The characters `#!` were chosen because they are the magic characters used in shell scripts for indicating that the name of the program for executing the script follows on the same line.

Thus a Guile script often starts like this.

```
#! /usr/local/bin/guile -s
!#
```

More details on Guile scripting can be found in the scripting section (см. Раздел 4.3 [Guile Scripting], страница 44).

Similarly, Guile (starting from version 2.0) supports nested block comments as specified by R6RS and SRFI-30 (`http://srfi.schemers.org/srfi-30/srfi-30.html`):

```
(+ 1 #| this is a #| nested |# block comment |# 2)
⇒ 3
```

For backward compatibility, this syntax can be overridden with `read-hash-extend` (см. Раздел 6.18.1.6 [Reader Extensions], страница 407).

There is one special case where the contents of a comment can actually affect the interpretation of code. When a character encoding declaration, such as `coding: utf-8` appears in one of the first few lines of a source file, it indicates to Guile's default reader that this source code file is not ASCII. For details see Раздел 6.18.8 [Character Encoding of Source Files], страница 416.

6.18.1.4 Case Sensitivity

Scheme as defined in R5RS is not case sensitive when reading symbols. Guile, on the contrary is case sensitive by default, so the identifiers

```
guile-whuzzy
Guile-Whuzzy
```

are the same in R5RS Scheme, but are different in Guile.

It is possible to turn off case sensitivity in Guile by setting the reader option `case-insensitive`. For more information on reader options, См. Раздел 6.18.2 [Scheme Read], страница 407.

```
(read-enable 'case-insensitive)
```

It is also possible to disable (or enable) case sensitivity within a single file by placing the reader directives `#!fold-case` (or `#!no-fold-case`) within the file itself.

6.18.1.5 Keyword Syntax

6.18.1.6 Reader Extensions

`read-hash-extend` *chr proc* [Scheme Procedure]
`scm_read_hash_extend` (*chr, proc*) [C Function]
Install the procedure *proc* for reading expressions starting with the character sequence `#` and *chr*. *proc* will be called with two arguments: the character *chr* and the port to read further data from. The object returned will be the return value of `read`. Passing `#f` for *proc* will remove a previous setting.

6.18.2 Reading Scheme Code

`read` [*port*] [Scheme Procedure]
`scm_read` (*port*) [C Function]
Read an s-expression from the input port *port*, or from the current input port if *port* is not specified. Any whitespace before the next token is discarded.

The behaviour of Guile's Scheme reader can be modified by manipulating its read options.

`read-options` [*setting*] [Scheme Procedure]
Display the current settings of the global read options. If *setting* is omitted, only a short form of the current read options is printed. Otherwise if *setting* is the symbol `help`, a complete options description is displayed.

The set of available options, and their default values, may be had by invoking `read-options` at the prompt.

```
scheme@(guile-user)> (read-options)
(square-brackets keywords #f positions)
scheme@(guile-user)> (read-options 'help)
copy              no    Copy source code expressions.
positions         yes   Record positions of source code expressions.
case-insensitive  no    Convert symbols to lower case.
keywords          #f    Style of keyword recognition: #f, 'prefix or 'postfix.
r6rs-hex-escapes  no    Use R6RS variable-length character and string hex escapes.
square-brackets   yes   Treat `[' and `]' as parentheses, for R6RS compatibility.
hungry-eol-escapes no   In strings, consume leading whitespace after an
                        escaped end-of-line.
curly-infix       no    Support SRFI-105 curly infix expressions.
r7rs-symbols      no    Support R7RS |...| symbol notation.
```

Note that Guile also includes a preliminary mechanism for setting read options on a per-port basis. For instance, the `case-insensitive` read option is set (or unset) on the port when the reader encounters the `#!fold-case` or `#!no-fold-case` reader directives. Similarly, the `#!curly-infix` reader directive sets the `curly-infix` read option on the port, and `#!curly-infix-and-bracket-lists` sets `curly-infix` and unsets `square-brackets` on the port (см. Раздел 7.5.43 [SRFI-105], страница 676). There is currently no other way to access or set the per-port read options.

The boolean options may be toggled with `read-enable` and `read-disable`. The non-boolean `keywords` option must be set using `read-set!`.

`read-enable` *option-name* [Scheme Procedure]
`read-disable` *option-name* [Scheme Procedure]
`read-set!` *option-name value* [Scheme Syntax]

Modify the read options. `read-enable` should be used with boolean options and switches them on, `read-disable` switches them off.

`read-set!` can be used to set an option to a specific value. Due to historical oddities, it is a macro that expects an unquoted option name.

For example, to make `read` fold all symbols to their lower case (perhaps for compatibility with older Scheme code), you can enter:

```
(read-enable 'case-insensitive)
```

For more information on the effect of the `r6rs-hex-escapes` and `hungry-eol-escapes` options, see (см. Раздел 6.6.5.1 [String Syntax], страница 151).

For more information on the `r7rs-symbols` option, see (см. Раздел 6.6.6.6 [Symbol Read Syntax], страница 182).

6.18.3 Writing Scheme Values

Any scheme value may be written to a port. Not all values may be read back in (см. Раздел 6.18.2 [Scheme Read], страница 407), however.

`write` *obj* [*port*] [Scheme Procedure]

Send a representation of *obj* to *port* or to the current output port if not given.

The output is designed to be machine readable, and can be read back with `read` (см. Раздел 6.18.2 [Scheme Read], страница 407). Strings are printed in double quotes, with escapes if necessary, and characters are printed in '`#\`' notation.

`display` *obj* [*port*] [Scheme Procedure]

Send a representation of *obj* to *port* or to the current output port if not given.

The output is designed for human readability, it differs from `write` in that strings are printed without double quotes and escapes, and characters are printed as per `write-char`, not in '`#\`' form.

As was the case with the Scheme reader, there are a few options that affect the behavior of the Scheme printer.

`print-options` [*setting*] [Scheme Procedure]

Display the current settings of the read options. If *setting* is omitted, only a short form of the current read options is printed. Otherwise if *setting* is the symbol `help`, a complete options description is displayed.

The set of available options, and their default values, may be had by invoking `print-options` at the prompt.

```
scheme@(guile-user)> (print-options)
(quote-keywordish-symbols reader highlight-suffix "}" highlight-prefix "{")
scheme@(guile-user)> (print-options 'help)
highlight-prefix          {       The string to print before highlighted values.
highlight-suffix          }       The string to print after highlighted values.
quote-keywordish-symbols  reader  How to print symbols that have a colon
```

```
                                as their first or last character. The
                                value '#f' does not quote the colons;
                                '#t' quotes them; 'reader' quotes them
                                when the reader option 'keywords' is
                                not '#f'.
     escape-newlines      yes   Render newlines as \n when printing
                                using `write'.
     r7rs-symbols         no    Escape symbols using R7RS |...| symbol
                                notation.
```

These options may be modified with the print-set! syntax.

`print-set!` *option-name value* [Scheme Syntax]

Modify the print options. Due to historical oddities, `print-set!` is a macro that expects an unquoted option name.

6.18.4 Procedures for On the Fly Evaluation

Scheme has the lovely property that its expressions may be represented as data. The `eval` procedure takes a Scheme datum and evaluates it as code.

`eval` *exp module_or_state* [Scheme Procedure]
`scm_eval` (*exp, module_or_state*) [C Function]

Evaluate *exp*, a list representing a Scheme expression, in the top-level environment specified by *module_or_state*. While *exp* is evaluated (using `primitive-eval`), *module_or_state* is made the current module. The current module is reset to its previous value when `eval` returns. XXX - dynamic states. Example: (eval '(+ 1 2) (interaction-environment))

`interaction-environment` [Scheme Procedure]
`scm_interaction_environment` () [C Function]

Return a specifier for the environment that contains implementation–defined bindings, typically a superset of those listed in the report. The intent is that this procedure will return the environment in which the implementation would evaluate expressions dynamically typed by the user.

См. Раздел 6.20.11 [Environments], страница 446, for other environments.

One does not always receive code as Scheme data, of course, and this is especially the case for Guile's other language implementations (см. Раздел 6.24 [Other Languages], страница 482). For the case in which all you have is a string, we have `eval-string`. There is a legacy version of this procedure in the default environment, but you really want the one from `(ice-9 eval-string)`, so load it up:

```
(use-modules (ice-9 eval-string))
```

`eval-string` *string* [*#:module=#f*] [*#:file=#f*] [*#:line=#f*] [Scheme Procedure]
[*#:column=#f*] [*#:lang=*(*current-language*)] [*#:compile?=#f*]

Parse *string* according to the current language, normally Scheme. Evaluate or compile the expressions it contains, in order, returning the last expression.

If the *module* keyword argument is set, save a module excursion (см. Раздел 6.20.8 [Module System Reflection], страница 441) and set the current module to *module* before evaluation.

The *file*, *line*, and *column* keyword arguments can be used to indicate that the source string begins at a particular source location.

Finally, *lang* is a language, defaulting to the current language, and the expression is compiled if *compile?* is true or there is no evaluator for the given language.

`scm_eval_string` (*string*) [C Function]
`scm_eval_string_in_module` (*string, module*) [C Function]
These C bindings call `eval-string` from `(ice-9 eval-string)`, evaluating within *module* or the current module.

`SCM scm_c_eval_string` (*const char *string*) [C Function]
`scm_eval_string`, but taking a C string in locale encoding instead of an `SCM`.

`apply` *proc arg ... arglst* [Scheme Procedure]
`scm_apply_0` (*proc, arglst*) [C Function]
`scm_apply_1` (*proc, arg1, arglst*) [C Function]
`scm_apply_2` (*proc, arg1, arg2, arglst*) [C Function]
`scm_apply_3` (*proc, arg1, arg2, arg3, arglst*) [C Function]
`scm_apply` (*proc, arg, rest*) [C Function]
Call *proc* with arguments *arg ...* and the elements of the *arglst* list.

`scm_apply` takes parameters corresponding to a Scheme level `(lambda (proc arg1 . rest) ...)`. So *arg1* and all but the last element of the *rest* list make up *arg ...*, and the last element of *rest* is the *arglst* list. Or if *rest* is the empty list `SCM_EOL` then there's no *arg ...*, and (*arg1*) is the *arglst*.

arglst is not modified, but the *rest* list passed to `scm_apply` is modified.

`scm_call_0` (*proc*) [C Function]
`scm_call_1` (*proc, arg1*) [C Function]
`scm_call_2` (*proc, arg1, arg2*) [C Function]
`scm_call_3` (*proc, arg1, arg2, arg3*) [C Function]
`scm_call_4` (*proc, arg1, arg2, arg3, arg4*) [C Function]
`scm_call_5` (*proc, arg1, arg2, arg3, arg4, arg5*) [C Function]
`scm_call_6` (*proc, arg1, arg2, arg3, arg4, arg5, arg6*) [C Function]
`scm_call_7` (*proc, arg1, arg2, arg3, arg4, arg5, arg6, arg7*) [C Function]
`scm_call_8` (*proc, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8*) [C Function]
`scm_call_9` (*proc, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9*) [C Function]
Call *proc* with the given arguments.

`scm_call` (*proc, ...*) [C Function]
Call *proc* with any number of arguments. The argument list must be terminated by `SCM_UNDEFINED`. For example:

```
scm_call (scm_c_public_ref ("guile", "+"),
          scm_from_int (1),
          scm_from_int (2),
          SCM_UNDEFINED);
```

`scm_call_n` (*proc, argv, nargs*) [C Function]
Call *proc* with the array of arguments *argv*, as a `SCM*`. The length of the arguments should be passed in *nargs*, as a `size_t`.

`primitive-eval` *exp* [Scheme Procedure]
`scm_primitive_eval` (*exp*) [C Function]
Evaluate *exp* in the top-level environment specified by the current module.

6.18.5 Compiling Scheme Code

The `eval` procedure directly interprets the S-expression representation of Scheme. An alternate strategy for evaluation is to determine ahead of time what computations will be necessary to evaluate the expression, and then use that recipe to produce the desired results. This is known as *compilation*.

While it is possible to compile simple Scheme expressions such as `(+ 2 2)` or even `"Hello world!"`, compilation is most interesting in the context of procedures. Compiling a lambda expression produces a compiled procedure, which is just like a normal procedure except typically much faster, because it can bypass the generic interpreter.

Functions from system modules in a Guile installation are normally compiled already, so they load and run quickly.

Note that well-written Scheme programs will not typically call the procedures in this section, for the same reason that it is often bad taste to use `eval`. By default, Guile automatically compiles any files it encounters that have not been compiled yet (см. Раздел 4.2 [Invoking Guile], страница 37). The compiler can also be invoked explicitly from the shell as `guild compile foo.scm`.

(Why are calls to `eval` and `compile` usually in bad taste? Because they are limited, in that they can only really make sense for top-level expressions. Also, most needs for "compile-time" computation are fulfilled by macros and closures. Of course one good counterexample is the REPL itself, or any code that reads expressions from a port.)

Automatic compilation generally works transparently, without any need for user intervention. However Guile does not yet do proper dependency tracking, so that if file `a.scm` uses macros from `b.scm`, and *b.scm* changes, `a.scm` would not be automatically recompiled. To forcibly invalidate the auto-compilation cache, pass the `--fresh-auto-compile` option to Guile, or set the `GUILE_AUTO_COMPILE` environment variable to `fresh` (instead of to `0` or `1`).

For more information on the compiler itself, see Раздел 9.4 [Compiling to the Virtual Machine], страница 870. For information on the virtual machine, see Раздел 9.3 [A Virtual Machine for Guile], страница 844.

The command-line interface to Guile's compiler is the `guild compile` command:

`guild compile` [`option`...] *file*... [Command]
Compile *file*, a source file, and store bytecode in the compilation cache or in the file specified by the `-o` option. The following options are available:

`-L` *dir*
`--load-path=`*dir*
Add *dir* to the front of the module load path.

`-o` *ofile*
`--output=`*ofile*
: Write output bytecode to *ofile*. By convention, bytecode file names end in `.go`. When `-o` is omitted, the output file name is as for `compile-file` (see below).

`-W` *warning*
`--warn=`*warning*
: Emit warnings of type *warning*; use `--warn=help` for a list of available warnings and their description. Currently recognized warnings include `unused-variable`, `unused-toplevel`, `unbound-variable`, `arity-mismatch`, `format`, `duplicate-case-datum`, and `bad-case-datum`.

`-f` *lang*
`--from=`*lang*
: Use *lang* as the source language of *file*. If this option is omitted, `scheme` is assumed.

`-t` *lang*
`--to=`*lang*
: Use *lang* as the target language of *file*. If this option is omitted, `rtl` is assumed.

`-T` *target*
`--target=`*target*
: Produce code for *target* instead of *%host-type* (см. Раздел 6.23.1 [Build Config], страница 477). Target must be a valid GNU triplet, such as `armv5tel-unknown-linux-gnueabi` (см. Раздел "Specifying Target Triplets" в *GNU Autoconf Manual*).

Each *file* is assumed to be UTF-8-encoded, unless it contains a coding declaration as recognized by `file-encoding` (см. Раздел 6.18.8 [Character Encoding of Source Files], страница 416).

The compiler can also be invoked directly by Scheme code using the procedures below:

`compile` *exp* **[***#:env=#f***]** **[***#:from=(current-language)***]** **[***#:to=value***]** **[***#:opts=()***]** [Scheme Procedure]

Compile the expression *exp* in the environment *env*. If *exp* is a procedure, the result will be a compiled procedure; otherwise `compile` is mostly equivalent to `eval`.

For a discussion of languages and compiler options, См. Раздел 9.4 [Compiling to the Virtual Machine], страница 870.

`compile-file` *file* **[***#:output-file=#f***]** **[***#:from=(current-language)***]** **[***#:to='rtl***]** **[***#:env=(default-environment from)***]** **[***#:opts='()***]** **[***#:canonicalization='relative***]** [Scheme Procedure]

Compile the file named *file*.

Output will be written to a *output-file*. If you do not supply an output file name, output is written to a file in the cache directory, as computed by `(compiled-file-name` *file*`)`.

from and *to* specify the source and target languages. См. Раздел 9.4 [Compiling to the Virtual Machine], страница 870, for more information on these options, and on *env* and *opts*.

As with `guild compile`, *file* is assumed to be UTF-8-encoded unless it contains a coding declaration.

`compiled-file-name` *file* [Scheme Procedure]

Compute a cached location for a compiled version of a Scheme file named *file*.

This file will usually be below the `$HOME/.cache/guile/ccache` directory, depending on the value of the `XDG_CACHE_HOME` environment variable. The intention is that `compiled-file-name` provides a fallback location for caching auto-compiled files. If you want to place a compile file in the `%load-compiled-path`, you should pass the *output-file* option to `compile-file`, explicitly.

`%auto-compilation-options` [Scheme Variable]

This variable contains the options passed to the `compile-file` procedure when auto-compiling source files. By default, it enables useful compilation warnings. It can be customized from `~/.guile`.

6.18.6 Loading Scheme Code from File

`load` *filename* [*reader*] [Scheme Procedure]

Load *filename* and evaluate its contents in the top-level environment.

reader if provided should be either `#f`, or a procedure with the signature `(lambda (port) ...)` which reads the next expression from *port*. If *reader* is `#f` or absent, Guile's built-in `read` procedure is used (см. Раздел 6.18.2 [Scheme Read], страница 407).

The *reader* argument takes effect by setting the value of the `current-reader` fluid (see below) before loading the file, and restoring its previous value when loading is complete. The Scheme code inside *filename* can itself change the current reader procedure on the fly by setting `current-reader` fluid.

If the variable `%load-hook` is defined, it should be bound to a procedure that will be called before any code is loaded. See documentation for `%load-hook` later in this section.

`load-compiled` *filename* [Scheme Procedure]

Load the compiled file named *filename*.

Compiling a source file (см. Раздел 6.18 [Read/Load/Eval/Compile], страница 404) and then calling `load-compiled` on the resulting file is equivalent to calling `load` on the source file.

`primitive-load` *filename* [Scheme Procedure]
`scm_primitive_load` (*filename*) [C Function]

Load the file named *filename* and evaluate its contents in the top-level environment. *filename* must either be a full pathname or be a pathname relative to the current directory. If the variable `%load-hook` is defined, it should be bound to a procedure that will be called before any code is loaded. See the documentation for `%load-hook` later in this section.

`SCM scm_c_primitive_load` (*const char *filename*) [C Function]
: `scm_primitive_load`, but taking a C string instead of an `SCM`.

`current-reader` [Переменная]
: `current-reader` holds the read procedure that is currently being used by the above loading procedures to read expressions (from the file that they are loading). `current-reader` is a fluid, so it has an independent value in each dynamic root and should be read and set using `fluid-ref` and `fluid-set!` (см. Раздел 6.13.11 [Fluids and Dynamic States], страница 343).

  Changing `current-reader` is typically useful to introduce local syntactic changes, such that code following the `fluid-set!` call is read using the newly installed reader. The `current-reader` change should take place at evaluation time when the code is evaluated, or at compilation time when the code is compiled:

```
(eval-when (compile eval)
  (fluid-set! current-reader my-own-reader))
```

  The `eval-when` form above ensures that the `current-reader` change occurs at the right time.

`%load-hook` [Переменная]
: A procedure to be called `(%load-hook` *`filename`*`)` whenever a file is loaded, or `#f` for no such call. `%load-hook` is used by all of the loading functions (`load` and `primitive-load`, and `load-from-path` and `primitive-load-path` documented in the next section).

  For example an application can set this to show what's loaded,

```
(set! %load-hook (lambda (filename)
                   (format #t "Loading ~a ...\n" filename)))
(load-from-path "foo.scm")
⊣ Loading /usr/local/share/guile/site/foo.scm ...
```

`current-load-port` [Scheme Procedure]
`scm_current_load_port` () [C Function]
: Return the current-load-port. The load port is used internally by `primitive-load`.

6.18.7 Load Paths

The procedure in the previous section look for Scheme code in the file system at specific location. Guile also has some procedures to search the load path for code.

`%load-path` [Переменная]
: List of directories which should be searched for Scheme modules and libraries. When Guile starts up, `%load-path` is initialized to the default load path `(list (%library-dir) (%site-dir) (%global-site-dir) (%package-data-dir))`. The `GUILE_LOAD_PATH` environment variable can be used to prepend or append additional directories (см. ⟨undefined⟩ [Environment Variables], страница ⟨undefined⟩).

  См. Раздел 6.23.1 [Build Config], страница 477, for more on `%site-dir` and related procedures.

`load-from-path` *filename* [Scheme Procedure]
Similar to `load`, but searches for *filename* in the load paths. Preferentially loads a compiled version of the file, if it is available and up-to-date.

A user can extend the load path by calling `add-to-load-path`.

`add-to-load-path` *dir* [Scheme Syntax]
Add *dir* to the load path.

For example, a script might include this form to add the directory that it is in to the load path:

```
(add-to-load-path (dirname (current-filename)))
```

It's better to use `add-to-load-path` than to modify `%load-path` directly, because `add-to-load-path` takes care of modifying the path both at compile-time and at run-time.

`primitive-load-path` *filename* [*exception-on-not-found*] [Scheme Procedure]
`scm_primitive_load_path` (*filename*) [C Function]
Search `%load-path` for the file named *filename* and load it into the top-level environment. If *filename* is a relative pathname and is not found in the list of search paths, an error is signalled. Preferentially loads a compiled version of the file, if it is available and up-to-date.

If *filename* is a relative pathname and is not found in the list of search paths, one of three things may happen, depending on the optional second argument, *exception-on-not-found*. If it is `#f`, `#f` will be returned. If it is a procedure, it will be called with no arguments. (This allows a distinction to be made between exceptions raised by loading a file, and exceptions related to the loader itself.) Otherwise an error is signalled.

For compatibility with Guile 1.8 and earlier, the C function takes only one argument, which can be either a string (the file name) or an argument list.

`%search-load-path` *filename* [Scheme Procedure]
`scm_sys_search_load_path` (*filename*) [C Function]
Search `%load-path` for the file named *filename*, which must be readable by the current user. If *filename* is found in the list of paths to search or is an absolute pathname, return its full pathname. Otherwise, return `#f`. Filenames may have any of the optional extensions in the `%load-extensions` list; `%search-load-path` will try each extension automatically.

`%load-extensions` [Переменная]
A list of default file extensions for files containing Scheme code. `%search-load-path` tries each of these extensions when looking for a file to load. By default, `%load-extensions` is bound to the list `("" ".scm")`.

As mentioned above, when Guile searches the `%load-path` for a source file, it will also search the `%load-compiled-path` for a corresponding compiled file. If the compiled file is as new or newer than the source file, it will be loaded instead of the source file, using `load-compiled`.

`%load-compiled-path` [Переменная]

Like `%load-path`, but for compiled files. By default, this path has two entries: one for compiled files from Guile itself, and one for site packages. The `GUILE_LOAD_COMPILED_PATH` environment variable can be used to prepend or append additional directories (см. ⟨undefined⟩ [Environment Variables], страница ⟨undefined⟩).

When `primitive-load-path` searches the `%load-compiled-path` for a corresponding compiled file for a relative path it does so by appending `.go` to the relative path. For example, searching for `ice-9/popen` could find `/usr/lib/guile/2.2/ccache/ice-9/popen.go`, and use it instead of `/usr/share/guile/2.2/ice-9/popen.scm`.

If `primitive-load-path` does not find a corresponding `.go` file in the `%load-compiled-path`, or the `.go` file is out of date, it will search for a corresponding auto-compiled file in the fallback path, possibly creating one if one does not exist.

См. Раздел 4.7 [Installing Site Packages], страница 61, for more on how to correctly install site packages. См. Раздел 6.20.4 [Modules and the File System], страница 437, for more on the relationship between load paths and modules. См. Раздел 6.18.5 [Compilation], страница 411, for more on the fallback path and auto-compilation.

Finally, there are a couple of helper procedures for general path manipulation.

`parse-path` *path* [*tail*] [Scheme Procedure]
`scm_parse_path` (*path, tail*) [C Function]

Parse *path*, which is expected to be a colon-separated string, into a list and return the resulting list with *tail* appended. If *path* is `#f`, *tail* is returned.

`parse-path-with-ellipsis` *path base* [Scheme Procedure]
`scm_parse_path_with_ellipsis` (*path, base*) [C Function]

Parse *path*, which is expected to be a colon-separated string, into a list and return the resulting list with *base* (a list) spliced in place of the `...` path component, if present, or else *base* is added to the end. If *path* is `#f`, *base* is returned.

`search-path` *path filename* [*extensions* [*require-exts?*]] [Scheme Procedure]
`scm_search_path` (*path, filename, rest*) [C Function]

Search *path* for a directory containing a file named *filename*. The file must be readable, and not a directory. If we find one, return its full filename; otherwise, return `#f`. If *filename* is absolute, return it unchanged. If given, *extensions* is a list of strings; for each directory in *path*, we search for *filename* concatenated with each *extension*. If *require-exts?* is true, require that the returned file name have one of the given extensions; if *require-exts?* is not given, it defaults to `#f`.

For compatibility with Guile 1.8 and earlier, the C function takes only three arguments.

6.18.8 Character Encoding of Source Files

Scheme source code files are usually encoded in ASCII or UTF-8, but the built-in reader can interpret other character encodings as well. When Guile loads Scheme source code, it uses the `file-encoding` procedure (described below) to try to guess the encoding of the file. In the absence of any hints, UTF-8 is assumed. One way to provide a hint about the

encoding of a source file is to place a coding declaration in the top 500 characters of the file.

A coding declaration has the form `coding: XXXXXX`, where `XXXXXX` is the name of a character encoding in which the source code file has been encoded. The coding declaration must appear in a scheme comment. It can either be a semicolon-initiated comment, or the first block `#!` comment in the file.

The name of the character encoding in the coding declaration is typically lower case and containing only letters, numbers, and hyphens, as recognized by `set-port-encoding!` (см. Раздел 6.14.1 [Ports], страница 352). Common examples of character encoding names are `utf-8` and `iso-8859-1`, as defined by IANA (`http://www.iana.org/assignments/character-sets`). Thus, the coding declaration is mostly compatible with Emacs.

However, there are some differences in encoding names recognized by Emacs and encoding names defined by IANA, the latter being essentially a subset of the former. For instance, `latin-1` is a valid encoding name for Emacs, but it's not according to the IANA standard, which Guile follows; instead, you should use `iso-8859-1`, which is both understood by Emacs and dubbed by IANA (IANA writes it uppercase but Emacs wants it lowercase and Guile is case insensitive.)

For source code, only a subset of all possible character encodings can be interpreted by the built-in source code reader. Only those character encodings in which ASCII text appears unmodified can be used. This includes `UTF-8` and `ISO-8859-1` through `ISO-8859-15`. The multi-byte character encodings `UTF-16` and `UTF-32` may not be used because they are not compatible with ASCII.

There might be a scenario in which one would want to read non-ASCII code from a port, such as with the function `read`, instead of with `load`. If the port's character encoding is the same as the encoding of the code to be read by the port, not other special handling is necessary. The port will automatically do the character encoding conversion. The functions `setlocale` or by `set-port-encoding!` are used to set port encodings (см. Раздел 6.14.1 [Ports], страница 352).

If a port is used to read code of unknown character encoding, it can accomplish this in three steps. First, the character encoding of the port should be set to ISO-8859-1 using `set-port-encoding!`. Then, the procedure `file-encoding`, described below, is used to scan for a coding declaration when reading from the port. As a side effect, it rewinds the port after its scan is complete. After that, the port's character encoding should be set to the encoding returned by `file-encoding`, if any, again by using `set-port-encoding!`. Then the code can be read as normal.

Alternatively, one can use the `#:guess-encoding` keyword argument of `open-file` and related procedures. См. Раздел 6.14.10.1 [File Ports], страница 365.

`file-encoding` *port* [Scheme Procedure]
`scm_file_encoding` (*port*) [C Function]

Attempt to scan the first few hundred bytes from the *port* for hints about its character encoding. Return a string containing the encoding name or `#f` if the encoding cannot be determined. The port is rewound.

Currently, the only supported method is to look for an Emacs-like character coding declaration (см. Раздел "Recognize Coding" в *The GNU Emacs Reference Manual*).

The coding declaration is of the form `coding: XXXXX` and must appear in a Scheme comment. Additional heuristics may be added in the future.

6.18.9 Delayed Evaluation

Promises are a convenient way to defer a calculation until its result is actually needed, and to run such a calculation only once. Also см. Раздел 7.5.31 [SRFI-45], страница 668.

`delay` *expr* [syntax]

Return a promise object which holds the given *expr* expression, ready to be evaluated by a later `force`.

`promise?` *obj* [Scheme Procedure]
`scm_promise_p` (*obj*) [C Function]

Return true if *obj* is a promise.

`force` *p* [Scheme Procedure]
`scm_force` (*p*) [C Function]

Return the value obtained from evaluating the *expr* in the given promise *p*. If *p* has previously been forced then its *expr* is not evaluated again, instead the value obtained at that time is simply returned.

During a `force`, an *expr* can call `force` again on its own promise, resulting in a recursive evaluation of that *expr*. The first evaluation to return gives the value for the promise. Higher evaluations run to completion in the normal way, but their results are ignored, `force` always returns the first value.

6.18.10 Local Evaluation

Guile includes a facility to capture a lexical environment, and later evaluate a new expression within that environment. This code is implemented in a module.

```
(use-modules (ice-9 local-eval))
```

`the-environment` [syntax]

Captures and returns a lexical environment for use with `local-eval` or `local-compile`.

`local-eval` *exp env* [Scheme Procedure]
`scm_local_eval` (*exp, env*) [C Function]
`local-compile` *exp env* [*opts*=()] [Scheme Procedure]

Evaluate or compile the expression *exp* in the lexical environment *env*.

Here is a simple example, illustrating that it is the variable that gets captured, not just its value at one point in time.

```
(define e (let ((x 100)) (the-environment)))
(define fetch-x (local-eval '(lambda () x) e))
(fetch-x)
⇒ 100
(local-eval '(set! x 42) e)
(fetch-x)
⇒ 42
```

While *exp* is evaluated within the lexical environment of `(the-environment)`, it has the dynamic environment of the call to `local-eval`.

`local-eval` and `local-compile` can only evaluate expressions, not definitions.

```
(local-eval '(define foo 42)
            (let ((x 100)) (the-environment)))
⇒ syntax error: definition in expression context
```

Note that the current implementation of `(the-environment)` only captures "normal" lexical bindings, and pattern variables bound by `syntax-case`. It does not currently capture local syntax transformers bound by `let-syntax`, `letrec-syntax` or non-top-level `define-syntax` forms. Any attempt to reference such captured syntactic keywords via `local-eval` or `local-compile` produces an error.

6.18.11 Local Inclusion

This section has discussed various means of linking Scheme code together: fundamentally, loading up files at run-time using `load` and `load-compiled`. Guile provides another option to compose parts of programs together at expansion-time instead of at run-time.

`include` *file-name* [Scheme Syntax]

Open *file-name*, at expansion-time, and read the Scheme forms that it contains, splicing them into the location of the `include`, within a `begin`.

If *file-name* is a relative path, it is searched for relative to the path that contains the file that the `include` form appears in.

If you are a C programmer, if `load` in Scheme is like `dlopen` in C, consider `include` to be like the C preprocessor's `#include`. When you use `include`, it is as if the contents of the included file were typed in instead of the `include` form.

Because the code is included at compile-time, it is available to the macroexpander. Syntax definitions in the included file are available to later code in the form in which the `include` appears, without the need for `eval-when`. (См. Раздел 6.10.8 [Eval When], страница 296.)

For the same reason, compiling a form that uses `include` results in one compilation unit, composed of multiple files. Loading the compiled file is one `stat` operation for the compilation unit, instead of `2*`*`n`* in the case of `load` (once for each loaded source file, and once each corresponding compiled file, in the best case).

Unlike `load`, `include` also works within nested lexical contexts. It so happens that the optimizer works best within a lexical context, because all of the uses of bindings in a lexical context are visible, so composing files by including them within a `(let () ...)` can sometimes lead to important speed improvements.

On the other hand, `include` does have all the disadvantages of early binding: once the code with the `include` is compiled, no change to the included file is reflected in the future behavior of the including form.

Also, the particular form of `include`, which requires an absolute path, or a path relative to the current directory at compile-time, is not very amenable to compiling the source in one place, but then installing the source to another place. For this reason, Guile provides another form, `include-from-path`, which looks for the source file to include within a load path.

`include-from-path` *file-name* [Scheme Syntax]
Like `include`, but instead of expecting `file-name` to be an absolute file name, it is expected to be a relative path to search in the `%load-path`.

`include-from-path` is more useful when you want to install all of the source files for a package (as you should!). It makes it possible to evaluate an installed file from source, instead of relying on the `.go` file being up to date.

6.18.12 Sandboxed Evaluation

Sometimes you would like to evaluate code that comes from an untrusted party. The safest way to do this is to buy a new computer, evaluate the code on that computer, then throw the machine away. However if you are unwilling to take this simple approach, Guile does include a limited "sandbox" facility that can allow untrusted code to be evaluated with some confidence.

To use the sandboxed evaluator, load its module:

```
(use-modules (ice-9 sandbox))
```

Guile's sandboxing facility starts with the ability to restrict the time and space used by a piece of code.

`call-with-time-limit` *limit thunk limit-reached* [Scheme Procedure]
Call *thunk*, but cancel it if *limit* seconds of wall-clock time have elapsed. If the computation is cancelled, call *limit-reached* in tail position. *thunk* must not disable interrupts or prevent an abort via a `dynamic-wind` unwind handler.

`call-with-allocation-limit` *limit thunk limit-reached* [Scheme Procedure]
Call *thunk*, but cancel it if *limit* bytes have been allocated. If the computation is cancelled, call *limit-reached* in tail position. *thunk* must not disable interrupts or prevent an abort via a `dynamic-wind` unwind handler.

This limit applies to both stack and heap allocation. The computation will not be aborted before *limit* bytes have been allocated, but for the heap allocation limit, the check may be postponed until the next garbage collection.

Note that as a current shortcoming, the heap size limit applies to all threads; concurrent allocation by other unrelated threads counts towards the allocation limit.

`call-with-time-and-allocation-limits` *time-limit allocation-limit thunk* [Scheme Procedure]
Invoke *thunk* in a dynamic extent in which its execution is limited to *time-limit* seconds of wall-clock time, and its allocation to *allocation-limit* bytes. *thunk* must not disable interrupts or prevent an abort via a `dynamic-wind` unwind handler.

If successful, return all values produced by invoking *thunk*. Any uncaught exception thrown by the thunk will propagate out. If the time or allocation limit is exceeded, an exception will be thrown to the `limit-exceeded` key.

The time limit and stack limit are both very precise, but the heap limit only gets checked asynchronously, after a garbage collection. In particular, if the heap is already very large, the number of allocated bytes between garbage collections will be large, and therefore the precision of the check is reduced.

Additionally, due to the mechanism used by the allocation limit (the `after-gc-hook`), large single allocations like `(make-vector #e1e7)` are only detected after the allocation completes, even if the allocation itself causes garbage collection. It's possible therefore for user code to not only exceed the allocation limit set, but also to exhaust all available memory, causing out-of-memory conditions at any allocation site. Failure to allocate memory in Guile itself should be safe and cause an exception to be thrown, but most systems are not designed to handle `malloc` failures. An allocation failure may therefore exercise unexpected code paths in your system, so it is a weakness of the sandbox (and therefore an interesting point of attack).

The main sandbox interface is `eval-in-sandbox`.

`eval-in-sandbox` *exp* **[***#:time-limit 0.1***]** **[***#:allocation-limit #e10e6***]** **[***#:bindings all-pure-bindings***]** **[***#:module* (*make-sandbox-module bindings*)**]** **[***#:sever-module? #t***]** [Scheme Procedure]

Evaluate the Scheme expression *exp* within an isolated "sandbox". Limit its execution to *time-limit* seconds of wall-clock time, and limit its allocation to *allocation-limit* bytes.

The evaluation will occur in *module*, which defaults to the result of calling `make-sandbox-module` on *bindings*, which itself defaults to `all-pure-bindings`. This is the core of the sandbox: creating a scope for the expression that is *safe*.

A safe sandbox module has two characteristics. Firstly, it will not allow the expression being evaluated to avoid being cancelled due to time or allocation limits. This ensures that the expression terminates in a timely fashion.

Secondly, a safe sandbox module will prevent the evaluation from receiving information from previous evaluations, or from affecting future evaluations. All combinations of binding sets exported by `(ice-9 sandbox)` form safe sandbox modules.

The *bindings* should be given as a list of import sets. One import set is a list whose car names an interface, like `(ice-9 q)`, and whose cdr is a list of imports. An import is either a bare symbol or a pair of `(`*`out . in`*`)`, where *out* and *in* are both symbols and denote the name under which a binding is exported from the module, and the name under which to make the binding available, respectively. Note that *bindings* is only used as an input to the default initializer for the *module* argument; if you pass `#:module`, *bindings* is unused. If *sever-module?* is true (the default), the module will be unlinked from the global module tree after the evaluation returns, to allow *mod* to be garbage-collected.

If successful, return all values produced by *exp*. Any uncaught exception thrown by the expression will propagate out. If the time or allocation limit is exceeded, an exception will be thrown to the `limit-exceeded` key.

Constructing a safe sandbox module is tricky in general. Guile defines an easy way to construct safe modules from predefined sets of bindings. Before getting to that interface, here are some general notes on safety.

1. The time and allocation limits rely on the ability to interrupt and cancel a computation. For this reason, no binding included in a sandbox module should be able to indefinitely postpone interrupt handling, nor should a binding be able to prevent an abort. In

practice this second consideration means that `dynamic-wind` should not be included in any binding set.

2. The time and allocation limits apply only to the `eval-in-sandbox` call. If the call returns a procedure which is later called, no limit is "automatically" in place. Users of `eval-in-sandbox` have to be very careful to reimpose limits when calling procedures that escape from sandboxes.
3. Similarly, the dynamic environment of the `eval-in-sandbox` call is not necessarily in place when any procedure that escapes from the sandbox is later called.

   This detail prevents us from exposing `primitive-eval` to the sandbox, for two reasons. The first is that it's possible for legacy code to forge references to any binding, if the `allow-legacy-syntax-objects?` parameter is true. The default for this parameter is true; см. Раздел 6.10.4 [Syntax Transformer Helpers], страница 289, for the details. The parameter is bound to `#f` for the duration of the `eval-in-sandbox` call itself, but that will not be in place during calls to escaped procedures.

   The second reason we don't expose `primitive-eval` is that `primitive-eval` implicitly works in the current module, which for an escaped procedure will probably be different than the module that is current for the `eval-in-sandbox` call itself.

   The common denominator here is that if an interface exposed to the sandbox relies on dynamic environments, it is easy to mistakenly grant the sandboxed procedure additional capabilities in the form of bindings that it should not have access to. For this reason, the default sets of predefined bindings do not depend on any dynamically scoped value.
4. Mutation may allow a sandboxed evaluation to break some invariant in users of data supplied to it. A lot of code culturally doesn't expect mutation, but if you hand mutable data to a sandboxed evaluation and you also grant mutating capabilities to that evaluation, then the sandboxed code may indeed mutate that data. The default set of bindings to the sandbox do not include any mutating primitives.

   Relatedly, `set!` may allow a sandbox to mutate a primitive, invalidating many system-wide invariants. Guile is currently quite permissive when it comes to imported bindings and mutability. Although `set!` to a module-local or lexically bound variable would be fine, we don't currently have an easy way to disallow `set!` to an imported binding, so currently no binding set includes `set!`.
5. Mutation may allow a sandboxed evaluation to keep state, or make a communication mechanism with other code. On the one hand this sounds cool, but on the other hand maybe this is part of your threat model. Again, the default set of bindings doesn't include mutating primitives, preventing sandboxed evaluations from keeping state.
6. The sandbox should probably not be able to open a network connection, or write to a file, or open a file from disk. The default binding set includes no interaction with the operating system.

If you, dear reader, find the above discussion interesting, you will enjoy Jonathan Rees' dissertation, "A Security Kernel Based on the Lambda Calculus".

`all-pure-bindings` [Scheme Variable]

All "pure" bindings that together form a safe subset of those bindings available by default to Guile user code.

`all-pure-and-impure-bindings` [Scheme Variable]
: Like `all-pure-bindings`, but additionally including mutating primitives like `vector-set!`. This set is still safe in the sense mentioned above, with the caveats about mutation.

The components of these composite sets are as follows:

`alist-bindings` [Scheme Variable]
`array-bindings` [Scheme Variable]
`bit-bindings` [Scheme Variable]
`bitvector-bindings` [Scheme Variable]
`char-bindings` [Scheme Variable]
`char-set-bindings` [Scheme Variable]
`clock-bindings` [Scheme Variable]
`core-bindings` [Scheme Variable]
`error-bindings` [Scheme Variable]
`fluid-bindings` [Scheme Variable]
`hash-bindings` [Scheme Variable]
`iteration-bindings` [Scheme Variable]
`keyword-bindings` [Scheme Variable]
`list-bindings` [Scheme Variable]
`macro-bindings` [Scheme Variable]
`nil-bindings` [Scheme Variable]
`number-bindings` [Scheme Variable]
`pair-bindings` [Scheme Variable]
`predicate-bindings` [Scheme Variable]
`procedure-bindings` [Scheme Variable]
`promise-bindings` [Scheme Variable]
`prompt-bindings` [Scheme Variable]
`regexp-bindings` [Scheme Variable]
`sort-bindings` [Scheme Variable]
`srfi-4-bindings` [Scheme Variable]
`string-bindings` [Scheme Variable]
`symbol-bindings` [Scheme Variable]
`unspecified-bindings` [Scheme Variable]
`variable-bindings` [Scheme Variable]
`vector-bindings` [Scheme Variable]
`version-bindings` [Scheme Variable]
: The components of `all-pure-bindings`.

`mutating-alist-bindings` [Scheme Variable]
`mutating-array-bindings` [Scheme Variable]
`mutating-bitvector-bindings` [Scheme Variable]
`mutating-fluid-bindings` [Scheme Variable]
`mutating-hash-bindings` [Scheme Variable]
`mutating-list-bindings` [Scheme Variable]
`mutating-pair-bindings` [Scheme Variable]
`mutating-sort-bindings` [Scheme Variable]

`mutating-srfi-4-bindings` [Scheme Variable]
`mutating-string-bindings` [Scheme Variable]
`mutating-variable-bindings` [Scheme Variable]
`mutating-vector-bindings` [Scheme Variable]
The additional components of `all-pure-and-impure-bindings`.

Finally, what do you do with a binding set? What is a binding set anyway? `make-sandbox-module` is here for you.

`make-sandbox-module` *bindings* [Scheme Procedure]
Return a fresh module that only contains *bindings*.

The *bindings* should be given as a list of import sets. One import set is a list whose car names an interface, like `(ice-9 q)`, and whose cdr is a list of imports. An import is either a bare symbol or a pair of `(`*`out . in`*`)`, where *out* and *in* are both symbols and denote the name under which a binding is exported from the module, and the name under which to make the binding available, respectively.

So you see that binding sets are just lists, and `all-pure-and-impure-bindings` is really just the result of appending all of the component binding sets.

6.18.13 REPL Servers

The procedures in this section are provided by

```
(use-modules (system repl server))
```

When an application is written in Guile, it is often convenient to allow the user to be able to interact with it by evaluating Scheme expressions in a REPL.

The procedures of this module allow you to spawn a *REPL server*, which permits interaction over a local or TCP connection. Guile itself uses them internally to implement the `--listen` switch, Раздел 4.2.1 [Command-line Options], страница 37.

`make-tcp-server-socket` **[***#:host=#f***]** **[***#:addr***]** **[***#:port=37146***]** [Scheme Procedure]
Return a stream socket bound to a given address *addr* and port number *port*. If the *host* is given, and *addr* is not, then the *host* string is converted to an address. If neither is given, we use the loopback address.

`make-unix-domain-server-socket` [Scheme Procedure]
[*#:path="/tmp/guile-socket"***]**
Return a UNIX domain socket, bound to a given *path*.

`run-server` **[***server-socket***]** [Scheme Procedure]
`spawn-server` **[***server-socket***]** [Scheme Procedure]
Create and run a REPL, making it available over the given *server-socket*. If *server-socket* is not provided, it defaults to the socket created by calling `make-tcp-server-socket` with no arguments.

`run-server` runs the server in the current thread, whereas `spawn-server` runs the server in a new thread.

`stop-server-and-clients!` [Scheme Procedure]

Closes the connection on all running server sockets.

Please note that in the current implementation, the REPL threads are cancelled without unwinding their stacks. If any of them are holding mutexes or are within a critical section, the results are unspecified.

6.18.14 Cooperative REPL Servers

The procedures in this section are provided by

```
(use-modules (system repl coop-server))
```

Whereas ordinary REPL servers run in their own threads (см. Раздел 6.18.13 [REPL Servers], страница 424), sometimes it is more convenient to provide REPLs that run at specified times within an existing thread, for example in programs utilizing an event loop or in single-threaded programs. This allows for safe access and mutation of a program's data structures from the REPL, without concern for thread synchronization.

Although the REPLs are run in the thread that calls `spawn-coop-repl-server` and `poll-coop-repl-server`, dedicated threads are spawned so that the calling thread is not blocked. The spawned threads read input for the REPLs and to listen for new connections.

Cooperative REPL servers must be polled periodically to evaluate any pending expressions by calling `poll-coop-repl-server` with the object returned from `spawn-coop-repl-server`. The thread that calls `poll-coop-repl-server` will be blocked for as long as the expression takes to be evaluated or if the debugger is entered.

`spawn-coop-repl-server` [*server-socket*] [Scheme Procedure]

Create and return a new cooperative REPL server object, and spawn a new thread to listen for connections on *server-socket*. Proper functioning of the REPL server requires that `poll-coop-repl-server` be called periodically on the returned server object.

`poll-coop-repl-server` *coop-server* [Scheme Procedure]

Poll the cooperative REPL server *coop-server* and apply a pending operation if there is one, such as evaluating an expression typed at the REPL prompt. This procedure must be called from the same thread that called `spawn-coop-repl-server`.

6.19 Memory Management and Garbage Collection

Guile uses a *garbage collector* to manage most of its objects. While the garbage collector is designed to be mostly invisible, you sometimes need to interact with it explicitly.

See Раздел 5.4.2 [Garbage Collection], страница 72, for a general discussion of how garbage collection relates to using Guile from C.

6.19.1 Function related to Garbage Collection

`gc` [Scheme Procedure]

`scm_gc ()` [C Function]

Finds all of the "live" `SCM` objects and reclaims for further use those that are no longer accessible. You normally don't need to call this function explicitly. Its functionality is invoked automatically as needed.

`SCM scm_gc_protect_object (SCM obj)` [C Function]
Protects *obj* from being freed by the garbage collector, when it otherwise might be. When you are done with the object, call `scm_gc_unprotect_object` on the object. Calls to `scm_gc_protect_object`/`scm_gc_unprotect_object` can be nested, and the object remains protected until it has been unprotected as many times as it was protected. It is an error to unprotect an object more times than it has been protected. Returns the SCM object it was passed.

Note that storing *obj* in a C global variable has the same effect[9].

`SCM scm_gc_unprotect_object (SCM obj)` [C Function]
Unprotects an object from the garbage collector which was protected by `scm_gc_unprotect_object`. Returns the SCM object it was passed.

`SCM scm_permanent_object (SCM obj)` [C Function]
Similar to `scm_gc_protect_object` in that it causes the collector to always mark the object, except that it should not be nested (only call `scm_permanent_object` on an object once), and it has no corresponding unpermanent function. Once an object is declared permanent, it will never be freed. Returns the SCM object it was passed.

`void scm_remember_upto_here_1 (SCM obj)` [C Macro]
`void scm_remember_upto_here_2 (SCM obj1, SCM obj2)` [C Macro]
Create a reference to the given object or objects, so they're certain to be present on the stack or in a register and hence will not be freed by the garbage collector before this point.

Note that these functions can only be applied to ordinary C local variables (ie. "automatics"). Objects held in global or static variables or some malloced block or the like cannot be protected with this mechanism.

`gc-stats` [Scheme Procedure]
`scm_gc_stats ()` [C Function]
Return an association list of statistics about Guile's current use of storage.

`gc-live-object-stats` [Scheme Procedure]
`scm_gc_live_object_stats ()` [C Function]
Return an alist of statistics of the current live objects.

`void scm_gc_mark (SCM x)` [Функция]
Mark the object *x*, and recurse on any objects *x* refers to. If *x*'s mark bit is already set, return immediately. This function must only be called during the mark-phase of garbage collection, typically from a smob *mark* function.

6.19.2 Memory Blocks

In C programs, dynamic management of memory blocks is normally done with the functions malloc, realloc, and free. Guile has additional functions for dynamic memory allocation that are integrated into the garbage collector and the error reporting system.

[9] In Guile up to version 1.8, C global variables were not visited by the garbage collector in the mark phase; hence, `scm_gc_protect_object` was the only way in C to prevent a Scheme object from being freed.

Memory blocks that are associated with Scheme objects (for example a foreign object) should be allocated with `scm_gc_malloc` or `scm_gc_malloc_pointerless`. These two functions will either return a valid pointer or signal an error. Memory blocks allocated this way may be released explicitly; however, this is not strictly needed, and we recommend *not* calling `scm_gc_free`. All memory allocated with `scm_gc_malloc` or `scm_gc_malloc_pointerless` is automatically reclaimed when the garbage collector no longer sees any live reference to it[10].

When garbage collection occurs, Guile will visit the words in memory allocated with `scm_gc_malloc`, looking for live pointers. This means that if `scm_gc_malloc`-allocated memory contains a pointer to some other part of the memory, the garbage collector notices it and prevents it from being reclaimed[11]. Conversely, memory allocated with `scm_gc_malloc_pointerless` is assumed to be "pointer-less" and is not scanned for pointers.

For memory that is not associated with a Scheme object, you can use `scm_malloc` instead of `malloc`. Like `scm_gc_malloc`, it will either return a valid pointer or signal an error. However, it will not assume that the new memory block can be freed by a garbage collection. The memory must be explicitly freed with `free`.

There is also `scm_gc_realloc` and `scm_realloc`, to be used in place of `realloc` when appropriate, and `scm_gc_calloc` and `scm_calloc`, to be used in place of `calloc` when appropriate.

The function `scm_dynwind_free` can be useful when memory should be freed with libc's `free` when leaving a dynwind context, См. Раздел 6.13.10 [Dynamic Wind], страница 339.

`void * scm_malloc` (*size_t size*) [C Function]
`void * scm_calloc` (*size_t size*) [C Function]

Allocate *size* bytes of memory and return a pointer to it. When *size* is 0, return `NULL`. When not enough memory is available, signal an error. This function runs the GC to free up some memory when it deems it appropriate.

The memory is allocated by the libc `malloc` function and can be freed with `free`. There is no `scm_free` function to go with `scm_malloc` to make it easier to pass memory back and forth between different modules.

The function `scm_calloc` is similar to `scm_malloc`, but initializes the block of memory to zero as well.

These functions will (indirectly) call `scm_gc_register_allocation`.

`void * scm_realloc` (*void *mem, size_t new_size*) [C Function]

Change the size of the memory block at *mem* to *new_size* and return its new location. When *new_size* is 0, this is the same as calling `free` on *mem* and `NULL` is returned. When *mem* is `NULL`, this function behaves like `scm_malloc` and allocates a new block of size *new_size*.

When not enough memory is available, signal an error. This function runs the GC to free up some memory when it deems it appropriate.

This function will call `scm_gc_register_allocation`.

[10] In Guile up to version 1.8, memory allocated with `scm_gc_malloc` *had* to be freed with `scm_gc_free`.

[11] In Guile up to 1.8, memory allocated with `scm_gc_malloc` was *not* visited by the collector in the mark phase. Consequently, the GC had to be told explicitly about pointers to live objects contained in the memory block, e.g., *via* SMOB mark functions (см. Раздел 6.8 [Smobs], страница 259)

`void * scm_gc_malloc (size_t size, const char *what)` [C Function]
`void * scm_gc_malloc_pointerless (size_t size, const char *what)` [C Function]
`void * scm_gc_realloc (void *mem, size_t old_size, size_t new_size, const char *what);` [C Function]
`void * scm_gc_calloc (size_t size, const char *what)` [C Function]

Allocate *size* bytes of automatically-managed memory. The memory is automatically freed when no longer referenced from any live memory block.

When garbage collection occurs, Guile will visit the words in memory allocated with `scm_gc_malloc` or `scm_gc_calloc`, looking for pointers to other memory allocations that are managed by the GC. In contrast, memory allocated by `scm_gc_malloc_pointerless` is not scanned for pointers.

The `scm_gc_realloc` call preserves the "pointerlessness" of the memory area pointed to by *mem*. Note that you need to pass the old size of a reallocated memory block as well. See below for a motivation.

`void scm_gc_free (void *mem, size_t size, const char *what)` [C Function]

Explicitly free the memory block pointed to by *mem*, which was previously allocated by one of the above `scm_gc` functions. This function is almost always unnecessary, except for codebases that still need to compile on Guile 1.8.

Note that you need to explicitly pass the *size* parameter. This is done since it should normally be easy to provide this parameter (for memory that is associated with GC controlled objects) and help keep the memory management overhead very low. However, in Guile 2.x, *size* is always ignored.

`void scm_gc_register_allocation (size_t size)` [C Function]

Informs the garbage collector that *size* bytes have been allocated, which the collector would otherwise not have known about.

In general, Scheme will decide to collect garbage only after some amount of memory has been allocated. Calling this function will make the Scheme garbage collector know about more allocation, and thus run more often (as appropriate).

It is especially important to call this function when large unmanaged allocations, like images, may be freed by small Scheme allocations, like foreign objects.

`void scm_dynwind_free (void *mem)` [C Function]

Equivalent to `scm_dynwind_unwind_handler (free, mem, SCM_F_WIND_EXPLICITLY)`. That is, the memory block at *mem* will be freed (using `free` from the C library) when the current dynwind is left.

`malloc-stats` [Scheme Procedure]

Return an alist ((*what* . *n*) ...) describing number of malloced objects. *what* is the second argument to `scm_gc_malloc`, *n* is the number of objects of that type currently allocated.

This function is only available if the `GUILE_DEBUG_MALLOC` preprocessor macro was defined when Guile was compiled.

6.19.3 Weak References

[FIXME: This chapter is based on Mikael Djurfeldt's answer to a question by Michael Livshin. Any mistakes are not theirs, of course.]

Weak references let you attach bookkeeping information to data so that the additional information automatically disappears when the original data is no longer in use and gets garbage collected. In a weak key hash, the hash entry for that key disappears as soon as the key is no longer referenced from anywhere else. For weak value hashes, the same happens as soon as the value is no longer in use. Entries in a doubly weak hash disappear when either the key or the value are not used anywhere else anymore.

Object properties offer the same kind of functionality as weak key hashes in many situations. (см. Раздел 6.11.2 [Object Properties], страница 303)

Here's an example (a little bit strained perhaps, but one of the examples is actually used in Guile):

Assume that you're implementing a debugging system where you want to associate information about filename and position of source code expressions with the expressions themselves.

Hashtables can be used for that, but if you use ordinary hash tables it will be impossible for the scheme interpreter to "forget" old source when, for example, a file is reloaded.

To implement the mapping from source code expressions to positional information it is necessary to use weak-key tables since we don't want the expressions to be remembered just because they are in our table.

To implement a mapping from source file line numbers to source code expressions you would use a weak-value table.

To implement a mapping from source code expressions to the procedures they constitute a doubly-weak table has to be used.

6.19.3.1 Weak hash tables

`make-weak-key-hash-table` [*size*] [Scheme Procedure]
`make-weak-value-hash-table` [*size*] [Scheme Procedure]
`make-doubly-weak-hash-table` [*size*] [Scheme Procedure]
`scm_make_weak_key_hash_table` (*size*) [C Function]
`scm_make_weak_value_hash_table` (*size*) [C Function]
`scm_make_doubly_weak_hash_table` (*size*) [C Function]

Return a weak hash table with *size* buckets. As with any hash table, choosing a good size for the table requires some caution.

You can modify weak hash tables in exactly the same way you would modify regular hash tables, with the exception of the routines that act on handles. Weak tables have a different implementation behind the scenes that doesn't have handles. см. Раздел 6.6.22 [Hash Tables], страница 251, for more on `hashq-ref` et al.

Note that in a weak-key hash table, the reference to the value is strong. This means that if the value references the key, even indirectly, the key will never be collected, which can lead to a memory leak. The reverse is true for weak value tables.

`weak-key-hash-table?` *obj* [Scheme Procedure]
`weak-value-hash-table?` *obj* [Scheme Procedure]
`doubly-weak-hash-table?` *obj* [Scheme Procedure]
`scm_weak_key_hash_table_p` (*obj*) [C Function]
`scm_weak_value_hash_table_p` (*obj*) [C Function]
`scm_doubly_weak_hash_table_p` (*obj*) [C Function]
Return `#t` if *obj* is the specified weak hash table. Note that a doubly weak hash table is neither a weak key nor a weak value hash table.

6.19.3.2 Weak vectors

`make-weak-vector` *size* [*fill*] [Scheme Procedure]
`scm_make_weak_vector` (*size*, *fill*) [C Function]
Return a weak vector with *size* elements. If the optional argument *fill* is given, all entries in the vector will be set to *fill*. The default value for *fill* is the empty list.

`weak-vector` *elem ...* [Scheme Procedure]
`list->weak-vector` *l* [Scheme Procedure]
`scm_weak_vector` (*l*) [C Function]
Construct a weak vector from a list: `weak-vector` uses the list of its arguments while `list->weak-vector` uses its only argument *l* (a list) to construct a weak vector the same way `list->vector` would.

`weak-vector?` *obj* [Scheme Procedure]
`scm_weak_vector_p` (*obj*) [C Function]
Return `#t` if *obj* is a weak vector.

`weak-vector-ref` *wvect k* [Scheme Procedure]
`scm_weak_vector_ref` (*wvect*, *k*) [C Function]
Return the *k*th element of the weak vector *wvect*, or `#f` if that element has been collected.

`weak-vector-set!` *wvect k elt* [Scheme Procedure]
`scm_weak_vector_set_x` (*wvect*, *k*, *elt*) [C Function]
Set the *k*th element of the weak vector *wvect* to *elt*.

6.19.4 Guardians

Guardians provide a way to be notified about objects that would otherwise be collected as garbage. Guarding them prevents the objects from being collected and cleanup actions can be performed on them, for example.

See R. Kent Dybvig, Carl Bruggeman, and David Eby (1993) "Guardians in a Generation-Based Garbage Collector". ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1993.

`make-guardian` [Scheme Procedure]
`scm_make_guardian` () [C Function]
Create a new guardian. A guardian protects a set of objects from garbage collection, allowing a program to apply cleanup or other actions.

`make-guardian` returns a procedure representing the guardian. Calling the guardian procedure with an argument adds the argument to the guardian's set of protected objects. Calling the guardian procedure without an argument returns one of the protected objects which are ready for garbage collection, or `#f` if no such object is available. Objects which are returned in this way are removed from the guardian.

You can put a single object into a guardian more than once and you can put a single object into more than one guardian. The object will then be returned multiple times by the guardian procedures.

An object is eligible to be returned from a guardian when it is no longer referenced from outside any guardian.

There is no guarantee about the order in which objects are returned from a guardian. If you want to impose an order on finalization actions, for example, you can do that by keeping objects alive in some global data structure until they are no longer needed for finalizing other objects.

Being an element in a weak vector, a key in a hash table with weak keys, or a value in a hash table with weak values does not prevent an object from being returned by a guardian. But as long as an object can be returned from a guardian it will not be removed from such a weak vector or hash table. In other words, a weak link does not prevent an object from being considered collectable, but being inside a guardian prevents a weak link from being broken.

A key in a weak key hash table can be thought of as having a strong reference to its associated value as long as the key is accessible. Consequently, when the key is only accessible from within a guardian, the reference from the key to the value is also considered to be coming from within a guardian. Thus, if there is no other reference to the value, it is eligible to be returned from a guardian.

6.20 Modules

When programs become large, naming conflicts can occur when a function or global variable defined in one file has the same name as a function or global variable in another file. Even just a *similarity* between function names can cause hard-to-find bugs, since a programmer might type the wrong function name.

The approach used to tackle this problem is called *information encapsulation*, which consists of packaging functional units into a given name space that is clearly separated from other name spaces.

The language features that allow this are usually called *the module system* because programs are broken up into modules that are compiled separately (or loaded separately in an interpreter).

Older languages, like C, have limited support for name space manipulation and protection. In C a variable or function is public by default, and can be made local to a module with the `static` keyword. But you cannot reference public variables and functions from another module with different names.

More advanced module systems have become a common feature in recently designed languages: ML, Python, Perl, and Modula 3 all allow the *renaming* of objects from a foreign module, so they will not clutter the global name space.

In addition, Guile offers variables as first-class objects. They can be used for interacting with the module system.

6.20.1 General Information about Modules

A Guile module can be thought of as a collection of named procedures, variables and macros. More precisely, it is a set of *bindings* of symbols (names) to Scheme objects.

Within a module, all bindings are visible. Certain bindings can be declared *public*, in which case they are added to the module's so-called *export list*; this set of public bindings is called the module's *public interface* (см. Раздел 6.20.3 [Creating Guile Modules], страница 434).

A client module *uses* a providing module's bindings by either accessing the providing module's public interface, or by building a custom interface (and then accessing that). In a custom interface, the client module can *select* which bindings to access and can also algorithmically *rename* bindings. In contrast, when using the providing module's public interface, the entire export list is available without renaming (см. Раздел 6.20.2 [Using Guile Modules], страница 432).

All Guile modules have a unique *module name*, for example `(ice-9 popen)` or `(srfi srfi-11)`. Module names are lists of one or more symbols.

When Guile goes to use an interface from a module, for example `(ice-9 popen)`, Guile first looks to see if it has loaded `(ice-9 popen)` for any reason. If the module has not been loaded yet, Guile searches a *load path* for a file that might define it, and loads that file.

The following subsections go into more detail on using, creating, installing, and otherwise manipulating modules and the module system.

6.20.2 Using Guile Modules

To use a Guile module is to access either its public interface or a custom interface (см. Раздел 6.20.1 [General Information about Modules], страница 432). Both types of access are handled by the syntactic form `use-modules`, which accepts one or more interface specifications and, upon evaluation, arranges for those interfaces to be available to the current module. This process may include locating and loading code for a given module if that code has not yet been loaded, following `%load-path` (см. Раздел 6.20.4 [Modules and the File System], страница 437).

An *interface specification* has one of two forms. The first variation is simply to name the module, in which case its public interface is the one accessed. For example:

```
(use-modules (ice-9 popen))
```

Here, the interface specification is `(ice-9 popen)`, and the result is that the current module now has access to `open-pipe`, `close-pipe`, `open-input-pipe`, and so on (см. Раздел 7.2.10 [Pipes], страница 552).

Note in the previous example that if the current module had already defined `open-pipe`, that definition would be overwritten by the definition in `(ice-9 popen)`. For this reason (and others), there is a second variation of interface specification that not only names a module to be accessed, but also selects bindings from it and renames them to suit the current module's needs. For example:

```
(use-modules ((ice-9 popen)
```

```
                #:select ((open-pipe . pipe-open) close-pipe)
                #:renamer (symbol-prefix-proc 'unixy:)))
```

or more simply:

```
(use-modules ((ice-9 popen)
              #:select ((open-pipe . pipe-open) close-pipe)
              #:prefix unixy:))
```

Here, the interface specification is more complex than before, and the result is that a custom interface with only two bindings is created and subsequently accessed by the current module. The mapping of old to new names is as follows:

```
(ice-9 popen) sees:          current module sees:
open-pipe                    unixy:pipe-open
close-pipe                   unixy:close-pipe
```

This example also shows how to use the convenience procedure `symbol-prefix-proc`.

You can also directly refer to bindings in a module by using the `@` syntax. For example, instead of using the `use-modules` statement from above and writing `unixy:pipe-open` to refer to the `pipe-open` from the `(ice-9 popen)`, you could also write `(@ (ice-9 popen) open-pipe)`. Thus an alternative to the complete `use-modules` statement would be

```
(define unixy:pipe-open (@ (ice-9 popen) open-pipe))
(define unixy:close-pipe (@ (ice-9 popen) close-pipe))
```

There is also `@@`, which can be used like `@`, but does not check whether the variable that is being accessed is actually exported. Thus, `@@` can be thought of as the impolite version of `@` and should only be used as a last resort or for debugging, for example.

Note that just as with a `use-modules` statement, any module that has not yet been loaded will be loaded when referenced by a `@` or `@@` form.

You can also use the `@` and `@@` syntaxes as the target of a `set!` when the binding refers to a variable.

`symbol-prefix-proc` *prefix-sym* [Scheme Procedure]

Return a procedure that prefixes its arg (a symbol) with *prefix-sym*.

`use-modules` *spec ...* [syntax]

Resolve each interface specification *spec* into an interface and arrange for these to be accessible by the current module. The return value is unspecified.

spec can be a list of symbols, in which case it names a module whose public interface is found and used.

spec can also be of the form:

```
(MODULE-NAME [#:select SELECTION]
             [#:prefix PREFIX]
             [#:renamer RENAMER])
```

in which case a custom interface is newly created and used. *module-name* is a list of symbols, as above; *selection* is a list of selection-specs; *prefix* is a symbol that is prepended to imported names; and *renamer* is a procedure that takes a symbol and returns its new name. A selection-spec is either a symbol or a pair of symbols `(ORIG . SEEN)`, where *orig* is the name in the used module and *seen* is the name in the using module. Note that *seen* is also modified by *prefix* and *renamer*.

The `#:select`, `#:prefix`, and `#:renamer` clauses are optional. If all are omitted, the returned interface has no bindings. If the `#:select` clause is omitted, *prefix* and *renamer* operate on the used module's public interface.

In addition to the above, *spec* can also include a `#:version` clause, of the form:

```
#:version VERSION-SPEC
```

where *version-spec* is an R6RS-compatible version reference. An error will be signaled in the case in which a module with the same name has already been loaded, if that module specifies a version and that version is not compatible with *version-spec*. См. Раздел 6.20.5 [R6RS Version References], страница 438, for more on version references.

If the module name is not resolvable, `use-modules` will signal an error.

`@` *module-name binding-name* [syntax]

Refer to the binding named *binding-name* in module *module-name*. The binding must have been exported by the module.

`@@` *module-name binding-name* [syntax]

Refer to the binding named *binding-name* in module *module-name*. The binding must not have been exported by the module. This syntax is only intended for debugging purposes or as a last resort.

6.20.3 Creating Guile Modules

When you want to create your own modules, you have to take the following steps:

- Create a Scheme source file and add all variables and procedures you wish to export, or which are required by the exported procedures.
- Add a `define-module` form at the beginning.
- Export all bindings which should be in the public interface, either by using `define-public` or `export` (both documented below).

`define-module` *module-name option ...* [syntax]

module-name is a list of one or more symbols.

```
(define-module (ice-9 popen))
```

`define-module` makes this module available to Guile programs under the given *module-name*.

option ... are keyword/value pairs which specify more about the defined module. The recognized options and their meaning are shown in the following table.

`#:use-module` *interface-specification*

Equivalent to a `(use-modules` *interface-specification*`)` (см. Раздел 6.20.2 [Using Guile Modules], страница 432).

`#:autoload` *module symbol-list*

Load *module* when any of *symbol-list* are accessed. For example,

```
(define-module (my mod)
  #:autoload (srfi srfi-1) (partition delete-duplicates))
...
```

```
(if something
    (set! foo (delete-duplicates ...)))
```

When a module is autoloaded, all its bindings become available. *symbol-list* is just those that will first trigger the load.

An autoload is a good way to put off loading a big module until it's really needed, for instance for faster startup or if it will only be needed in certain circumstances.

`@` can do a similar thing (см. Раздел 6.20.2 [Using Guile Modules], страница 432), but in that case an `@` form must be written every time a binding from the module is used.

`#:export` *list*

Export all identifiers in *list* which must be a list of symbols or pairs of symbols. This is equivalent to `(export` *list*`)` in the module body.

`#:re-export` *list*

Re-export all identifiers in *list* which must be a list of symbols or pairs of symbols. The symbols in *list* must be imported by the current module from other modules. This is equivalent to `re-export` below.

`#:replace` *list*

Export all identifiers in *list* (a list of symbols or pairs of symbols) and mark them as *replacing bindings*. In the module user's name space, this will have the effect of replacing any binding with the same name that is not also "replacing". Normally a replacement results in an "override" warning message, `#:replace` avoids that.

In general, a module that exports a binding for which the `(guile)` module already has a definition should use `#:replace` instead of `#:export`. `#:replace`, in a sense, lets Guile know that the module *purposefully* replaces a core binding. It is important to note, however, that this binding replacement is confined to the name space of the module user. In other words, the value of the core binding in question remains unchanged for other modules.

Note that although it is often a good idea for the replaced binding to remain compatible with a binding in `(guile)`, to avoid surprising the user, sometimes the bindings will be incompatible. For example, SRFI-19 exports its own version of `current-time` (см. Раздел 7.5.16.2 [SRFI-19 Time], страница 638) which is not compatible with the core `current-time` function (см. Раздел 7.2.5 [Time], страница 537). Guile assumes that a user importing a module knows what she is doing, and uses `#:replace` for this binding rather than `#:export`.

A `#:replace` clause is equivalent to `(export!` *list*`)` in the module body.

The `#:duplicates` (see below) provides fine-grain control about duplicate binding handling on the module-user side.

`#:version` *list*

Specify a version for the module in the form of *list*, a list of zero or more exact, nonnegative integers. The corresponding `#:version` option in the

`use-modules` form allows callers to restrict the value of this option in various ways.

`#:duplicates` *list*

Tell Guile to handle duplicate bindings for the bindings imported by the current module according to the policy defined by *list*, a list of symbols. *list* must contain symbols representing a duplicate binding handling policy chosen among the following:

`check` Raises an error when a binding is imported from more than one place.

`warn` Issue a warning when a binding is imported from more than one place and leave the responsibility of actually handling the duplication to the next duplicate binding handler.

`replace` When a new binding is imported that has the same name as a previously imported binding, then do the following:

1. If the old binding was said to be *replacing* (via the `#:replace` option above) and the new binding is not replacing, the keep the old binding.
2. If the old binding was not said to be replacing and the new binding is replacing, then replace the old binding with the new one.
3. If neither the old nor the new binding is replacing, then keep the old one.

`warn-override-core`

Issue a warning when a core binding is being overwritten and actually override the core binding with the new one.

`first` In case of duplicate bindings, the firstly imported binding is always the one which is kept.

`last` In case of duplicate bindings, the lastly imported binding is always the one which is kept.

`noop` In case of duplicate bindings, leave the responsibility to the next duplicate handler.

If *list* contains more than one symbol, then the duplicate binding handlers which appear first will be used first when resolving a duplicate binding situation. As mentioned above, some resolution policies may explicitly leave the responsibility of handling the duplication to the next handler in *list*.

If GOOPS has been loaded before the `#:duplicates` clause is processed, there are additional strategies available for dealing with generic functions. См. Раздел 8.6.3 [Merging Generics], страница 798, for more information.

The default duplicate binding resolution policy is given by the `default-duplicate-binding-handler` procedure, and is

```
(replace warn-override-core warn last)
```

`#:pure` Create a *pure* module, that is a module which does not contain any of the standard procedure bindings except for the syntax forms. This is useful if you want to create *safe* modules, that is modules which do not know anything about dangerous procedures.

`export` *variable ...* [syntax]
Add all *variable*s (which must be symbols or pairs of symbols) to the list of exported bindings of the current module. If *variable* is a pair, its `car` gives the name of the variable as seen by the current module and its `cdr` specifies a name for the binding in the current module's public interface.

`define-public ...` [syntax]
Equivalent to `(begin (define foo ...) (export foo))`.

`re-export` *variable ...* [syntax]
Add all *variable*s (which must be symbols or pairs of symbols) to the list of re-exported bindings of the current module. Pairs of symbols are handled as in `export`. Re-exported bindings must be imported by the current module from some other module.

`export!` *variable ...* [syntax]
Like `export`, but marking the exported variables as replacing. Using a module with replacing bindings will cause any existing bindings to be replaced without issuing any warnings. See the discussion of `#:replace` above.

6.20.4 Modules and the File System

Typical programs only use a small subset of modules installed on a Guile system. In order to keep startup time down, Guile only loads modules when a program uses them, on demand.

When a program evaluates `(use-modules (ice-9 popen))`, and the module is not loaded, Guile searches for a conventionally-named file from in the *load path*.

In this case, loading `(ice-9 popen)` will eventually cause Guile to run `(primitive-load-path "ice-9/popen")`. `primitive-load-path` will search for a file `ice-9/popen` in the `%load-path` (см. Раздел 6.18.7 [Load Paths], страница 414). For each directory in `%load-path`, Guile will try to find the file name, concatenated with the extensions from `%load-extensions`. By default, this will cause Guile to `stat ice-9/popen.scm`, and then `ice-9/popen`. См. Раздел 6.18.7 [Load Paths], страница 414, for more on `primitive-load-path`.

If a corresponding compiled `.go` file is found in the `%load-compiled-path` or in the fallback path, and is as fresh as the source file, it will be loaded instead of the source file. If no compiled file is found, Guile may try to compile the source file and cache away the resulting `.go` file. См. Раздел 6.18.5 [Compilation], страница 411, for more on compilation.

Once Guile finds a suitable source or compiled file is found, the file will be loaded. If, after loading the file, the module under consideration is still not defined, Guile will signal an error.

For more information on where and how to install Scheme modules, См. Раздел 4.7 [Installing Site Packages], страница 61.

6.20.5 R6RS Version References

Guile's module system includes support for locating modules based on a declared version specifier of the same form as the one described in R6RS (см. Раздел "Library form" в *The Revised^6 Report on the Algorithmic Language Scheme*). By using the `#:version` keyword in a `define-module` form, a module may specify a version as a list of zero or more exact, nonnegative integers.

This version can then be used to locate the module during the module search process. Client modules and callers of the `use-modules` function may specify constraints on the versions of target modules by providing a *version reference*, which has one of the following forms:

```
(sub-version-reference ...)
(and version-reference ...)
(or version-reference ...)
(not version-reference)
```

in which *sub-version-reference* is in turn one of:

```
(sub-version)
(>= sub-version)
(<= sub-version)
(and sub-version-reference ...)
(or sub-version-reference ...)
(not sub-version-reference)
```

in which *sub-version* is an exact, nonnegative integer as above. A version reference matches a declared module version if each element of the version reference matches a corresponding element of the module version, according to the following rules:

- The `and` sub-form matches a version or version element if every element in the tail of the sub-form matches the specified version or version element.
- The `or` sub-form matches a version or version element if any element in the tail of the sub-form matches the specified version or version element.
- The `not` sub-form matches a version or version element if the tail of the sub-form does not match the version or version element.
- The `>=` sub-form matches a version element if the element is greater than or equal to the *sub-version* in the tail of the sub-form.
- The `<=` sub-form matches a version element if the version is less than or equal to the *sub-version* in the tail of the sub-form.
- A *sub-version* matches a version element if one is *eqv?* to the other.

For example, a module declared as:

```
(define-module (mylib mymodule) #:version (1 2 0))
```

would be successfully loaded by any of the following `use-modules` expressions:

```
(use-modules ((mylib mymodule) #:version (1 2 (>= 0))))
(use-modules ((mylib mymodule) #:version (or (1 2 0) (1 2 1))))
(use-modules ((mylib mymodule) #:version ((and (>= 1) (not 2)) 2 0)))
```

6.20.6 R6RS Libraries

In addition to the API described in the previous sections, you also have the option to create modules using the portable `library` form described in R6RS (см. Раздел "Library form" в *The Revised^6 Report on the Algorithmic Language Scheme*), and to import libraries created in this format by other programmers. Guile's R6RS library implementation takes advantage of the flexibility built into the module system by expanding the R6RS library form into a corresponding Guile `define-module` form that specifies equivalent import and export requirements and includes the same body expressions. The library expression:

```
(library (mylib (1 2))
  (export mybinding)
  (import (otherlib (3))))
```

is equivalent to the module definition:

```
(define-module (mylib)
  #:version (1 2)
  #:use-module ((otherlib) #:version (3))
  #:export (mybinding))
```

Central to the mechanics of R6RS libraries is the concept of import and export *levels*, which control the visibility of bindings at various phases of a library's lifecycle — macros necessary to expand forms in the library's body need to be available at expand time; variables used in the body of a procedure exported by the library must be available at runtime. R6RS specifies the optional `for` sub-form of an *import set* specification (see below) as a mechanism by which a library author can indicate that a particular library import should take place at a particular phase with respect to the lifecycle of the importing library.

Guile's library implementation uses a technique called *implicit phasing* (first described by Abdulaziz Ghuloum and R. Kent Dybvig), which allows the expander and compiler to automatically determine the necessary visibility of a binding imported from another library. As such, the `for` sub-form described below is ignored by Guile (but may be required by Schemes in which phasing is explicit).

`library` *name* (*export export-spec ...*) (*import import-spec ...*) *body ...* [Scheme Syntax]

Defines a new library with the specified name, exports, and imports, and evaluates the specified body expressions in this library's environment.

The library *name* is a non-empty list of identifiers, optionally ending with a version specification of the form described above (см. Раздел 6.20.3 [Creating Guile Modules], страница 434).

Each *export-spec* is the name of a variable defined or imported by the library, or must take the form `(rename (internal-name external-name) ...)`, where the identifier *internal-name* names a variable defined or imported by the library and *external-name* is the name by which the variable is seen by importing libraries.

Each *import-spec* must be either an *import set* (see below) or must be of the form `(for import-set import-level ...)`, where each *import-level* is one of:

```
run
expand
```

```
(meta level)
```

where *level* is an integer. Note that since Guile does not require explicit phase specification, any *import-set*s found inside of `for` sub-forms will be "unwrapped" during expansion and processed as if they had been specified directly.

Import sets in turn take one of the following forms:

```
library-reference
(library library-reference)
(only import-set identifier ...)
(except import-set identifier ...)
(prefix import-set identifier)
(rename import-set (internal-identifier external-identifier) ...)
```

where *library-reference* is a non-empty list of identifiers ending with an optional version reference (см. Раздел 6.20.5 [R6RS Version References], страница 438), and the other sub-forms have the following semantics, defined recursively on nested *import-set*s:

- The `library` sub-form is used to specify libraries for import whose names begin with the identifier "library."
- The `only` sub-form imports only the specified *identifier*s from the given *import-set*.
- The `except` sub-form imports all of the bindings exported by *import-set* except for those that appear in the specified list of *identifier*s.
- The `prefix` sub-form imports all of the bindings exported by *import-set*, first prefixing them with the specified *identifier*.
- The `rename` sub-form imports all of the identifiers exported by *import-set*. The binding for each *internal-identifier* among these identifiers is made visible to the importing library as the corresponding *external-identifier*; all other bindings are imported using the names provided by *import-set*.

Note that because Guile translates R6RS libraries into module definitions, an import specification may be used to declare a dependency on a native Guile module — although doing so may make your libraries less portable to other Schemes.

`import` *import-spec ...* [Scheme Syntax]

Import into the current environment the libraries specified by the given import specifications, where each *import-spec* takes the same form as in the `library` form described above.

6.20.7 Variables

Each module has its own hash table, sometimes known as an *obarray*, that maps the names defined in that module to their corresponding variable objects.

A variable is a box-like object that can hold any Scheme value. It is said to be *undefined* if its box holds a special Scheme value that denotes undefined-ness (which is different from all other Scheme values, including for example `#f`); otherwise the variable is *defined*.

On its own, a variable object is anonymous. A variable is said to be *bound* when it is associated with a name in some way, usually a symbol in a module obarray. When this happens, the name is said to be bound to the variable, in that module.

(That's the theory, anyway. In practice, defined-ness and bound-ness sometimes get confused, because Lisp and Scheme implementations have often conflated — or deliberately drawn no distinction between — a name that is unbound and a name that is bound to a variable whose value is undefined. We will try to be clear about the difference and explain any confusion where it is unavoidable.)

Variables do not have a read syntax. Most commonly they are created and bound implicitly by `define` expressions: a top-level `define` expression of the form

```
(define name value)
```

creates a variable with initial value *value* and binds it to the name *name* in the current module. But they can also be created dynamically by calling one of the constructor procedures `make-variable` and `make-undefined-variable`.

`make-undefined-variable` [Scheme Procedure]
`scm_make_undefined_variable` () [C Function]
Return a variable that is initially unbound.

`make-variable` *init* [Scheme Procedure]
`scm_make_variable` (*init*) [C Function]
Return a variable initialized to value *init*.

`variable-bound?` *var* [Scheme Procedure]
`scm_variable_bound_p` (*var*) [C Function]
Return `#t` if *var* is bound to a value, or `#f` otherwise. Throws an error if *var* is not a variable object.

`variable-ref` *var* [Scheme Procedure]
`scm_variable_ref` (*var*) [C Function]
Dereference *var* and return its value. *var* must be a variable object; see `make-variable` and `make-undefined-variable`.

`variable-set!` *var val* [Scheme Procedure]
`scm_variable_set_x` (*var, val*) [C Function]
Set the value of the variable *var* to *val*. *var* must be a variable object, *val* can be any value. Return an unspecified value.

`variable-unset!` *var* [Scheme Procedure]
`scm_variable_unset_x` (*var*) [C Function]
Unset the value of the variable *var*, leaving *var* unbound.

`variable?` *obj* [Scheme Procedure]
`scm_variable_p` (*obj*) [C Function]
Return `#t` if *obj* is a variable object, else return `#f`.

6.20.8 Module System Reflection

The previous sections have described a declarative view of the module system. You can also work with it programmatically by accessing and modifying various parts of the Scheme objects that Guile uses to implement the module system.

At any time, there is a *current module*. This module is the one where a top-level `define` and similar syntax will add new bindings. You can find other module objects with `resolve-module`, for example.

These module objects can be used as the second argument to `eval`.

`current-module` [Scheme Procedure]
`scm_current_module` () [C Function]
Return the current module object.

`set-current-module` *module* [Scheme Procedure]
`scm_set_current_module` (*module*) [C Function]
Set the current module to *module* and return the previous current module.

`save-module-excursion` *thunk* [Scheme Procedure]
Call *thunk* within a `dynamic-wind` such that the module that is current at invocation time is restored when *thunk*'s dynamic extent is left (см. Раздел 6.13.10 [Dynamic Wind], страница 339).

More precisely, if *thunk* escapes non-locally, the current module (at the time of escape) is saved, and the original current module (at the time *thunk*'s dynamic extent was last entered) is restored. If *thunk*'s dynamic extent is re-entered, then the current module is saved, and the previously saved inner module is set current again.

`resolve-module` *name* [*autoload*=*#t*] [*version*=*#f*] [*#:ensure*=*#t*] [Scheme Procedure]
`scm_resolve_module` (*name*) [C Function]
Find the module named *name* and return it. When it has not already been defined and *autoload* is true, try to auto-load it. When it can't be found that way either, create an empty module if *ensure* is true, otherwise return `#f`. If *version* is true, ensure that the resulting module is compatible with the given version reference (см. Раздел 6.20.5 [R6RS Version References], страница 438). The name is a list of symbols.

`resolve-interface` *name* [*#:select*=*#f*] [*#:hide*='()] [*#:prefix*=*#f*] [*#:renamer*=*#f*] [*#:version*=*#f*] [Scheme Procedure]
Find the module named *name* as with `resolve-module` and return its interface. The interface of a module is also a module object, but it contains only the exported bindings.

`module-uses` *module* [Scheme Procedure]
Return a list of the interfaces used by *module*.

`module-use!` *module interface* [Scheme Procedure]
Add *interface* to the front of the use-list of *module*. Both arguments should be module objects, and *interface* should very likely be a module returned by `resolve-interface`.

`reload-module` *module* [Scheme Procedure]
Revisit the source file that corresponds to *module*. Raises an error if no source file is associated with the given module.

As mentioned in the previous section, modules contain a mapping between identifiers (as symbols) and storage locations (as variables). Guile defines a number of procedures to allow access to this mapping. If you are programming in C, Раздел 6.20.9 [Accessing Modules from C], страница 443.

`module-variable` *module name* [Scheme Procedure]
Return the variable bound to *name* (a symbol) in *module*, or `#f` if *name* is unbound.

`module-add!` *module name var* [Scheme Procedure]
Define a new binding between *name* (a symbol) and *var* (a variable) in *module*.

`module-ref` *module name* [Scheme Procedure]
Look up the value bound to *name* in *module*. Like `module-variable`, but also does a `variable-ref` on the resulting variable, raising an error if *name* is unbound.

`module-define!` *module name value* [Scheme Procedure]
Locally bind *name* to *value* in *module*. If *name* was already locally bound in *module*, i.e., defined locally and not by an imported module, the value stored in the existing variable will be updated. Otherwise, a new variable will be added to the module, via `module-add!`.

`module-set!` *module name value* [Scheme Procedure]
Update the binding of *name* in *module* to *value*, raising an error if *name* is not already bound in *module*.

There are many other reflective procedures available in the default environment. If you find yourself using one of them, please contact the Guile developers so that we can commit to stability for that interface.

6.20.9 Accessing Modules from C

The last sections have described how modules are used in Scheme code, which is the recommended way of creating and accessing modules. You can also work with modules from C, but it is more cumbersome.

The following procedures are available.

`SCM scm_c_call_with_current_module` (*SCM module, SCM (*func)(void *), void *data*) [C Function]
Call *func* and make *module* the current module during the call. The argument *data* is passed to *func*. The return value of `scm_c_call_with_current_module` is the return value of *func*.

`SCM scm_public_variable` (*SCM module_name, SCM name*) [C Function]
`SCM scm_c_public_variable` (*const char *module_name, const char *name*) [C Function]
Find a the variable bound to the symbol *name* in the public interface of the module named *module_name*.

module_name should be a list of symbols, when represented as a Scheme object, or a space-separated string, in the `const char *` case. See `scm_c_define_module` below, for more examples.

Signals an error if no module was found with the given name. If *name* is not bound in the module, just returns `#f`.

`SCM scm_private_variable (SCM module_name, SCM name)` [C Function]
`SCM scm_c_private_variable (const char *module_name, const char *name)` [C Function]
Like `scm_public_variable`, but looks in the internals of the module named *module_name* instead of the public interface. Logically, these procedures should only be called on modules you write.

`SCM scm_public_lookup (SCM module_name, SCM name)` [C Function]
`SCM scm_c_public_lookup (const char *module_name, const char *name)` [C Function]
`SCM scm_private_lookup (SCM module_name, SCM name)` [C Function]
`SCM scm_c_private_lookup (const char *module_name, const char *name)` [C Function]
Like `scm_public_variable` or `scm_private_variable`, but if the *name* is not bound in the module, signals an error. Returns a variable, always.

```
static SCM eval_string_var;

/* NOTE: It is important that the call to 'my_init'
   happens-before all calls to 'my_eval_string'. */
void my_init (void)
{
  eval_string_var = scm_c_public_lookup ("ice-9 eval-string",
                                         "eval-string");
}

SCM my_eval_string (SCM str)
{
  return scm_call_1 (scm_variable_ref (eval_string_var), str);
}
```

`SCM scm_public_ref (SCM module_name, SCM name)` [C Function]
`SCM scm_c_public_ref (const char *module_name, const char *name)` [C Function]
`SCM scm_private_ref (SCM module_name, SCM name)` [C Function]
`SCM scm_c_private_ref (const char *module_name, const char *name)` [C Function]
Like `scm_public_lookup` or `scm_private_lookup`, but additionally dereferences the variable. If the variable object is unbound, signals an error. Returns the value bound to *name* in *module_name*.

In addition, there are a number of other lookup-related procedures. We suggest that you use the `scm_public_` and `scm_private_` family of procedures instead, if possible.

`SCM scm_c_lookup (const char *name)` [C Function]
Return the variable bound to the symbol indicated by *name* in the current module. If there is no such binding or the symbol is not bound to a variable, signal an error.

`SCM scm_lookup (SCM name)` [C Function]
Like `scm_c_lookup`, but the symbol is specified directly.

`SCM scm_c_module_lookup (SCM module, const char *name)` [C Function]
`SCM scm_module_lookup (SCM module, SCM name)` [C Function]
Like `scm_c_lookup` and `scm_lookup`, but the specified module is used instead of the current one.

`SCM scm_module_variable (SCM module, SCM name)` [C Function]
Like `scm_module_lookup`, but if the binding does not exist, just returns `#f` instead of raising an error.

To define a value, use `scm_define`:

`SCM scm_c_define (const char *name, SCM val)` [C Function]
Bind the symbol indicated by *name* to a variable in the current module and set that variable to *val*. When *name* is already bound to a variable, use that. Else create a new variable.

`SCM scm_define (SCM name, SCM val)` [C Function]
Like `scm_c_define`, but the symbol is specified directly.

`SCM scm_c_module_define (SCM module, const char *name, SCM val)` [C Function]
`SCM scm_module_define (SCM module, SCM name, SCM val)` [C Function]
Like `scm_c_define` and `scm_define`, but the specified module is used instead of the current one.

In some rare cases, you may need to access the variable that `scm_module_define` would have accessed, without changing the binding of the existing variable, if one is present. In that case, use `scm_module_ensure_local_variable`:

`SCM scm_module_ensure_local_variable (SCM module, SCM sym)` [C Function]
Like `scm_module_define`, but if the *sym* is already locally bound in that module, the variable's existing binding is not reset. Returns a variable.

`SCM scm_module_reverse_lookup (SCM module, SCM variable)` [C Function]
Find the symbol that is bound to *variable* in *module*. When no such binding is found, return `#f`.

`SCM scm_c_define_module (const char *name, void (*init)(void *), void *data)` [C Function]
Define a new module named *name* and make it current while *init* is called, passing it *data*. Return the module.

The parameter *name* is a string with the symbols that make up the module name, separated by spaces. For example, '`"foo bar"`' names the module '`(foo bar)`'.

When there already exists a module named *name*, it is used unchanged, otherwise, an empty module is created.

`SCM scm_c_resolve_module (const char *name)` [C Function]
Find the module name *name* and return it. When it has not already been defined, try to auto-load it. When it can't be found that way either, create an empty module. The name is interpreted as for `scm_c_define_module`.

`SCM scm_c_use_module` (*const char *name*) [C Function]
: Add the module named *name* to the uses list of the current module, as with `(use-modules name)`. The name is interpreted as for `scm_c_define_module`.

`void scm_c_export` (*const char *name, ...*) [C Function]
: Add the bindings designated by *name*, ... to the public interface of the current module. The list of names is terminated by `NULL`.

6.20.10 provide and require

Aubrey Jaffer, mostly to support his portable Scheme library SLIB, implemented a provide/require mechanism for many Scheme implementations. Library files in SLIB *provide* a feature, and when user programs *require* that feature, the library file is loaded in.

For example, the file `random.scm` in the SLIB package contains the line

```
(provide 'random)
```

so to use its procedures, a user would type

```
(require 'random)
```

and they would magically become available, *but still have the same names!* So this method is nice, but not as good as a full-featured module system.

When SLIB is used with Guile, provide and require can be used to access its facilities.

6.20.11 Environments

Scheme, as defined in R5RS, does *not* have a full module system. However it does define the concept of a top-level *environment*. Such an environment maps identifiers (symbols) to Scheme objects such as procedures and lists: Раздел 3.4 [About Closure], страница 27. In other words, it implements a set of *bindings*.

Environments in R5RS can be passed as the second argument to `eval` (см. Раздел 6.18.4 [Fly Evaluation], страница 409). Three procedures are defined to return environments: `scheme-report-environment`, `null-environment` and `interaction-environment` (см. Раздел 6.18.4 [Fly Evaluation], страница 409).

In addition, in Guile any module can be used as an R5RS environment, i.e., passed as the second argument to `eval`.

Note: the following two procedures are available only when the `(ice-9 r5rs)` module is loaded:

```
(use-modules (ice-9 r5rs))
```

`scheme-report-environment` *version* [Scheme Procedure]
`null-environment` *version* [Scheme Procedure]
: *version* must be the exact integer '5', corresponding to revision 5 of the Scheme report (the Revised^5 Report on Scheme). `scheme-report-environment` returns a specifier for an environment that is empty except for all bindings defined in the report that are either required or both optional and supported by the implementation. `null-environment` returns a specifier for an environment that is empty except for the (syntactic) bindings for all syntactic keywords defined in the report that are either required or both optional and supported by the implementation.

Currently Guile does not support values of *version* for other revisions of the report.

The effect of assigning (through the use of `eval`) a variable bound in a `scheme-report-environment` (for example `car`) is unspecified. Currently the environments specified by `scheme-report-environment` are not immutable in Guile.

6.21 Интерфейс Внешних Функций

Чем больше хакеров в Scheme, тем больше осознается, что на самом деле существуют два мира вычислений: один теплый и живой, это мир круглых скобок и один холодный и мертвый, это мир Си и ему подобный.

Но все же мы, как программисты живем в обоих мирах, а сам Guile частично реализован на Си. Таким образом живая половина Guile платит дань уважения к ее мертвой половине, через спектр интерфейсов к Си, начиная от динамической загрузки примитивов Scheme до динамического связывания библиотечных Си процедур.

6.21.1 Внешние Библиотеки

У большинства современных Юниксов есть что-то, что называется разделяемыми библиотеками(*shared libraries*). Это обычно означает, что они имеют возможность совместно использовать исполняемый образ библиотеки между несколькими запущенными программами для экономии памяти и дискового пространства. Но как правило, разделяемые(общие) библиотеки дают большую дополнительную гибкость по сравнению с традиционными статическими библиотеками. Фактически, название их динмаическими('dynamic') библиотеками так же корректно, как и название их общими('shared').

Разделяемые библиотеки действительно дают вам большую гибкость в дополнении к экономии памяти и пространства диска. Когда вы связываете программу с разделяемой библиотекой, эта библиотека не жестко включается в окончательный исполняемый файл. Вместо этого исполняемый файл вашей программы содержит только необходимую информацию для поиска необходимых разделяемых библиотек необходимых для запуска вашей программы. Только тогда, когда программа запускается, происходит последний шаг процесса связвания. Это означает, что вам не нужно перекомпилировать все программы при установке новой, только слегка измененной версии разделяемой библиотеки. Программы автоматически получат изменения при следующем запуске.

Теперь, когда все необходимое для машины должно выполнять часть связывания во время выполнения, почему бы не сделать следующий шаг и позволить программисту явно воспользоваться преимуществами этого в рамках своей программы? Конечно, многие операционные системы поддерживающие разделяемые библиотеки делают именно это, и скорее всего, Guile позволит вам получить доступ к этой функции из ваших программ Scheme. Как вы уже догадались эта функция называется динамическое связывание(*dynamic linking*).[12]

[12] Некоторые люди также ссылаются на конечный этап компоновки при запуске программы как на 'динамическое связывание', поэтому если вы хотите прояснить этот вопрос окончательно, вероятно лучше использовать более технический термин *dlopening*, как было предложено Gordon Matzigkeit в его документации по libtool.

Мы назвали этот разде Внешние библиотеки(“foreign libraries”), потому что, хотя название Внешние(“foreign”) не является утечкой в API, мир Си действительно внешний по отношению к Scheme – и это отчуждение распространяется и на компоненты внешних библиотек, как мы увидим в следующих разделах.

`dynamic-link` [*library*] [Scheme Procedure]
`scm_dynamic_link` (*library*) [C Function]

Ищет разделяемую библиотеку, указанную *library* (строка) и связывает ее с текущим выполняемым приложением Guile. Когда все сработает, возвращает объект Scheme подходящий для представления связванного объектного файла. В противном случае возникает ошибка. Как объектные файлы ищутся зависит от системы.

Обычно, *library* это просто имя файла разделяемой библиотеки, которую нужно искать в местах, где обычно находятся разделяемые библиотеки, например в `/usr/lib` и `/usr/local/lib`.

library не должна содержать расширений, таких как `.so`. Правильное расширение имени файла предоставляется автоматически в зависимости от операционной системы хоста, в соответствии с правилами libltdl (см. см. Раздел “`lt_dlopenext`” в *Поддержка разделяемых библиотек для GNU*).

Когда *library* пропущено, возвращается хендл/дескриптор(ручка, а по существу указатель) глобальных символов*global symbol handle*. Этот дескриптор обеспечивает доступ к символам, доступным программе во время выполнения, включая экспортированные самой программой и уже загруженные разделяемые библиотеки.

Обратите внимание, что на машине(хосте) использующем динамически загружаемые(компонуемые) библиотеки(DLLs), дескриптор глобальных символов возможно не сможет обеспечить доступк к символам из рекурсивно загружаемых библиотек DLL. Только экспортируемые символы из этих DLL, непосредственно загружаемых программой, могут быть доступны.

`dynamic-object?` *obj* [Scheme Procedure]
`scm_dynamic_object_p` (*obj*) [C Function]

Возвращает `#t` если *obj* является дескриптором/хендлом динамической библиотеки или `#f` в противном случае.

`dynamic-unlink` *dobj* [Scheme Procedure]
`scm_dynamic_unlink` (*dobj*) [C Function]

Unlink указывает файловый объект отключен от приложения. Аргумент *dobj* должен быть получен путем вызова `dynamic-link`. После вызова `dynamic-unlink` содержимое *dobj* больше не доступно.

```
(define libgl-obj (dynamic-link "libGL"))
libgl-obj
⇒ #<dynamic-object "libGL">
(dynamic-unlink libGL-obj)
libGL-obj
⇒ #<dynamic-object "libGL" (unlinked)>
```

Как вы можете видеть, после вызова `dynamic-unlink` с динамически связанной библиотекой, она помечается как отсоединенная(‘`(unlinked)`’) и вы больше не можете

использовать ее с `dynamic-call`, и т.д. Независимо от того, действительно ли библиотека удалена из вашей программы в зависимости от системы и как правило ничего не произойдет, когда некоторые другие части вашей программы все еще используют ее.

Когда динамическое связывание не доступно или не поддерживается вашей системой, вышеуказанные функции выбрасывают ошибки(исключения), но они все еще доступны.

6.21.2 Внешние Функции

Самое естественное что можно сделать с динамической библиотекой это найти в ней(взять из нее) указатель на функцию: внешнюю функцию(*foreign function*). Для этой цели служит функция `dynamic-func` .

`dynamic-func` *name dobj* [Scheme Procedure]
`scm_dynamic_func` (*name, dobj*) [C Function]

Возвращает дескриптор("handle") для функции с именем *name* в разделяемом объекте на который ссылается *dobj*. Дескриптор может быть передан `dynamic-call` для фактического вызова функции.

Независимо от того, добавляет ли ваш Си компилятор символ подчеркивания '_' к глобальным именам в программе, вы **не** должны включать это подчеркивание в имя, так как оно будет автоматически добавлено в *name* при необходимости.

Guile имеет статическую поддержку для вызовов функций без аргументов, `dynamic-call`.

`dynamic-call` *func dobj* [Scheme Procedure]
`scm_dynamic_call` (*func, dobj*) [C Function]

Вызывает Си функцию указанную *func* и *dobj*. Функции не передаются аргументы и возвращаемое ей значение игнорируется. Когда *function* возвращается как результат `dynamic-func`, вызовите эту функцию и игнорируйте *dobj*. Когда *func* является строкой, она ищет дескриптор функции в *dynobj*; этот вызов эквивалентен коду

```
(dynamic-call (dynamic-func func dobj) #f)
```

`dynamic-call` не очень мощная функция. Она в основном предназначена для использования специально написанных инициализирующих фунций, которые затем добавят новые примитивы в Guile. Например, мы не ожидаем что вы будете динамически связывать `libX11` используя `dynamic-link`, а затем строить красивый графический интерфейс пользователя, используя `dynamic-call`. Вместо этого, обычным способом было бы написать спициальную библиотеку склеивающую Guile-с-X11, имеющую глубокие знания как о Guile, так и о X11 и делающую все, что необходимо для обеспечения их взаимодействия. Затем эту склеивающую библиотеку можно было бы динамически связать с чистым интерпретатором Guile и активировать ее вызовом функции инициализации. Эта функция добаит все новые типы и примитивы к интерпретатору Guile, всё что она может предложить.

(Здесь предлагается другой, лучший способ: просто создать обертку `libX11` в Scheme используя динамический FFI. см. См. Раздел 6.21.6 [Dynamic FFI], страница 460, дополнительную информацию.)

Учитывая некоторый набор Си расширений для Guile, следующим логическим шагом является интеграция этих склеивающих библиотек в модульную систему Guile, чтобы вы могли загружать новые примитивы во время выполнения системы, также как вы можете загрузить новый код Scheme.

`load-extension` *lib init* [Scheme Procedure]
`scm_load_extension` (*lib, init*) [C Function]

Загружает и инициализирует расширение указанное LIB и INIT. Когда нет предварительно зарегистрированной функции для LIB/INIT, это эквивалентно

```
(dynamic-call INIT (dynamic-link LIB))
```

Когда есть предварительно зарегистрированная функция, это функция вызывается вместо указанной(?).

Обычно, нет предварительно зарегистрированной функции. Этот вариант существует только для ситуаций где динамическая компоновка недоступна или нежелательна. В этом случае вы статически связываете свою программу с нужной библиотекой и зарегистрируете ее функуию инициализации(init) сразу после инициализации Guile.

Что касается `dynamic-link`, *lib* не должен содержать никакого суффикса, такого как `.so` (см. Раздел 6.21.1 [Foreign Libraries], страница 447). Он также не должен содержать компонетов каталога. Библиотеки которые реализуют Расширения Guile должны быть помещены в обычные места для разделяемых библиотек. Мы рекомендуем использовать соглашение об именах `libguile-bla-blum` для расширения связанного с модулем `(bla blum)`.

Обычным способом использования расширения является написание небольшого файла Scheme который определяет модуль и загружает расширение в этот модуль. Когда модуль загружается автоматически, загружается и расширение. Например,

```
(define-module (bla blum))

(load-extension "libguile-bla-blum" "bla_init_blum")
```

6.21.3 Си Расширения

Наиболее интересным применением динамически связываемых библиотек вероятно является их использование для предоставления скомпилированных модулей кода(*compiled code modules*) для программ Scheme. Программирование на Scheme является очень веселым, но время от времени возникает потребоность написать некоторый низко уровневый Си код, чтобы сделать программирование на Scheme еще веселей.

Вы можете не только добавить эти новые примитивы в свой собственный модуль (см. предыдущий раздел), вы можете даже поместить их в разделяемую библиотеку, которая подсоединяется к запущеному образу Guile только тогда, когда она действительно необходима.

Пример, надеюсь, все разяснит. Предположим, мы хотим чтобы сделать доступной функцию Бесселя(Bessel) библиотеки Си для Scheme в модуле '`(math bessel)`'. Первое что нам необходимо сделать это написать соответствующий код клея, чтобы

преобразовать аргументы и возвращаемые значения функций от Scheme в Си и обратно. Кроме того, нам нужна функция которая добавит их к набору примитивов Guile. Поскольку это всего лишь пример, мы будем реализовывать его лишь для функции `j0`.

```
#include <math.h>
#include <libguile.h>

SCM
j0_wrapper (SCM x)
{
  return scm_from_double (j0 (scm_to_double (x, "j0")));
}

void
init_math_bessel ()
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
}
```

Мы уже можем попытаться привести это в действие, вручную вызвав функции низкого уровня для выполнения динамического связвания. Исходный файл Си должен быть скомпилирован в разделяемую библиотеку. Вот как это делается в GNU/Linux, пожалуйста обратитесь к документации по `libtool` для того чтоыбы узнать как создавать переносимые динамически связываемые библиотеки.

```
gcc -shared -o libbessel.so -fPIC bessel.c
```

Теперь запустите Guile:

```
(define bessel-lib (dynamic-link "./libbessel.so"))
(dynamic-call "init_math_bessel" bessel-lib)
(j0 2)
⇒ 0.223890779141236
```

Имя файла `./libbessel.so` должно указывать на разделяемую библиотеку созданную с помощью команды `gcc` выше, конечно. Вторая строка взаимодействия с Guile вызовет функцию `init_math_bessel` которая в свою очередь зарегистрирует Си функцию `j0_wrapper` в интерпретаторе Guile под именем `j0`. Эта функция становиться сразу доступной и мы можем вызвать ее из Scheme.

Всело, не так ли? Но мы только на полпути. Это то, что, `apropos` говорит о `j0`:

```
(apropos "j0")
⊣ (guile-user): j0     #<primitive-procedure j0>
```

Как вы можете видеть, `j0` содержиться в корневом модуле, где и все остальные примитивы Guile, такие как `display`, и т.д. В общем, примитив помещается в любой модуль являющийся текущим(*current module*) в момент вызова `scm_c_define_gsubr`.

Скомпилированный модуль должен иметь специально именованную функцию инициализации (*module init function*. Guile знает об этом специальном имени и вызовет эту функцию автоматически после связывания(linked) с разделяемой библиотекой. В нашем примере, мы заменим `init_math_bessel` следующим кодом в `bessel.c`:

```
void
init_math_bessel (void *unused)
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
  scm_c_export ("j0", NULL);
```

```
}

void
scm_init_math_bessel_module ()
{
  scm_c_define_module ("math bessel", init_math_bessel, NULL);
}
```

Общий шаблон для имени функции инициализации модуля является: ‘scm_init_’, затем имя модуля, в котором отдельные иерархические компоненты объединяются символами подчеркивания, а затем следует ‘_module’.

После того как libbessel.so будет перестроен, нам нужно поместить разделяемую библиотеку в нужное место.

Как только модуль будет правильно установлен, его можно будет использовать следующим образом:

```
guile> (load-extension "./libbessel.so" "scm_init_math_bessel_module")
guile> (use-modules (math bessel))
guile> (j0 2)
0.223890779141236
guile> (apropos "j0")
⊣ (math bessel): j0      #<primitive-procedure j0>
```

Это то что надо!

6.21.4 Модули и Расширения

Новые примитивы которые вы добавляете в Guile с помощью scm_c_define_gsubr (см. Раздел 6.9.2 [Primitive Procedures], страница 264) или используя любые другие механизмы, помещаются в модуль, являющийся текущим на момент выполнения scm_c_define_gsubr. Например, расширения, загруженные из REPL, будут помещены в модуль (guile-user), если модуль REPL не был изменен.

Чтобы определить примитивы Си в определенном модуле, самый простой способ:

```
(define-module (foo bar))
(load-extension "foobar-c-code" "foo_bar_init")
```

При загрузке с помощью (use-modules (foo bar)), вызов load-extension ищет файл разделяемой библиотеки foobar-c-code.so (etc) в директории расширений Guile extensiondir, который обычно является подкаталогом из libdir. Например, если ваш libdir это /usr/lib, то директория для расширений extensiondir для Guile 2.2.*x* версии /usr/lib/guile/2.2/.

Путь к расширениям включает основную и второстепенную версии Guile (т.е. эффективную версию/“effective version”), поскольку Guile гарантирует совместимость в рамках эффективной версии. Это дает вам возможность инсталировать различные версии одного и того же расширения для разных версий Guile.

Если расширение не найдено в extensiondir, Guile также будет искать его в стандартных местах размещения библиотек, таких как /usr/lib или /usr/local/lib. Однако предпочтительно убрать ваше расширение от туда, чтобы предотвратить непреднамеренное вмешательство в другие динамически связываемые библиотеки Си.

Если кто-то устанавливает ваш модуль в нестандартное местоположение, тогда ваш объектный файл не будет найден. Вы можете решить эту проблему добавив место установки в файл foo/bar.scm. Это удобно для пользователя, а также гарантирует,

что предполагаемый объект прочтется, даже если старые или более новые версии находятся в пути загрузки.

Обычный способ указать место установки с префиксом(`prefix`) на этапе конфигурации(configure), для команды '`./configure prefix=/opt`' результирующие библиотечные файлы будут размещены: `/opt/lib/foobar-c-code.so`. При использовании Autoconf (см. Раздел "Introduction" в *The GNU Autoconf Manual*), расположение библиотеки находиться в переменной `libdir`. Его значение предполагается видно в команде `make`, и может быть подставлено в исходный файл, например `foo.scm.in`

```
(define-module (foo bar))
(load-extension "XXextensiondirXX/foobar-c-code" "foo_bar_init")
```

в следующем файле `Makefile`, используется команда `sed` (см. Раздел "Introduction" в *SED*),

```
foo.scm: foo.scm.in
        sed 's|XXextensiondirXX|$(libdir)/guile/2.2|' <foo.scm.in >foo.scm
```

Фактический шаблон `XXextensiondirXX` выбран произвольно, лишь бы небыло совпадений с другими выражениями в файле. Если несколько модулей нуждаются в значении, его может быть проще создать в файле `foo/config.scm` определяющим местоположение `extensiondir` и используемым по мере необходимости.

```
(define-module (foo config))
(define-public foo-config-extensiondir "XXextensiondirXX"")
```

Такой файл может содержать и другие местоположения, например каталог для вспомогательных файлов данных, или `localedir` если имеет свой собственный каталог сообщений `gettext`, см (см. Раздел 6.25 [Internationalization], страница 486).

Следует отметить, что все вышеперечисленное требует, чтобы код Scheme был найден в пути загрузки `%load-path` (см. Раздел 6.18.7 [Load Paths], страница 414). В настоящее время он задается системным администратором или каждым пользователем отдельно когда модули Guile устанавливаются в нестандартные места. Но достигнув кода Scheme, уже этот код должен заботится о том, чтобы найти любой из своих файлов и т.д.

6.21.5 Внешние Указатели

В предыдущих разделах показано, как Guile может быть расширен во время выполнения загрузкой скомпилированных Си расширений. Этот подход всегда хорош, но было бы неплохо, если бы у нас небыло необходимости вообще писать на Си? В этом разделе рассматривается проблема доступа к значениям Си из Scheme, а в следующей обсуждаются Си функции.

6.21.5.1 Внешние Типы

Первое несоответствие, которое наблюдается между Си и Scheme заключается в том, что в Си расположение хранилищ(переменные) является типизированным, но в Scheme типы связаны со значениями, а не переменными. См. Раздел 3.1.2 [Values and Variables], страница 16.

Таким образом, при описании функции Си или структуры Си, чтобы к ней можно было получить доступ из Scheme, типы данных параметров или полей должны передаваться явно.

Эти "значения типов Си" могут быть построены с использованием констант и процедур из модуля `(system foreign)`, который может быть загружен следующим образом:

```
(use-modules (system foreign))
```

`(system foreign)` экспортирует ряд значений, выражающих основные Си типы:

`int8` [Scheme Variable]
`uint8` [Scheme Variable]
`uint16` [Scheme Variable]
`int16` [Scheme Variable]
`uint32` [Scheme Variable]
`int32` [Scheme Variable]
`uint64` [Scheme Variable]
`int64` [Scheme Variable]
`float` [Scheme Variable]
`double` [Scheme Variable]

Эти значения представляют собой числовые типы Си указанных размеров и типов.

Кроме того, есть некоторые удобные привязки для указания типов размер которых зависит от платформы:

`int` [Scheme Variable]
`unsigned-int` [Scheme Variable]
`long` [Scheme Variable]
`unsigned-long` [Scheme Variable]
`short` [Scheme Variable]
`unsigned-short` [Scheme Variable]
`size_t` [Scheme Variable]
`ssize_t` [Scheme Variable]
`ptrdiff_t` [Scheme Variable]
`intptr_t` [Scheme Variable]
`uintptr_t` [Scheme Variable]

Значения, экспортируемые модулем `(system foreign)`, представляющие числовые типы Си. Например, `long` может быть `equal?`(равным) `int64` на 64-битной платформе.

`void` [Scheme Variable]

Тип `void`. Его можно использовать в качестве первого аргумента для `pointer->procedure` создающей обертку Си функции которая ничего не возвращает.

Кроме того, знак `*` используется как условное обозначение разименовывающее указатель. Процедуры подробно описаны в следующих разделах, такие как `pointer->procedure`, принимают его как дескриптор(определитель) типа.

6.21.5.2 Внешние Переменные

Указатели на переменные в текущем адресном пространстве можно искать динамически используя `dynamic-pointer`.

`dynamic-pointer` *name dobj* [Scheme Procedure]
`scm_dynamic_pointer` (*name*, *dobj*) [C Function]

Возвращает "обернутый указатель(wrapped pointer)" для символа *name* в разделяемом объекте, на который ссылается *dobj*. Возвращаемый указатель указывает на объект Си.

Независимо от того, добавляет ли ваш Си компилятор подчеркивание '_' к глобальным именам в программе, вы **НЕ** должны включать включать это подчеркивание в *name*, так как оно будет автоматически добавлено при необходимости.

Например, в настоящее время Guile имеет переменную `scm_numptob` как часть своего API. Она объявлена в как Си `long`. Итак, чтобы создать дескриптор, указывающий на это внешнее значени, мы делаем следующее:

```
(use-modules (system foreign))
(define numptob (dynamic-pointer "scm_numptob" (dynamic-link)))
numptob
⇒ #<pointer 0x7fb35b1b4688>
```

(В следующем разделе рассматривается способ разыменовывания указателей(т.е получения значений).)

Значение возвращаемое `dynamic-pointer` представляет собой обертку (оболочку) в Scheme для указателя Си.

`pointer-address` *pointer* [Scheme Procedure]
`scm_pointer_address` (*pointer*) [C Function]

Возвращает числовое значение указателя *pointer*.

```
(pointer-address numptob)
⇒ 139984413364296 ; YMMV
```

`make-pointer` *address* **[***finalizer***]** [Scheme Procedure]

Возвращает объект внешний указатель указывающий на адрес *address*. Если финализатор(*finalizer*) передан, он должен быть указателем на Си функцию с одним аргументом, которая будет вызываться, когда объект указатель становится недостижымым в Scheme(вызывается сборщиком мусора/GC).

`pointer?` *obj* [Scheme Procedure]

Возвращает `#t` если *obj* является объектом-указателем, `#f` в противном случае.

`%null-pointer` [Scheme Variable]

Внешний указатель значение которого равно 0.

`null-pointer?` *pointer* [Scheme Procedure]

Возвращает `#t` если указатель *pointer* является нулевым указателем, `#f` в противном случае.

С целью передачи значений SCM непосредственно во внешние функции и разрешения им возвращать SCM значения, Guile также поддерживает некоторые небезопасные операторы приведения(указания) типов.

`scm->pointer` *scm* [Scheme Procedure]

Возвращает объект внешний указатель с адресом объекта `object-address` который имеет *scm*.

`pointer->scm` *pointer* [Scheme Procedure]
Небезопасное преобразование указателя *pointer* в объект Scheme. Скрестите пальцы!

Иногда ван надо предоставить Си расширениям доступ к динамическому FFI. В этот момент имена путаются, поскольку указатель "pointer" может ссылаться на объект `SCM` который оборачивает указатель, или значение `void*`. Мы попытаемся использовать "объект-указатель(pointer object)" для обозначения объекта Scheme, и "значение-указатель(pointer value)" для обозначения значений `void *`.

`SCM scm_from_pointer` (*void *ptr, void* (**finalizer*) (*void**)) [C Function]
Создает объект-указатель из значения указателя(pointer).

Если финализатор(*finalizer*) не нулевой, Guile организует его вызов по значению указателя в некторой точке после того, как объект-указатель станет собираемым(мусором).

`void* scm_to_pointer` (*SCM obj*) [C Function]
Распаковывает значение указателя из объекта-указателя.

6.21.5.3 Указатели типа Void и Байтовый Доступ

Обернутые указатели являются нетипизированными, поэтому они по сущетсву эквивалентны Си укзателям на `void`. Как и в Си, в Scheme область памяти на которую указывает указатель, может быть доступна на уровне байта. Это достигается использованием байтовых векторов *bytevectors* (см. Раздел 6.6.12 [Bytevectors], страница 205). Модуль `(rnrs bytevectors)` содержит процедуры, которые могут использоваться для преобразования последовательностей байтов в объекты Scheme, такие как строки(string), числа с плавающей запятой(floating point) или целые числа(integers).

`pointer->bytevector` *pointer len* [*offset* [*uvec_type*]] [Scheme Procedure]
`scm_pointer_to_bytevector` (*pointer, len, offset, uvec_type*) [C Function]
Возвращает байт-вектор(bytevector) наложенный на некоторое число *len* байтов начинающееся с адреса указываемого *pointer*.

Пользователь может указать альтернативную имеющейся по умолчанию интерпретацию для памяти путем передачи аргумента *uvec_type*, чтобы указать, что память представляет собой массив элементов этого типа. *uvec_type* должен быть чем-то вроде `array-type`, например `f32` или `s16`.

Когда передается смещение(*offset*) оно определяет смещение в байтах относительно указателя *pointer* на регион памяти возвращаемого конструктором bytevector.

Изменение возвращаемого bytevector изменяет указываемую *pointer* память, поэтому пристегите ремни безопасности.

`bytevector->pointer` *bv* [*offset*] [Scheme Procedure]
`scm_bytevector_to_pointer` (*bv, offset*) [C Function]
Возвращает указатель pointer налагающийся на память, на которую указывает *bv* или смещение *offset* в байтах после *bv* когда передается *offset*.

В дополнении к этим примитивам, доступны удобные процедуры:

`dereference-pointer` *pointer* [Scheme Procedure]
Предполагая что указатель *pointer* указывает на область памяти, которая содержит указатель, возвращает этот указатель.

`string->pointer` *string* [*encoding*] [Scheme Procedure]
Возвращает внешний указатель на копию строки заканчивающуюся нулем(си строку) *string* в данной кодировке *encoding*, по умолчанию это текущая кодировка локали. Си строка освобождается, когда возвращаемый внешний указатель становиться недоступным.

Это Scheme Это эквивалент вызова `scm_to_stringn`.

`pointer->string` *pointer* [*length*] [*encoding*] [Scheme Procedure]
Возвращает строку, представляющую Си строку, на которую указывает by *pointer*. Если *length* опущена или `-1`, предполагается что строка является (nul-terminated), т.е оканчивается нулем. В противном случае *length* это количество байтов памяти, на которые указывает указатель *pointer*. Предполагается, что Си строка находиться в заданной кодировке *encoding*, по умолчанию это текущая кодировка локали.

Это Scheme эквивалент функции `scm_from_stringn`.

Большинство объектно-ориентированных библиотек Си используют укзатели на конкретные структуры данных для идентификации объектов. В таких случаях полезно использовать различные типы указателей как непересекающиеся типы Scheme. Макрос `define-wrapped-pointer-type` упрощает это.

`define-wrapped-pointer-type` *type-name pred wrap unwrap print* [Scheme Syntax]
Определяет вспомогательные процедуры для обертывания указываемых внешних объектов в объекты Scheme с помощью непересекающихся типов. В частности этот макрос определяет:

- *pred*, предикат для нового типа Scheme;
- *wrap*, процедуру которая принимает указатель на объект и возвращает объект который удовлетворяет *pred*;
- *unwrap*, процедуру которая делает обратное *wrap* преобразование.

wrap сохраняет идентификацию указателя, для двух объектов указателей *p1* и *p2* которые `equal?`, `(eq? (`*wrap p1*`) (`*wrap p2*`))` ⇒ `#t`.

Наконец, *print* должна именовать(указывать) пользовательскую процедуру для печати таких объектов. Процедуре передается обернутый объект и порт для записи.

Например, предположим что мы создаем обертку для библиотеки Си, которая определяет тип `bottle_t`, и функции которым могут быть переданы указатели не этот тип `bottle_t *`, чтобы они могли манипулировать им. Мы можем записать:

```
(define-wrapped-pointer-type bottle
  bottle?
  wrap-bottle unwrap-bottle
  (lambda (b p)
```

```
    (format p "#<bottle of ~a ~x>"
            (bottle-contents b)
            (pointer-address (unwrap-bottle b)))))

(define grab-bottle
  ;; Wrapper for `bottle_t *grab (void)'.
  (let ((grab (pointer->procedure '*
                                  (dynamic-func "grab_bottle" libbottle)
                                  '())))
    (lambda ()
      "Return a new bottle."
      (wrap-bottle (grab)))))

(define bottle-contents
  ;; Wrapper for `const char *bottle_contents (bottle_t *)'.
  (let ((contents (pointer->procedure '*
                                      (dynamic-func "bottle_contents"
                                                    libbottle)
                                      '(*))))
    (lambda (b)
      "Return the contents of B."
      (pointer->string (contents (unwrap-bottle b))))))

(write (grab-bottle))
⇒ #<bottle of Château Haut-Brion 803d36>
```

В этом примере, `grab-bottle` гарантированно возвращает подлинный объект `bottle` удовлетворяющий предикату `bottle?`. Аналогичным образом, `bottle-contents` возвращает ошибку когда ее аргумент не является подлинным объектом `bottle`.

Возвращаясь к приведенному выше примеру с `scm_numptob`, мы можем прочитать его значение как Си длинное целое(`long` integer):

```
(use-modules (rnrs bytevectors))

(bytevector-uint-ref (pointer->bytevector numptob (sizeof long))
                     0 (native-endianness)
                     (sizeof long))
⇒ 8
```

Если бы мы хотели повредить внутреннее состояние Guile, мы могли бы установить `scm_numptob` в другое значение; но мы этого делать недолжны, потому что эта переменная не предназначена для установки(присваивания). Действительно, этот момент применяется широко: Си API является опасным местом. Не только установка значения может вызвать крах вашей программы, простой доступ к данным, на которые указывает "висячий" указатель или аналогичное действие может оказаться стольже катастрофическим.

6.21.5.4 Внешние Структуры

Наконец, последнее замечание по внешним значениям, прежде чем перейти к фактическим вызвовам внешних функций. Иногда вам приходиться иметь дело с Си структурами, что требует итерпретатции каждого элемента структуры в соответствии с его типом, смещением и выравниванием. У Guile есть несколько примитивова для поддержки этого.

`sizeof` *type* [Scheme Procedure]
`scm_sizeof` (*type*) [C Function]

Возвращает размер *type*, в байтах.

type должен быть допустимым Си типом, например `int`. Альтернативным *type* может быть символ `*`, в этом случае возвращается размер указателя. *type* также может быть списком типов, в этом случае возвращается размер структуры `struct` с обычной для ABI упаковкой(размещением элементов).

`alignof` *type* [Scheme Procedure]
`scm_alignof` (*type*) [C Function]

Возвращается выравнивание *type*, в байтах.

type должен быть допустимым Си типом, например `int`. Альтернативным *type* может быть символ `*`, в этом случае возвращается размер указателя. *type* также может быть списком типов, в этом случае возвращается размер структуры `struct` с обычной для ABI упаковкой(размещением элементов).

Guile также предоставляет некоторые удобные методы для упаковки и распаковки внешних указателей обертывающих Си структуры.

`make-c-struct` *types vals* [Scheme Procedure]

Создает внешний указатель на Си структуру содержащую значения *vals* с типами `types`.

vals и `types` должны быть списками одинаковой длины.

`parse-c-struct` *foreign types* [Scheme Procedure]

Разбирает внешний указатель на Си структуру, возвращая список значений.

`types` должен быть списком Си типов.

Например, создадим и разберем эквивалент структуры `struct { int64_t a; uint8_t b; }`:

```
(parse-c-struct (make-c-struct (list int64 uint8)
                               (list 300 43))
                (list int64 uint8))
⇒ (300 43)
```

Пока у Guile есть только удобные процедуры поддержки упакованных структур поддерживающих соглашение. Но учитывая процедуры `bytevector->pointer` и `pointer->bytevector`, можно создавать и разбирать плотно упакованные структуры и объединения(unions) в ручную. См. код для `(system foreign)` для ознакомления с деталями.

6.21.6 Динамический FFI

Конечно, земля Си это не только существительные и не глаголы: есть также функции, и Guile позволяет вам их вызывать.

`pointer->procedure` *return_type func_ptr arg_types* [*#:return-errno?=#f*] [Scheme Procedure]

`scm_pointer_to_procedure` (*return_type, func_ptr, arg_types*) [C Function]

`scm_pointer_to_procedure_with_errno` (*return_type, func_ptr, arg_types*) [C Function]

Создает внешнюю функцию.

Данный внешний свободный/неопределенный(void) указатель *func_ptr*, является ее аргументом, как и типы аргументов *arg_types* и возвращаемый тип *return_type*, она возвращает процедуру, которая будет передавать аргументы внешней функции и возвращать соответствующее значение.

arg_types должен быть списком внешних типов. `return_type` должно быть внешним типом. См. Раздел 6.21.5.1 [Foreign Types], страница 453, для получения дополнительной информации о внешних типах.

Если *return-errno?* равно истине, или при вызове `scm_pointer_to_procedure_with_errno`, возвращается процедура возвращающая два значения, вторым значением является кодом ошибки `errno`.

Вот лучшее определение `(math bessel)`:

```
(define-module (math bessel)
  #:use-module (system foreign)
  #:export (j0))

(define libm (dynamic-link "libm"))

(define j0
  (pointer->procedure double
                      (dynamic-func "j0" libm)
                      (list double)))
```

Вот так! Никаких Си вызовов.

Числовые аргументы и возвращаемые значения из внешних функций представлены в виде значений Scheme. Например, `j0` в приведенном выше примере принимает в качестве аргумента число Scheme в качестве аргумента, и возвращает число Scheme.

Указатели могут быть переданы и возвращены из внешних функций. В этом случае тип аргумента или возвращаемого значения должен быть символом `*`, обозначающим указатель. Например, следующий код делает функцию `memcpy` доступным для Scheme:

```
(define memcpy
  (let ((this (dynamic-link)))
    (pointer->procedure '*
                        (dynamic-func "memcpy" this)
                        (list '* '* size_t))))
```

Чтобы вызвать `memcpy`, нужно передать ей внешние указатели:

```
(use-modules (rnrs bytevectors))
```

```
(define src-bits
  (u8-list->bytevector '(0 1 2 3 4 5 6 7)))
(define src
  (bytevector->pointer src-bits))
(define dest
  (bytevector->pointer (make-bytevector 16 0)))

(memcpy dest src (bytevector-length src-bits))

(bytevector->u8-list (pointer->bytevector dest 16))
⇒ (0 1 2 3 4 5 6 7 0 0 0 0 0 0 0 0)
```

Можно также передавать структуры как значения, передавая структуры как внешние указатели. См. Раздел 6.21.5.4 [Foreign Structs], страница 459, для получения дальнейшей информации о том как выразить структуру через типы и значения элементов структуры.

Аргументы “Out” передаются как внешние указатели. Памеять на которую указывают внешние указатели изменяется на месте(просто изменяется внешней функцией).

```
;; struct timeval {
;;      time_t      tv_sec;     /* seconds */
;;      suseconds_t tv_usec;    /* microseconds */
;; };
;; assuming fields are of type "long"

(define gettimeofday
  (let ((f (pointer->procedure
            int
            (dynamic-func "gettimeofday" (dynamic-link))
            (list '* '*)))
        (tv-type (list long long)))
    (lambda ()
      (let* ((timeval (make-c-struct tv-type (list 0 0)))
             (ret (f timeval %null-pointer)))
        (if (zero? ret)
            (apply values (parse-c-struct timeval tv-type))
            (error "gettimeofday returned an error" ret))))))

(gettimeofday)
⇒ 1270587589
⇒ 499553
```

Как вы можете видеть, этот интерфейс для внешних функций работает но очень низком уровне, очень опасном уровне[13].

[13] Весьма приветствуется вклад в Guile в на высоком уровне FFI.

FFI также может работать и в обратном направлении: создании процедур Scheme вызываемых из Си. Это позволяет использовать процедуры Scheme как “обратные вызовы(callbacks)”, ожидаемые Си функцией.

`procedure->pointer` *return-type proc arg-types* [Scheme Procedure]
`scm_procedure_to_pointer` (*return_type, proc, arg_types*) [C Function]

Возвращает указатель на функцию Си возвращающую значение типа *return-type* и принимающую аргументы типов *arg-types* (это список) и ведет себя как посредник к процедуре *proc*. Таким образом арность(размерность) *proc* поддерживает типы аргументов *arg-types* и возвращает тип который должен соответствовать *return-type*.

В качестве примера можно привести функцию сортировки массива `qsort` библиотеки Си доступной для Scheme (см. Раздел “Array Sort Function” в *The GNU C Library Reference Manual*):

```
(define qsort!
  (let ((qsort (pointer->procedure void
                                   (dynamic-func "qsort"
                                                 (dynamic-link))
                                   (list '* size_t size_t '*))))
    (lambda (bv compare)
      ;; Sort bytevector BV in-place according to comparison
      ;; procedure COMPARE.
      (let ((ptr (procedure->pointer int
                                     (lambda (x y)
                                       ;; X and Y are pointers so,
                                       ;; for convenience, dereference
                                       ;; them before calling COMPARE.
                                       (compare (dereference-uint8* x)
                                                (dereference-uint8* y)))
                                     (list '* '*))))
        (qsort (bytevector->pointer bv)
               (bytevector-length bv) 1 ;; we're sorting bytes
               ptr)))))

(define (dereference-uint8* ptr)
  ;; Helper function: dereference the byte pointed to by PTR.
  (let ((b (pointer->bytevector ptr 1)))
    (bytevector-u8-ref b 0)))

(define bv
  ;; An unsorted array of bytes.
  (u8-list->bytevector '(7 1 127 3 5 4 77 2 9 0)))

;; Sort BV.
(qsort! bv (lambda (x y) (- x y)))
```

```
;; Let's see what the sorted array looks like:
(bytevector->u8-list bv)
⇒ (0 1 2 3 4 5 7 9 77 127)
```

И вуаля!

Обратите внимание, что `procedure->pointer` не поддерживат(и не определена) на нескольких экзотических архитектурах. Таким образом, пользовательскому коду возможно потребуется проверять определенали эта процедура `(defined? 'procedure->pointer)`. Тем не менее, она доступна на многих архитектурах, включая (как libffi 3.0.9) x86, ia64, SPARC, PowerPC, ARM, и MIPS.

6.22 Threads, Mutexes, Asyncs and Dynamic Roots

6.22.1 Threads

Guile supports POSIX threads, unless it was configured with `--without-threads` or the host lacks POSIX thread support. When thread support is available, the `threads` feature is provided (см. Раздел 6.23.2.1 [Feature Manipulation], страница 479).

The procedures below manipulate Guile threads, which are wrappers around the system's POSIX threads. For application-level parallelism, using higher-level constructs, such as futures, is recommended (см. Раздел 6.22.7 [Futures], страница 474).

To use these facilities, load the `(ice-9 threads)` module.

```
(use-modules (ice-9 threads))
```

`all-threads` [Scheme Procedure]
`scm_all_threads ()` [C Function]
Return a list of all threads.

`current-thread` [Scheme Procedure]
`scm_current_thread ()` [C Function]
Return the thread that called this function.

`call-with-new-thread` *thunk* **[***handler***]** [Scheme Procedure]
Call `thunk` in a new thread and with a new dynamic state, returning the new thread. The procedure *thunk* is called via `with-continuation-barrier`.

When *handler* is specified, then *thunk* is called from within a `catch` with tag `#t` that has *handler* as its handler. This catch is established inside the continuation barrier.

Once *thunk* or *handler* returns, the return value is made the *exit value* of the thread and the thread is terminated.

`SCM scm_spawn_thread` (*scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data*) [C Function]
Call *body* in a new thread, passing it *body_data*, returning the new thread. The function *body* is called via `scm_c_with_continuation_barrier`.

When *handler* is non-`NULL`, *body* is called via `scm_internal_catch` with tag `SCM_BOOL_T` that has *handler* and *handler_data* as the handler and its data. This catch is established inside the continuation barrier.

Once *body* or *handler* returns, the return value is made the *exit value* of the thread and the thread is terminated.

`thread?` *obj* [Scheme Procedure]
`scm_thread_p` (*obj*) [C Function]
Return `#t` ff *obj* is a thread; otherwise, return `#f`.

`join-thread` *thread* [*timeout* [*timeoutval*]] [Scheme Procedure]
`scm_join_thread` (*thread*) [C Function]
`scm_join_thread_timed` (*thread, timeout, timeoutval*) [C Function]
Wait for *thread* to terminate and return its exit value. Only threads that were created with `call-with-new-thread` or `scm_spawn_thread` can be joinable; attempting to join a foreign thread will raise an error.

When *timeout* is given, it specifies a point in time where the waiting should be aborted. It can be either an integer as returned by `current-time` or a pair as returned by `gettimeofday`. When the waiting is aborted, *timeoutval* is returned (if it is specified; `#f` is returned otherwise).

`thread-exited?` *thread* [Scheme Procedure]
`scm_thread_exited_p` (*thread*) [C Function]
Return `#t` if *thread* has exited, or `#f` otherwise.

`yield` [Scheme Procedure]
`scm_yield` (*thread*) [C Function]
If one or more threads are waiting to execute, calling yield forces an immediate context switch to one of them. Otherwise, yield has no effect.

`cancel-thread` *thread . values* [Scheme Procedure]
`scm_cancel_thread` (*thread*) [C Function]
Asynchronously interrupt *thread* and ask it to terminate. `dynamic-wind` post thunks will run, but throw handlers will not. If *thread* has already terminated or been signaled to terminate, this function is a no-op. Calling `join-thread` on the thread will return the given *values*, if the cancel succeeded.

Under the hood, thread cancellation uses `system-async-mark` and `abort-to-prompt`. См. Раздел 6.22.3 [Asyncs], страница 466, for more on asynchronous interrupts.

`make-thread` *proc arg ...* [macro]
Apply *proc* to *arg ...* in a new thread formed by `call-with-new-thread` using a default error handler that display the error to the current error port. The *arg ...* expressions are evaluated in the new thread.

`begin-thread` *expr1 expr2 ...* [macro]
Evaluate forms *expr1 expr2 ...* in a new thread formed by `call-with-new-thread` using a default error handler that display the error to the current error port.

One often wants to limit the number of threads running to be proportional to the number of available processors. These interfaces are therefore exported by (ice-9 threads) as well.

`total-processor-count` [Scheme Procedure]
`scm_total_processor_count` () [C Function]
Return the total number of processors of the machine, which is guaranteed to be at least 1. A "processor" here is a thread execution unit, which can be either:

- an execution core in a (possibly multi-core) chip, in a (possibly multi- chip) module, in a single computer, or
- a thread execution unit inside a core in the case of *hyper-threaded* CPUs.

Which of the two definitions is used, is unspecified.

`current-processor-count` [Scheme Procedure]
`scm_current_processor_count ()` [C Function]
Like `total-processor-count`, but return the number of processors available to the current process. See `setaffinity` and `getaffinity` for more information.

6.22.2 Thread-Local Variables

Sometimes you want to establish a variable binding that is only valid for a given thread: a "thread-local variable".

You would think that fluids or parameters would be Guile's answer for thread-local variables, since establishing a new fluid binding doesn't affect bindings in other threads. См. Раздел 6.13.11 [Fluids and Dynamic States], страница 343, or См. Раздел 6.13.12 [Parameters], страница 347. However, new threads inherit the fluid bindings that were in place in their creator threads. In this way, a binding established using a fluid (or a parameter) in a thread can escape to other threads, which might not be what you want. Or, it might escape via explicit reification via `current-dynamic-state`.

Of course, this dynamic scoping might be exactly what you want; that's why fluids and parameters work this way, and is what you want for for many common parameters such as the current input and output ports, the current locale conversion parameters, and the like. Perhaps this is the case for most parameters, even. If your use case for thread-local bindings comes from a desire to isolate a binding from its setting in unrelated threads, then fluids and parameters apply nicely.

On the other hand, if your use case is to prevent concurrent access to a value from multiple threads, then using vanilla fluids or parameters is not appropriate. For this purpose, Guile has *thread-local fluids*. A fluid created with `make-thread-local-fluid` won't be captured by `current-dynamic-state` and won't be propagated to new threads.

`make-thread-local-fluid` [*dflt*] [Scheme Procedure]
`scm_make_thread_local_fluid` (*dflt*) [C Function]
Return a newly created fluid, whose initial value is *dflt*, or `#f` if *dflt* is not given. Unlike fluids made with `make-fluid`, thread local fluids are not captured by `make-dynamic-state`. Similarly, a newly spawned child thread does not inherit thread-local fluid values from the parent thread.

`fluid-thread-local?` *fluid* [Scheme Procedure]
`scm_fluid_thread_local_p` (*fluid*) [C Function]
Return `#t` if the fluid *fluid* is is thread-local, or `#f` otherwise.

For example:

```
(define %thread-local (make-thread-local-fluid))

(with-fluids ((%thread-local (compute-data)))
```

```
... (fluid-ref %thread-local) ...)
```

You can also make a thread-local parameter out of a thread-local fluid using the normal `fluid->parameter`:

```
(define param (fluid->parameter (make-thread-local-fluid)))

(parameterize ((param (compute-data)))
  ... (param) ...)
```

6.22.3 Asynchronous Interrupts

Every Guile thread can be interrupted. Threads running Guile code will periodically check if there are pending interrupts and run them if necessary. To interrupt a thread, call `system-async-mark` on that thread.

`system-async-mark` *proc* [*thread*] [Scheme Procedure]
`scm_system_async_mark` (*proc*) [C Function]
`scm_system_async_mark_for_thread` (*proc*, *thread*) [C Function]

Enqueue *proc* (a procedure with zero arguments) for future execution in *thread*. When *proc* has already been enqueued for *thread* but has not been executed yet, this call has no effect. When *thread* is omitted, the thread that called `system-async-mark` is used.

Note that `scm_system_async_mark_for_thread` is not "async-signal-safe" and so cannot be called from a C signal handler. (Indeed in general, `libguile` functions are not safe to call from C signal handlers.)

Though an interrupt procedure can have any side effect permitted to Guile code, asynchronous interrupts are generally used either for profiling or for prematurely cancelling a computation. The former case is mostly transparent to the program being run, by design, but the latter case can introduce bugs. Like finalizers (см. Раздел 5.5.4 [Foreign Object Memory Management], страница 82), asynchronous interrupts introduce concurrency in a program. An asyncronous interrupt can run in the middle of some mutex-protected operation, for example, and potentially corrupt the program's state.

If some bit of Guile code needs to temporarily inhibit interrupts, it can use `call-with-blocked-asyncs`. This function works by temporarily increasing the *async blocking level* of the current thread while a given procedure is running. The blocking level starts out at zero, and whenever a safe point is reached, a blocking level greater than zero will prevent the execution of queued asyncs.

Analogously, the procedure `call-with-unblocked-asyncs` will temporarily decrease the blocking level of the current thread. You can use it when you want to disable asyncs by default and only allow them temporarily.

In addition to the C versions of `call-with-blocked-asyncs` and `call-with-unblocked-asyncs`, C code can use `scm_dynwind_block_asyncs` and `scm_dynwind_unblock_asyncs` inside a *dynamic context* (см. Раздел 6.13.10 [Dynamic Wind], страница 339) to block or unblock asyncs temporarily.

`call-with-blocked-asyncs` *proc* [Scheme Procedure]
`scm_call_with_blocked_asyncs` (*proc*) [C Function]
Call *proc* and block the execution of asyncs by one level for the current thread while it is running. Return the value returned by *proc*. For the first two variants, call *proc* with no arguments; for the third, call it with *data*.

`void * scm_c_call_with_blocked_asyncs` (*void * (*proc) (void *data), void *data*) [C Function]
The same but with a C function *proc* instead of a Scheme thunk.

`call-with-unblocked-asyncs` *proc* [Scheme Procedure]
`scm_call_with_unblocked_asyncs` (*proc*) [C Function]
Call *proc* and unblock the execution of asyncs by one level for the current thread while it is running. Return the value returned by *proc*. For the first two variants, call *proc* with no arguments; for the third, call it with *data*.

`void * scm_c_call_with_unblocked_asyncs` (*void *(*proc) (void *data), void *data*) [C Function]
The same but with a C function *proc* instead of a Scheme thunk.

`void scm_dynwind_block_asyncs` () [C Function]
During the current dynwind context, increase the blocking of asyncs by one level. This function must be used inside a pair of calls to `scm_dynwind_begin` and `scm_dynwind_end` (см. Раздел 6.13.10 [Dynamic Wind], страница 339).

`void scm_dynwind_unblock_asyncs` () [C Function]
During the current dynwind context, decrease the blocking of asyncs by one level. This function must be used inside a pair of calls to `scm_dynwind_begin` and `scm_dynwind_end` (см. Раздел 6.13.10 [Dynamic Wind], страница 339).

Sometimes you want to interrupt a thread that might be waiting for something to happen, for example on a file descriptor or a condition variable. In that case you can inform Guile of how to interrupt that wait using the following procedures:

`int scm_c_prepare_to_wait_on_fd` (*int fd*) [C Function]
Inform Guile that the current thread is about to sleep, and that if an asynchronous interrupt is signalled on this thread, Guile should wake up the thread by writing a zero byte to *fd*. Returns zero if the prepare succeeded, or nonzero if the thread already has a pending async and that it should avoid waiting.

`int scm_c_prepare_to_wait_on_cond` (*scm_i_pthread_mutex_t *mutex, scm_i_pthread_cond_t *cond*) [C Function]
Inform Guile that the current thread is about to sleep, and that if an asynchronous interrupt is signalled on this thread, Guile should wake up the thread by acquiring *mutex* and signalling *cond*. The caller must already hold *mutex* and only drop it as part of the `pthread_cond_wait` call. Returns zero if the prepare succeeded, or nonzero if the thread already has a pending async and that it should avoid waiting.

`void scm_c_wait_finished` (*void*) [C Function]
Inform Guile that the current thread has finished waiting, and that asynchronous interrupts no longer need any special wakeup action; the current thread will periodically poll its internal queue instead.

Guile's own interface to `sleep`, `wait-condition-variable`, `select`, and so on all call the above routines as appropriate.

Finally, note that threads can also be interrupted via POSIX signals. См. Раздел 7.2.8 [Signals], страница 548. As an implementation detail, signal handlers will effectively call `system-async-mark` in a signal-safe way, eventually running the signal handler using the same async mechanism. In this way you can temporarily inhibit signal handlers from running using the above interfaces.

6.22.4 Atomics

When accessing data in parallel from multiple threads, updates made by one thread are not generally guaranteed to be visible by another thread. It could be that your hardware requires special instructions to be emitted to propagate a change from one CPU core to another. Or, it could be that your hardware updates values with a sequence of instructions, and a parallel thread could see a value that is in the process of being updated but not fully updated.

Atomic references solve this problem. Atomics are a standard, primitive facility to allow for concurrent access and update of mutable variables from multiple threads with guaranteed forward-progress and well-defined intermediate states.

Atomic references serve not only as a hardware memory barrier but also as a compiler barrier. Normally a compiler might choose to reorder or elide certain memory accesses due to optimizations like common subexpression elimination. Atomic accesses however will not be reordered relative to each other, and normal memory accesses will not be reordered across atomic accesses.

As an implementation detail, currently all atomic accesses and updates use the sequential consistency memory model from C11. We may relax this in the future to the acquire/release semantics, which still issues a memory barrier so that non-atomic updates are not reordered across atomic accesses or updates.

To use Guile's atomic operations, load the `(ice-9 atomic)` module:

```
(use-modules (ice-9 atomic))
```

`make-atomic-box` *init* [Scheme Procedure]
Return an atomic box initialized to value *init*.

`atomic-box?` *obj* [Scheme Procedure]
Return `#t` if *obj* is an atomic-box object, else return `#f`.

`atomic-box-ref` *box* [Scheme Procedure]
Fetch the value stored in the atomic box *box* and return it.

`atomic-box-set!` *box val* [Scheme Procedure]
Store *val* into the atomic box *box*.

`atomic-box-swap!` *box val* [Scheme Procedure]
Store *val* into the atomic box *box*, and return the value that was previously stored in the box.

`atomic-box-compare-and-swap!` *box expected desired* [Scheme Procedure]
If the value of the atomic box *box* is the same as, *expected* (in the sense of `eq?`), replace the contents of the box with *desired*. Otherwise does not update the box. Returns the previous value of the box in either case, so you can know if the swap worked by checking if the return value is `eq?` to *expected*.

6.22.5 Mutexes and Condition Variables

Mutexes are low-level primitives used to coordinate concurrent access to mutable data. Short for "mutual exclusion", the name "mutex" indicates that only one thread at a time can acquire access to data that is protected by a mutex – threads are excluded from accessing data at the same time. If one thread has locked a mutex, then another thread attempting to lock that same mutex will wait until the first thread is done.

Mutexes can be used to build robust multi-threaded programs that take advantage of multiple cores. However, they provide very low-level functionality and are somewhat dangerous; usually you end up wanting to acquire multiple mutexes at the same time to perform a multi-object access, but this can easily lead to deadlocks if the program is not carefully written. For example, if objects A and B are protected by associated mutexes M and N, respectively, then to access both of them then you need to acquire both mutexes. But what if one thread acquires M first and then N, at the same time that another thread acquires N them M? You can easily end up in a situation where one is waiting for the other.

There's no easy way around this problem on the language level. A function A that uses mutexes does not necessarily compose nicely with a function B that uses mutexes. For this reason we suggest using atomic variables when you can (см. Раздел 6.22.4 [Atomics], страница 468), as they do not have this problem.

Still, if you as a programmer are responsible for a whole system, then you can use mutexes as a primitive to provide safe concurrent abstractions to your users. (For example, given all locks in a system, if you establish an order such that M is consistently acquired before N, you can avoid the "deadly-embrace" deadlock described above. The problem is enumerating all mutexes and establishing this order from a system perspective.) Guile gives you the low-level facilities to build such systems.

In Guile there are additional considerations beyond the usual ones in other programming languages: non-local control flow and asynchronous interrupts. What happens if you hold a mutex, but somehow you cause an exception to be thrown? There is no one right answer. You might want to keep the mutex locked to prevent any other code from ever entering that critical section again. Or, your critical section might be fine if you unlock the mutex "on the way out", via a catch handler or `dynamic-wind`. См. Раздел 6.13.8.2 [Catch], страница 333, and См. Раздел 6.13.10 [Dynamic Wind], страница 339.

But if you arrange to unlock the mutex when leaving a dynamic extent via `dynamic-wind`, what to do if control re-enters that dynamic extent via a continuation invocation? Surely re-entering the dynamic extent without the lock is a bad idea, so there are two options on the table: either prevent re-entry via `with-continuation-barrier` or similar, or reacquire the lock in the entry thunk of a `dynamic-wind`.

You might think that because you don't use continuations, that you don't have to think about this, and you might be right. If you control the whole system, you can reason about continuation use globally. Or, if you know all code that can be called in a dynamic extent, and none of that code can call continuations, then you don't have to worry about re-entry, and you might not have to worry about early exit either.

However, do consider the possibility of asynchronous interrupts (см. Раздел 6.22.3 [Asyncs], страница 466). If the user interrupts your code interactively, that can cause an exception; or your thread might be cancelled, which does the same; or the user could be running your code under some pre-emptive system that periodically causes lightweight task switching. (Guile does not currently include such a system, but it's possible to implement as a library.) Probably you also want to defer asynchronous interrupt processing while you hold the mutex, and probably that also means that you should not hold the mutex for very long.

All of these additional Guile-specific considerations mean that from a system perspective, you would do well to avoid these hazards if you can by not requiring mutexes. Instead, work with immutable data that can be shared between threads without hazards, or use persistent data structures with atomic updates based on the atomic variable library (см. Раздел 6.22.4 [Atomics], страница 468).

There are three types of mutexes in Guile: "standard", "recursive", and "unowned".

Calling `make-mutex` with no arguments makes a standard mutex. A standard mutex can only be locked once. If you try to lock it again from the thread that locked it to begin with (the `"owner"` thread), it throws an error. It can only be unlocked from the thread that locked it in the first place.

Calling `make-mutex` with the symbol `recursive` as the argument, or calling `make-recursive-mutex`, will give you a recursive mutex. A recursive mutex can be locked multiple times by its owner. It then has to be unlocked the corresponding number of times, and like standard mutexes can only be unlocked by the owner thread.

Finally, calling `make-mutex` with the symbol `allow-external-unlock` creates an unowned mutex. An unowned mutex is like a standard mutex, except that it can be unlocked by any thread. A corollary of this behavior is that a thread's attempt to lock a mutex that it already owns will block instead of signalling an error, as it could be that some other thread unlocks the mutex, allowing the owner thread to proceed. This kind of mutex is a bit strange and is here for use by SRFI-18.

The mutex procedures in Guile can operate on all three kinds of mutexes.

To use these facilities, load the `(ice-9 threads)` module.

```
(use-modules (ice-9 threads))
```

`make-mutex` [*kind*] [Scheme Procedure]
`scm_make_mutex` () [C Function]
`scm_make_mutex_with_kind` (*SCM kind*) [C Function]

Return a new mutex. It will be a standard non-recursive mutex, unless the `recursive` symbol is passed as the optional *kind* argument, in which case it will be recursive. It's also possible to pass `unowned` for semantics tailored to SRFI-18's use case; see above for details.

`mutex?` *obj* [Scheme Procedure]
`scm_mutex_p` (*obj*) [C Function]
Return `#t` if *obj* is a mutex; otherwise, return `#f`.

`make-recursive-mutex` [Scheme Procedure]
`scm_make_recursive_mutex` () [C Function]
Create a new recursive mutex. It is initially unlocked. Calling this function is equivalent to calling `make-mutex` with the `recursive` kind.

`lock-mutex` *mutex* [*timeout*] [Scheme Procedure]
`scm_lock_mutex` (*mutex*) [C Function]
`scm_timed_lock_mutex` (*mutex*, *timeout*) [C Function]
Lock *mutex* and return `#t`. If the mutex is already locked, then block and return only when *mutex* has been acquired.

When *timeout* is given, it specifies a point in time where the waiting should be aborted. It can be either an integer as returned by `current-time` or a pair as returned by `gettimeofday`. When the waiting is aborted, `#f` is returned.

For standard mutexes (`make-mutex`), an error is signalled if the thread has itself already locked *mutex*.

For a recursive mutex (`make-recursive-mutex`), if the thread has itself already locked *mutex*, then a further `lock-mutex` call increments the lock count. An additional `unlock-mutex` will be required to finally release.

When an asynchronous interrupt (см. Раздел 6.22.3 [Asyncs], страница 466) is scheduled for a thread blocked in `lock-mutex`, Guile will interrupt the wait, run the interrupts, and then resume the wait.

`void scm_dynwind_lock_mutex` (*SCM mutex*) [C Function]
Arrange for *mutex* to be locked whenever the current dynwind context is entered and to be unlocked when it is exited.

`try-mutex` *mx* [Scheme Procedure]
`scm_try_mutex` (*mx*) [C Function]
Try to lock *mutex* and return `#t` if successful, or `#f` otherwise. This is like calling `lock-mutex` with an expired timeout.

`unlock-mutex` *mutex* [Scheme Procedure]
`scm_unlock_mutex` (*mutex*) [C Function]
Unlock *mutex*. An error is signalled if *mutex* is not locked.

"Standard" and "recursive" mutexes can only be unlocked by the thread that locked them; Guile detects this situation and signals an error. "Unowned" mutexes can be unlocked by any thread.

`mutex-owner` *mutex* [Scheme Procedure]
`scm_mutex_owner` (*mutex*) [C Function]
Return the current owner of *mutex*, in the form of a thread or `#f` (indicating no owner). Note that a mutex may be unowned but still locked.

`mutex-level` *mutex* [Scheme Procedure]
`scm_mutex_level` (*mutex*) [C Function]
Return the current lock level of *mutex*. If *mutex* is currently unlocked, this value will be 0; otherwise, it will be the number of times *mutex* has been recursively locked by its current owner.

`mutex-locked?` *mutex* [Scheme Procedure]
`scm_mutex_locked_p` (*mutex*) [C Function]
Return `#t` if *mutex* is locked, regardless of ownership; otherwise, return `#f`.

`make-condition-variable` [Scheme Procedure]
`scm_make_condition_variable` () [C Function]
Return a new condition variable.

`condition-variable?` *obj* [Scheme Procedure]
`scm_condition_variable_p` (*obj*) [C Function]
Return `#t` if *obj* is a condition variable; otherwise, return `#f`.

`wait-condition-variable` *condvar mutex* [*time*] [Scheme Procedure]
`scm_wait_condition_variable` (*condvar, mutex, time*) [C Function]
Wait until *condvar* has been signalled. While waiting, *mutex* is atomically unlocked (as with `unlock-mutex`) and is locked again when this function returns. When *time* is given, it specifies a point in time where the waiting should be aborted. It can be either a integer as returned by `current-time` or a pair as returned by `gettimeofday`. When the waiting is aborted, `#f` is returned. When the condition variable has in fact been signalled, `#t` is returned. The mutex is re-locked in any case before `wait-condition-variable` returns.

When an async is activated for a thread that is blocked in a call to `wait-condition-variable`, the waiting is interrupted, the mutex is locked, and the async is executed. When the async returns, the mutex is unlocked again and the waiting is resumed. When the thread block while re-acquiring the mutex, execution of asyncs is blocked.

`signal-condition-variable` *condvar* [Scheme Procedure]
`scm_signal_condition_variable` (*condvar*) [C Function]
Wake up one thread that is waiting for *condvar*.

`broadcast-condition-variable` *condvar* [Scheme Procedure]
`scm_broadcast_condition_variable` (*condvar*) [C Function]
Wake up all threads that are waiting for *condvar*.

Guile also includes some higher-level abstractions for working with mutexes.

`with-mutex` *mutex body1 body2 ...* [macro]
Lock *mutex*, evaluate the body *body1 body2 ...*, then unlock *mutex*. The return value is that returned by the last body form.

The lock, body and unlock form the branches of a `dynamic-wind` (см. Раздел 6.13.10 [Dynamic Wind], страница 339), so *mutex* is automatically unlocked if an error or new continuation exits the body, and is re-locked if the body is re-entered by a captured continuation.

`monitor` *body1 body2 ...* [macro]

Evaluate the body form *body1 body2 ...* with a mutex locked so only one thread can execute that code at any one time. The return value is the return from the last body form.

Each `monitor` form has its own private mutex and the locking and evaluation is as per `with-mutex` above. A standard mutex (`make-mutex`) is used, which means the body must not recursively re-enter the `monitor` form.

The term "monitor" comes from operating system theory, where it means a particular bit of code managing access to some resource and which only ever executes on behalf of one process at any one time.

6.22.6 Blocking in Guile Mode

Up to Guile version 1.8, a thread blocked in guile mode would prevent the garbage collector from running. Thus threads had to explicitly leave guile mode with `scm_without_guile ()` before making a potentially blocking call such as a mutex lock, a `select ()` system call, etc. The following functions could be used to temporarily leave guile mode or to perform some common blocking operations in a supported way.

Starting from Guile 2.0, blocked threads no longer hinder garbage collection. Thus, the functions below are not needed anymore. They can still be used to inform the GC that a thread is about to block, giving it a (small) optimization opportunity for "stop the world" garbage collections, should they occur while the thread is blocked.

`void * scm_without_guile` (*void *(*func) (void *), void *data*) [C Function]

Leave guile mode, call *func* on *data*, enter guile mode and return the result of calling *func*.

While a thread has left guile mode, it must not call any libguile functions except `scm_with_guile` or `scm_without_guile` and must not use any libguile macros. Also, local variables of type `SCM` that are allocated while not in guile mode are not protected from the garbage collector.

When used from non-guile mode, calling `scm_without_guile` is still allowed: it simply calls *func*. In that way, you can leave guile mode without having to know whether the current thread is in guile mode or not.

`int scm_pthread_mutex_lock` (*pthread_mutex_t *mutex*) [C Function]

Like `pthread_mutex_lock`, but leaves guile mode while waiting for the mutex.

`int scm_pthread_cond_wait` (*pthread_cond_t *cond, pthread_mutex_t *mutex*) [C Function]

`int scm_pthread_cond_timedwait` (*pthread_cond_t *cond, pthread_mutex_t *mutex, struct timespec *abstime*) [C Function]

Like `pthread_cond_wait` and `pthread_cond_timedwait`, but leaves guile mode while waiting for the condition variable.

`int scm_std_select` (*int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout*) [C Function]

Like `select` but leaves guile mode while waiting. Also, the delivery of an async causes this function to be interrupted with error code `EINTR`.

`unsigned int scm_std_sleep` (*unsigned int seconds*) [C Function]
Like `sleep`, but leaves guile mode while sleeping. Also, the delivery of an async causes this function to be interrupted.

`unsigned long scm_std_usleep` (*unsigned long usecs*) [C Function]
Like `usleep`, but leaves guile mode while sleeping. Also, the delivery of an async causes this function to be interrupted.

6.22.7 Futures

The `(ice-9 futures)` module provides *futures*, a construct for fine-grain parallelism. A future is a wrapper around an expression whose computation may occur in parallel with the code of the calling thread, and possibly in parallel with other futures. Like promises, futures are essentially proxies that can be queried to obtain the value of the enclosed expression:

```
(touch (future (+ 2 3)))
⇒ 5
```

However, unlike promises, the expression associated with a future may be evaluated on another CPU core, should one be available. This supports *fine-grain parallelism*, because even relatively small computations can be embedded in futures. Consider this sequential code:

```
(define (find-prime lst1 lst2)
  (or (find prime? lst1)
      (find prime? lst2)))
```

The two arms of `or` are potentially computation-intensive. They are independent of one another, yet, they are evaluated sequentially when the first one returns `#f`. Using futures, one could rewrite it like this:

```
(define (find-prime lst1 lst2)
  (let ((f (future (find prime? lst2))))
    (or (find prime? lst1)
        (touch f))))
```

This preserves the semantics of `find-prime`. On a multi-core machine, though, the computation of `(find prime? lst2)` may be done in parallel with that of the other `find` call, which can reduce the execution time of `find-prime`.

Futures may be nested: a future can itself spawn and then `touch` other futures, leading to a directed acyclic graph of futures. Using this facility, a parallel `map` procedure can be defined along these lines:

```
(use-modules (ice-9 futures) (ice-9 match))

(define (par-map proc lst)
  (match lst
    (()
     '())
    ((head tail ...)
     (let ((tail (future (par-map proc tail)))
           (head (proc head)))
       (cons head (touch tail))))))
```

Note that futures are intended for the evaluation of purely functional expressions. Expressions that have side-effects or rely on I/O may require additional care, such as explicit synchronization (см. Раздел 6.22.5 [Mutexes and Condition Variables], страница 469).

Guile's futures are implemented on top of POSIX threads (см. Раздел 6.22.1 [Threads], страница 463). Internally, a fixed-size pool of threads is used to evaluate futures, such that offloading the evaluation of an expression to another thread doesn't incur thread creation costs. By default, the pool contains one thread per available CPU core, minus one, to account for the main thread. The number of available CPU cores is determined using `current-processor-count` (см. Раздел 7.2.7 [Processes], страница 542).

When a thread touches a future that has not completed yet, it processes any pending future while waiting for it to complete, or just waits if there are no pending futures. When `touch` is called from within a future, the execution of the calling future is suspended, allowing its host thread to process other futures, and resumed when the touched future has completed. This suspend/resume is achieved by capturing the calling future's continuation, and later reinstating it (см. Раздел 6.13.5 [Prompts], страница 323).

`future` *exp* [Scheme Syntax]

Return a future for expression *exp*. This is equivalent to:

```
(make-future (lambda () exp))
```

`make-future` *thunk* [Scheme Procedure]

Return a future for *thunk*, a zero-argument procedure.

This procedure returns immediately. Execution of *thunk* may begin in parallel with the calling thread's computations, if idle CPU cores are available, or it may start when `touch` is invoked on the returned future.

If the execution of *thunk* throws an exception, that exception will be re-thrown when `touch` is invoked on the returned future.

`future?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a future.

`touch` *f* [Scheme Procedure]

Return the result of the expression embedded in future *f*.

If the result was already computed in parallel, `touch` returns instantaneously. Otherwise, it waits for the computation to complete, if it already started, or initiates it. In the former case, the calling thread may process other futures in the meantime.

6.22.8 Parallel forms

The functions described in this section are available from

```
(use-modules (ice-9 threads))
```

They provide high-level parallel constructs. The following functions are implemented in terms of futures (см. Раздел 6.22.7 [Futures], страница 474). Thus they are relatively cheap as they re-use existing threads, and portable, since they automatically use one thread per available CPU core.

`parallel` *expr ...* [syntax]
: Evaluate each *expr* expression in parallel, each in its own thread. Return the results of *n* expressions as a set of *n* multiple values (см. Раздел 6.13.7 [Multiple Values], страница 330).

`letpar` ((*var expr*) ...) *body1 body2 ...* [syntax]
: Evaluate each *expr* in parallel, each in its own thread, then bind the results to the corresponding *var* variables, and then evaluate *body1 body2 ...*

  `letpar` is like `let` (см. Раздел 6.12.2 [Local Bindings], страница 314), but all the expressions for the bindings are evaluated in parallel.

`par-map` *proc lst1 lst2 ...* [Scheme Procedure]
`par-for-each` *proc lst1 lst2 ...* [Scheme Procedure]
: Call *proc* on the elements of the given lists. `par-map` returns a list comprising the return values from *proc*. `par-for-each` returns an unspecified value, but waits for all calls to complete.

  The *proc* calls are (*proc elem1 elem2 ...*), where each *elem* is from the corresponding *lst* . Each *lst* must be the same length. The calls are potentially made in parallel, depending on the number of CPU cores available.

  These functions are like `map` and `for-each` (см. Раздел 6.6.9.8 [List Mapping], страница 197), but make their *proc* calls in parallel.

Unlike those above, the functions described below take a number of threads as an argument. This makes them inherently non-portable since the specified number of threads may differ from the number of available CPU cores as returned by `current-processor-count` (см. Раздел 7.2.7 [Processes], страница 542). In addition, these functions create the specified number of threads when they are called and terminate them upon completion, which makes them quite expensive.

Therefore, they should be avoided.

`n-par-map` *n proc lst1 lst2 ...* [Scheme Procedure]
`n-par-for-each` *n proc lst1 lst2 ...* [Scheme Procedure]
: Call *proc* on the elements of the given lists, in the same way as `par-map` and `par-for-each` above, but use no more than *n* threads at any one time. The order in which calls are initiated within that threads limit is unspecified.

  These functions are good for controlling resource consumption if *proc* calls might be costly, or if there are many to be made. On a dual-CPU system for instance $n = 4$ might be enough to keep the CPUs utilized, and not consume too much memory.

`n-for-each-par-map` *n sproc pproc lst1 lst2 ...* [Scheme Procedure]
: Apply *pproc* to the elements of the given lists, and apply *sproc* to each result returned by *pproc*. The final return value is unspecified, but all calls will have been completed before returning.

  The calls made are (*sproc* (*pproc elem1 ... elemN*)), where each *elem* is from the corresponding *lst*. Each *lst* must have the same number of elements.

  The *pproc* calls are made in parallel, in separate threads. No more than *n* threads are used at any one time. The order in which *pproc* calls are initiated within that limit is unspecified.

The *sproc* calls are made serially, in list element order, one at a time. *pproc* calls on later elements may execute in parallel with the *sproc* calls. Exactly which thread makes each *sproc* call is unspecified.

This function is designed for individual calculations that can be done in parallel, but with results needing to be handled serially, for instance to write them to a file. The *n* limit on threads controls system resource usage when there are many calculations or when they might be costly.

It will be seen that `n-for-each-par-map` is like a combination of `n-par-map` and `for-each`,

```
(for-each sproc (n-par-map n pproc lst1 ... lstN))
```

But the actual implementation is more efficient since each *sproc* call, in turn, can be initiated once the relevant *pproc* call has completed, it doesn't need to wait for all to finish.

6.23 Configuration, Features and Runtime Options

Why is my Guile different from your Guile? There are three kinds of possible variation:

- build differences — different versions of the Guile source code, installation directories, configuration flags that control pieces of functionality being included or left out, etc.
- differences in dynamically loaded code — behaviour and features provided by modules that can be dynamically loaded into a running Guile
- different runtime options — some of the options that are provided for controlling Guile's behaviour may be set differently.

Guile provides "introspective" variables and procedures to query all of these possible variations at runtime. For runtime options, it also provides procedures to change the settings of options and to obtain documentation on what the options mean.

6.23.1 Configuration, Build and Installation

The following procedures and variables provide information about how Guile was configured, built and installed on your system.

`version` [Scheme Procedure]
`effective-version` [Scheme Procedure]
`major-version` [Scheme Procedure]
`minor-version` [Scheme Procedure]
`micro-version` [Scheme Procedure]
`scm_version ()` [C Function]
`scm_effective_version ()` [C Function]
`scm_major_version ()` [C Function]
`scm_minor_version ()` [C Function]
`scm_micro_version ()` [C Function]

Return a string describing Guile's full version number, effective version number, major, minor or micro version number, respectively. The `effective-version` function returns the version name that should remain unchanged during a stable series. Currently that means that it omits the micro version. The effective

version should be used for items like the versioned share directory name i.e. `/usr/share/guile/2.2/`

```
(version) ⇒ "2.2.0"
(effective-version) ⇒ "2.2"
(major-version) ⇒ "2"
(minor-version) ⇒ "2"
(micro-version) ⇒ "0"
```

`%package-data-dir` [Scheme Procedure]
`scm_sys_package_data_dir ()` [C Function]

Return the name of the directory under which Guile Scheme files in general are stored. On Unix-like systems, this is usually `/usr/local/share/guile` or `/usr/share/guile`.

`%library-dir` [Scheme Procedure]
`scm_sys_library_dir ()` [C Function]

Return the name of the directory where the Guile Scheme files that belong to the core Guile installation (as opposed to files from a 3rd party package) are installed. On Unix-like systems this is usually `/usr/local/share/guile/`*`GUILE_EFFECTIVE_VERSION`* or `/usr/share/guile/`*`GUILE_EFFECTIVE_VERSION`*;

for example `/usr/local/share/guile/2.2`.

`%site-dir` [Scheme Procedure]
`scm_sys_site_dir ()` [C Function]

Return the name of the directory where Guile Scheme files specific to your site should be installed. On Unix-like systems, this is usually `/usr/local/share/guile/site` or `/usr/share/guile/site`.

`%site-ccache-dir` [Scheme Procedure]
`scm_sys_site_ccache_dir ()` [C Function]

Return the directory where users should install compiled `.go` files for use with this version of Guile. Might look something like `/usr/lib/guile/2.2/site-ccache`.

`%guile-build-info` [Переменная]

Alist of information collected during the building of a particular Guile. Entries can be grouped into one of several categories: directories, env vars, and versioning info.

Briefly, here are the keys in `%guile-build-info`, by group:

directories srcdir, top_srcdir, prefix, exec_prefix, bindir, sbindir, libexecdir, datadir, sysconfdir, sharedstatedir, localstatedir, libdir, infodir, mandir, includedir, pkgdatadir, pkglibdir, pkgincludedir

env vars LIBS

versioning info
guileversion, libguileinterface, buildstamp

Values are all strings. The value for `LIBS` is typically found also as a part of `pkg-config --libs guile-2.2` output. The value for `guileversion` has form X.Y.Z, and should be the same as returned by `(version)`. The value for

`libguileinterface` is libtool compatible and has form CURRENT:REVISION:AGE (см. Раздел "Library interface versions" в *GNU Libtool*). The value for `buildstamp` is the output of the command '`date -u +'%Y-%m-%d %T''` (UTC).

In the source, `%guile-build-info` is initialized from libguile/libpath.h, which is completely generated, so deleting this file before a build guarantees up-to-date values for that build.

`%host-type` [Переменная]

The canonical host type (GNU triplet) of the host Guile was configured for, e.g., `"x86_64-unknown-linux-gnu"` (см. Раздел "Canonicalizing" в *The GNU Autoconf Manual*).

6.23.2 Feature Tracking

Guile has a Scheme level variable `*features*` that keeps track to some extent of the features that are available in a running Guile. `*features*` is a list of symbols, for example `threads`, each of which describes a feature of the running Guile process.

`*features*` [Переменная]

A list of symbols describing available features of the Guile process.

You shouldn't modify the `*features*` variable directly using `set!`. Instead, see the procedures that are provided for this purpose in the following subsection.

6.23.2.1 Feature Manipulation

To check whether a particular feature is available, use the `provided?` procedure:

`provided?` *feature* [Scheme Procedure]
`feature?` *feature* [Deprecated Scheme Procedure]

Return `#t` if the specified *feature* is available, otherwise `#f`.

To advertise a feature from your own Scheme code, you can use the `provide` procedure:

`provide` *feature* [Scheme Procedure]

Add *feature* to the list of available features in this Guile process.

For C code, the equivalent function takes its feature name as a `char *` argument for convenience:

`void scm_add_feature` (*const char *str*) [C Function]

Add a symbol with name *str* to the list of available features in this Guile process.

6.23.2.2 Common Feature Symbols

In general, a particular feature may be available for one of two reasons. Either because the Guile library was configured and compiled with that feature enabled — i.e. the feature is built into the library on your system. Or because some C or Scheme code that was dynamically loaded by Guile has added that feature to the list.

In the first category, here are the features that the current version of Guile may define (depending on how it is built), and what they mean.

`array` Indicates support for arrays (см. Раздел 6.6.13 [Arrays], страница 212).

`array-for-each`
: Indicates availability of `array-for-each` and other array mapping procedures (см. Раздел 6.6.13 [Arrays], страница 212).

`char-ready?`
: Indicates that the `char-ready?` function is available (см. Раздел 6.14.11 [Venerable Port Interfaces], страница 372).

`complex`
: Indicates support for complex numbers.

`current-time`
: Indicates availability of time-related functions: `times`, `get-internal-run-time` and so on (см. Раздел 7.2.5 [Time], страница 537).

`debug-extensions`
: Indicates that the debugging evaluator is available, together with the options for controlling it.

`delay`
: Indicates support for promises (см. Раздел 6.18.9 [Delayed Evaluation], страница 418).

`EIDs`
: Indicates that the `geteuid` and `getegid` really return effective user and group IDs (см. Раздел 7.2.7 [Processes], страница 542).

`inexact`
: Indicates support for inexact numbers.

`i/o-extensions`
: Indicates availability of the following extended I/O procedures: `ftell`, `redirect-port`, `dup->fdes`, `dup2`, `fileno`, `isatty?`, `fdopen`, `primitive-move->fdes` and `fdes->ports` (см. Раздел 7.2.2 [Ports and File Descriptors], страница 521).

`net-db`
: Indicates availability of network database functions: `scm_gethost`, `scm_getnet`, `scm_getproto`, `scm_getserv`, `scm_sethost`, `scm_setnet`, `scm_setproto`, `scm_setserv`, and their 'byXXX' variants (см. Раздел 7.2.11.2 [Network Databases], страница 555).

`posix`
: Indicates support for POSIX functions: `pipe`, `getgroups`, `kill`, `execl` and so on (см. Раздел 7.2 [POSIX], страница 520).

`fork`
: Indicates support for the POSIX `fork` function (см. Раздел 7.2.7 [Processes], страница 542).

`popen`
: Indicates support for `open-pipe` in the `(ice-9 popen)` module (см. Раздел 7.2.10 [Pipes], страница 552).

`random`
: Indicates availability of random number generation functions: `random`, `copy-random-state`, `random-uniform` and so on (см. Раздел 6.6.2.14 [Random], страница 136).

`reckless`
: Indicates that Guile was built with important checks omitted — you should never see this!

`regex`
: Indicates support for POSIX regular expressions using `make-regexp`, `regexp-exec` and friends (см. Раздел 6.15.1 [Regexp Functions], страница 380).

`socket` Indicates availability of socket-related functions: `socket`, `bind`, `connect` and so on (см. Раздел 7.2.11.4 [Network Sockets and Communication], страница 564).

`sort` Indicates availability of sorting and merging functions (см. Раздел 6.11.3 [Sorting], страница 304).

`system` Indicates that the `system` function is available (см. Раздел 7.2.7 [Processes], страница 542).

`threads` Indicates support for multithreading (см. Раздел 6.22.1 [Threads], страница 463).

`values` Indicates support for multiple return values using `values` and `call-with-values` (см. Раздел 6.13.7 [Multiple Values], страница 330).

Available features in the second category depend, by definition, on what additional code your Guile process has loaded in. The following table lists features that you might encounter for this reason.

`defmacro` Indicates that the `defmacro` macro is available (см. Раздел 6.10 [Macros], страница 276).

`describe` Indicates that the `(oop goops describe)` module has been loaded, which provides a procedure for describing the contents of GOOPS instances.

`readline` Indicates that Guile has loaded in Readline support, for command line editing (см. Раздел 7.8 [Readline Support], страница 723).

`record` Indicates support for record definition using `make-record-type` and friends (см. Раздел 6.6.17 [Records], страница 234).

Although these tables may seem exhaustive, it is probably unwise in practice to rely on them, as the correspondences between feature symbols and available procedures/behaviour are not strictly defined. If you are writing code that needs to check for the existence of some procedure, it is probably safer to do so directly using the `defined?` procedure than to test for the corresponding feature using `provided?`.

6.23.3 Runtime Options

There are a number of runtime options available for paramaterizing built-in procedures, like `read`, and built-in behavior, like what happens on an uncaught error.

For more information on reader options, См. Раздел 6.18.2 [Scheme Read], страница 407.

For more information on print options, См. Раздел 6.18.3 [Scheme Write], страница 408.

Finally, for more information on debugger options, См. Раздел 6.26.3.5 [Debug Options], страница 507.

6.23.3.1 Examples of option use

Here is an example of a session in which some read and debug option handling procedures are used. In this example, the user

1. Notices that the symbols `abc` and `aBc` are not the same

2. Examines the `read-options`, and sees that `case-insensitive` is set to "no".
3. Enables `case-insensitive`
4. Quits the recursive prompt
5. Verifies that now `aBc` and `abc` are the same

```
scheme@(guile-user)> (define abc "hello")
scheme@(guile-user)> abc
$1 = "hello"
scheme@(guile-user)> aBc
<unknown-location>: warning: possibly unbound variable `aBc'
ERROR: In procedure module-lookup:
ERROR: Unbound variable: aBc
Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
scheme@(guile-user) [1]> (read-options 'help)
copy              no    Copy source code expressions.
positions         yes   Record positions of source code expressions.
case-insensitive  no    Convert symbols to lower case.
keywords          #f    Style of keyword recognition: #f, 'prefix or 'postfix.
r6rs-hex-escapes  no    Use R6RS variable-length character and string hex escapes.
square-brackets   yes   Treat `[' and `]' as parentheses, for R6RS compatibility.
hungry-eol-escapes no   In strings, consume leading whitespace after an
                        escaped end-of-line.
curly-infix       no    Support SRFI-105 curly infix expressions.
scheme@(guile-user) [1]> (read-enable 'case-insensitive)
$2 = (square-brackets keywords #f case-insensitive positions)
scheme@(guile-user) [1]> ,q
scheme@(guile-user)> aBc
$3 = "hello"
```

6.24 Support for Other Languages

In addition to Scheme, a user may write a Guile program in an increasing number of other languages. Currently supported languages include Emacs Lisp and ECMAScript.

Guile is still fundamentally a Scheme, but it tries to support a wide variety of language building-blocks, so that other languages can be implemented on top of Guile. This allows users to write or extend applications in languages other than Scheme, too. This section describes the languages that have been implemented.

(For details on how to implement a language, См. Раздел 9.4 [Compiling to the Virtual Machine], страница 870.)

6.24.1 Using Other Languages

There are currently only two ways to access other languages from within Guile: at the REPL, and programmatically, via `compile`, `read-and-compile`, and `compile-file`.

The REPL is Guile's command prompt (см. Раздел 4.4 [Using Guile Interactively], страница 51). The REPL has a concept of the "current language", which defaults to Scheme. The user may change that language, via the meta-command `,language`.

For example, the following meta-command enables Emacs Lisp input:

```
scheme@(guile-user)> ,language elisp
Happy hacking with Emacs Lisp!  To switch back, type `,L scheme'.
elisp@(guile-user)> (eq 1 2)
$1 = #nil
```

Each language has its short name: for example, `elisp`, for Elisp. The same short name may be used to compile source code programmatically, via `compile`:

```
elisp@(guile-user)> ,L scheme
Happy hacking with Guile Scheme!  To switch back, type `,L elisp'.
scheme@(guile-user)> (compile '(eq 1 2) #:from 'elisp)
$2 = #nil
```

Granted, as the input to `compile` is a datum, this works best for Lispy languages, which have a straightforward datum representation. Other languages that need more parsing are better dealt with as strings.

The easiest way to deal with syntax-heavy language is with files, via `compile-file` and friends. However it is possible to invoke a language's reader on a port, and then compile the resulting expression (which is a datum at that point). For more information, См. Раздел 6.18.5 [Compilation], страница 411.

For more details on introspecting aspects of different languages, См. Раздел 9.4.1 [Compiler Tower], страница 870.

6.24.2 Emacs Lisp

Emacs Lisp (Elisp) is a dynamically-scoped Lisp dialect used in the Emacs editor. См. Раздел "Overview" в *Emacs Lisp*, for more information on Emacs Lisp.

We hope that eventually Guile's implementation of Elisp will be good enough to replace Emacs' own implementation of Elisp. For that reason, we have thought long and hard about how to support the various features of Elisp in a performant and compatible manner.

Readers familiar with Emacs Lisp might be curious about how exactly these various Elisp features are supported in Guile. The rest of this section focuses on addressing these concerns of the Elisp elect.

6.24.2.1 Nil

`nil` in ELisp is an amalgam of Scheme's `#f` and `'()`. It is false, and it is the end-of-list; thus it is a boolean, and a list as well.

Guile has chosen to support `nil` as a separate value, distinct from `#f` and `'()`. This allows existing Scheme and Elisp code to maintain their current semantics. `nil`, which in Elisp would just be written and read as `nil`, in Scheme has the external representation `#nil`.

This decision to have `nil` as a low-level distinct value facilitates interoperability between the two languages. Guile has chosen to have Scheme deal with `nil` as follows:

```
(boolean? #nil) ⇒ #t
(not #nil) ⇒ #t
(null? #nil) ⇒ #t
```

And in C, one has:

```
scm_is_bool (SCM_ELISP_NIL) ⇒ 1
scm_is_false (SCM_ELISP_NIL) ⇒ 1
scm_is_null (SCM_ELISP_NIL) ⇒ 1
```

In this way, a version of `fold` written in Scheme can correctly fold a function written in Elisp (or in fact any other language) over a nil-terminated list, as Elisp makes. The

converse holds as well; a version of `fold` written in Elisp can fold over a `'()`-terminated list, as made by Scheme.

On a low level, the bit representations for `#f`, `#t`, `nil`, and `'()` are made in such a way that they differ by only one bit, and so a test for, for example, `#f`-or-`nil` may be made very efficiently. See `libguile/boolean.h`, for more information.

Equality

Since Scheme's `equal?` must be transitive, and `'()` is not `equal?` to `#f`, to Scheme `nil` is not `equal?` to `#f` or `'()`.

```
(eq? #f '()) ⇒ #f
(eq? #nil '()) ⇒ #f
(eq? #nil #f) ⇒ #f
(eqv? #f '()) ⇒ #f
(eqv? #nil '()) ⇒ #f
(eqv? #nil #f) ⇒ #f
(equal? #f '()) ⇒ #f
(equal? #nil '()) ⇒ #f
(equal? #nil #f) ⇒ #f
```

However, in Elisp, `'()`, `#f`, and `nil` are all `equal` (though not `eq`).

```
(defvar f (make-scheme-false))
(defvar eol (make-scheme-null))
(eq f eol) ⇒ nil
(eq nil eol) ⇒ nil
(eq nil f) ⇒ nil
(equal f eol) ⇒ t
(equal nil eol) ⇒ t
(equal nil f) ⇒ t
```

These choices facilitate interoperability between Elisp and Scheme code, but they are not perfect. Some code that is correct standard Scheme is not correct in the presence of a second false and null value. For example:

```
(define (truthiness x)
  (if (eq? x #f)
      #f
      #t))
```

This code seems to be meant to test a value for truth, but now that there are two false values, `#f` and `nil`, it is no longer correct.

Similarly, there is the loop:

```
(define (my-length l)
  (let lp ((l l) (len 0))
    (if (eq? l '())
        len
        (lp (cdr l) (1+ len)))))
```

Here, `my-length` will raise an error if *l* is a `nil`-terminated list.

Both of these examples are correct standard Scheme, but, depending on what they really want to do, they are not correct Guile Scheme. Correctly written, they would test

the *properties* of falsehood or nullity, not the individual members of that set. That is to say, they should use `not` or `null?` to test for falsehood or nullity, not `eq?` or `memv` or the like.

Fortunately, using `not` and `null?` is in good style, so all well-written standard Scheme programs are correct, in Guile Scheme.

Here are correct versions of the above examples:

```
(define (truthiness* x)
  (if (not x)
      #f
      #t))
;; or: (define (t* x) (not (not x)))
;; or: (define (t** x) x)

(define (my-length* l)
  (let lp ((l l) (len 0))
    (if (null? l)
        len
        (lp (cdr l) (1+ len)))))
```

This problem has a mirror-image case in Elisp:

```
(defun my-falsep (x)
  (if (eq x nil)
      t
      nil))
```

Guile can warn when compiling code that has equality comparisons with `#f`, `'()`, or `nil`. См. Раздел 6.18.5 [Compilation], страница 411, for details.

6.24.2.2 Dynamic Binding

In contrast to Scheme, which uses "lexical scoping", Emacs Lisp scopes its variables dynamically. Guile supports dynamic scoping with its "fluids" facility. См. Раздел 6.13.11 [Fluids and Dynamic States], страница 343, for more information.

6.24.2.3 Other Elisp Features

Buffer-local and mode-local variables should be mentioned here, along with buckybits on characters, Emacs primitive data types, the Lisp-2-ness of Elisp, and other things. Contributions to the documentation are most welcome!

6.24.3 ECMAScript

ECMAScript (`http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`) was not the first non-Schemey language implemented by Guile, but it was the first implemented for Guile's bytecode compiler. The goal was to support ECMAScript version 3.1, a relatively small language, but the implementor was completely irresponsible and got distracted by other things before finishing the standard library, and even some bits of the syntax. So, ECMAScript does deserve a mention in the manual, but it doesn't deserve an endorsement until its implementation is completed, perhaps by some more responsible hacker.

In the meantime, the charitable user might investigate such invocations as `,L ecmascript` and `cat test-suite/tests/ecmascript.test`.

6.25 Support for Internationalization

Guile provides internationalization[14] support for Scheme programs in two ways. First, procedures to manipulate text and data in a way that conforms to particular cultural conventions (i.e., in a "locale-dependent" way) are provided in the `(ice-9 i18n)`. Second, Guile allows the use of GNU `gettext` to translate program message strings.

6.25.1 Internationalization with Guile

In order to make use of the functions described thereafter, the `(ice-9 i18n)` module must be imported in the usual way:

```
(use-modules (ice-9 i18n))
```

The `(ice-9 i18n)` module provides procedures to manipulate text and other data in a way that conforms to the cultural conventions chosen by the user. Each region of the world or language has its own customs to, for instance, represent real numbers, classify characters, collate text, etc. All these aspects comprise the so-called "cultural conventions" of that region or language.

Computer systems typically refer to a set of cultural conventions as a *locale*. For each particular aspect that comprise those cultural conventions, a *locale category* is defined. For instance, the way characters are classified is defined by the `LC_CTYPE` category, while the language in which program messages are issued to the user is defined by the `LC_MESSAGES` category (см. Раздел 7.2.13 [Locales], страница 571, for details).

The procedures provided by this module allow the development of programs that adapt automatically to any locale setting. As we will see later, many of these procedures can optionally take a *locale object* argument. This additional argument defines the locale settings that must be followed by the invoked procedure. When it is omitted, then the current locale settings of the process are followed (см. Раздел 7.2.13 [Locales], страница 571).

The following procedures allow the manipulation of such locale objects.

`make-locale` *category-list locale-name* **[***base-locale***]** [Scheme Procedure]
`scm_make_locale` (*category_list, locale_name, base_locale*) [C Function]

Return a reference to a data structure representing a set of locale datasets. *locale-name* should be a string denoting a particular locale (e.g., `"aa_DJ"`) and *category-list* should be either a list of locale categories or a single category as used with `setlocale` (см. Раздел 7.2.13 [Locales], страница 571). Optionally, if `base-locale` is passed, it should be a locale object denoting settings for categories not listed in *category-list*.

The following invocation creates a locale object that combines the use of Swedish for messages and character classification with the default settings for the other categories (i.e., the settings of the default `C` locale which usually represents conventions in use in the USA):

```
(make-locale (list LC_MESSAGES LC_CTYPE) "sv_SE")
```

[14] For concision and style, programmers often like to refer to internationalization as "i18n".

The following example combines the use of Esperanto messages and conventions with monetary conventions from Croatia:

```
(make-locale LC_MONETARY "hr_HR"
             (make-locale LC_ALL "eo_EO"))
```

A `system-error` exception (см. Раздел 6.13.13 [Handling Errors], страница 349) is raised by `make-locale` when *locale-name* does not match any of the locales compiled on the system. Note that on non-GNU systems, this error may be raised later, when the locale object is actually used.

`locale?` *obj* [Scheme Procedure]
`scm_locale_p` (*obj*) [C Function]
Return true if *obj* is a locale object.

`%global-locale` [Scheme Variable]
`scm_global_locale` [C Variable]
This variable is bound to a locale object denoting the current process locale as installed using `setlocale ()` (см. Раздел 7.2.13 [Locales], страница 571). It may be used like any other locale object, including as a third argument to `make-locale`, for instance.

6.25.2 Text Collation

The following procedures provide support for text collation, i.e., locale-dependent string and character sorting.

`string-locale<?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_lt` (*s1, s2, locale*) [C Function]
`string-locale>?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_gt` (*s1, s2, locale*) [C Function]
`string-locale-ci<?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_ci_lt` (*s1, s2, locale*) [C Function]
`string-locale-ci>?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_ci_gt` (*s1, s2, locale*) [C Function]
Compare strings *s1* and *s2* in a locale-dependent way. If *locale* is provided, it should be locale object (as returned by `make-locale`) and will be used to perform the comparison; otherwise, the current system locale is used. For the `-ci` variants, the comparison is made in a case-insensitive way.

`string-locale-ci=?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_ci_eq` (*s1, s2, locale*) [C Function]
Compare strings *s1* and *s2* in a case-insensitive, and locale-dependent way. If *locale* is provided, it should be a locale object (as returned by `make-locale`) and will be used to perform the comparison; otherwise, the current system locale is used.

`char-locale<?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_lt` (*c1, c2, locale*) [C Function]
`char-locale>?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_gt` (*c1, c2, locale*) [C Function]
`char-locale-ci<?` *c1 c2* [*locale*] [Scheme Procedure]

`scm_char_locale_ci_lt` (*c1, c2, locale*) [C Function]
`char-locale-ci>?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_ci_gt` (*c1, c2, locale*) [C Function]
Compare characters *c1* and *c2* according to either *locale* (a locale object as returned by `make-locale`) or the current locale. For the `-ci` variants, the comparison is made in a case-insensitive way.

`char-locale-ci=?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_ci_eq` (*c1, c2, locale*) [C Function]
Return true if character *c1* is equal to *c2*, in a case insensitive way according to *locale* or to the current locale.

6.25.3 Character Case Mapping

The procedures below provide support for "character case mapping", i.e., to convert characters or strings to their upper-case or lower-case equivalent. Note that SRFI-13 provides procedures that look similar (см. Раздел 6.6.5.9 [Alphabetic Case Mapping], страница 164). However, the SRFI-13 procedures are locale-independent. Therefore, they do not take into account specificities of the customs in use in a particular language or region of the world. For instance, while most languages using the Latin alphabet map lower-case letter "i" to upper-case letter "I", Turkish maps lower-case "i" to "Latin capital letter I with dot above". The following procedures allow programmers to provide idiomatic character mapping.

`char-locale-downcase` *chr* [*locale*] [Scheme Procedure]
`scm_char_locale_upcase` (*chr, locale*) [C Function]
Return the lowercase character that corresponds to *chr* according to either *locale* or the current locale.

`char-locale-upcase` *chr* [*locale*] [Scheme Procedure]
`scm_char_locale_downcase` (*chr, locale*) [C Function]
Return the uppercase character that corresponds to *chr* according to either *locale* or the current locale.

`char-locale-titlecase` *chr* [*locale*] [Scheme Procedure]
`scm_char_locale_titlecase` (*chr, locale*) [C Function]
Return the titlecase character that corresponds to *chr* according to either *locale* or the current locale.

`string-locale-upcase` *str* [*locale*] [Scheme Procedure]
`scm_string_locale_upcase` (*str, locale*) [C Function]
Return a new string that is the uppercase version of *str* according to either *locale* or the current locale.

`string-locale-downcase` *str* [*locale*] [Scheme Procedure]
`scm_string_locale_downcase` (*str, locale*) [C Function]
Return a new string that is the down-case version of *str* according to either *locale* or the current locale.

`string-locale-titlecase` *str* [*locale*] [Scheme Procedure]
`scm_string_locale_titlecase` (*str, locale*) [C Function]
Return a new string that is the titlecase version of *str* according to either *locale* or the current locale.

6.25.4 Number Input and Output

The following procedures allow programs to read and write numbers written according to a particular locale. As an example, in English, "ten thousand and a half" is usually written `10,000.5` while in French it is written `10 000,5`. These procedures allow such differences to be taken into account.

`locale-string->integer` *str* [*base* [*locale*]] [Scheme Procedure]
`scm_locale_string_to_integer` (*str, base, locale*) [C Function]
Convert string *str* into an integer according to either *locale* (a locale object as returned by `make-locale`) or the current process locale. If *base* is specified, then it determines the base of the integer being read (e.g., `16` for an hexadecimal number, `10` for a decimal number); by default, decimal numbers are read. Return two values (см. Раздел 6.13.7 [Multiple Values], страница 330): an integer (on success) or `#f`, and the number of characters read from *str* (`0` on failure).

This function is based on the C library's `strtol` function (см. Раздел "Parsing of Integers" в *The GNU C Library Reference Manual*).

`locale-string->inexact` *str* [*locale*] [Scheme Procedure]
`scm_locale_string_to_inexact` (*str, locale*) [C Function]
Convert string *str* into an inexact number according to either *locale* (a locale object as returned by `make-locale`) or the current process locale. Return two values (см. Раздел 6.13.7 [Multiple Values], страница 330): an inexact number (on success) or `#f`, and the number of characters read from *str* (`0` on failure).

This function is based on the C library's `strtod` function (см. Раздел "Parsing of Floats" в *The GNU C Library Reference Manual*).

`number->locale-string` *number* [*fraction-digits* [*locale*]] [Scheme Procedure]
Convert *number* (an inexact) into a string according to the cultural conventions of either *locale* (a locale object) or the current locale. By default, print as many fractional digits as necessary, up to an upper bound. Optionally, *fraction-digits* may be bound to an integer specifying the number of fractional digits to be displayed.

`monetary-amount->locale-string` *amount intl?* [*locale*] [Scheme Procedure]
Convert *amount* (an inexact denoting a monetary amount) into a string according to the cultural conventions of either *locale* (a locale object) or the current locale. If *intl?* is true, then the international monetary format for the given locale is used (см. Раздел "Currency Symbol" в *The GNU C Library Reference Manual*).

6.25.5 Accessing Locale Information

It is sometimes useful to obtain very specific information about a locale such as the word it uses for days or months, its format for representing floating-point figures, etc. The `(ice-9`

`i18n)` module provides support for this in a way that is similar to the libc functions `nl_langinfo ()` and `localeconv ()` (см. Раздел "Locale Information" в *The GNU C Library Reference Manual*). The available functions are listed below.

`locale-encoding` [*locale*] [Scheme Procedure]
: Return the name of the encoding (a string whose interpretation is system-dependent) of either *locale* or the current locale.

The following functions deal with dates and times.

`locale-day` *day* [*locale*] [Scheme Procedure]
`locale-day-short` *day* [*locale*] [Scheme Procedure]
`locale-month` *month* [*locale*] [Scheme Procedure]
`locale-month-short` *month* [*locale*] [Scheme Procedure]
: Return the word (a string) used in either *locale* or the current locale to name the day (or month) denoted by *day* (or *month*), an integer between 1 and 7 (or 1 and 12). The `-short` variants provide an abbreviation instead of a full name.

`locale-am-string` [*locale*] [Scheme Procedure]
`locale-pm-string` [*locale*] [Scheme Procedure]
: Return a (potentially empty) string that is used to denote *ante meridiem* (or *post meridiem*) hours in 12-hour format.

`locale-date+time-format` [*locale*] [Scheme Procedure]
`locale-date-format` [*locale*] [Scheme Procedure]
`locale-time-format` [*locale*] [Scheme Procedure]
`locale-time+am/pm-format` [*locale*] [Scheme Procedure]
`locale-era-date-format` [*locale*] [Scheme Procedure]
`locale-era-date+time-format` [*locale*] [Scheme Procedure]
`locale-era-time-format` [*locale*] [Scheme Procedure]
: These procedures return format strings suitable to `strftime` (см. Раздел 7.2.5 [Time], страница 537) that may be used to display (part of) a date/time according to certain constraints and to the conventions of either *locale* or the current locale (см. Раздел "The Elegant and Fast Way" в *The GNU C Library Reference Manual*).

`locale-era` [*locale*] [Scheme Procedure]
`locale-era-year` [*locale*] [Scheme Procedure]
: These functions return, respectively, the era and the year of the relevant era used in *locale* or the current locale. Most locales do not define this value. In this case, the empty string is returned. An example of a locale that does define this value is the Japanese one.

The following procedures give information about number representation.

`locale-decimal-point` [*locale*] [Scheme Procedure]
`locale-thousands-separator` [*locale*] [Scheme Procedure]
: These functions return a string denoting the representation of the decimal point or that of the thousand separator (respectively) for either *locale* or the current locale.

`locale-digit-grouping` [*locale*] [Scheme Procedure]
: Return a (potentially circular) list of integers denoting how digits of the integer part of a number are to be grouped, starting at the decimal point and going to the left. The list contains integers indicating the size of the successive groups, from right to left. If the list is non-circular, then no grouping occurs for digits beyond the last group.

  For instance, if the returned list is a circular list that contains only `3` and the thousand separator is `","` (as is the case with English locales), then the number `12345678` should be printed `12,345,678`.

The following procedures deal with the representation of monetary amounts. Some of them take an additional *intl?* argument (a boolean) that tells whether the international or local monetary conventions for the given locale are to be used.

`locale-monetary-decimal-point` [*locale*] [Scheme Procedure]
`locale-monetary-thousands-separator` [*locale*] [Scheme Procedure]
`locale-monetary-grouping` [*locale*] [Scheme Procedure]
: These are the monetary counterparts of the above procedures. These procedures apply to monetary amounts.

`locale-currency-symbol` *intl?* [*locale*] [Scheme Procedure]
: Return the currency symbol (a string) of either *locale* or the current locale.

  The following example illustrates the difference between the local and international monetary formats:

```
(define us (make-locale LC_MONETARY "en_US"))
(locale-currency-symbol #f us)
⇒ "-$"
(locale-currency-symbol #t us)
⇒ "USD "
```

`locale-monetary-fractional-digits` *intl?* [*locale*] [Scheme Procedure]
: Return the number of fractional digits to be used when printing monetary amounts according to either *locale* or the current locale. If the locale does not specify it, then `#f` is returned.

`locale-currency-symbol-precedes-positive?` *intl?* [*locale*] [Scheme Procedure]
`locale-currency-symbol-precedes-negative?` *intl?* [*locale*] [Scheme Procedure]
`locale-positive-separated-by-space?` *intl?* [*locale*] [Scheme Procedure]
`locale-negative-separated-by-space?` *intl?* [*locale*] [Scheme Procedure]
: These procedures return a boolean indicating whether the currency symbol should precede a positive/negative number, and whether a whitespace should be inserted between the currency symbol and a positive/negative amount.

`locale-monetary-positive-sign` [*locale*] [Scheme Procedure]
`locale-monetary-negative-sign` [*locale*] [Scheme Procedure]
: Return a string denoting the positive (respectively negative) sign that should be used when printing a monetary amount.

`locale-positive-sign-position` [Scheme Procedure]
`locale-negative-sign-position` [Scheme Procedure]

These functions return a symbol telling where a sign of a positive/negative monetary amount is to appear when printing it. The possible values are:

`parenthesize`
: The currency symbol and quantity should be surrounded by parentheses.

`sign-before`
: Print the sign string before the quantity and currency symbol.

`sign-after`
: Print the sign string after the quantity and currency symbol.

`sign-before-currency-symbol`
: Print the sign string right before the currency symbol.

`sign-after-currency-symbol`
: Print the sign string right after the currency symbol.

`unspecified`
: Unspecified. We recommend you print the sign after the currency symbol.

Finally, the two following procedures may be helpful when programming user interfaces:

`locale-yes-regexp` [*locale*] [Scheme Procedure]
`locale-no-regexp` [*locale*] [Scheme Procedure]

Return a string that can be used as a regular expression to recognize a positive (respectively, negative) response to a yes/no question. For the C locale, the default values are typically `"^[yY]"` and `"^[nN]"`, respectively.

Here is an example:

```
(use-modules (ice-9 rdelim))
(format #t "Does Guile rock?~%")
(let lp ((answer (read-line)))
  (cond ((string-match (locale-yes-regexp) answer)
         (format #t "High fives!~%"))
        ((string-match (locale-no-regexp) answer)
         (format #t "How about now? Does it rock yet?~%")
         (lp (read-line)))
        (else
         (format #t "What do you mean?~%")
         (lp (read-line)))))
```

For an internationalized yes/no string output, `gettext` should be used (см. Раздел 6.25.6 [Gettext Support], страница 493).

Example uses of some of these functions are the implementation of the `number->locale-string` and `monetary-amount->locale-string` procedures (см. Раздел 6.25.4 [Number Input and Output], страница 489), as well as that the SRFI-19 date and time conversion to/from strings (см. Раздел 7.5.16 [SRFI-19], страница 637).

6.25.6 Gettext Support

Guile provides an interface to GNU `gettext` for translating message strings (см. Раздел "Introduction" в *GNU* `gettext` *utilities*).

Messages are collected in domains, so different libraries and programs maintain different message catalogues. The *domain* parameter in the functions below is a string (it becomes part of the message catalog filename).

When `gettext` is not available, or if Guile was configured '`--without-nls`', dummy functions doing no translation are provided. When `gettext` support is available in Guile, the `i18n` feature is provided (см. Раздел 6.23.2 [Feature Tracking], страница 479).

`gettext` *msg* [*domain* [*category*]] [Scheme Procedure]
`scm_gettext` (*msg, domain, category*) [C Function]

Return the translation of *msg* in *domain*. *domain* is optional and defaults to the domain set through `textdomain` below. *category* is optional and defaults to `LC_MESSAGES` (см. Раздел 7.2.13 [Locales], страница 571).

Normal usage is for *msg* to be a literal string. `xgettext` can extract those from the source to form a message catalogue ready for translators (см. Раздел "Invoking the `xgettext` Program" в *GNU* `gettext` *utilities*).

```
(display (gettext "You are in a maze of twisty passages."))
```

`_` is a commonly used shorthand, an application can make that an alias for `gettext`. Or a library can make a definition that uses its specific *domain* (so an application can change the default without affecting the library).

```
(define (_ msg) (gettext msg "mylibrary"))
(display (_ "File not found."))
```

`_` is also a good place to perhaps strip disambiguating extra text from the message string, as for instance in Раздел "How to use `gettext` in GUI programs" в *GNU* `gettext` *utilities*.

`ngettext` *msg msgplural n* [*domain* [*category*]] [Scheme Procedure]
`scm_ngettext` (*msg, msgplural, n, domain, category*) [C Function]

Return the translation of *msg*/*msgplural* in *domain*, with a plural form chosen appropriately for the number *n*. *domain* is optional and defaults to the domain set through `textdomain` below. *category* is optional and defaults to `LC_MESSAGES` (см. Раздел 7.2.13 [Locales], страница 571).

msg is the singular form, and *msgplural* the plural. When no translation is available, *msg* is used if $n = 1$, or *msgplural* otherwise. When translated, the message catalogue can have a different rule, and can have more than two possible forms.

As per `gettext` above, normal usage is for *msg* and *msgplural* to be literal strings, since `xgettext` can extract them from the source to build a message catalogue. For example,

```
(define (done n)
  (format #t (ngettext "~a file processed\n"
                       "~a files processed\n" n)
             n))
```

```
(done 1) ⊣ 1 file processed
(done 3) ⊣ 3 files processed
```

It's important to use `ngettext` rather than plain `gettext` for plurals, since the rules for singular and plural forms in English are not the same in other languages. Only `ngettext` will allow translators to give correct forms (см. Раздел "Additional functions for plural forms" в *GNU* `gettext` *utilities*).

`textdomain` [*domain*] [Scheme Procedure]
`scm_textdomain` (*domain*) [C Function]

Get or set the default gettext domain. When called with no parameter the current domain is returned. When called with a parameter, *domain* is set as the current domain, and that new value returned. For example,

```
(textdomain "myprog")
⇒ "myprog"
```

`bindtextdomain` *domain* [*directory*] [Scheme Procedure]
`scm_bindtextdomain` (*domain, directory*) [C Function]

Get or set the directory under which to find message files for *domain*. When called without a *directory* the current setting is returned. When called with a *directory*, *directory* is set for *domain* and that new setting returned. For example,

```
(bindtextdomain "myprog" "/my/tree/share/locale")
⇒ "/my/tree/share/locale"
```

When using Autoconf/Automake, an application should arrange for the configured `localedir` to get into the program (by substituting, or by generating a config file) and set that for its domain. This ensures the catalogue can be found even when installed in a non-standard location.

`bind-textdomain-codeset` *domain* [*encoding*] [Scheme Procedure]
`scm_bind_textdomain_codeset` (*domain, encoding*) [C Function]

Get or set the text encoding to be used by `gettext` for messages from *domain*. *encoding* is a string, the name of a coding system, for instance `"8859_1"`. (On a Unix/POSIX system the `iconv` program can list all available encodings.)

When called without an *encoding* the current setting is returned, or `#f` if none yet set. When called with an *encoding*, it is set for *domain* and that new setting returned. For example,

```
(bind-textdomain-codeset "myprog")
⇒ #f
(bind-textdomain-codeset "myprog" "latin-9")
⇒ "latin-9"
```

The encoding requested can be different from the translated data file, messages will be recoded as necessary. But note that when there is no translation, `gettext` returns its *msg* unchanged, ie. without any recoding. For that reason source message strings are best as plain ASCII.

Currently Guile has no understanding of multi-byte characters, and string functions won't recognise character boundaries in multi-byte strings. An application will at least be able to pass such strings through to some output though. Perhaps this will change in the future.

6.26 Debugging Infrastructure

In order to understand Guile's debugging facilities, you first need to understand a little about how Guile represent the Scheme control stack. With that in place we explain the low level trap calls that the virtual machine can be configured to make, and the trap and breakpoint infrastructure that builds on top of those calls.

6.26.1 Evaluation and the Scheme Stack

The idea of the Scheme stack is central to a lot of debugging. The Scheme stack is a reified representation of the pending function returns in an expression's continuation. As Guile implements function calls using a stack, this reification takes the form of a number of nested stack frames, each of which corresponds to the application of a procedure to a set of arguments.

A Scheme stack always exists implicitly, and can be summoned into concrete existence as a first-class Scheme value by the `make-stack` call, so that an introspective Scheme program – such as a debugger – can present it in some way and allow the user to query its details. The first thing to understand, therefore, is how Guile's function call convention creates the stack.

Broadly speaking, Guile represents all control flow on a stack. Calling a function involves pushing an empty frame on the stack, then evaluating the procedure and its arguments, then fixing up the new frame so that it points to the old one. Frames on the stack are thus linked together. A tail call is the same, except it reuses the existing frame instead of pushing on a new one.

In this way, the only frames that are on the stack are "active" frames, frames which need to do some work before the computation is complete. On the other hand, a function that has tail-called another function will not be on the stack, as it has no work left to do.

Therefore, when an error occurs in a running program, or the program hits a breakpoint, or in fact at any point that the programmer chooses, its state at that point can be represented by a *stack* of all the procedure applications that are logically in progress at that time, each of which is known as a *frame*. The programmer can learn more about the program's state at that point by inspecting the stack and its frames.

6.26.1.1 Stack Capture

A Scheme program can use the `make-stack` primitive anywhere in its code, with first arg `#t`, to construct a Scheme value that describes the Scheme stack at that point.

```
(make-stack #t)
⇒
#<stack 25205a0>
```

Use `start-stack` to limit the stack extent captured by future `make-stack` calls.

`make-stack` *obj arg ...* [Scheme Procedure]
`scm_make_stack` (*obj, args*) [C Function]

Create a new stack. If *obj* is `#t`, the current evaluation stack is used for creating the stack frames, otherwise the frames are taken from *obj* (which must be a continuation or a frame object).

arg ... can be any combination of integer, procedure, address range, and prompt tag values.

These values specify various ways of cutting away uninteresting stack frames from the top and bottom of the stack that `make-stack` returns. They come in pairs like this: (*inner_cut_1 outer_cut_1 inner_cut_2 outer_cut_2 ...*).

Each *inner_cut_i* can be an integer, a procedure, an address range, or a prompt tag. An integer means to cut away exactly that number of frames. A procedure means to cut away all frames up to but excluding the frame whose procedure matches the specified one. An address range is a pair of integers indicating the low and high addresses of a procedure's code, and is the same as cutting away to a procedure (though with less work). Anything else is interpreted as a prompt tag which cuts away all frames that are inside a prompt with the given tag.

Each *outer_cut_i* can likewise be an integer, a procedure, an address range, or a prompt tag. An integer means to cut away that number of frames. A procedure means to cut away frames down to but excluding the frame whose procedure matches the specified one. An address range is the same, but with the procedure's code specified as an address range. Anything else is taken to be a prompt tag, which cuts away all frames that are outside a prompt with the given tag.

If the *outer_cut_i* of the last pair is missing, it is taken as 0.

`start-stack` *id exp* [Scheme Syntax]

Evaluate *exp* on a new calling stack with identity *id*. If *exp* is interrupted during evaluation, backtraces will not display frames farther back than *exp*'s top-level form. This macro is a way of artificially limiting backtraces and stack procedures, largely as a convenience to the user.

6.26.1.2 Stacks

`stack?` *obj* [Scheme Procedure]
`scm_stack_p` (*obj*) [C Function]

Return `#t` if *obj* is a calling stack.

`stack-id` *stack* [Scheme Procedure]
`scm_stack_id` (*stack*) [C Function]

Return the identifier given to *stack* by `start-stack`.

`stack-length` *stack* [Scheme Procedure]
`scm_stack_length` (*stack*) [C Function]

Return the length of *stack*.

`stack-ref` *stack index* [Scheme Procedure]
`scm_stack_ref` (*stack, index*) [C Function]

Return the *index*'th frame from *stack*.

`display-backtrace` *stack port* [*first* [*depth* [*highlights*]]] [Scheme Procedure]
`scm_display_backtrace_with_highlights` (*stack, port, first, depth, highlights*) [C Function]

`scm_display_backtrace` (*stack*, *port*, *first*, *depth*) [C Function]
Display a backtrace to the output port *port*. *stack* is the stack to take the backtrace from, *first* specifies where in the stack to start and *depth* how many frames to display. *first* and *depth* can be `#f`, which means that default values will be used. If *highlights* is given it should be a list; the elements of this list will be highlighted wherever they appear in the backtrace.

6.26.1.3 Frames

`frame?` *obj* [Scheme Procedure]
`scm_frame_p` (*obj*) [C Function]
Return `#t` if *obj* is a stack frame.

`frame-previous` *frame* [Scheme Procedure]
`scm_frame_previous` (*frame*) [C Function]
Return the previous frame of *frame*, or `#f` if *frame* is the first frame in its stack.

`frame-procedure-name` *frame* [Scheme Procedure]
`scm_frame_procedure_name` (*frame*) [C Function]
Return the name of the procedure being applied in *frame*, as a symbol, or `#f` if the procedure has no name.

`frame-arguments` *frame* [Scheme Procedure]
`scm_frame_arguments` (*frame*) [C Function]
Return the arguments of *frame*.

`frame-address` *frame* [Scheme Procedure]
`frame-instruction-pointer` *frame* [Scheme Procedure]
`frame-stack-pointer` *frame* [Scheme Procedure]
Accessors for the three VM registers associated with this frame: the frame pointer (fp), instruction pointer (ip), and stack pointer (sp), respectively. См. Раздел 9.3.2 [VM Concepts], страница 845, for more information.

`frame-dynamic-link` *frame* [Scheme Procedure]
`frame-return-address` *frame* [Scheme Procedure]
`frame-mv-return-address` *frame* [Scheme Procedure]
Accessors for the three saved VM registers in a frame: the previous frame pointer, the single-value return address, and the multiple-value return address. См. Раздел 9.3.3 [Stack Layout], страница 846, for more information.

`frame-bindings` *frame* [Scheme Procedure]
Return a list of binding records indicating the local variables that are live in a frame.

`frame-lookup-binding` *frame var* [Scheme Procedure]
Fetch the bindings in *frame*, and return the first one whose name is *var*, or `#f` otherwise.

`binding-index` *binding* [Scheme Procedure]
`binding-name` *binding* [Scheme Procedure]
`binding-slot` *binding* [Scheme Procedure]

`binding-representation` *binding* [Scheme Procedure]

Accessors for the various fields in a binding. The implicit "callee" argument is index 0, the first argument is index 1, and so on to the end of the arguments. After that are temporary variables. Note that if a variable is dead, it might not be available.

`binding-ref` *binding* [Scheme Procedure]
`binding-set!` *binding val* [Scheme Procedure]

Accessors for the values of local variables in a frame.

`display-application` *frame* [*port* [*indent*]] [Scheme Procedure]
`scm_display_application` (*frame, port, indent*) [C Function]

Display a procedure application *frame* to the output port *port*. *indent* specifies the indentation of the output.

Additionally, the `(system vm frame)` module defines a number of higher-level introspective procedures, for example to retrieve the names of local variables, and the source location to correspond to a frame. See its source code for more details.

6.26.2 Source Properties

As Guile reads in Scheme code from file or from standard input, it remembers the file name, line number and column number where each expression begins. These pieces of information are known as the *source properties* of the expression. Syntax expanders and the compiler propagate these source properties to compiled procedures, so that, if an error occurs when evaluating the transformed expression, Guile's debugger can point back to the file and location where the expression originated.

The way that source properties are stored means that Guile cannot associate source properties with individual symbols, keywords, characters, booleans, or small integers. This can be seen by typing `(xxx)` and `xxx` at the Guile prompt (where the variable `xxx` has not been defined):

```
scheme@(guile-user)> (xxx)
<unnamed port>:4:1: In procedure module-lookup:
<unnamed port>:4:1: Unbound variable: xxx

scheme@(guile-user)> xxx
ERROR: In procedure module-lookup:
ERROR: Unbound variable: xxx
```

In the latter case, no source properties were stored, so the error doesn't have any source information.

`supports-source-properties?` *obj* [Scheme Procedure]
`scm_supports_source_properties_p` (*obj*) [C Function]

Return #t if source properties can be associated with *obj*, otherwise return #f.

The recording of source properties is controlled by the read option named "positions" (см. Раздел 6.18.2 [Scheme Read], страница 407). This option is switched *on* by default.

The following procedures can be used to access and set the source properties of read expressions.

`set-source-properties!` *obj alist* [Scheme Procedure]
`scm_set_source_properties_x` (*obj, alist*) [C Function]
Install the association list *alist* as the source property list for *obj*.

`set-source-property!` *obj key datum* [Scheme Procedure]
`scm_set_source_property_x` (*obj, key, datum*) [C Function]
Set the source property of object *obj*, which is specified by *key* to *datum*. Normally, the key will be a symbol.

`source-properties` *obj* [Scheme Procedure]
`scm_source_properties` (*obj*) [C Function]
Return the source property association list of *obj*.

`source-property` *obj key* [Scheme Procedure]
`scm_source_property` (*obj, key*) [C Function]
Return the property specified by *key* from *obj*'s source properties.

If the `positions` reader option is enabled, supported expressions will have values set for the `filename`, `line` and `column` properties.

Source properties are also associated with syntax objects. Procedural macros can get at the source location of their input using the `syntax-source` accessor. См. Раздел 6.10.4 [Syntax Transformer Helpers], страница 289, for more.

Guile also defines a couple of convenience macros built on `syntax-source`:

`current-source-location` [Scheme Syntax]
Expands to the source properties corresponding to the location of the `(current-source-location)` form.

`current-filename` [Scheme Syntax]
Expands to the current filename: the filename that the `(current-filename)` form appears in. Expands to `#f` if this information is unavailable.

If you're stuck with defmacros (см. Раздел 6.10.5 [Defmacros], страница 292), and want to preserve source information, the following helper function might be useful to you:

`cons-source` *xorig x y* [Scheme Procedure]
`scm_cons_source` (*xorig, x, y*) [C Function]
Create and return a new pair whose car and cdr are *x* and *y*. Any source properties associated with *xorig* are also associated with the new pair.

6.26.3 Programmatic Error Handling

For better or for worse, all programs have bugs, and dealing with bugs is part of programming. This section deals with that class of bugs that causes an exception to be raised – from your own code, from within a library, or from Guile itself.

6.26.3.1 Catching Exceptions

A common requirement is to be able to show as much useful context as possible when a Scheme program hits an error. The most immediate information about an error is the kind of error that it is – such as "division by zero" – and any parameters that the code which

signalled the error chose explicitly to provide. This information originates with the `error` or `throw` call (or their C code equivalents, if the error is detected by C code) that signals the error, and is passed automatically to the handler procedure of the innermost applicable `catch` or `with-throw-handler` expression.

Therefore, to catch errors that occur within a chunk of Scheme code, and to intercept basic information about those errors, you need to execute that code inside the dynamic context of a `catch` or `with-throw-handler` expression, or the equivalent in C. In Scheme, this means you need something like this:

```
(catch #t
       (lambda ()
         ;; Execute the code in which
         ;; you want to catch errors here.
         ...)
       (lambda (key . parameters)
         ;; Put the code which you want
         ;; to handle an error here.
         ...))
```

The `catch` here can also be `with-throw-handler`; see Раздел 6.13.8.3 [Throw Handlers], страница 335, for information on the when you might want to use `with-throw-handler` instead of `catch`.

For example, to print out a message and return #f when an error occurs, you might use:

```
(define (catch-all thunk)
  (catch #t
    thunk
    (lambda (key . parameters)
      (format (current-error-port)
              "Uncaught throw to '~a: ~a\n" key parameters)
      #f)))

(catch-all
 (lambda () (error "Not a vegetable: tomato")))
⊣ Uncaught throw to 'misc-error: (#f ~A (Not a vegetable: tomato) #f)
⇒ #f
```

The `#t` means that the catch is applicable to all kinds of error. If you want to restrict your catch to just one kind of error, you can put the symbol for that kind of error instead of `#t`. The equivalent to this in C would be something like this:

```
SCM my_body_proc (void *body_data)
{
  /* Execute the code in which
     you want to catch errors here. */
  ...
}

SCM my_handler_proc (void *handler_data,
                     SCM key,
                     SCM parameters)
{
```

```
  /* Put the code which you want
     to handle an error here. */
  ...
}

{
  ...
  scm_c_catch (SCM_BOOL_T,
               my_body_proc, body_data,
               my_handler_proc, handler_data,
               NULL, NULL);
  ...
}
```

Again, as with the Scheme version, `scm_c_catch` could be replaced by `scm_c_with_throw_handler`, and `SCM_BOOL_T` could instead be the symbol for a particular kind of error.

6.26.3.2 Capturing the full error stack

The other interesting information about an error is the full Scheme stack at the point where the error occurred; in other words what innermost expression was being evaluated, what was the expression that called that one, and so on. If you want to write your code so that it captures and can display this information as well, there are a couple important things to understand.

Firstly, the stack at the point of the error needs to be explicitly captured by a `make-stack` call (or the C equivalent `scm_make_stack`). The Guile library does not do this "automatically" for you, so you will need to write code with a `make-stack` or `scm_make_stack` call yourself. (We emphasise this point because some people are misled by the fact that the Guile interactive REPL code *does* capture and display the stack automatically. But the Guile interactive REPL is itself a Scheme program[15] running on top of the Guile library, and which uses `catch` and `make-stack` in the way we are about to describe to capture the stack when an error occurs.)

And secondly, in order to capture the stack effectively at the point where the error occurred, the `make-stack` call must be made before Guile unwinds the stack back to the location of the prevailing catch expression. This means that the `make-stack` call must be made within the handler of a `with-throw-handler` expression, or the optional `"pre-unwind"` handler of a `catch`. (For the full story of how these alternatives differ from each other, see Раздел 6.13.8 [Exceptions], страница 332. The main difference is that `catch` terminates the error, whereas `with-throw-handler` only intercepts it temporarily and then allow it to continue propagating up to the next innermost handler.)

So, here are some examples of how to do all this in Scheme and in C. For the purpose of these examples we assume that the captured stack should be stored in a variable, so that it can be displayed or arbitrarily processed later on. In Scheme:

```
(let ((captured-stack #f))
  (catch #t
```

[15] In effect, it is the default program which is run when no commands or script file are specified on the Guile command line.

```
          (lambda ()
            ;; Execute the code in which
            ;; you want to catch errors here.
            ...)
          (lambda (key . parameters)
            ;; Put the code which you want
            ;; to handle an error after the
            ;; stack has been unwound here.
            ...)
          (lambda (key . parameters)
            ;; Capture the stack here:
            (set! captured-stack (make-stack #t))))
    ...
    (if captured-stack
        (begin
          ;; Display or process the captured stack.
          ...))
    ...)
```

And in C:

```
SCM my_body_proc (void *body_data)
{
  /* Execute the code in which
     you want to catch errors here. */
  ...
}

SCM my_handler_proc (void *handler_data,
                     SCM key,
                     SCM parameters)
{
  /* Put the code which you want
     to handle an error after the
     stack has been unwound here. */
  ...
}

SCM my_preunwind_proc (void *handler_data,
                       SCM key,
                       SCM parameters)
{
  /* Capture the stack here: */
  *(SCM *)handler_data = scm_make_stack (SCM_BOOL_T, SCM_EOL);
}

{
  SCM captured_stack = SCM_BOOL_F;
```

```
  ...
  scm_c_catch (SCM_BOOL_T,
               my_body_proc, body_data,
               my_handler_proc, handler_data,
               my_preunwind_proc, &captured_stack);
  ...
  if (captured_stack != SCM_BOOL_F)
  {
    /* Display or process the captured stack. */
    ...
  }
  ...
}
```

Once you have a captured stack, you can interrogate and display its details in any way that you want, using the `stack-...` and `frame-...` API described in Раздел 6.26.1.2 [Stacks], страница 496, and Раздел 6.26.1.3 [Frames], страница 497.

If you want to print out a backtrace in the same format that the Guile REPL does, you can use the `display-backtrace` procedure to do so. You can also use `display-application` to display an individual frame in the Guile REPL format.

6.26.3.3 Pre-Unwind Debugging

Instead of saving a stack away and waiting for the `catch` to return, you can handle errors directly, from within the pre-unwind handler.

For example, to show a backtrace when an error is thrown, you might want to use a procedure like this:

```
(define (with-backtrace thunk)
  (with-throw-handler #t
                      thunk
                      (lambda args (backtrace))))
(with-backtrace (lambda () (error "Not a vegetable: tomato")))
```

Since we used `with-throw-handler` here, we didn't actually catch the error. См. Раздел 6.13.8.3 [Throw Handlers], страница 335, for more information. However, we did print out a context at the time of the error, using the built-in procedure, `backtrace`.

`backtrace` [*highlights*] [Scheme Procedure]
`scm_backtrace_with_highlights` (*highlights*) [C Function]
`scm_backtrace` () [C Function]

Display a backtrace of the current stack to the current output port. If *highlights* is given it should be a list; the elements of this list will be highlighted wherever they appear in the backtrace.

The Guile REPL code (in `system/repl/repl.scm` and related files) uses a `catch` with a pre-unwind handler to capture the stack when an error occurs in an expression that was typed into the REPL, and debug that stack interactively in the context of the error.

These procedures are available for use by user programs, in the `(system repl error-handling)` module.

```
(use-modules (system repl error-handling))
```

`call-with-error-handling` *thunk* **[***#:on-error on-error='debug***]** [Scheme Procedure] **[***#:post-error post-error='catch***]** **[***#:pass-keys pass-keys='(quit)***]** **[***#:report-keys report-keys='(stack-overflow)***]** **[***#:trap-handler trap-handler='debug***]**

Call a thunk in a context in which errors are handled.

There are five keyword arguments:

on-error Specifies what to do before the stack is unwound.

Valid options are `debug` (the default), which will enter a debugger; `pass`, in which case nothing is done, and the exception is rethrown; or a procedure, which will be the pre-unwind handler.

post-error Specifies what to do after the stack is unwound.

Valid options are `catch` (the default), which will silently catch errors, returning the unspecified value; `report`, which prints out a description of the error (via `display-error`), and then returns the unspecified value; or a procedure, which will be the catch handler.

trap-handler

Specifies a trap handler: what to do when a breakpoint is hit.

Valid options are `debug`, which will enter the debugger; `pass`, which does nothing; or `disabled`, which disables traps entirely. См. Раздел 6.26.4 [Traps], страница 507, for more information.

pass-keys A set of keys to ignore, as a list.

report-keys

A set of keys to always report even if the post-error handler is `catch`, as a list.

6.26.3.4 Stack Overflow

Every time a Scheme program makes a call that is not in tail position, it pushes a new frame onto the stack. Returning a value from a function pops the top frame off the stack. Stack frames take up memory, and as nobody has an infinite amount of memory, deep recursion could cause Guile to run out of memory. Running out of stack memory is called *stack overflow*.

Stack Limits

Most languages have a terrible stack overflow story. For example, in C, if you use too much stack, your program will exhibit "undefined behavior", which if you are lucky means that it will crash. It's especially bad in C, as you neither know ahead of time how much stack your functions use, nor the stack limit imposed by the user's system, and the stack limit is often quite small relative to the total memory size.

Managed languages like Python have a better error story, as they are defined to raise an exception on stack overflow – but like C, Python and most dynamic languages still have a fixed stack size limit that is usually much smaller than the heap.

Arbitrary stack limits would have an unfortunate effect on Guile programs. For example, the following implementation of the inner loop of `map` is clean and elegant:

```
(define (map f l)
```

```
(if (pair? l)
    (cons (f (car l))
          (map f (cdr l)))
    '())
```

However, if there were a stack limit, that would limit the size of lists that can be processed with this `map`. Eventually, you would have to rewrite it to use iteration with an accumulator:

```
(define (map f l)
  (let lp ((l l) (out '()))
    (if (pair? l)
        (lp (cdr l) (cons (f (car l)) out))
        (reverse out))))
```

This second version is sadly not as clear, and it also allocates more heap memory (once to build the list in reverse, and then again to reverse the list). You would be tempted to use the destructive `reverse!` to save memory and time, but then your code would not be continuation-safe – if *f* returned again after the map had finished, it would see an *out* list that had already been reversed. The recursive `map` has none of these problems.

Guile has no stack limit for Scheme code. When a thread makes its first Guile call, a small stack is allocated – just one page of memory. Whenever that memory limit would be reached, Guile arranges to grow the stack by a factor of two. When garbage collection happens, Guile arranges to return the unused part of the stack to the operating system, but without causing the stack to shrink. In this way, the stack can grow to consume up to all memory available to the Guile process, and when the recursive computation eventually finishes, that stack memory is returned to the system.

Exceptional Situations

Of course, it's still possible to run out of stack memory. The most common cause of this is program bugs that cause unbounded recursion, as in:

```
(define (faulty-map f l)
  (if (pair? l)
      (cons (f (car l)) (faulty-map f l))
      '()))
```

Did you spot the bug? The recursive call to `faulty-map` recursed on *l*, not `(cdr l)`. Running this program would cause Guile to use up all memory in your system, and eventually Guile would fail to grow the stack. At that point you have a problem: Guile needs to raise an exception to unwind the stack and return memory to the system, but the user might have throw handlers in place (см. Раздел 6.13.8.3 [Throw Handlers], страница 335) that want to run before the stack is unwound, and we don't have any stack in which to run them.

Therefore in this case, Guile throws an unwind-only exception that does not run pre-unwind handlers. Because this is such an odd case, Guile prints out a message on the console, in case the user was expecting to be able to get a backtrace from any pre-unwind handler.

Runaway Recursion

Still, this failure mode is not so nice. If you are running an environment in which you are interactively building a program while it is running, such as at a REPL, you might want to impose an artificial stack limit on the part of your program that you are building to detect accidental runaway recursion. For that purpose, there is `call-with-stack-overflow-handler`, from `(system vm vm)`.

```
(use-module (system vm vm))
```

`call-with-stack-overflow-handler` *limit thunk handler* [Scheme Procedure]

Call *thunk* in an environment in which the stack limit has been reduced to *limit* additional words. If the limit is reached, *handler* (a thunk) will be invoked in the dynamic environment of the error. For the extent of the call to *handler*, the stack limit and handler are restored to the values that were in place when `call-with-stack-overflow-handler` was called.

Usually, *handler* should raise an exception or abort to an outer prompt. However if *handler* does return, it should return a number of additional words of stack space to allow to the inner environment.

A stack overflow handler may only ever "credit" the inner thunk with stack space that was available when the handler was instated. When Guile first starts, there is no stack limit in place, so the outer handler may allow the inner thunk an arbitrary amount of space, but any nested stack overflow handler will not be able to consume more than its limit.

Unlike the unwind-only exception that is thrown if Guile is unable to grow its stack, any exception thrown by a stack overflow handler might invoke pre-unwind handlers. Indeed, the stack overflow handler is itself a pre-unwind handler of sorts. If the code imposing the stack limit wants to protect itself against malicious pre-unwind handlers from the inner thunk, it should abort to a prompt of its own making instead of throwing an exception that might be caught by the inner thunk.

C Stack Usage

It is also possible for Guile to run out of space on the C stack. If you call a primitive procedure which then calls a Scheme procedure in a loop, you will consume C stack space. Guile tries to detect excessive consumption of C stack space, throwing an error when you have hit 80% of the process' available stack (as allocated by the operating system), or 160 kilowords in the absence of a strict limit.

For example, looping through `call-with-vm`, a primitive that calls a thunk, gives us the following:

```
scheme@(guile-user)> (use-modules (system vm vm))
scheme@(guile-user)> (let lp () (call-with-vm lp))
ERROR: Stack overflow
```

Unfortunately, that's all the information we get. Overrunning the C stack will throw an unwind-only exception, because it's not safe to do very much when you are close to the C stack limit.

If you get an error like this, you can either try rewriting your code to use less stack space, or increase the maximum stack size. To increase the maximum stack size, use `debug-set!`, for example:

```
(debug-set! stack 200000)
```

The next section describes `debug-set!` more thoroughly. Of course the best thing is to have your code operate without so much resource consumption by avoiding loops through C trampolines.

6.26.3.5 Debug options

The behavior of the `backtrace` procedure and of the default error handler can be parameterized via the debug options.

`debug-options` [*setting*] [Scheme Procedure]

Display the current settings of the debug options. If *setting* is omitted, only a short form of the current read options is printed. Otherwise if *setting* is the symbol `help`, a complete options description is displayed.

The set of available options, and their default values, may be had by invoking `debug-options` at the prompt.

```
scheme@(guile-user)>
backwards       no      Display backtrace in anti-chronological order.
width           79      Maximal width of backtrace.
depth           20      Maximal length of printed backtrace.
backtrace       yes     Show backtrace on error.
stack           1048576 Stack size limit (measured in words;
                        0 = no check).
show-file-name  #t      Show file names and line numbers in backtraces
                        when not `#f'.  A value of `base' displays only
                        base names, while `#t' displays full names.
warn-deprecated no      Warn when deprecated features are used.
```

The boolean options may be toggled with `debug-enable` and `debug-disable`. The non-boolean options must be set using `debug-set!`.

`debug-enable` *option-name* [Scheme Procedure]
`debug-disable` *option-name* [Scheme Procedure]
`debug-set!` *option-name value* [Scheme Syntax]

Modify the debug options. `debug-enable` should be used with boolean options and switches them on, `debug-disable` switches them off.

`debug-set!` can be used to set an option to a specific value. Due to historical oddities, it is a macro that expects an unquoted option name.

6.26.4 Traps

Guile's virtual machine can be configured to call out at key points to arbitrary user-specified procedures.

In principle, these *hooks* allow Scheme code to implement any model it chooses for examining the evaluation stack as program execution proceeds, and for suspending execution to be resumed later.

VM hooks are very low-level, though, and so Guile also has a library of higher-level *traps* on top of the VM hooks. A trap is an execution condition that, when fulfilled, will fire a handler. For example, Guile defines a trap that fires when control reaches a certain source location.

Finally, Guile also defines a third level of abstractions: per-thread *trap states*. A trap state exists to give names to traps, and to hold on to the set of traps so that they can be enabled, disabled, or removed. The trap state infrastructure defines the most useful abstractions for most cases. For example, Guile's REPL uses trap state functions to set breakpoints and tracepoints.

The following subsections describe all this in detail, for both the user wanting to use traps, and the developer interested in understanding how the interface hangs together.

6.26.4.1 VM Hooks

Everything that runs in Guile runs on its virtual machine, a C program that defines a number of operations that Scheme programs can perform.

Note that there are multiple VM "engines" for Guile. Only some of them have support for hooks compiled in. Normally the deal is that you get hooks if you are running interactively, and otherwise they are disabled, as they do have some overhead (about 10 or 20 percent).

To ensure that you are running with hooks, pass `--debug` to Guile when running your program, or otherwise use the `call-with-vm` and `set-vm-engine!` procedures to ensure that you are running in a VM with the `debug` engine.

To digress, Guile's VM has 6 different hooks (см. Раздел 6.11.6 [Hooks], страница 306) that can be fired at different times, which may be accessed with the following procedures.

The first argument of calls to these hooks is the frame in question. См. Раздел 6.26.1.3 [Frames], страница 497. Some hooks may call their procedures with more arguments. Since these hooks may be fired very frequently, Guile does a terrible thing: it allocates the frames on the C stack instead of the garbage-collected heap.

The upshot here is that the frames are only valid within the dynamic extent of the call to the hook. If a hook procedure keeps a reference to the frame outside the extent of the hook, bad things will happen.

The interface to hooks is provided by the `(system vm vm)` module:

```
(use-modules (system vm vm))
```

All of these functions implicitly act on the VM for the current thread only.

`vm-next-hook` [Scheme Procedure]
: The hook that will be fired before an instruction is retired (and executed).

`vm-push-continuation-hook` [Scheme Procedure]
: The hook that will be fired after preparing a new frame. Fires just before applying a procedure in a non-tail context, just before the corresponding apply-hook.

`vm-pop-continuation-hook` [Scheme Procedure]
: The hook that will be fired before returning from a frame.

  This hook fires with a variable number of arguments, corresponding to the values that the frame returns to its continuation.

`vm-apply-hook` [Scheme Procedure]
: The hook that will be fired before a procedure is applied. The frame's procedure will have already been set to the new procedure.

Note that procedure application is somewhat orthogonal to continuation pushes and pops. A non-tail call to a procedure will result first in a firing of the push-continuation hook, then this application hook, whereas a tail call will run without having fired a push-continuation hook.

`vm-abort-continuation-hook` [Scheme Procedure]
The hook that will be called after aborting to a prompt. См. Раздел 6.13.5 [Prompts], страница 323.

Like the pop-continuation hook, this hook fires with a variable number of arguments, corresponding to the values that returned to the continuation.

`vm-restore-continuation-hook` [Scheme Procedure]
The hook that will be called after restoring an undelimited continuation. Unfortunately it's not currently possible to introspect on the values that were given to the continuation.

These hooks do impose a performance penalty, if they are on. Obviously, the `vm-next-hook` has quite an impact, performance-wise. Therefore Guile exposes a single, heavy-handed knob to turn hooks on or off, the *VM trace level*. If the trace level is positive, hooks run; otherwise they don't.

For convenience, when the VM fires a hook, it does so with the trap level temporarily set to 0. That way the hooks don't fire while you're handling a hook. The trace level is restored to whatever it was once the hook procedure finishes.

`vm-trace-level` [Scheme Procedure]
Retrieve the "trace level" of the VM. If positive, the trace hooks associated with *vm* will be run. The initial trace level is 0.

`set-vm-trace-level!` *level* [Scheme Procedure]
Set the "trace level" of the VM.

См. Раздел 9.3 [A Virtual Machine for Guile], страница 844, for more information on Guile's virtual machine.

6.26.4.2 Trap Interface

The capabilities provided by hooks are great, but hooks alone rarely correspond to what users want to do.

For example, if a user wants to break when and if control reaches a certain source location, how do you do it? If you install a "next" hook, you get unacceptable overhead for the execution of the entire program. It would be possible to install an "apply" hook, then if the procedure encompasses those source locations, install a "next" hook, but already you're talking about one concept that might be implemented by a varying number of lower-level concepts.

It's best to be clear about things and define one abstraction for all such conditions: the *trap*.

Considering the myriad capabilities offered by the hooks though, there is only a minimum of functionality shared by all traps. Guile's current take is to reduce this to the

absolute minimum, and have the only standard interface of a trap be "turn yourself on" or "turn yourself off".

This interface sounds a bit strange, but it is useful to procedurally compose higher-level traps from lower-level building blocks. For example, Guile defines a trap that calls one handler when control enters a procedure, and another when control leaves the procedure. Given that trap, one can define a trap that adds to the next-hook only when within a given procedure. Building further, one can define a trap that fires when control reaches particular instructions within a procedure.

Or of course you can stop at any of these intermediate levels. For example, one might only be interested in calls to a given procedure. But the point is that a simple enable/disable interface is all the commonality that exists between the various kinds of traps, and furthermore that such an interface serves to allow "higher-level" traps to be composed from more primitive ones.

Specifically, a trap, in Guile, is a procedure. When a trap is created, by convention the trap is enabled; therefore, the procedure that is the trap will, when called, disable the trap, and return a procedure that will enable the trap, and so on.

Trap procedures take one optional argument: the current frame. (A trap may want to add to different sets of hooks depending on the frame that is current at enable-time.)

If this all sounds very complicated, it's because it is. Some of it is essential, but probably most of it is not. The advantage of using this minimal interface is that composability is more lexically apparent than when, for example, using a stateful interface based on GOOPS. But perhaps this reflects the cognitive limitations of the programmer who made the current interface more than anything else.

6.26.4.3 Low-Level Traps

To summarize the last sections, traps are enabled or disabled, and when they are enabled, they add to various VM hooks.

Note, however, that *traps do not increase the VM trace level*. So if you create a trap, it will be enabled, but unless something else increases the VM's trace level (см. Раздел 6.26.4.1 [VM Hooks], страница 508), the trap will not fire. It turns out that getting the VM trace level right is tricky without a global view of what traps are enabled. См. Раздел 6.26.4.5 [Trap States], страница 514, for Guile's answer to this problem.

Traps are created by calling procedures. Most of these procedures share a set of common keyword arguments, so rather than document them separately, we discuss them all together here:

`#:vm` The VM to instrument. Defaults to the current thread's VM.

`#:current-frame`
For traps that enable more hooks depending on their dynamic context, this argument gives the current frame that the trap is running in. Defaults to `#f`.

To have access to these procedures, you'll need to have imported the `(system vm traps)` module:

```
(use-modules (system vm traps))
```

`trap-at-procedure-call` *proc handler* **[***#:vm***]** [Scheme Procedure]
A trap that calls *handler* when *proc* is applied.

`trap-in-procedure` *proc enter-handler exit-handler* [Scheme Procedure]
[*#:current-frame***]** **[***#:vm***]**

A trap that calls *enter-handler* when control enters *proc*, and *exit-handler* when control leaves *proc*.

Control can enter a procedure via:

- A procedure call.
- A return to a procedure's frame on the stack.
- A continuation returning directly to an application of this procedure.

Control can leave a procedure via:

- A normal return from the procedure.
- An application of another procedure.
- An invocation of a continuation.
- An abort.

`trap-instructions-in-procedure` *proc next-handler* [Scheme Procedure]
exit-handler **[***#:current-frame***]** **[***#:vm***]**

A trap that calls *next-handler* for every instruction executed in *proc*, and *exit-handler* when execution leaves *proc*.

`trap-at-procedure-ip-in-range` *proc range handler* [Scheme Procedure]
[*#:current-frame***]** **[***#:vm***]**

A trap that calls *handler* when execution enters a range of instructions in *proc*. *range* is a simple of pairs, `((start . end) ...)`. The *start* addresses are inclusive, and *end* addresses are exclusive.

`trap-at-source-location` *file user-line handler* [Scheme Procedure]
[*#:current-frame***]** **[***#:vm***]**

A trap that fires when control reaches a given source location. The *user-line* parameter is one-indexed, as a user counts lines, instead of zero-indexed, as Guile counts lines.

`trap-frame-finish` *frame return-handler abort-handler* **[***#:vm***]** [Scheme Procedure]

A trap that fires when control leaves the given frame. *frame* should be a live frame in the current continuation. *return-handler* will be called on a normal return, and *abort-handler* on a nonlocal exit.

`trap-in-dynamic-extent` *proc enter-handler return-handler* [Scheme Procedure]
abort-handler **[***#:vm***]**

A more traditional dynamic-wind trap, which fires *enter-handler* when control enters *proc*, *return-handler* on a normal return, and *abort-handler* on a nonlocal exit.

Note that rewinds are not handled, so there is no rewind handler.

`trap-calls-in-dynamic-extent` *proc apply-handler* [Scheme Procedure]
return-handler **[***#:current-frame***]** **[***#:vm***]**

A trap that calls *apply-handler* every time a procedure is applied, and *return-handler* for returns, but only during the dynamic extent of an application of *proc*.

`trap-instructions-in-dynamic-extent` *proc next-handler* [Scheme Procedure]
[*#:current-frame***]** **[***#:vm***]**

A trap that calls *next-handler* for all retired instructions within the dynamic extent of a call to *proc*.

`trap-calls-to-procedure` *proc apply-handler return-handler* [Scheme Procedure]
[*#:vm***]**

A trap that calls *apply-handler* whenever *proc* is applied, and *return-handler* when it returns, but with an additional argument, the call depth.

That is to say, the handlers will get two arguments: the frame in question, and the call depth (a non-negative integer).

`trap-matching-instructions` *frame-pred handler* **[***#:vm***]** [Scheme Procedure]

A trap that calls *frame-pred* at every instruction, and if *frame-pred* returns a true value, calls *handler* on the frame.

6.26.4.4 Tracing Traps

The `(system vm trace)` module defines a number of traps for tracing of procedure applications. When a procedure is *traced*, it means that every call to that procedure is reported to the user during a program run. The idea is that you can mark a collection of procedures for tracing, and Guile will subsequently print out a line of the form

```
|  |  (procedure args ...)
```

whenever a marked procedure is about to be applied to its arguments. This can help a programmer determine whether a function is being called at the wrong time or with the wrong set of arguments.

In addition, the indentation of the output is useful for demonstrating how the traced applications are or are not tail recursive with respect to each other. Thus, a trace of a non-tail recursive factorial implementation looks like this:

```
scheme@(guile-user)> (define (fact1 n)
                       (if (zero? n) 1
                           (* n (fact1 (1- n)))))
scheme@(guile-user)> ,trace (fact1 4)
trace: (fact1 4)
trace: |  (fact1 3)
trace: |  |  (fact1 2)
trace: |  |  |  (fact1 1)
trace: |  |  |  |  (fact1 0)
trace: |  |  |  |  1
trace: |  |  |  1
trace: |  |  2
trace: |  6
trace: 24
```

While a typical tail recursive implementation would look more like this:

```
scheme@(guile-user)> (define (facti acc n)
                       (if (zero? n) acc
                           (facti (* n acc) (1- n))))
```

```
scheme@(guile-user)> (define (fact2 n) (facti 1 n))
scheme@(guile-user)> ,trace (fact2 4)
trace: (fact2 4)
trace: (facti 1 4)
trace: (facti 4 3)
trace: (facti 12 2)
trace: (facti 24 1)
trace: (facti 24 0)
trace: 24
```

The low-level traps below (см. Раздел 6.26.4.3 [Low-Level Traps], страница 510) share some common options:

`#:width` The maximum width of trace output. Trace printouts will try not to exceed this column, but for highly nested procedure calls, it may be unavoidable. Defaults to 80.

`#:vm` The VM on which to add the traps. Defaults to the current thread's VM.

`#:prefix` A string to print out before each trace line. As seen above in the examples, defaults to `"trace: "`.

To have access to these procedures, you'll need to have imported the `(system vm trace)` module:

```
(use-modules (system vm trace))
```

`trace-calls-to-procedure` *proc* [*#:width*] [*#:vm*] [*#:prefix*] [Scheme Procedure]
Print a trace at applications of and returns from *proc*.

`trace-calls-in-procedure` *proc* [*#:width*] [*#:vm*] [*#:prefix*] [Scheme Procedure]
Print a trace at all applications and returns within the dynamic extent of calls to *proc*.

`trace-instructions-in-procedure` *proc* [*#:width*] [*#:vm*] [Scheme Procedure]
Print a trace at all instructions executed in the dynamic extent of calls to *proc*.

In addition, Guile defines a procedure to call a thunk, tracing all procedure calls and returns within the thunk.

`call-with-trace` *thunk* [*#:calls?=#t*] [*#:instructions?=#f*] [*#:width=80*] [Scheme Procedure]
Call *thunk*, tracing all execution within its dynamic extent.

If *calls?* is true, Guile will print a brief report at each procedure call and return, as given above.

If *instructions?* is true, Guile will also print a message each time an instruction is executed. This is a lot of output, but it is sometimes useful when doing low-level optimization.

Note that because this procedure manipulates the VM trace level directly, it doesn't compose well with traps at the REPL.

См. Раздел 4.4.4.5 [Profile Commands], страница 55, for more information on tracing at the REPL.

6.26.4.5 Trap States

When multiple traps are present in a system, we begin to have a bookkeeping problem. How are they named? How does one disable, enable, or delete them?

Guile's answer to this is to keep an implicit per-thread *trap state*. The trap state object is not exposed to the user; rather, API that works on trap states fetches the current trap state from the dynamic environment.

Traps are identified by integers. A trap can be enabled, disabled, or removed, and can have an associated user-visible name.

These procedures have their own module:

```
(use-modules (system vm trap-state))
```

`add-trap!` *trap name* [Scheme Procedure]
Add a trap to the current trap state, associating the given *name* with it. Returns a fresh trap identifier (an integer).

Note that usually the more specific functions detailed in Раздел 6.26.4.6 [High-Level Traps], страница 514, are used in preference to this one.

`list-traps` [Scheme Procedure]
List the current set of traps, both enabled and disabled. Returns a list of integers.

`trap-name` *idx* [Scheme Procedure]
Returns the name associated with trap *idx*, or `#f` if there is no such trap.

`trap-enabled?` *idx* [Scheme Procedure]
Returns `#t` if trap *idx* is present and enabled, or `#f` otherwise.

`enable-trap!` *idx* [Scheme Procedure]
Enables trap *idx*.

`disable-trap!` *idx* [Scheme Procedure]
Disables trap *idx*.

`delete-trap!` *idx* [Scheme Procedure]
Removes trap *idx*, disabling it first, if necessary.

6.26.4.6 High-Level Traps

The low-level trap API allows one to make traps that call procedures, and the trap state API allows one to keep track of what traps are there. But neither of these APIs directly helps you when you want to set a breakpoint, because it's unclear what to do when the trap fires. Do you enter a debugger, or mail a summary of the situation to your great-aunt, or what?

So for the common case in which you just want to install breakpoints, and then have them all result in calls to one parameterizable procedure, we have the high-level trap interface.

Perhaps we should have started this section with this interface, as it's clearly the one most people should use. But as its capabilities and limitations proceed from the lower

layers, we felt that the character-building exercise of building a mental model might be helpful.

These procedures share a module with trap states:

```
(use-modules (system vm trap-state))
```

`with-default-trap-handler` *handler thunk* [Scheme Procedure]
Call *thunk* in a dynamic context in which *handler* is the current trap handler.

Additionally, during the execution of *thunk*, the VM trace level (см. Раздел 6.26.4.1 [VM Hooks], страница 508) is set to the number of enabled traps. This ensures that traps will in fact fire.

handler may be `#f`, in which case VM hooks are not enabled as they otherwise would be, as there is nothing to handle the traps.

The trace-level-setting behavior of `with-default-trap-handler` is one of its more useful aspects, but if you are willing to forgo that, and just want to install a global trap handler, there's a function for that too:

`install-trap-handler!` *handler* [Scheme Procedure]
Set the current thread's trap handler to *handler*.

Trap handlers are called when traps installed by procedures from this module fire. The current "consumer" of this API is Guile's REPL, but one might easily imagine other trap handlers being used to integrate with other debugging tools.

`add-trap-at-procedure-call!` *proc* [Scheme Procedure]
Install a trap that will fire when *proc* is called.

This is a breakpoint.

`add-trace-at-procedure-call!` *proc* [Scheme Procedure]
Install a trap that will print a tracing message when *proc* is called. См. Раздел 6.26.4.4 [Tracing Traps], страница 512, for more information.

This is a tracepoint.

`add-trap-at-source-location!` *file user-line* [Scheme Procedure]
Install a trap that will fire when control reaches the given source location. *user-line* is one-indexed, as users count lines, instead of zero-indexed, as Guile counts lines.

This is a source breakpoint.

`add-ephemeral-trap-at-frame-finish!` *frame handler* [Scheme Procedure]
Install a trap that will call *handler* when *frame* finishes executing. The trap will be removed from the trap state after firing, or on nonlocal exit.

This is a finish trap, used to implement the "finish" REPL command.

`add-ephemeral-stepping-trap!` *frame handler* **[***#:into?***]** **[***#:instruction?***]** [Scheme Procedure]
Install a trap that will call *handler* after stepping to a different source line or instruction. The trap will be removed from the trap state after firing, or on nonlocal exit.

If *instruction?* is false (the default), the trap will fire when control reaches a new source line. Otherwise it will fire when control reaches a new instruction.

Additionally, if *into?* is false (not the default), the trap will only fire for frames at or prior to the given frame. If *into?* is true (the default), the trap may step into nested procedure invocations.

This is a stepping trap, used to implement the "step", "next", "step-instruction", and "next-instruction" REPL commands.

6.26.5 GDB Support

Sometimes, you may find it necessary to debug Guile applications at the C level. Doing so can be tedious, in particular because the debugger is oblivious to Guile's `SCM` type, and thus unable to display `SCM` values in any meaningful way:

```
(gdb) frame
#0  scm_display (obj=0xf04310, port=0x6f9f30) at print.c:1437
```

To address that, Guile comes with an extension of the GNU Debugger (GDB) that contains a "pretty-printer" for `SCM` values. With this GDB extension, the C frame in the example above shows up like this:

```
(gdb) frame
#0  scm_display (obj=("hello" GDB!), port=#<port file 6f9f30>) at print.c:1437
```

Here GDB was able to decode the list pointed to by *obj*, and to print it using Scheme's read syntax.

That extension is a `.scm` file installed alongside the `libguile` shared library. When GDB 7.8 or later is installed and compiled with support for extensions written in Guile, the extension is automatically loaded when debugging a program linked against `libguile` (см. Раздел "Auto-loading" в *Debugging with GDB*). Note that the directory where `libguile` is installed must be among GDB's auto-loading "safe directories" (см. Раздел "Auto-loading safe path" в *Debugging with GDB*).

6.27 Code Coverage Reports

When writing a test suite for a program or library, it is desirable to know what part of the code is *covered* by the test suite. The `(system vm coverage)` module provides tools to gather code coverage data and to present them, as detailed below.

`with-code-coverage` *thunk* [Scheme Procedure]

Run *thunk*, a zero-argument procedure, while instrumenting Guile's virtual machine to collect code coverage data. Return code coverage data and the values returned by *thunk*.

`coverage-data?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a *coverage data* object as returned by `with-code-coverage`.

`coverage-data->lcov` *data port #:key modules* [Scheme Procedure]

Traverse code coverage information *data*, as obtained with `with-code-coverage`, and write coverage information to port in the `.info` format used by LCOV (`http://ltp.sourceforge.net/coverage/lcov.php`). The report will include all of *modules* (or, by default, all the currently loaded modules) even if their code was not executed.

The generated data can be fed to LCOV's `genhtml` command to produce an HTML report, which aids coverage data visualization.

Here's an example use:

```
(use-modules (system vm coverage)
             (system vm vm))

(call-with-values (lambda ()
                    (with-code-coverage
                      (lambda ()
                        (do-something-tricky))))
  (lambda (data result)
    (let ((port (open-output-file "lcov.info")))
      (coverage-data->lcov data port)
      (close file))))
```

In addition, the module provides low-level procedures that would make it possible to write other user interfaces to the coverage data.

`instrumented-source-files` *data* [Scheme Procedures]

Return the list of "instrumented" source files, i.e., source files whose code was loaded at the time *data* was collected.

`line-execution-counts` *data file* [Scheme Procedures]

Return a list of line number/execution count pairs for *file*, or `#f` if *file* is not among the files covered by *data*. This includes lines with zero count.

`instrumented/executed-lines` *data file* [Scheme Procedures]

Return the number of instrumented and the number of executed source lines in *file* according to *data*.

`procedure-execution-count` *data proc* [Scheme Procedures]

Return the number of times *proc*'s code was executed, according to *data*, or `#f` if *proc* was not executed. When *proc* is a closure, the number of times its code was executed is returned, not the number of times this code associated with this particular closure was executed.

7 Guile Modules

7.1 SLIB

SLIB is a portable library of Scheme packages which can be used with Guile and other Scheme implementations. SLIB is not included in the Guile distribution, but can be installed separately (см. Раздел 7.1.1 [SLIB installation], страница 519). It is available from `http://people.csail.mit.edu/jaffer/SLIB.html`.

After SLIB is installed, the following Scheme expression must be executed before the SLIB facilities can be used:

```
(use-modules (ice-9 slib))
```

`require` can then be used in the usual way (см. Раздел "Require" в *The SLIB Manual*). For example,

```
(use-modules (ice-9 slib))
(require 'primes)
(prime? 13)
⇒ #t
```

A few Guile core functions are overridden by the SLIB setups; for example the SLIB version of `delete-file` returns a boolean indicating success or failure, whereas the Guile core version throws an error for failure. In general (and as might be expected) when SLIB is loaded it's the SLIB specifications that are followed.

7.1.1 SLIB installation

The following procedure works, e.g., with SLIB version 3a3 (см. Раздел "Installation" в *The SLIB Portable Scheme Library*):

1. Unpack SLIB and install it using `make install` from its directory. By default, this will install SLIB in `/usr/local/lib/slib/`. Running `make install-info` installs its documentation, by default under `/usr/local/info/`.
2. Define the `SCHEME_LIBRARY_PATH` environment variable:

   ```
   $ SCHEME_LIBRARY_PATH=/usr/local/lib/slib/
   $ export SCHEME_LIBRARY_PATH
   ```

   Alternatively, you can create a symlink in the Guile directory to SLIB, e.g.:

   ```
   ln -s /usr/local/lib/slib /usr/local/share/guile/2.2/slib
   ```
3. Use Guile to create the catalog file, e.g.,:

   ```
   # guile
   guile> (use-modules (ice-9 slib))
   guile> (require 'new-catalog)
   guile> (quit)
   ```

   The catalog data should now be in `/usr/local/share/guile/2.2/slibcat`.

   If instead you get an error such as:

   ```
   Unbound variable: scheme-implementation-type
   ```

   then a solution is to get a newer version of Guile, or to modify `ice-9/slib.scm` to use `define-public` for the offending variables.

7.1.2 JACAL

Jacal is a symbolic math package written in Scheme by Aubrey Jaffer. It is usually installed as an extra package in SLIB.

You can use Guile's interface to SLIB to invoke Jacal:

```
(use-modules (ice-9 slib))
(slib:load "math")
(math)
```

For complete documentation on Jacal, please read the Jacal manual. If it has been installed on line, you can look at Раздел "Jacal" в *JACAL Symbolic Mathematics System*. Otherwise you can find it on the web at `http://www-swiss.ai.mit.edu/~jaffer/JACAL.html`

7.2 POSIX System Calls and Networking

7.2.1 POSIX Interface Conventions

These interfaces provide access to operating system facilities. They provide a simple wrapping around the underlying C interfaces to make usage from Scheme more convenient. They are also used to implement the Guile port of scsh (см. Раздел 7.17 [The Scheme shell (scsh)], страница 756).

Generally there is a single procedure for each corresponding Unix facility. There are some exceptions, such as procedures implemented for speed and convenience in Scheme with no primitive Unix equivalent, e.g. `copy-file`.

The interfaces are intended as far as possible to be portable across different versions of Unix. In some cases procedures which can't be implemented on particular systems may become no-ops, or perform limited actions. In other cases they may throw errors.

General naming conventions are as follows:

- The Scheme name is often identical to the name of the underlying Unix facility.
- Underscores in Unix procedure names are converted to hyphens.
- Procedures which destructively modify Scheme data have exclamation marks appended, e.g., `recv!`.
- Predicates (returning only `#t` or `#f`) have question marks appended, e.g., `access?`.
- Some names are changed to avoid conflict with dissimilar interfaces defined by scsh, e.g., `primitive-fork`.
- Unix preprocessor names such as `EPERM` or `R_OK` are converted to Scheme variables of the same name (underscores are not replaced with hyphens).

Unexpected conditions are generally handled by raising exceptions. There are a few procedures which return a special value if they don't succeed, e.g., `getenv` returns `#f` if it the requested string is not found in the environment. These cases are noted in the documentation.

For ways to deal with exceptions, see Раздел 6.13.8 [Exceptions], страница 332.

Errors which the C library would report by returning a null pointer or through some other means are reported by raising a `system-error` exception with `scm-error` (см.

Раздел 6.13.9 [Error Reporting], страница 338). The *data* parameter is a list containing the Unix `errno` value (an integer). For example,

```
(define (my-handler key func fmt fmtargs data)
  (display key) (newline)
  (display func) (newline)
  (apply format #t fmt fmtargs) (newline)
  (display data) (newline))

(catch 'system-error
  (lambda () (dup2 -123 -456))
  my-handler)

⊣
system-error
dup2
Bad file descriptor
(9)
```

`system-error-errno` *arglist* [Функция]

Return the `errno` value from a list which is the arguments to an exception handler. If the exception is not a `system-error`, then the return is `#f`. For example,

```
(catch
 'system-error
 (lambda ()
   (mkdir "/this-ought-to-fail-if-I'm-not-root"))
 (lambda stuff
   (let ((errno (system-error-errno stuff)))
     (cond
      ((= errno EACCES)
       (display "You're not allowed to do that."))
      ((= errno EEXIST)
       (display "Already exists."))
      (#t
       (display (strerror errno))))
     (newline))))
```

7.2.2 Ports and File Descriptors

Conventions generally follow those of scsh, Раздел 7.17 [The Scheme shell (scsh)], страница 756.

Each open file port has an associated operating system file descriptor. File descriptors are generally not useful in Scheme programs; however they may be needed when interfacing with foreign code and the Unix environment.

A file descriptor can be extracted from a port and a new port can be created from a file descriptor. However a file descriptor is just an integer and the garbage collector doesn't recognize it as a reference to the port. If all other references to the port were dropped, then

it's likely that the garbage collector would free the port, with the side-effect of closing the file descriptor prematurely.

To assist the programmer in avoiding this problem, each port has an associated *revealed count* which can be used to keep track of how many times the underlying file descriptor has been stored in other places. If a port's revealed count is greater than zero, the file descriptor will not be closed when the port is garbage collected. A programmer can therefore ensure that the revealed count will be greater than zero if the file descriptor is needed elsewhere.

For the simple case where a file descriptor is "imported" once to become a port, it does not matter if the file descriptor is closed when the port is garbage collected. There is no need to maintain a revealed count. Likewise when "exporting" a file descriptor to the external environment, setting the revealed count is not required provided the port is kept open (i.e., is pointed to by a live Scheme binding) while the file descriptor is in use.

To correspond with traditional Unix behaviour, three file descriptors (0, 1, and 2) are automatically imported when a program starts up and assigned to the initial values of the current/standard input, output, and error ports, respectively. The revealed count for each is initially set to one, so that dropping references to one of these ports will not result in its garbage collection: it could be retrieved with `fdopen` or `fdes->ports`.

Guile's ports can be buffered. This means that writing a byte to a file port goes to the internal buffer first, and only when the buffer is full (or the user invokes `force-output` on the port) is the data actually written to the file descriptor. Likewise on input, bytes are read in from the file descriptor in blocks and placed in a buffer. Reading a character via `read-char` first goes to the buffer, filling it as needed. Usually read buffering is more or less transparent, but write buffering can sometimes cause writes to be delayed unexpectedly, if you forget to call `force-output`. См. Раздел 6.14.6 [Buffering], страница 360, for more on how to control port buffers.

Note however that some procedures (e.g., `recv!`) will accept ports as arguments, but will actually operate directly on the file descriptor underlying the port. Any port buffering is ignored, including the buffer which implements `peek-char` and `unread-char`.

`port-revealed` *port* [Scheme Procedure]
`scm_port_revealed` (*port*) [C Function]
Return the revealed count for *port*.

`set-port-revealed!` *port rcount* [Scheme Procedure]
`scm_set_port_revealed_x` (*port*, *rcount*) [C Function]
Sets the revealed count for a *port* to *rcount*. The return value is unspecified.

`fileno` *port* [Scheme Procedure]
`scm_fileno` (*port*) [C Function]
Return the integer file descriptor underlying *port*. Does not change its revealed count.

`port->fdes` *port* [Scheme Procedure]
Returns the integer file descriptor underlying *port*. As a side effect the revealed count of *port* is incremented.

`fdopen` *fdes modes* [Scheme Procedure]
`scm_fdopen` (*fdes*, *modes*) [C Function]
Return a new port based on the file descriptor *fdes*. Modes are given by the string *modes*. The revealed count of the port is initialized to zero. The *modes* string is the same as that accepted by `open-file` (см. Раздел 6.14.10.1 [File Ports], страница 365).

`fdes->ports` *fdes* [Scheme Procedure]
`scm_fdes_to_ports` (*fdes*) [C Function]
Return a list of existing ports which have *fdes* as an underlying file descriptor, without changing their revealed counts.

`fdes->inport` *fdes* [Scheme Procedure]
Returns an existing input port which has *fdes* as its underlying file descriptor, if one exists, and increments its revealed count. Otherwise, returns a new input port with a revealed count of 1.

`fdes->outport` *fdes* [Scheme Procedure]
Returns an existing output port which has *fdes* as its underlying file descriptor, if one exists, and increments its revealed count. Otherwise, returns a new output port with a revealed count of 1.

`primitive-move->fdes` *port fdes* [Scheme Procedure]
`scm_primitive_move_to_fdes` (*port*, *fdes*) [C Function]
Moves the underlying file descriptor for *port* to the integer value *fdes* without changing the revealed count of *port*. Any other ports already using this descriptor will be automatically shifted to new descriptors and their revealed counts reset to zero. The return value is `#f` if the file descriptor already had the required value or `#t` if it was moved.

`move->fdes` *port fdes* [Scheme Procedure]
Moves the underlying file descriptor for *port* to the integer value *fdes* and sets its revealed count to one. Any other ports already using this descriptor will be automatically shifted to new descriptors and their revealed counts reset to zero. The return value is unspecified.

`release-port-handle` *port* [Scheme Procedure]
Decrements the revealed count for a port.

`fsync` *port_or_fd* [Scheme Procedure]
`scm_fsync` (*port_or_fd*) [C Function]
Copies any unwritten data for the specified output file descriptor to disk. If *port_or_fd* is a port, its buffer is flushed before the underlying file descriptor is fsync'd. The return value is unspecified.

`open` *path flags* [*mode*] [Scheme Procedure]
`scm_open` (*path*, *flags*, *mode*) [C Function]
Open the file named by *path* for reading and/or writing. *flags* is an integer specifying how the file should be opened. *mode* is an integer specifying the permission bits of

the file, if it needs to be created, before the umask (см. Раздел 7.2.7 [Processes], страница 542) is applied. The default is 666 (Unix itself has no default).

flags can be constructed by combining variables using `logior`. Basic flags are:

`O_RDONLY` [Переменная]
Open the file read-only.

`O_WRONLY` [Переменная]
Open the file write-only.

`O_RDWR` [Переменная]
Open the file read/write.

`O_APPEND` [Переменная]
Append to the file instead of truncating.

`O_CREAT` [Переменная]
Create the file if it does not already exist.

См. Раздел "File Status Flags" в *The GNU C Library Reference Manual*, for additional flags.

`open-fdes` *path flags* [*mode*] [Scheme Procedure]
`scm_open_fdes` (*path, flags, mode*) [C Function]
Similar to `open` but return a file descriptor instead of a port.

`close` *fd_or_port* [Scheme Procedure]
`scm_close` (*fd_or_port*) [C Function]
Similar to `close-port` (см. Раздел 6.14.1 [Ports], страница 352), but also works on file descriptors. A side effect of closing a file descriptor is that any ports using that file descriptor are moved to a different file descriptor and have their revealed counts set to zero.

`close-fdes` *fd* [Scheme Procedure]
`scm_close_fdes` (*fd*) [C Function]
A simple wrapper for the `close` system call. Close file descriptor *fd*, which must be an integer. Unlike `close`, the file descriptor will be closed even if a port is using it. The return value is unspecified.

`pipe` [Scheme Procedure]
`scm_pipe` () [C Function]
Return a newly created pipe: a pair of ports which are linked together on the local machine. The CAR is the input port and the CDR is the output port. Data written (and flushed) to the output port can be read from the input port. Pipes are commonly used for communication with a newly forked child process. The need to flush the output port can be avoided by making it unbuffered using `setvbuf` (см. Раздел 6.14.6 [Buffering], страница 360).

`PIPE_BUF` [Переменная]
A write of up to `PIPE_BUF` many bytes to a pipe is atomic, meaning when done it goes into the pipe instantaneously and as a contiguous block (см. Раздел "Atomicity of Pipe I/O" в *The GNU C Library Reference Manual*).

Note that the output port is likely to block if too much data has been written but not yet read from the input port. Typically the capacity is `PIPE_BUF` bytes.

The next group of procedures perform a `dup2` system call, if *newfd* (an integer) is supplied, otherwise a `dup`. The file descriptor to be duplicated can be supplied as an integer or contained in a port. The type of value returned varies depending on which procedure is used.

All procedures also have the side effect when performing `dup2` that any ports using *newfd* are moved to a different file descriptor and have their revealed counts set to zero.

`dup->fdes` *fd_or_port* **[***fd***]** [Scheme Procedure]
`scm_dup_to_fdes` (*fd_or_port*, *fd*) [C Function]
Return a new integer file descriptor referring to the open file designated by *fd_or_port*, which must be either an open file port or a file descriptor.

`dup->inport` *port/fd* **[***newfd***]** [Scheme Procedure]
Returns a new input port using the new file descriptor.

`dup->outport` *port/fd* **[***newfd***]** [Scheme Procedure]
Returns a new output port using the new file descriptor.

`dup` *port/fd* **[***newfd***]** [Scheme Procedure]
Returns a new port if *port/fd* is a port, with the same mode as the supplied port, otherwise returns an integer file descriptor.

`dup->port` *port/fd mode* **[***newfd***]** [Scheme Procedure]
Returns a new port using the new file descriptor. *mode* supplies a mode string for the port (см. Раздел 6.14.10.1 [File Ports], страница 365).

`duplicate-port` *port modes* [Scheme Procedure]
Returns a new port which is opened on a duplicate of the file descriptor underlying *port*, with mode string *modes* as for Раздел 6.14.10.1 [File Ports], страница 365. The two ports will share a file position and file status flags.

Unexpected behaviour can result if both ports are subsequently used and the original and/or duplicate ports are buffered. The mode string can include `0` to obtain an unbuffered duplicate port.

This procedure is equivalent to `(dup->port` *`port modes`*`)`.

`redirect-port` *old_port new_port* [Scheme Procedure]
`scm_redirect_port` (*old_port*, *new_port*) [C Function]
This procedure takes two ports and duplicates the underlying file descriptor from *old_port* into *new_port*. The current file descriptor in *new_port* will be closed. After the redirection the two ports will share a file position and file status flags.

The return value is unspecified.

Unexpected behaviour can result if both ports are subsequently used and the original and/or duplicate ports are buffered.

This procedure does not have any side effects on other ports or revealed counts.

`dup2` *oldfd newfd* [Scheme Procedure]
`scm_dup2` (*oldfd, newfd*) [C Function]

A simple wrapper for the `dup2` system call. Copies the file descriptor *oldfd* to descriptor number *newfd*, replacing the previous meaning of *newfd*. Both *oldfd* and *newfd* must be integers. Unlike for `dup->fdes` or `primitive-move->fdes`, no attempt is made to move away ports which are using *newfd*. The return value is unspecified.

`port-for-each` *proc* [Scheme Procedure]
`scm_port_for_each` (*SCM proc*) [C Function]
`scm_c_port_for_each` (*void* (**proc*)(*void *, SCM*), *void *data*) [C Function]

Apply *proc* to each port in the Guile port table (FIXME: what is the Guile port table?) in turn. The return value is unspecified. More specifically, *proc* is applied exactly once to every port that exists in the system at the time `port-for-each` is invoked. Changes to the port table while `port-for-each` is running have no effect as far as `port-for-each` is concerned.

The C function `scm_port_for_each` takes a Scheme procedure encoded as a `SCM` value, while `scm_c_port_for_each` takes a pointer to a C function and passes along a arbitrary *data* cookie.

`fcntl` *port/fd cmd* [*value*] [Scheme Procedure]
`scm_fcntl` (*object, cmd, value*) [C Function]

Apply *cmd* on *port/fd*, either a port or file descriptor. The *value* argument is used by the `SET` commands described below, it's an integer value.

Values for *cmd* are:

`F_DUPFD` [Переменная]

Duplicate the file descriptor, the same as `dup->fdes` above does.

`F_GETFD` [Переменная]
`F_SETFD` [Переменная]

Get or set flags associated with the file descriptor. The only flag is the following,

`FD_CLOEXEC` [Переменная]

"Close on exec", meaning the file descriptor will be closed on an `exec` call (a successful such call). For example to set that flag,

```
(fcntl port F_SETFD FD_CLOEXEC)
```

Or better, set it but leave any other possible future flags unchanged,

```
(fcntl port F_SETFD (logior FD_CLOEXEC
                            (fcntl port F_GETFD)))
```

`F_GETFL` [Переменная]
`F_SETFL` [Переменная]

Get or set flags associated with the open file. These flags are `O_RDONLY` etc described under `open` above.

A common use is to set `O_NONBLOCK` on a network socket. The following sets that flag, and leaves other flags unchanged.

```
(fcntl sock F_SETFL (logior O_NONBLOCK
                            (fcntl sock F_GETFL)))
```

`F_GETOWN` [Переменная]
`F_SETOWN` [Переменная]
Get or set the process ID of a socket's owner, for `SIGIO` signals.

`flock` *file operation* [Scheme Procedure]
`scm_flock` (*file, operation*) [C Function]
Apply or remove an advisory lock on an open file. *operation* specifies the action to be done:

`LOCK_SH` [Переменная]
Shared lock. More than one process may hold a shared lock for a given file at a given time.

`LOCK_EX` [Переменная]
Exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.

`LOCK_UN` [Переменная]
Unlock the file.

`LOCK_NB` [Переменная]
Don't block when locking. This is combined with one of the other operations using `logior` (см. Раздел 6.6.2.13 [Bitwise Operations], страница 134). If `flock` would block an `EWOULDBLOCK` error is thrown (см. Раздел 7.2.1 [Conventions], страница 520).

The return value is not specified. *file* may be an open file descriptor or an open file descriptor port.

Note that `flock` does not lock files across NFS.

`select` *reads writes excepts* **[***secs* **[***usecs***]]** [Scheme Procedure]
`scm_select` (*reads, writes, excepts, secs, usecs*) [C Function]
This procedure has a variety of uses: waiting for the ability to provide input, accept output, or the existence of exceptional conditions on a collection of ports or file descriptors, or waiting for a timeout to occur.

When an error occurs, this procedure throws a `system-error` exception (см. Раздел 7.2.1 [Conventions], страница 520). Note that `select` may return early for other reasons, for example due to pending interrupts. См. Раздел 6.22.3 [Asyncs], страница 466, for more on interrupts.

reads, *writes* and *excepts* can be lists or vectors, with each member a port or a file descriptor. The value returned is a list of three corresponding lists or vectors containing only the members which meet the specified requirement. The ability of port buffers to provide input or accept output is taken into account. Ordering of the input lists or vectors is not preserved.

The optional arguments *secs* and *usecs* specify the timeout. Either *secs* can be specified alone, as either an integer or a real number, or both *secs* and *usecs* can be specified as integers, in which case *usecs* is an additional timeout expressed in microseconds. If *secs* is omitted or is `#f` then select will wait for as long as it takes for one of the other conditions to be satisfied.

The scsh version of `select` differs as follows: Only vectors are accepted for the first three arguments. The *usecs* argument is not supported. Multiple values are returned instead of a list. Duplicates in the input vectors appear only once in output. An additional `select!` interface is provided.

While it is sometimes necessary to operate at the level of file descriptors, this is an operation whose correctness can only be considered as part of a whole program. So for example while the effects of `(string-set! x 34 #\y)` are limited to the bits of code that can access *x*, `(close-fdes 34)` mutates the state of the entire process. In particular if another thread is using file descriptor 34 then their state might be corrupted; and another thread which opens a file might cause file descriptor 34 to be re-used, so that corruption could manifest itself in a strange way.

However when working with file descriptors, it's common to want to associate information with the file descriptor, perhaps in a side table. To support this use case and to allow user code to remove an association when a file descriptor is closed, Guile offers *fdes finalizers*.

As the name indicates, fdes finalizers are finalizers – they can run in response to garbage collection, and they can also run in response to explicit calls to `close-port`, `close-fdes`, or the like. As such they inherit many of the pitfalls of finalizers: they may be invoked from concurrent threads, or not at all. См. Раздел 5.5.4 [Foreign Object Memory Management], страница 82, for more on finalizers.

To use fdes finalizers, import their module;

```
(use-modules (ice-9 fdes-finalizers))
```

`add-fdes-finalizer!` *fdes finalizer* [Scheme Procedure]
`remove-fdes-finalizer!` *fdes finalizer* [Scheme Procedure]

Add or remove a finalizer for *fdes*. A finalizer is a procedure that is called by Guile when a file descriptor is closed. The file descriptor being closed is passed as the one argument to the finalizer. If a finalizer has been added multiple times to a file descriptor, to remove it would require that number of calls to `remove-fdes-finalizer!`.

The finalizers added to a file descriptor are called by Guile in an unspecified order, and their return values are ignored.

7.2.3 File System

These procedures allow querying and setting file system attributes (such as owner, permissions, sizes and types of files); deleting, copying, renaming and linking files; creating and removing directories and querying their contents; syncing the file system and creating special files.

`access?` *path how* [Scheme Procedure]
`scm_access` (*path, how*) [C Function]

Test accessibility of a file under the real UID and GID of the calling process. The return is `#t` if *path* exists and the permissions requested by *how* are all allowed, or `#f` if not.

how is an integer which is one of the following values, or a bitwise-OR (`logior`) of multiple values.

`R_OK` [Переменная]
Test for read permission.

`W_OK` [Переменная]
Test for write permission.

`X_OK` [Переменная]
Test for execute permission.

`F_OK` [Переменная]
Test for existence of the file. This is implied by each of the other tests, so there's no need to combine it with them.

It's important to note that `access?` does not simply indicate what will happen on attempting to read or write a file. In normal circumstances it does, but in a set-UID or set-GID program it doesn't because `access?` tests the real ID, whereas an open or execute attempt uses the effective ID.

A program which will never run set-UID/GID can ignore the difference between real and effective IDs, but for maximum generality, especially in library functions, it's best not to use `access?` to predict the result of an open or execute, instead simply attempt that and catch any exception.

The main use for `access?` is to let a set-UID/GID program determine what the invoking user would have been allowed to do, without the greater (or perhaps lesser) privileges afforded by the effective ID. For more on this, see Раздел "Testing File Access" в *The GNU C Library Reference Manual*.

`stat` *object* [Scheme Procedure]
`scm_stat` (*object*) [C Function]
Return an object containing various information about the file determined by *object*. *object* can be a string containing a file name or a port or integer file descriptor which is open on a file (in which case `fstat` is used as the underlying system call).

The object returned by `stat` can be passed as a single parameter to the following procedures, all of which return integers:

`stat:dev` *st* [Scheme Procedure]
The device number containing the file.

`stat:ino` *st* [Scheme Procedure]
The file serial number, which distinguishes this file from all other files on the same device.

`stat:mode` *st* [Scheme Procedure]
The mode of the file. This is an integer which incorporates file type information and file permission bits. See also `stat:type` and `stat:perms` below.

`stat:nlink` *st* [Scheme Procedure]
The number of hard links to the file.

`stat:uid` *st* [Scheme Procedure]
The user ID of the file's owner.

`stat:gid` *st* [Scheme Procedure]
The group ID of the file.

`stat:rdev` *st* [Scheme Procedure]
Device ID; this entry is defined only for character or block special files. On some systems this field is not available at all, in which case `stat:rdev` returns `#f`.

`stat:size` *st* [Scheme Procedure]
The size of a regular file in bytes.

`stat:atime` *st* [Scheme Procedure]
The last access time for the file, in seconds.

`stat:mtime` *st* [Scheme Procedure]
The last modification time for the file, in seconds.

`stat:ctime` *st* [Scheme Procedure]
The last modification time for the attributes of the file, in seconds.

`stat:atimensec` *st* [Scheme Procedure]
`stat:mtimensec` *st* [Scheme Procedure]
`stat:ctimensec` *st* [Scheme Procedure]
The fractional part of a file's access, modification, or attribute modification time, in nanoseconds. Nanosecond timestamps are only available on some operating systems and file systems. If Guile cannot retrieve nanosecond-level timestamps for a file, these fields will be set to 0.

`stat:blksize` *st* [Scheme Procedure]
The optimal block size for reading or writing the file, in bytes. On some systems this field is not available, in which case `stat:blksize` returns a sensible suggested block size.

`stat:blocks` *st* [Scheme Procedure]
The amount of disk space that the file occupies measured in units of 512 byte blocks. On some systems this field is not available, in which case `stat:blocks` returns `#f`.

In addition, the following procedures return the information from `stat:mode` in a more convenient form:

`stat:type` *st* [Scheme Procedure]
A symbol representing the type of file. Possible values are '`regular`', '`directory`', '`symlink`', '`block-special`', '`char-special`', '`fifo`', '`socket`', and '`unknown`'.

`stat:perms` *st* [Scheme Procedure]
An integer representing the access permission bits.

`lstat` *path* [Scheme Procedure]
`scm_lstat` (*path*) [C Function]
Similar to `stat`, but does not follow symbolic links, i.e., it will return information about a symbolic link itself, not the file it points to. *path* must be a string.

`readlink` *path* [Scheme Procedure]
`scm_readlink` (*path*) [C Function]
Return the value of the symbolic link named by *path* (a string), i.e., the file that the link points to.

`chown` *object owner group* [Scheme Procedure]
`scm_chown` (*object, owner, group*) [C Function]
Change the ownership and group of the file referred to by *object* to the integer values *owner* and *group*. *object* can be a string containing a file name or, if the platform supports `fchown` (см. Раздел "File Owner" в *The GNU C Library Reference Manual*), a port or integer file descriptor which is open on the file. The return value is unspecified.

If *object* is a symbolic link, either the ownership of the link or the ownership of the referenced file will be changed depending on the operating system (lchown is unsupported at present). If *owner* or *group* is specified as `-1`, then that ID is not changed.

`chmod` *object mode* [Scheme Procedure]
`scm_chmod` (*object, mode*) [C Function]
Changes the permissions of the file referred to by *object*. *object* can be a string containing a file name or a port or integer file descriptor which is open on a file (in which case `fchmod` is used as the underlying system call). *mode* specifies the new permissions as a decimal number, e.g., `(chmod "foo" #o755)`. The return value is unspecified.

`utime` *pathname* [*actime* [*modtime* [*actimens* [*modtimens* [Scheme Procedure]
[*flags*]]]]]
`scm_utime` (*pathname, actime, modtime, actimens, modtimens, flags*) [C Function]
`utime` sets the access and modification times for the file named by *pathname*. If *actime* or *modtime* is not supplied, then the current time is used. *actime* and *modtime* must be integer time values as returned by the `current-time` procedure.

The optional *actimens* and *modtimens* are nanoseconds to add *actime* and *modtime*. Nanosecond precision is only supported on some combinations of file systems and operating systems.

```
(utime "foo" (- (current-time) 3600))
```

will set the access time to one hour in the past and the modification time to the current time.

`delete-file` *str* [Scheme Procedure]
`scm_delete_file` (*str*) [C Function]
Deletes (or "unlinks") the file whose path is specified by *str*.

`copy-file` *oldfile newfile* [Scheme Procedure]
`scm_copy_file` (*oldfile, newfile*) [C Function]
Copy the file specified by *oldfile* to *newfile*. The return value is unspecified.

`sendfile` *out in count* [*offset*] [Scheme Procedure]
`scm_sendfile` (*out, in, count, offset*) [C Function]
Send *count* bytes from *in* to *out*, both of which must be either open file ports or file descriptors. When *offset* is omitted, start reading from *in*'s current position; otherwise, start reading at *offset*. Return the number of bytes actually sent.

When *in* is a port, it is often preferable to specify *offset*, because *in*'s offset as a port may be different from the offset of its underlying file descriptor.

On systems that support it, such as GNU/Linux, this procedure uses the `sendfile` libc function, which usually corresponds to a system call. This is faster than doing a series of `read` and `write` system calls. A typical application is to send a file over a socket.

In some cases, the `sendfile` libc function may return `EINVAL` or `ENOSYS`. In that case, Guile's `sendfile` procedure automatically falls back to doing a series of `read` and `write` calls.

In other cases, the libc function may send fewer bytes than *count*—for instance because *out* is a slow or limited device, such as a pipe. When that happens, Guile's `sendfile` automatically retries until exactly *count* bytes were sent or an error occurs.

`rename-file` *oldname newname* [Scheme Procedure]
`scm_rename` (*oldname, newname*) [C Function]
Renames the file specified by *oldname* to *newname*. The return value is unspecified.

`link` *oldpath newpath* [Scheme Procedure]
`scm_link` (*oldpath, newpath*) [C Function]
Creates a new name *newpath* in the file system for the file named by *oldpath*. If *oldpath* is a symbolic link, the link may or may not be followed depending on the system.

`symlink` *oldpath newpath* [Scheme Procedure]
`scm_symlink` (*oldpath, newpath*) [C Function]
Create a symbolic link named *newpath* with the value (i.e., pointing to) *oldpath*. The return value is unspecified.

`mkdir` *path* [*mode*] [Scheme Procedure]
`scm_mkdir` (*path, mode*) [C Function]
Create a new directory named by *path*. If *mode* is omitted then the permissions of the directory are set to `#o777` masked with the current umask (см. Раздел 7.2.7 [Processes], страница 542). Otherwise they are set to the value specified with *mode*. The return value is unspecified.

`rmdir` *path* [Scheme Procedure]
`scm_rmdir` (*path*) [C Function]
Remove the existing directory named by *path*. The directory must be empty for this to succeed. The return value is unspecified.

`opendir` *dirname* [Scheme Procedure]
`scm_opendir` (*dirname*) [C Function]
Open the directory specified by *dirname* and return a directory stream.

Before using this and the procedures below, make sure to see the higher-level procedures for directory traversal that are available (см. Раздел 7.11 [File Tree Walk], страница 739).

`directory-stream?` *object* [Scheme Procedure]
`scm_directory_stream_p` (*object*) [C Function]
Return a boolean indicating whether *object* is a directory stream as returned by `opendir`.

`readdir` *stream* [Scheme Procedure]
`scm_readdir` (*stream*) [C Function]
Return (as a string) the next directory entry from the directory stream *stream*. If there is no remaining entry to be read then the end of file object is returned.

`rewinddir` *stream* [Scheme Procedure]
`scm_rewinddir` (*stream*) [C Function]
Reset the directory port *stream* so that the next call to `readdir` will return the first directory entry.

`closedir` *stream* [Scheme Procedure]
`scm_closedir` (*stream*) [C Function]
Close the directory stream *stream*. The return value is unspecified.

Here is an example showing how to display all the entries in a directory:

```
(define dir (opendir "/usr/lib"))
(do ((entry (readdir dir) (readdir dir)))
    ((eof-object? entry))
  (display entry)(newline))
(closedir dir)
```

`sync` [Scheme Procedure]
`scm_sync` () [C Function]
Flush the operating system disk buffers. The return value is unspecified.

`mknod` *path type perms dev* [Scheme Procedure]
`scm_mknod` (*path, type, perms, dev*) [C Function]
Creates a new special file, such as a file corresponding to a device. *path* specifies the name of the file. *type* should be one of the following symbols: '`regular`', '`directory`', '`symlink`', '`block-special`', '`char-special`', '`fifo`', or '`socket`'. *perms* (an integer) specifies the file permissions. *dev* (an integer) specifies which device the special file refers to. Its exact interpretation depends on the kind of special file being created.

E.g.,

```
(mknod "/dev/fd0" 'block-special #o660 (+ (* 2 256) 2))
```

The return value is unspecified.

`tmpnam` [Scheme Procedure]
`scm_tmpnam ()` [C Function]

Return an auto-generated name of a temporary file, a file which doesn't already exist. The name includes a path, it's usually in `/tmp` but that's system dependent.

Care must be taken when using `tmpnam`. In between choosing the name and creating the file another program might use that name, or an attacker might even make it a symlink pointing at something important and causing you to overwrite that.

The safe way is to create the file using `open` with `O_EXCL` to avoid any overwriting. A loop can try again with another name if the file exists (error `EEXIST`). `mkstemp!` below does that.

`mkstemp!` *tmpl* **[***mode***]** [Scheme Procedure]
`scm_mkstemp` (*tmpl*) [C Function]

Create a new unique file in the file system and return a new buffered port open for reading and writing to the file.

tmpl is a string specifying where the file should be created: it must end with '`XXXXXX`' and those '`X`'s will be changed in the string to return the name of the file. (`port-filename` on the port also gives the name.)

POSIX doesn't specify the permissions mode of the file, on GNU and most systems it's `#o600`. An application can use `chmod` to relax that if desired. For example `#o666` less `umask`, which is usual for ordinary file creation,

```
(let ((port (mkstemp! (string-copy "/tmp/myfile-XXXXXX"))))
  (chmod port (logand #o666 (lognot (umask))))
  ...)
```

The optional *mode* argument specifies a mode with which to open the new file, as a string in the same format that `open-file` takes. It defaults to `"w+"`.

`tmpfile` [Scheme Procedure]
`scm_tmpfile ()` [C Function]

Return an input/output port to a unique temporary file named using the path prefix `P_tmpdir` defined in `stdio.h`. The file is automatically deleted when the port is closed or the program terminates.

`dirname` *filename* [Scheme Procedure]
`scm_dirname` (*filename*) [C Function]

Return the directory name component of the file name *filename*. If *filename* does not contain a directory component, `.` is returned.

`basename` *filename* **[***suffix***]** [Scheme Procedure]
`scm_basename` (*filename, suffix*) [C Function]

Return the base name of the file name *filename*. The base name is the file name without any directory components. If *suffix* is provided, and is equal to the end of *basename*, it is removed also.

```
(basename "/tmp/test.xml" ".xml")
⇒ "test"
```

`file-exists?` *filename* [Scheme Procedure]

Return `#t` if the file named *filename* exists, `#f` if not.

Many operating systems, such as GNU, use / (forward slash) to separate the components of a file name; any file name starting with / is considered an *absolute file name*. These conventions are specified by the POSIX Base Definitions, which refer to conforming file names as "pathnames". Some operating systems use a different convention; in particular, Windows uses \ (backslash) as the file name separator, and also has the notion of *volume names* like `C:\` for absolute file names. The following procedures and variables provide support for portable file name manipulations.

`system-file-name-convention` [Scheme Procedure]
Return either `posix` or `windows`, depending on what kind of system this Guile is running on.

`file-name-separator?` *c* [Scheme Procedure]
Return true if character *c* is a file name separator on the host platform.

`absolute-file-name?` *file-name* [Scheme Procedure]
Return true if *file-name* denotes an absolute file name on the host platform.

`file-name-separator-string` [Scheme Variable]
The preferred file name separator.

Note that on MinGW builds for Windows, both / and \ are valid separators. Thus, programs should not assume that `file-name-separator-string` is the *only* file name separator—e.g., when extracting the components of a file name.

7.2.4 User Information

The facilities in this section provide an interface to the user and group database. They should be used with care since they are not reentrant.

The following functions accept an object representing user information and return a selected component:

`passwd:name` *pw* [Scheme Procedure]
The name of the userid.

`passwd:passwd` *pw* [Scheme Procedure]
The encrypted passwd.

`passwd:uid` *pw* [Scheme Procedure]
The user id number.

`passwd:gid` *pw* [Scheme Procedure]
The group id number.

`passwd:gecos` *pw* [Scheme Procedure]
The full name.

`passwd:dir` *pw* [Scheme Procedure]
The home directory.

`passwd:shell` *pw* [Scheme Procedure]
The login shell.

`getpwuid` *uid* [Scheme Procedure]
Look up an integer userid in the user database.

`getpwnam` *name* [Scheme Procedure]
Look up a user name string in the user database.

`setpwent` [Scheme Procedure]
Initializes a stream used by `getpwent` to read from the user database. The next use of `getpwent` will return the first entry. The return value is unspecified.

`getpwent` [Scheme Procedure]
Read the next entry in the user database stream. The return is a passwd user object as above, or `#f` when no more entries.

`endpwent` [Scheme Procedure]
Closes the stream used by `getpwent`. The return value is unspecified.

`setpw` [*arg*] [Scheme Procedure]
`scm_setpwent` (*arg*) [C Function]
If called with a true argument, initialize or reset the password data stream. Otherwise, close the stream. The `setpwent` and `endpwent` procedures are implemented on top of this.

`getpw` [*user*] [Scheme Procedure]
`scm_getpwuid` (*user*) [C Function]
Look up an entry in the user database. *user* can be an integer, a string, or omitted, giving the behaviour of getpwuid, getpwnam or getpwent respectively.

The following functions accept an object representing group information and return a selected component:

`group:name` *gr* [Scheme Procedure]
The group name.

`group:passwd` *gr* [Scheme Procedure]
The encrypted group password.

`group:gid` *gr* [Scheme Procedure]
The group id number.

`group:mem` *gr* [Scheme Procedure]
A list of userids which have this group as a supplementary group.

`getgrgid` *gid* [Scheme Procedure]
Look up an integer group id in the group database.

`getgrnam` *name* [Scheme Procedure]
Look up a group name in the group database.

`setgrent` [Scheme Procedure]
Initializes a stream used by `getgrent` to read from the group database. The next use of `getgrent` will return the first entry. The return value is unspecified.

`getgrent` [Scheme Procedure]
Return the next entry in the group database, using the stream set by `setgrent`.

`endgrent` [Scheme Procedure]
Closes the stream used by `getgrent`. The return value is unspecified.

`setgr` [*arg*] [Scheme Procedure]
`scm_setgrent` (*arg*) [C Function]
If called with a true argument, initialize or reset the group data stream. Otherwise, close the stream. The `setgrent` and `endgrent` procedures are implemented on top of this.

`getgr` [*group*] [Scheme Procedure]
`scm_getgrgid` (*group*) [C Function]
Look up an entry in the group database. *group* can be an integer, a string, or omitted, giving the behaviour of getgrgid, getgrnam or getgrent respectively.

In addition to the accessor procedures for the user database, the following shortcut procedure is also available.

`getlogin` [Scheme Procedure]
`scm_getlogin` () [C Function]
Return a string containing the name of the user logged in on the controlling terminal of the process, or `#f` if this information cannot be obtained.

7.2.5 Time

`current-time` [Scheme Procedure]
`scm_current_time` () [C Function]
Return the number of seconds since 1970-01-01 00:00:00 UTC, excluding leap seconds.

`gettimeofday` [Scheme Procedure]
`scm_gettimeofday` () [C Function]
Return a pair containing the number of seconds and microseconds since 1970-01-01 00:00:00 UTC, excluding leap seconds. Note: whether true microsecond resolution is available depends on the operating system.

The following procedures either accept an object representing a broken down time and return a selected component, or accept an object representing a broken down time and a value and set the component to the value. The numbers in parentheses give the usual range.

`tm:sec` *tm* [Scheme Procedure]
`set-tm:sec` *tm val* [Scheme Procedure]
Seconds (0-59).

`tm:min` *tm* [Scheme Procedure]
`set-tm:min` *tm val* [Scheme Procedure]
Minutes (0-59).

`tm:hour` *tm* [Scheme Procedure]
`set-tm:hour` *tm val* [Scheme Procedure]
Hours (0-23).

`tm:mday` *tm* [Scheme Procedure]
`set-tm:mday` *tm val* [Scheme Procedure]
Day of the month (1-31).

`tm:mon` *tm* [Scheme Procedure]
`set-tm:mon` *tm val* [Scheme Procedure]
Month (0-11).

`tm:year` *tm* [Scheme Procedure]
`set-tm:year` *tm val* [Scheme Procedure]
Year (70-), the year minus 1900.

`tm:wday` *tm* [Scheme Procedure]
`set-tm:wday` *tm val* [Scheme Procedure]
Day of the week (0-6) with Sunday represented as 0.

`tm:yday` *tm* [Scheme Procedure]
`set-tm:yday` *tm val* [Scheme Procedure]
Day of the year (0-364, 365 in leap years).

`tm:isdst` *tm* [Scheme Procedure]
`set-tm:isdst` *tm val* [Scheme Procedure]
Daylight saving indicator (0 for "no", greater than 0 for "yes", less than 0 for "unknown").

`tm:gmtoff` *tm* [Scheme Procedure]
`set-tm:gmtoff` *tm val* [Scheme Procedure]
Time zone offset in seconds west of UTC (-46800 to 43200). For example on East coast USA (zone '`EST+5`') this would be 18000 (ie. $5 \times 60 \times 60$) in winter, or 14400 (ie. $4 \times 60 \times 60$) during daylight savings.

Note `tm:gmtoff` is not the same as `tm_gmtoff` in the C `tm` structure. `tm_gmtoff` is seconds east and hence the negative of the value here.

`tm:zone` *tm* [Scheme Procedure]
`set-tm:zone` *tm val* [Scheme Procedure]
Time zone label (a string), not necessarily unique.

`localtime` *time* [*zone*] [Scheme Procedure]
`scm_localtime` (*time*, *zone*) [C Function]
Return an object representing the broken down components of *time*, an integer like the one returned by `current-time`. The time zone for the calculation is optionally specified by *zone* (a string), otherwise the `TZ` environment variable or the system default is used.

`gmtime` *time* [Scheme Procedure]
`scm_gmtime` (*time*) [C Function]
Return an object representing the broken down components of *time*, an integer like the one returned by `current-time`. The values are calculated for UTC.

`mktime` *sbd-time* [*zone*] [Scheme Procedure]
`scm_mktime` (*sbd_time, zone*) [C Function]
For a broken down time object *sbd-time*, return a pair the `car` of which is an integer time like `current-time`, and the `cdr` of which is a new broken down time with normalized fields.

zone is a timezone string, or the default is the `TZ` environment variable or the system default (см. Раздел "Specifying the Time Zone with `TZ`" в *GNU C Library Reference Manual*). *sbd-time* is taken to be in that *zone*.

The following fields of *sbd-time* are used: `tm:year`, `tm:mon`, `tm:mday`, `tm:hour`, `tm:min`, `tm:sec`, `tm:isdst`. The values can be outside their usual ranges. For example `tm:hour` normally goes up to 23, but a value say 33 would mean 9 the following day.

`tm:isdst` in *sbd-time* says whether the time given is with daylight savings or not. This is ignored if *zone* doesn't have any daylight savings adjustment amount.

The broken down time in the return normalizes the values of *sbd-time* by bringing them into their usual ranges, and using the actual daylight savings rule for that time in *zone* (which may differ from what *sbd-time* had). The easiest way to think of this is that *sbd-time* plus *zone* converts to the integer UTC time, then a `localtime` is applied to get the normal presentation of that time, in *zone*.

`tzset` [Scheme Procedure]
`scm_tzset` () [C Function]
Initialize the timezone from the `TZ` environment variable or the system default. It's not usually necessary to call this procedure since it's done automatically by other procedures that depend on the timezone.

`strftime` *format tm* [Scheme Procedure]
`scm_strftime` (*format, tm*) [C Function]
Return a string which is broken-down time structure *tm* formatted according to the given *format* string.

format contains field specifications introduced by a '`%`' character. See Раздел "Formatting Calendar Time" в *The GNU C Library Reference Manual*, or '`man 3 strftime`', for the available formatting.

```
(strftime "%c" (localtime (current-time)))
⇒ "Mon Mar 11 20:17:43 2002"
```

If `setlocale` has been called (см. Раздел 7.2.13 [Locales], страница 571), month and day names are from the current locale and in the locale character set.

`strptime` *format string* [Scheme Procedure]
`scm_strptime` (*format, string*) [C Function]
Performs the reverse action to `strftime`, parsing *string* according to the specification supplied in *format*. The interpretation of month and day names is dependent on

the current locale. The value returned is a pair. The CAR has an object with time components in the form returned by `localtime` or `gmtime`, but the time zone components are not usefully set. The CDR reports the number of characters from *string* which were used for the conversion.

`internal-time-units-per-second` [Переменная]
The value of this variable is the number of time units per second reported by the following procedures.

`times` [Scheme Procedure]
`scm_times ()` [C Function]
Return an object with information about real and processor time. The following procedures accept such an object as an argument and return a selected component:

`tms:clock` *tms* [Scheme Procedure]
The current real time, expressed as time units relative to an arbitrary base.

`tms:utime` *tms* [Scheme Procedure]
The CPU time units used by the calling process.

`tms:stime` *tms* [Scheme Procedure]
The CPU time units used by the system on behalf of the calling process.

`tms:cutime` *tms* [Scheme Procedure]
The CPU time units used by terminated child processes of the calling process, whose status has been collected (e.g., using `waitpid`).

`tms:cstime` *tms* [Scheme Procedure]
Similarly, the CPU times units used by the system on behalf of terminated child processes.

`get-internal-real-time` [Scheme Procedure]
`scm_get_internal_real_time ()` [C Function]
Return the number of time units since the interpreter was started.

`get-internal-run-time` [Scheme Procedure]
`scm_get_internal_run_time ()` [C Function]
Return the number of time units of processor time used by the interpreter. Both *system* and *user* time are included but subprocesses are not.

7.2.6 Runtime Environment

`program-arguments` [Scheme Procedure]
`command-line` [Scheme Procedure]
`set-program-arguments` [Scheme Procedure]
`scm_program_arguments ()` [C Function]
`scm_set_program_arguments_scm (lst)` [C Function]
Get the command line arguments passed to Guile, or set new arguments.

The arguments are a list of strings, the first of which is the invoked program name. This is just `"guile"` (or the executable path) when run interactively, or it's the script name when running a script with `-s` (см. Раздел 4.2 [Invoking Guile], страница 37).

```
guile -L /my/extra/dir -s foo.scm abc def

(program-arguments) ⇒ ("foo.scm" "abc" "def")
```

`set-program-arguments` allows a library module or similar to modify the arguments, for example to strip options it recognises, leaving the rest for the mainline.

The argument list is held in a fluid, which means it's separate for each thread. Neither the list nor the strings within it are copied at any point and normally should not be mutated.

The two names `program-arguments` and `command-line` are an historical accident, they both do exactly the same thing. The name `scm_set_program_arguments_scm` has an extra `_scm` on the end to avoid clashing with the C function below.

`void scm_set_program_arguments` (*int argc, char **argv, char *first*) [C Function]

Set the list of command line arguments for `program-arguments` and `command-line` above.

argv is an array of null-terminated strings, as in a C `main` function. *argc* is the number of strings in *argv*, or if it's negative then a `NULL` in *argv* marks its end.

first is an extra string put at the start of the arguments, or `NULL` for no such extra. This is a convenient way to pass the program name after advancing *argv* to strip option arguments. Eg.

```
{
  char *progname = argv[0];
  for (argv++; argv[0] != NULL && argv[0][0] == '-'; argv++)
    {
      /* munch option ... */
    }
  /* remaining args for scheme level use */
  scm_set_program_arguments (-1, argv, progname);
}
```

This sort of thing is often done at startup under `scm_boot_guile` with options handled at the C level removed. The given strings are all copied, so the C data is not accessed again once `scm_set_program_arguments` returns.

`getenv` *name* [Scheme Procedure]
`scm_getenv` (*name*) [C Function]

Looks up the string *name* in the current environment. The return value is `#f` unless a string of the form `NAME=VALUE` is found, in which case the string `VALUE` is returned.

`setenv` *name value* [Scheme Procedure]

Modifies the environment of the current process, which is also the default environment inherited by child processes.

If *value* is `#f`, then *name* is removed from the environment. Otherwise, the string *name*=*value* is added to the environment, replacing any existing string with name matching *name*.

The return value is unspecified.

`unsetenv` *name* [Scheme Procedure]
Remove variable *name* from the environment. The name can not contain a '=' character.

`environ` [*env*] [Scheme Procedure]
`scm_environ` (*env*) [C Function]
If *env* is omitted, return the current environment (in the Unix sense) as a list of strings. Otherwise set the current environment, which is also the default environment for child processes, to the supplied list of strings. Each member of *env* should be of the form *name*=*value* and values of *name* should not be duplicated. If *env* is supplied then the return value is unspecified.

`putenv` *str* [Scheme Procedure]
`scm_putenv` (*str*) [C Function]
Modifies the environment of the current process, which is also the default environment inherited by child processes.

If *str* is of the form `NAME=VALUE` then it will be written directly into the environment, replacing any existing environment string with name matching `NAME`. If *str* does not contain an equal sign, then any existing string with name matching *str* will be removed.

The return value is unspecified.

7.2.7 Processes

`chdir` *str* [Scheme Procedure]
`scm_chdir` (*str*) [C Function]
Change the current working directory to *str*. The return value is unspecified.

`getcwd` [Scheme Procedure]
`scm_getcwd` () [C Function]
Return the name of the current working directory.

`umask` [*mode*] [Scheme Procedure]
`scm_umask` (*mode*) [C Function]
If *mode* is omitted, returns a decimal number representing the current file creation mask. Otherwise the file creation mask is set to *mode* and the previous value is returned. См. Раздел "Assigning File Permissions" в *The GNU C Library Reference Manual*, for more on how to use umasks.

E.g., `(umask #o022)` sets the mask to octal 22/decimal 18.

`chroot` *path* [Scheme Procedure]
`scm_chroot` (*path*) [C Function]
Change the root directory to that specified in *path*. This directory will be used for path names beginning with `/`. The root directory is inherited by all children of the current process. Only the superuser may change the root directory.

`getpid` [Scheme Procedure]
`scm_getpid ()` [C Function]
Return an integer representing the current process ID.

`getgroups` [Scheme Procedure]
`scm_getgroups ()` [C Function]
Return a vector of integers representing the current supplementary group IDs.

`getppid` [Scheme Procedure]
`scm_getppid ()` [C Function]
Return an integer representing the process ID of the parent process.

`getuid` [Scheme Procedure]
`scm_getuid ()` [C Function]
Return an integer representing the current real user ID.

`getgid` [Scheme Procedure]
`scm_getgid ()` [C Function]
Return an integer representing the current real group ID.

`geteuid` [Scheme Procedure]
`scm_geteuid ()` [C Function]
Return an integer representing the current effective user ID. If the system does not support effective IDs, then the real ID is returned. `(provided? 'EIDs)` reports whether the system supports effective IDs.

`getegid` [Scheme Procedure]
`scm_getegid ()` [C Function]
Return an integer representing the current effective group ID. If the system does not support effective IDs, then the real ID is returned. `(provided? 'EIDs)` reports whether the system supports effective IDs.

`setgroups` *vec* [Scheme Procedure]
`scm_setgroups` (*vec*) [C Function]
Set the current set of supplementary group IDs to the integers in the given vector *vec*. The return value is unspecified.

Generally only the superuser can set the process group IDs (см. Раздел “Setting Groups” в *The GNU C Library Reference Manual*).

`setuid` *id* [Scheme Procedure]
`scm_setuid` (*id*) [C Function]
Sets both the real and effective user IDs to the integer *id*, provided the process has appropriate privileges. The return value is unspecified.

`setgid` *id* [Scheme Procedure]
`scm_setgid` (*id*) [C Function]
Sets both the real and effective group IDs to the integer *id*, provided the process has appropriate privileges. The return value is unspecified.

`seteuid` *id* [Scheme Procedure]
`scm_seteuid` (*id*) [C Function]
Sets the effective user ID to the integer *id*, provided the process has appropriate privileges. If effective IDs are not supported, the real ID is set instead—`(provided? 'EIDs)` reports whether the system supports effective IDs. The return value is unspecified.

`setegid` *id* [Scheme Procedure]
`scm_setegid` (*id*) [C Function]
Sets the effective group ID to the integer *id*, provided the process has appropriate privileges. If effective IDs are not supported, the real ID is set instead—`(provided? 'EIDs)` reports whether the system supports effective IDs. The return value is unspecified.

`getpgrp` [Scheme Procedure]
`scm_getpgrp` () [C Function]
Return an integer representing the current process group ID. This is the POSIX definition, not BSD.

`setpgid` *pid pgid* [Scheme Procedure]
`scm_setpgid` (*pid, pgid*) [C Function]
Move the process *pid* into the process group *pgid*. *pid* or *pgid* must be integers: they can be zero to indicate the ID of the current process. Fails on systems that do not support job control. The return value is unspecified.

`setsid` [Scheme Procedure]
`scm_setsid` () [C Function]
Creates a new session. The current process becomes the session leader and is put in a new process group. The process will be detached from its controlling terminal if it has one. The return value is an integer representing the new process group ID.

`getsid` *pid* [Scheme Procedure]
`scm_getsid` (*pid*) [C Function]
Returns the session ID of process *pid*. (The session ID of a process is the process group ID of its session leader.)

`waitpid` *pid* [*options*] [Scheme Procedure]
`scm_waitpid` (*pid, options*) [C Function]
This procedure collects status information from a child process which has terminated or (optionally) stopped. Normally it will suspend the calling process until this can be done. If more than one child process is eligible then one will be chosen by the operating system.

The value of *pid* determines the behaviour:

pid greater than 0
: Request status information from the specified child process.

pid equal to -1 or `WAIT_ANY`
: Request status information for any child process.

pid equal to 0 or `WAIT_MYPGRP`
: Request status information for any child process in the current process group.

pid less than -1
: Request status information for any child process whose process group ID is $-pid$.

The *options* argument, if supplied, should be the bitwise OR of the values of zero or more of the following variables:

`WNOHANG` [Переменная]
: Return immediately even if there are no child processes to be collected.

`WUNTRACED` [Переменная]
: Report status information for stopped processes as well as terminated processes.

The return value is a pair containing:

1. The process ID of the child process, or 0 if `WNOHANG` was specified and no process was collected.
2. The integer status value.

The following three functions can be used to decode the process status code returned by `waitpid`.

`status:exit-val` *status* [Scheme Procedure]
`scm_status_exit_val` (*status*) [C Function]
: Return the exit status value, as would be set if a process ended normally through a call to `exit` or `_exit`, if any, otherwise `#f`.

`status:term-sig` *status* [Scheme Procedure]
`scm_status_term_sig` (*status*) [C Function]
: Return the signal number which terminated the process, if any, otherwise `#f`.

`status:stop-sig` *status* [Scheme Procedure]
`scm_status_stop_sig` (*status*) [C Function]
: Return the signal number which stopped the process, if any, otherwise `#f`.

`system` [*cmd*] [Scheme Procedure]
`scm_system` (*cmd*) [C Function]
: Execute *cmd* using the operating system's "command processor". Under Unix this is usually the default shell `sh`. The value returned is *cmd*'s exit status as returned by `waitpid`, which can be interpreted using the functions above.

  If `system` is called without arguments, return a boolean indicating whether the command processor is available.

`system*` *arg1 arg2 ...* [Scheme Procedure]
`scm_system_star` (*args*) [C Function]
: Execute the command indicated by *arg1 arg2 ...*. The first element must be a string indicating the command to be executed, and the remaining items must be strings representing each of the arguments to that command.

This function returns the exit status of the command as provided by `waitpid`. This value can be handled with `status:exit-val` and the related functions.

`system*` is similar to `system`, but accepts only one string per-argument, and performs no shell interpretation. The command is executed using fork and execlp. Accordingly this function may be safer than `system` in situations where shell interpretation is not required.

Example: (system* "echo" "foo" "bar")

`quit` [*status*] [Scheme Procedure]
`exit` [*status*] [Scheme Procedure]

Terminate the current process with proper unwinding of the Scheme stack. The exit status zero if *status* is not supplied. If *status* is supplied, and it is an integer, that integer is used as the exit status. If *status* is `#t` or `#f`, the exit status is *EXIT_SUCCESS* or *EXIT_FAILURE*, respectively.

The procedure `exit` is an alias of `quit`. They have the same functionality.

`EXIT_SUCCESS` [Scheme Variable]
`EXIT_FAILURE` [Scheme Variable]

These constants represent the standard exit codes for success (zero) or failure (one.)

`primitive-exit` [*status*] [Scheme Procedure]
`primitive-_exit` [*status*] [Scheme Procedure]
`scm_primitive_exit` (*status*) [C Function]
`scm_primitive__exit` (*status*) [C Function]

Terminate the current process without unwinding the Scheme stack. The exit status is *status* if supplied, otherwise zero.

`primitive-exit` uses the C `exit` function and hence runs usual C level cleanups (flush output streams, call `atexit` functions, etc, see Раздел "Normal Termination" в *The GNU C Library Reference Manual*)).

`primitive-_exit` is the `_exit` system call (см. Раздел "Termination Internals" в *The GNU C Library Reference Manual*). This terminates the program immediately, with neither Scheme-level nor C-level cleanups.

The typical use for `primitive-_exit` is from a child process created with `primitive-fork`. For example in a Gdk program the child process inherits the X server connection and a C-level `atexit` cleanup which will close that connection. But closing in the child would upset the protocol in the parent, so `primitive-_exit` should be used to exit without that.

`execl` *filename arg ...* [Scheme Procedure]
`scm_execl` (*filename, args*) [C Function]

Executes the file named by *filename* as a new process image. The remaining arguments are supplied to the process; from a C program they are accessible as the `argv` argument to `main`. Conventionally the first *arg* is the same as *filename*. All arguments must be strings.

If *arg* is missing, *filename* is executed with a null argument list, which may have system-dependent side-effects.

This procedure is currently implemented using the `execv` system call, but we call it `execl` because of its Scheme calling interface.

`execlp` *filename arg ...* [Scheme Procedure]
`scm_execlp` (*filename, args*) [C Function]
Similar to `execl`, however if *filename* does not contain a slash then the file to execute will be located by searching the directories listed in the `PATH` environment variable.

This procedure is currently implemented using the `execvp` system call, but we call it `execlp` because of its Scheme calling interface.

`execle` *filename env arg ...* [Scheme Procedure]
`scm_execle` (*filename, env, args*) [C Function]
Similar to `execl`, but the environment of the new process is specified by *env*, which must be a list of strings as returned by the `environ` procedure.

This procedure is currently implemented using the `execve` system call, but we call it `execle` because of its Scheme calling interface.

`primitive-fork` [Scheme Procedure]
`scm_fork` () [C Function]
Creates a new "child" process by duplicating the current "parent" process. In the child the return value is 0. In the parent the return value is the integer process ID of the child.

Note that it is unsafe to fork a process that has multiple threads running, as only the thread that calls `primitive-fork` will persist in the child. Any resources that other threads held, such as locked mutexes or open file descriptors, are lost. Indeed, POSIX specifies that only async-signal-safe procedures are safe to call after a multithreaded fork, which is a very limited set. Guile issues a warning if it detects a fork from a multi-threaded program.

If you are going to `exec` soon after forking, the procedures in `(ice-9 popen)` may be useful to you, as they fork and exec within an async-signal-safe function carefully written to ensure robust program behavior, even in the presence of threads. См. Раздел 7.2.10 [Pipes], страница 552, for more.

This procedure has been renamed from `fork` to avoid a naming conflict with the scsh fork.

`nice` *incr* [Scheme Procedure]
`scm_nice` (*incr*) [C Function]
Increment the priority of the current process by *incr*. A higher priority value means that the process runs less often. The return value is unspecified.

`setpriority` *which who prio* [Scheme Procedure]
`scm_setpriority` (*which, who, prio*) [C Function]
Set the scheduling priority of the process, process group or user, as indicated by *which* and *who*. *which* is one of the variables `PRIO_PROCESS`, `PRIO_PGRP` or `PRIO_USER`, and *who* is interpreted relative to *which* (a process identifier for `PRIO_PROCESS`, process group identifier for `PRIO_PGRP`, and a user identifier for `PRIO_USER`. A zero value of *who* denotes the current process, process group, or user. *prio* is a value in the

range [−20,20]. The default priority is 0; lower priorities (in numerical terms) cause more favorable scheduling. Sets the priority of all of the specified processes. Only the super-user may lower priorities. The return value is not specified.

`getpriority` *which who* [Scheme Procedure]
`scm_getpriority` (*which, who*) [C Function]
Return the scheduling priority of the process, process group or user, as indicated by *which* and *who*. *which* is one of the variables `PRIO_PROCESS`, `PRIO_PGRP` or `PRIO_USER`, and *who* should be interpreted depending on *which* (a process identifier for `PRIO_PROCESS`, process group identifier for `PRIO_PGRP`, and a user identifier for `PRIO_USER`). A zero value of *who* denotes the current process, process group, or user. Return the highest priority (lowest numerical value) of any of the specified processes.

`getaffinity` *pid* [Scheme Procedure]
`scm_getaffinity` (*pid*) [C Function]
Return a bitvector representing the CPU affinity mask for process *pid*. Each CPU the process has affinity with has its corresponding bit set in the returned bitvector. The number of bits set is a good estimate of how many CPUs Guile can use without stepping on other processes' toes.

Currently this procedure is only defined on GNU variants (см. Раздел "CPU Affinity" в *The GNU C Library Reference Manual*).

`setaffinity` *pid mask* [Scheme Procedure]
`scm_setaffinity` (*pid, mask*) [C Function]
Install the CPU affinity mask *mask*, a bitvector, for the process or thread with ID *pid*. The return value is unspecified.

Currently this procedure is only defined on GNU variants (см. Раздел "CPU Affinity" в *The GNU C Library Reference Manual*).

См. Раздел 6.22.1 [Threads], страница 463, for information on how get the number of processors available on a system.

7.2.8 Signals

The following procedures raise, handle and wait for signals.

Scheme code signal handlers are run via an async (см. Раздел 6.22.3 [Asyncs], страница 466), so they're called in the handler's thread at the next safe opportunity. Generally this is after any currently executing primitive procedure finishes (which could be a long time for primitives that wait for an external event).

`kill` *pid sig* [Scheme Procedure]
`scm_kill` (*pid, sig*) [C Function]
Sends a signal to the specified process or group of processes.

pid specifies the processes to which the signal is sent:

pid greater than 0
: The process whose identifier is *pid*.

pid equal to 0
: All processes in the current process group.

pid less than -1
: The process group whose identifier is *-pid*

pid equal to -1
: If the process is privileged, all processes except for some special system processes. Otherwise, all processes with the current effective user ID.

sig should be specified using a variable corresponding to the Unix symbolic name, e.g.,

`SIGHUP` [Переменная]
: Hang-up signal.

`SIGINT` [Переменная]
: Interrupt signal.

A full list of signals on the GNU system may be found in Раздел "Standard Signals" в *The GNU C Library Reference Manual*.

`raise` *sig* [Scheme Procedure]
`scm_raise` (*sig*) [C Function]
: Sends a specified signal *sig* to the current process, where *sig* is as described for the `kill` procedure.

`sigaction` *signum* **[***handler* **[***flags* **[***thread***]]]** [Scheme Procedure]
`scm_sigaction` (*signum, handler, flags*) [C Function]
`scm_sigaction_for_thread` (*signum, handler, flags, thread*) [C Function]
: Install or report the signal handler for a specified signal.

signum is the signal number, which can be specified using the value of variables such as `SIGINT`.

If *handler* is omitted, `sigaction` returns a pair: the CAR is the current signal hander, which will be either an integer with the value `SIG_DFL` (default action) or `SIG_IGN` (ignore), or the Scheme procedure which handles the signal, or `#f` if a non-Scheme procedure handles the signal. The CDR contains the current `sigaction` flags for the handler.

If *handler* is provided, it is installed as the new handler for *signum*. *handler* can be a Scheme procedure taking one argument, or the value of `SIG_DFL` (default action) or `SIG_IGN` (ignore), or `#f` to restore whatever signal handler was installed before `sigaction` was first used. When a scheme procedure has been specified, that procedure will run in the given *thread*. When no thread has been given, the thread that made this call to `sigaction` is used.

flags is a `logior` (см. Раздел 6.6.2.13 [Bitwise Operations], страница 134) of the following (where provided by the system), or `0` for none.

`SA_NOCLDSTOP` [Переменная]
: By default, `SIGCHLD` is signalled when a child process stops (ie. receives `SIGSTOP`), and when a child process terminates. With the `SA_NOCLDSTOP` flag, `SIGCHLD` is only signalled for termination, not stopping.

`SA_NOCLDSTOP` has no effect on signals other than `SIGCHLD`.

`SA_RESTART` [Переменная]

If a signal occurs while in a system call, deliver the signal then restart the system call (as opposed to returning an `EINTR` error from that call).

Guile handles signals asynchronously. When it receives a signal, the synchronous signal handler just records the fact that a signal was received and sets a flag to tell the relevant Guile thread that it has a pending signal. When the Guile thread checks the pending-interrupt flag, it will arrange to run the asynchronous part of the signal handler, which is the handler attached by `sigaction`.

This strategy has some perhaps-unexpected interactions with the `SA_RESTART` flag, though: because the synchronous handler doesn't do very much, and notably it doesn't run the Guile handler, it's impossible to interrupt a thread stuck in a long-running system call via a signal handler that is installed with `SA_RESTART`: the synchronous handler just records the pending interrupt, but then the system call resumes and Guile doesn't have a chance to actually check the flag and run the asynchronous handler. That's just how it is.

The return value is a pair with information about the old handler as described above.

This interface does not provide access to the "signal blocking" facility. Maybe this is not needed, since the thread support may provide solutions to the problem of consistent access to data structures.

`restore-signals` [Scheme Procedure]
`scm_restore_signals ()` [C Function]

Return all signal handlers to the values they had before any call to `sigaction` was made. The return value is unspecified.

`alarm` *i* [Scheme Procedure]
`scm_alarm` (*i*) [C Function]

Set a timer to raise a `SIGALRM` signal after the specified number of seconds (an integer). It's advisable to install a signal handler for `SIGALRM` beforehand, since the default action is to terminate the process.

The return value indicates the time remaining for the previous alarm, if any. The new value replaces the previous alarm. If there was no previous alarm, the return value is zero.

`pause` [Scheme Procedure]
`scm_pause ()` [C Function]

Pause the current process (thread?) until a signal arrives whose action is to either terminate the current process or invoke a handler procedure. The return value is unspecified.

`sleep` *secs* [Scheme Procedure]
`usleep` *usecs* [Scheme Procedure]
`scm_sleep` (*secs*) [C Function]
`scm_usleep` (*usecs*) [C Function]

Wait the given period *secs* seconds or *usecs* microseconds (both integers). If a signal arrives the wait stops and the return value is the time remaining, in seconds or microseconds respectively. If the period elapses with no signal the return is zero.

On most systems the process scheduler is not microsecond accurate and the actual period slept by `usleep` might be rounded to a system clock tick boundary, which might be 10 milliseconds for instance.

See `scm_std_sleep` and `scm_std_usleep` for equivalents at the C level (см. Раздел 6.22.6 [Blocking], страница 473).

`getitimer` *which_timer* [Scheme Procedure]
`setitimer` *which_timer interval_seconds interval_microseconds value_seconds value_microseconds* [Scheme Procedure]
`scm_getitimer` (*which_timer*) [C Function]
`scm_setitimer` (*which_timer, interval_seconds, interval_microseconds, value_seconds, value_microseconds*) [C Function]

Get or set the periods programmed in certain system timers.

These timers have two settings. The first setting, the interval, is the value at which the timer will be reset when the current timer expires. The second is the current value of the timer, indicating when the next expiry will be signalled.

which_timer is one of the following values:

`ITIMER_REAL` [Переменная]
: A real-time timer, counting down elapsed real time. At zero it raises `SIGALRM`. This is like `alarm` above, but with a higher resolution period.

`ITIMER_VIRTUAL` [Переменная]
: A virtual-time timer, counting down while the current process is actually using CPU. At zero it raises `SIGVTALRM`.

`ITIMER_PROF` [Переменная]
: A profiling timer, counting down while the process is running (like `ITIMER_VIRTUAL`) and also while system calls are running on the process's behalf. At zero it raises a `SIGPROF`.

  This timer is intended for profiling where a program is spending its time (by looking where it is when the timer goes off).

`getitimer` returns the restart timer value and its current value, as a list containing two pairs. Each pair is a time in seconds and microseconds: `((`*`interval_secs`* `.` *`interval_usecs`*`) (`*`value_secs`* `.` *`value_usecs`*`))`.

`setitimer` sets the timer values similarly, in seconds and microseconds (which must be integers). The interval value can be zero to have the timer run down just once. The return value is the timer's previous setting, in the same form as `getitimer` returns.

```
(setitimer ITIMER_REAL
           5 500000     ;; Raise SIGALRM every 5.5 seconds
           2 0)         ;; with the first SIGALRM in 2 seconds
```

Although the timers are programmed in microseconds, the actual accuracy might not be that high.

Note that `ITIMER_PROF` and `ITIMER_VIRTUAL` are not functional on all platforms and may always error when called. `(provided? 'ITIMER_PROF)` and `(provided?`

'ITIMER_VIRTUAL) can be used to test if the those itimers are supported on the given host. ITIMER_REAL is supported on all platforms that support setitimer.

7.2.9 Terminals and Ptys

isatty? *port* [Scheme Procedure]
scm_isatty_p (*port*) [C Function]
Return #t if *port* is using a serial non–file device, otherwise #f.

ttyname *port* [Scheme Procedure]
scm_ttyname (*port*) [C Function]
Return a string with the name of the serial terminal device underlying *port*.

ctermid [Scheme Procedure]
scm_ctermid () [C Function]
Return a string containing the file name of the controlling terminal for the current process.

tcgetpgrp *port* [Scheme Procedure]
scm_tcgetpgrp (*port*) [C Function]
Return the process group ID of the foreground process group associated with the terminal open on the file descriptor underlying *port*.

If there is no foreground process group, the return value is a number greater than 1 that does not match the process group ID of any existing process group. This can happen if all of the processes in the job that was formerly the foreground job have terminated, and no other job has yet been moved into the foreground.

tcsetpgrp *port pgid* [Scheme Procedure]
scm_tcsetpgrp (*port*, *pgid*) [C Function]
Set the foreground process group ID for the terminal used by the file descriptor underlying *port* to the integer *pgid*. The calling process must be a member of the same session as *pgid* and must have the same controlling terminal. The return value is unspecified.

7.2.10 Pipes

The following procedures are similar to the popen and pclose system routines. The code is in a separate "popen" module[1]:

```
(use-modules (ice-9 popen))
```

open-pipe *command mode* [Scheme Procedure]
open-pipe* *mode prog* [*args...*] [Scheme Procedure]
Execute a command in a subprocess, with a pipe to it or from it, or with pipes in both directions.

open-pipe runs the shell *command* using '/bin/sh -c'. open-pipe* executes *prog* directly, with the optional *args* arguments (all strings).

mode should be one of the following values. OPEN_READ is an input pipe, ie. to read from the subprocess. OPEN_WRITE is an output pipe, ie. to write to it.

[1] This module is only available on systems where the popen feature is provided (см. Раздел 6.23.2.2 [Common Feature Symbols], страница 479).

`OPEN_READ` [Переменная]
`OPEN_WRITE` [Переменная]
`OPEN_BOTH` [Переменная]

For an input pipe, the child's standard output is the pipe and standard input is inherited from `current-input-port`. For an output pipe, the child's standard input is the pipe and standard output is inherited from `current-output-port`. In all cases the child's standard error is inherited from `current-error-port` (см. Раздел 6.14.9 [Default Ports], страница 364).

If those `current-X-ports` are not files of some kind, and hence don't have file descriptors for the child, then `/dev/null` is used instead.

Care should be taken with `OPEN_BOTH`, a deadlock will occur if both parent and child are writing, and waiting until the write completes before doing any reading. Each direction has `PIPE_BUF` bytes of buffering (см. Раздел 6.14.6 [Buffering], страница 360), which will be enough for small writes, but not for say putting a big file through a filter.

`open-input-pipe` *command* [Scheme Procedure]

Equivalent to `open-pipe` with mode `OPEN_READ`.

```
(let* ((port (open-input-pipe "date --utc"))
       (str  (read-line port)))
  (close-pipe port)
  str)
⇒ "Mon Mar 11 20:10:44 UTC 2002"
```

`open-output-pipe` *command* [Scheme Procedure]

Equivalent to `open-pipe` with mode `OPEN_WRITE`.

```
(let ((port (open-output-pipe "lpr")))
  (display "Something for the line printer.\n" port)
  (if (not (eqv? 0 (status:exit-val (close-pipe port))))
      (error "Cannot print")))
```

`open-input-output-pipe` *command* [Scheme Procedure]

Equivalent to `open-pipe` with mode `OPEN_BOTH`.

`close-pipe` *port* [Scheme Procedure]

Close a pipe created by `open-pipe`, wait for the process to terminate, and return the wait status code. The status is as per `waitpid` and can be decoded with `status:exit-val` etc (см. Раздел 7.2.7 [Processes], страница 542)

`waitpid WAIT_ANY` should not be used when pipes are open, since it can reap a pipe's child process, causing an error from a subsequent `close-pipe`.

`close-port` (см. Раздел 6.14.1 [Ports], страница 352) can close a pipe, but it doesn't reap the child process.

The garbage collector will close a pipe no longer in use, and reap the child process with `waitpid`. If the child hasn't yet terminated the garbage collector doesn't block, but instead checks again in the next GC.

Many systems have per-user and system-wide limits on the number of processes, and a system-wide limit on the number of pipes, so pipes should be closed explicitly when no longer needed, rather than letting the garbage collector pick them up at some later time.

7.2.11 Networking

7.2.11.1 Network Address Conversion

This section describes procedures which convert internet addresses between numeric and string formats.

IPv4 Address Conversion

An IPv4 Internet address is a 4-byte value, represented in Guile as an integer in host byte order, so that say "0.0.0.1" is 1, or "1.0.0.0" is 16777216.

Some underlying C functions use network byte order for addresses, Guile converts as necessary so that at the Scheme level its host byte order everywhere.

`INADDR_ANY` [Переменная]
: For a server, this can be used with `bind` (см. Раздел 7.2.11.4 [Network Sockets and Communication], страница 564) to allow connections from any interface on the machine.

`INADDR_BROADCAST` [Переменная]
: The broadcast address on the local network.

`INADDR_LOOPBACK` [Переменная]
: The address of the local host using the loopback device, ie. '`127.0.0.1`'.

`inet-aton` *address* [Scheme Procedure]
`scm_inet_aton` (*address*) [C Function]
: This function is deprecated in favor of `inet-pton`.

  Convert an IPv4 Internet address from printable string (dotted decimal notation) to an integer. E.g.,

```
(inet-aton "127.0.0.1") ⇒ 2130706433
```

`inet-ntoa` *inetid* [Scheme Procedure]
`scm_inet_ntoa` (*inetid*) [C Function]
: This function is deprecated in favor of `inet-ntop`.

  Convert an IPv4 Internet address to a printable (dotted decimal notation) string. E.g.,

```
(inet-ntoa 2130706433) ⇒ "127.0.0.1"
```

`inet-netof` *address* [Scheme Procedure]
`scm_inet_netof` (*address*) [C Function]
: Return the network number part of the given IPv4 Internet address. E.g.,

```
(inet-netof 2130706433) ⇒ 127
```

`inet-lnaof` *address* [Scheme Procedure]
`scm_lnaof` (*address*) [C Function]
Return the local-address-with-network part of the given IPv4 Internet address, using the obsolete class A/B/C system. E.g.,

```
(inet-lnaof 2130706433) ⇒ 1
```

`inet-makeaddr` *net lna* [Scheme Procedure]
`scm_inet_makeaddr` (*net*, *lna*) [C Function]
Make an IPv4 Internet address by combining the network number *net* with the local-address-within-network number *lna*. E.g.,

```
(inet-makeaddr 127 1) ⇒ 2130706433
```

IPv6 Address Conversion

An IPv6 Internet address is a 16-byte value, represented in Guile as an integer in host byte order, so that say "::1" is 1.

`inet-ntop` *family address* [Scheme Procedure]
`scm_inet_ntop` (*family*, *address*) [C Function]
Convert a network address from an integer to a printable string. *family* can be `AF_INET` or `AF_INET6`. E.g.,

```
(inet-ntop AF_INET 2130706433) ⇒ "127.0.0.1"
(inet-ntop AF_INET6 (- (expt 2 128) 1))
  ⇒ "ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff"
```

`inet-pton` *family address* [Scheme Procedure]
`scm_inet_pton` (*family*, *address*) [C Function]
Convert a string containing a printable network address to an integer address. *family* can be `AF_INET` or `AF_INET6`. E.g.,

```
(inet-pton AF_INET "127.0.0.1") ⇒ 2130706433
(inet-pton AF_INET6 "::1") ⇒ 1
```

7.2.11.2 Network Databases

This section describes procedures which query various network databases. Care should be taken when using the database routines since they are not reentrant.

`getaddrinfo`

The `getaddrinfo` procedure maps host and service names to socket addresses and associated information in a protocol-independent way.

`getaddrinfo` *name service* [*hint_flags* [*hint_family* [*hint_socktype* [*hint_protocol*]]]] [Scheme Procedure]
`scm_getaddrinfo` (*name*, *service*, *hint_flags*, *hint_family*, *hint_socktype*, *hint_protocol*) [C Function]
Return a list of `addrinfo` structures containing a socket address and associated information for host *name* and/or *service* to be used in creating a socket with which to address the specified service.

```
(let* ((ai (car (getaddrinfo "www.gnu.org" "http")))
```

```
        (s  (socket (addrinfo:fam ai) (addrinfo:socktype ai)
                    (addrinfo:protocol ai))))
    (connect s (addrinfo:addr ai))
    s)
```

When *service* is omitted or is `#f`, return network-level addresses for *name*. When *name* is `#f` *service* must be provided and service locations local to the caller are returned.

Additional hints can be provided. When specified, *hint_flags* should be a bitwise-or of zero or more constants among the following:

`AI_PASSIVE`
: Socket address is intended for `bind`.

`AI_CANONNAME`
: Request for canonical host name, available via `addrinfo:canonname`. This makes sense mainly when DNS lookups are involved.

`AI_NUMERICHOST`
: Specifies that *name* is a numeric host address string (e.g., `"127.0.0.1"`), meaning that name resolution will not be used.

`AI_NUMERICSERV`
: Likewise, specifies that *service* is a numeric port string (e.g., `"80"`).

`AI_ADDRCONFIG`
: Return only addresses configured on the local system It is highly recommended to provide this flag when the returned socket addresses are to be used to make connections; otherwise, some of the returned addresses could be unreachable or use a protocol that is not supported.

`AI_V4MAPPED`
: When looking up IPv6 addresses, return mapped IPv4 addresses if there is no IPv6 address available at all.

`AI_ALL` If this flag is set along with `AI_V4MAPPED` when looking up IPv6 addresses, return all IPv6 addresses as well as all IPv4 addresses, the latter mapped to IPv6 format.

When given, *hint_family* should specify the requested address family, e.g., `AF_INET6`. Similarly, *hint_socktype* should specify the requested socket type (e.g., `SOCK_DGRAM`), and *hint_protocol* should specify the requested protocol (its value is interpreted as in calls to `socket`).

On error, an exception with key `getaddrinfo-error` is thrown, with an error code (an integer) as its argument:

```
(catch 'getaddrinfo-error
  (lambda ()
    (getaddrinfo "www.gnu.org" "gopher"))
  (lambda (key errcode)
    (cond ((= errcode EAI_SERVICE)
   (display "doesn't know about Gopher!\n"))
```

```
        ((= errcode EAI_NONAME)
         (display "www.gnu.org not found\\n"))
        (else
         (format #t "something wrong: ~a\n"
         (gai-strerror errcode))))))
```

Error codes are:

`EAI_AGAIN`
: The name or service could not be resolved at this time. Future attempts may succeed.

`EAI_BADFLAGS`
: *hint_flags* contains an invalid value.

`EAI_FAIL`
: A non-recoverable error occurred when attempting to resolve the name.

`EAI_FAMILY`
: *hint_family* was not recognized.

`EAI_NONAME`
: Either *name* does not resolve for the supplied parameters, or neither *name* nor *service* were supplied.

`EAI_NODATA`
: This non-POSIX error code can be returned on some systems (GNU and Darwin, at least), for example when *name* is known but requests that were made turned out no data. Error handling code should be prepared to handle it when it is defined.

`EAI_SERVICE`
: *service* was not recognized for the specified socket type.

`EAI_SOCKTYPE`
: *hint_socktype* was not recognized.

`EAI_SYSTEM`
: A system error occurred. In C, the error code can be found in `errno`; this value is not accessible from Scheme, but in practice it provides little information about the actual error cause.

Users are encouraged to read the "POSIX specification (`http://www.opengroup.org/onlinepubs/9699919799/functions/getaddrinfo.html`) for more details.

The following procedures take an `addrinfo` object as returned by `getaddrinfo`:

`addrinfo:flags` *ai* [Scheme Procedure]
: Return flags for *ai* as a bitwise or of `AI_` values (see above).

`addrinfo:fam` *ai* [Scheme Procedure]
: Return the address family of *ai* (a `AF_` value).

`addrinfo:socktype` *ai* [Scheme Procedure]
: Return the socket type for *ai* (a `SOCK_` value).

`addrinfo:protocol` *ai* [Scheme Procedure]
Return the protocol of *ai*.

`addrinfo:addr` *ai* [Scheme Procedure]
Return the socket address associated with *ai* as a `sockaddr` object (см. Раздел 7.2.11.3 [Network Socket Address], страница 562).

`addrinfo:canonname` *ai* [Scheme Procedure]
Return a string for the canonical name associated with *ai* if the `AI_CANONNAME` flag was supplied.

The Host Database

A *host object* is a structure that represents what is known about a network host, and is the usual way of representing a system's network identity inside software.

The following functions accept a host object and return a selected component:

`hostent:name` *host* [Scheme Procedure]
The "official" hostname for *host*.

`hostent:aliases` *host* [Scheme Procedure]
A list of aliases for *host*.

`hostent:addrtype` *host* [Scheme Procedure]
The host address type, one of the `AF` constants, such as `AF_INET` or `AF_INET6`.

`hostent:length` *host* [Scheme Procedure]
The length of each address for *host*, in bytes.

`hostent:addr-list` *host* [Scheme Procedure]
The list of network addresses associated with *host*. For `AF_INET` these are integer IPv4 address (см. Раздел 7.2.11.1 [Network Address Conversion], страница 554).

The following procedures can be used to search the host database. However, `getaddrinfo` should be preferred over them since it's more generic and thread-safe.

`gethost` [*host*] [Scheme Procedure]
`gethostbyname` *hostname* [Scheme Procedure]
`gethostbyaddr` *address* [Scheme Procedure]
`scm_gethost` (*host*) [C Function]
Look up a host by name or address, returning a host object. The `gethost` procedure will accept either a string name or an integer address; if given no arguments, it behaves like `gethostent` (see below). If a name or address is supplied but the address can not be found, an error will be thrown to one of the keys: `host-not-found`, `try-again`, `no-recovery` or `no-data`, corresponding to the equivalent `h_error` values. Unusual conditions may result in errors thrown to the `system-error` or `misc_error` keys.

```
(gethost "www.gnu.org")
⇒ #("www.gnu.org" () 2 4 (3353880842))

(gethostbyname "www.emacs.org")
⇒ #("emacs.org" ("www.emacs.org") 2 4 (1073448978))
```

The following procedures may be used to step through the host database from beginning to end.

`sethostent` [*stayopen*] [Scheme Procedure]
: Initialize an internal stream from which host objects may be read. This procedure must be called before any calls to `gethostent`, and may also be called afterward to reset the host entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `gethostbyname` or `gethostbyaddr` calls, possibly giving an efficiency gain.

`gethostent` [Scheme Procedure]
: Return the next host object from the host database, or `#f` if there are no more hosts to be found (or an error has been encountered). This procedure may not be used before `sethostent` has been called.

`endhostent` [Scheme Procedure]
: Close the stream used by `gethostent`. The return value is unspecified.

`sethost` [*stayopen*] [Scheme Procedure]
`scm_sethost` (*stayopen*) [C Function]
: If *stayopen* is omitted, this is equivalent to `endhostent`. Otherwise it is equivalent to `sethostent stayopen`.

The Network Database

The following functions accept an object representing a network and return a selected component:

`netent:name` *net* [Scheme Procedure]
: The "official" network name.

`netent:aliases` *net* [Scheme Procedure]
: A list of aliases for the network.

`netent:addrtype` *net* [Scheme Procedure]
: The type of the network number. Currently, this returns only `AF_INET`.

`netent:net` *net* [Scheme Procedure]
: The network number.

The following procedures are used to search the network database:

`getnet` [*net*] [Scheme Procedure]
`getnetbyname` *net-name* [Scheme Procedure]
`getnetbyaddr` *net-number* [Scheme Procedure]
`scm_getnet` (*net*) [C Function]
: Look up a network by name or net number in the network database. The *net-name* argument must be a string, and the *net-number* argument must be an integer. `getnet` will accept either type of argument, behaving like `getnetent` (see below) if no arguments are given.

The following procedures may be used to step through the network database from beginning to end.

`setnetent` [*stayopen*] [Scheme Procedure]
: Initialize an internal stream from which network objects may be read. This procedure must be called before any calls to `getnetent`, and may also be called afterward to reset the net entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `getnetbyname` or `getnetbyaddr` calls, possibly giving an efficiency gain.

`getnetent` [Scheme Procedure]
: Return the next entry from the network database.

`endnetent` [Scheme Procedure]
: Close the stream used by `getnetent`. The return value is unspecified.

`setnet` [*stayopen*] [Scheme Procedure]
`scm_setnet` (*stayopen*) [C Function]
: If *stayopen* is omitted, this is equivalent to `endnetent`. Otherwise it is equivalent to `setnetent stayopen`.

The Protocol Database

The following functions accept an object representing a protocol and return a selected component:

`protoent:name` *protocol* [Scheme Procedure]
: The "official" protocol name.

`protoent:aliases` *protocol* [Scheme Procedure]
: A list of aliases for the protocol.

`protoent:proto` *protocol* [Scheme Procedure]
: The protocol number.

The following procedures are used to search the protocol database:

`getproto` [*protocol*] [Scheme Procedure]
`getprotobyname` *name* [Scheme Procedure]
`getprotobynumber` *number* [Scheme Procedure]
`scm_getproto` (*protocol*) [C Function]
: Look up a network protocol by name or by number. `getprotobyname` takes a string argument, and `getprotobynumber` takes an integer argument. `getproto` will accept either type, behaving like `getprotoent` (see below) if no arguments are supplied.

The following procedures may be used to step through the protocol database from beginning to end.

`setprotoent` [*stayopen*] [Scheme Procedure]
: Initialize an internal stream from which protocol objects may be read. This procedure must be called before any calls to `getprotoent`, and may also be called afterward to

reset the protocol entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `getprotobyname` or `getprotobynumber` calls, possibly giving an efficiency gain.

`getprotoent` [Scheme Procedure]
Return the next entry from the protocol database.

`endprotoent` [Scheme Procedure]
Close the stream used by `getprotoent`. The return value is unspecified.

`setproto` [*stayopen*] [Scheme Procedure]
`scm_setproto` (*stayopen*) [C Function]
If *stayopen* is omitted, this is equivalent to `endprotoent`. Otherwise it is equivalent to `setprotoent stayopen`.

The Service Database

The following functions accept an object representing a service and return a selected component:

`servent:name` *serv* [Scheme Procedure]
The "official" name of the network service.

`servent:aliases` *serv* [Scheme Procedure]
A list of aliases for the network service.

`servent:port` *serv* [Scheme Procedure]
The Internet port used by the service.

`servent:proto` *serv* [Scheme Procedure]
The protocol used by the service. A service may be listed many times in the database under different protocol names.

The following procedures are used to search the service database:

`getserv` [*name* [*protocol*]] [Scheme Procedure]
`getservbyname` *name protocol* [Scheme Procedure]
`getservbyport` *port protocol* [Scheme Procedure]
`scm_getserv` (*name*, *protocol*) [C Function]
Look up a network service by name or by service number, and return a network service object. The *protocol* argument specifies the name of the desired protocol; if the protocol found in the network service database does not match this name, a system error is signalled.

The `getserv` procedure will take either a service name or number as its first argument; if given no arguments, it behaves like `getservent` (see below).

```
(getserv "imap" "tcp")
⇒ #("imap2" ("imap") 143 "tcp")

(getservbyport 88 "udp")
⇒ #("kerberos" ("kerberos5" "krb5") 88 "udp")
```

The following procedures may be used to step through the service database from beginning to end.

`setservent` [*stayopen*] [Scheme Procedure]
: Initialize an internal stream from which service objects may be read. This procedure must be called before any calls to `getservent`, and may also be called afterward to reset the service entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `getservbyname` or `getservbyport` calls, possibly giving an efficiency gain.

`getservent` [Scheme Procedure]
: Return the next entry from the services database.

`endservent` [Scheme Procedure]
: Close the stream used by `getservent`. The return value is unspecified.

`setserv` [*stayopen*] [Scheme Procedure]
`scm_setserv` (*stayopen*) [C Function]
: If *stayopen* is omitted, this is equivalent to `endservent`. Otherwise it is equivalent to `setservent stayopen`.

7.2.11.3 Network Socket Address

A *socket address* object identifies a socket endpoint for communication. In the case of `AF_INET` for instance, the socket address object comprises the host address (or interface on the host) and a port number which specifies a particular open socket in a running client or server process. A socket address object can be created with,

`make-socket-address` *AF_INET ipv4addr port* [Scheme Procedure]
`make-socket-address` *AF_INET6 ipv6addr port* [*flowinfo* [*scopeid*]] [Scheme Procedure]
`make-socket-address` *AF_UNIX path* [Scheme Procedure]
`scm_make_socket_address` (*family, address, arglist*) [C Function]
: Return a new socket address object. The first argument is the address family, one of the `AF` constants, then the arguments vary according to the family.

  For `AF_INET` the arguments are an IPv4 network address number (см. Раздел 7.2.11.1 [Network Address Conversion], страница 554), and a port number.

  For `AF_INET6` the arguments are an IPv6 network address number and a port number. Optional *flowinfo* and *scopeid* arguments may be given (both integers, default 0).

  For `AF_UNIX` the argument is a filename (a string).

  The C function `scm_make_socket_address` takes the *family* and *address* arguments directly, then *arglist* is a list of further arguments, being the port for IPv4, port and optional flowinfo and scopeid for IPv6, or the empty list `SCM_EOL` for Unix domain.

The following functions access the fields of a socket address object,

`sockaddr:fam` *sa* [Scheme Procedure]
: Return the address family from socket address object *sa*. This is one of the `AF` constants (e.g. `AF_INET`).

`sockaddr:path` *sa* [Scheme Procedure]
For an `AF_UNIX` socket address object *sa*, return the filename.

`sockaddr:addr` *sa* [Scheme Procedure]
For an `AF_INET` or `AF_INET6` socket address object *sa*, return the network address number.

`sockaddr:port` *sa* [Scheme Procedure]
For an `AF_INET` or `AF_INET6` socket address object *sa*, return the port number.

`sockaddr:flowinfo` *sa* [Scheme Procedure]
For an `AF_INET6` socket address object *sa*, return the flowinfo value.

`sockaddr:scopeid` *sa* [Scheme Procedure]
For an `AF_INET6` socket address object *sa*, return the scope ID value.

The functions below convert to and from the C `struct sockaddr` (см. Раздел "Address Formats" в *The GNU C Library Reference Manual*). That structure is a generic type, an application can cast to or from `struct sockaddr_in`, `struct sockaddr_in6` or `struct sockaddr_un` according to the address family.

In a `struct sockaddr` taken or returned, the byte ordering in the fields follows the C conventions (см. Раздел "Byte Order Conversion" в *The GNU C Library Reference Manual*). This means network byte order for `AF_INET` host address (`sin_addr.s_addr`) and port number (`sin_port`), and `AF_INET6` port number (`sin6_port`). But at the Scheme level these values are taken or returned in host byte order, so the port is an ordinary integer, and the host address likewise is an ordinary integer (as described in Раздел 7.2.11.1 [Network Address Conversion], страница 554).

`struct sockaddr * scm_c_make_socket_address` (*SCM family, SCM address, SCM args, size_t *outsize*) [C Function]
Return a newly-`malloc`ed `struct sockaddr` created from arguments like those taken by `scm_make_socket_address` above.

The size (in bytes) of the `struct sockaddr` return is stored into `*outsize`. An application must call `free` to release the returned structure when no longer required.

`SCM scm_from_sockaddr` (*const struct sockaddr *address, unsigned address_size*) [C Function]
Return a Scheme socket address object from the C *address* structure. *address_size* is the size in bytes of *address*.

`struct sockaddr * scm_to_sockaddr` (*SCM address, size_t *address_size*) [C Function]
Return a newly-`malloc`ed `struct sockaddr` from a Scheme level socket address object.

The size (in bytes) of the `struct sockaddr` return is stored into `*outsize`. An application must call `free` to release the returned structure when no longer required.

7.2.11.4 Network Sockets and Communication

Socket ports can be created using `socket` and `socketpair`. The ports are initially unbuffered, to make reading and writing to the same port more reliable. A buffer can be added to the port using `setvbuf` (см. Раздел 6.14.6 [Buffering], страница 360).

Most systems have limits on how many files and sockets can be open, so it's strongly recommended that socket ports be closed explicitly when no longer required (см. Раздел 6.14.1 [Ports], страница 352).

Some of the underlying C functions take values in network byte order, but the convention in Guile is that at the Scheme level everything is ordinary host byte order and conversions are made automatically where necessary.

`socket` *family style proto* [Scheme Procedure]
`scm_socket` (*family, style, proto*) [C Function]
Return a new socket port of the type specified by *family*, *style* and *proto*. All three parameters are integers. The possible values for *family* are as follows, where supported by the system,

`PF_UNIX` [Переменная]
`PF_INET` [Переменная]
`PF_INET6` [Переменная]
The possible values for *style* are as follows, again where supported by the system,

`SOCK_STREAM` [Переменная]
`SOCK_DGRAM` [Переменная]
`SOCK_RAW` [Переменная]
`SOCK_RDM` [Переменная]
`SOCK_SEQPACKET` [Переменная]
proto can be obtained from a protocol name using `getprotobyname` (см. Раздел 7.2.11.2 [Network Databases], страница 555). A value of zero means the default protocol, which is usually right.

A socket cannot by used for communication until it has been connected somewhere, usually with either `connect` or `accept` below.

`socketpair` *family style proto* [Scheme Procedure]
`scm_socketpair` (*family, style, proto*) [C Function]
Return a pair, the `car` and `cdr` of which are two unnamed socket ports connected to each other. The connection is full-duplex, so data can be transferred in either direction between the two.

family, *style* and *proto* are as per `socket` above. But many systems only support socket pairs in the `PF_UNIX` family. Zero is likely to be the only meaningful value for *proto*.

`getsockopt` *sock level optname* [Scheme Procedure]
`setsockopt` *sock level optname value* [Scheme Procedure]
`scm_getsockopt` (*sock, level, optname*) [C Function]
`scm_setsockopt` (*sock, level, optname, value*) [C Function]
Get or set an option on socket port *sock*. `getsockopt` returns the current value. `setsockopt` sets a value and the return is unspecified.

level is an integer specifying a protocol layer, either `SOL_SOCKET` for socket level options, or a protocol number from the `IPPROTO` constants or `getprotoent` (см. Раздел 7.2.11.2 [Network Databases], страница 555).

`SOL_SOCKET` [Переменная]
`IPPROTO_IP` [Переменная]
`IPPROTO_TCP` [Переменная]
`IPPROTO_UDP` [Переменная]

optname is an integer specifying an option within the protocol layer.

For `SOL_SOCKET` level the following *optname*s are defined (when provided by the system). For their meaning see Раздел "Socket-Level Options" в *The GNU C Library Reference Manual*, or `man 7 socket`.

`SO_DEBUG` [Переменная]
`SO_REUSEADDR` [Переменная]
`SO_STYLE` [Переменная]
`SO_TYPE` [Переменная]
`SO_ERROR` [Переменная]
`SO_DONTROUTE` [Переменная]
`SO_BROADCAST` [Переменная]
`SO_SNDBUF` [Переменная]
`SO_RCVBUF` [Переменная]
`SO_KEEPALIVE` [Переменная]
`SO_OOBINLINE` [Переменная]
`SO_NO_CHECK` [Переменная]
`SO_PRIORITY` [Переменная]
`SO_REUSEPORT` [Переменная]

The *value* taken or returned is an integer.

`SO_LINGER` [Переменная]

The *value* taken or returned is a pair of integers `(`*`ENABLE`*` . `*`TIMEOUT`*`)`. On old systems without timeout support (ie. without `struct linger`), only *ENABLE* has an effect but the value in Guile is always a pair.

For IP level (`IPPROTO_IP`) the following *optname*s are defined (when provided by the system). See `man ip` for what they mean.

`IP_MULTICAST_IF` [Переменная]

This sets the source interface used by multicast traffic.

`IP_MULTICAST_TTL` [Переменная]

This sets the default TTL for multicast traffic. This defaults to 1 and should be increased to allow traffic to pass beyond the local network.

`IP_ADD_MEMBERSHIP` [Переменная]
`IP_DROP_MEMBERSHIP` [Переменная]

These can be used only with `setsockopt`, not `getsockopt`. *value* is a pair `(`*`MULTIADDR`*` . `*`INTERFACEADDR`*`)` of integer IPv4 addresses (см. Раздел 7.2.11.1 [Network Address Conversion], страница 554). *MULTIADDR* is a multicast

address to be added to or dropped from the interface *INTERFACEADDR*. *INTERFACEADDR* can be `INADDR_ANY` to have the system select the interface. *INTERFACEADDR* can also be an interface index number, on systems supporting that.

For `IPPROTO_TCP` level the following *optname*s are defined (when provided by the system). For their meaning see `man 7 tcp`.

`TCP_NODELAY` [Переменная]
`TCP_CORK` [Переменная]
The *value* taken or returned is an integer.

`shutdown` *sock how* [Scheme Procedure]
`scm_shutdown` (*sock, how*) [C Function]
Sockets can be closed simply by using `close-port`. The `shutdown` procedure allows reception or transmission on a connection to be shut down individually, according to the parameter *how*:

0 Stop receiving data for this socket. If further data arrives, reject it.

1 Stop trying to transmit data from this socket. Discard any data waiting to be sent. Stop looking for acknowledgement of data already sent; don't retransmit it if it is lost.

2 Stop both reception and transmission.

The return value is unspecified.

`connect` *sock sockaddr* [Scheme Procedure]
`connect` *sock AF_INET ipv4addr port* [Scheme Procedure]
`connect` *sock AF_INET6 ipv6addr port* **[***flowinfo* **[***scopeid***]]** [Scheme Procedure]
`connect` *sock AF_UNIX path* [Scheme Procedure]
`scm_connect` (*sock, fam, address, args*) [C Function]
Initiate a connection on socket port *sock* to a given address. The destination is either a socket address object, or arguments the same as `make-socket-address` would take to make such an object (см. Раздел 7.2.11.3 [Network Socket Address], страница 562). Return true unless the socket was configured as non-blocking and the connection could not be made immediately.

```
(connect sock AF_INET INADDR_LOOPBACK 23)
(connect sock (make-socket-address AF_INET INADDR_LOOPBACK 23))
```

`bind` *sock sockaddr* [Scheme Procedure]
`bind` *sock AF_INET ipv4addr port* [Scheme Procedure]
`bind` *sock AF_INET6 ipv6addr port* **[***flowinfo* **[***scopeid***]]** [Scheme Procedure]
`bind` *sock AF_UNIX path* [Scheme Procedure]
`scm_bind` (*sock, fam, address, args*) [C Function]
Bind socket port *sock* to the given address. The address is either a socket address object, or arguments the same as `make-socket-address` would take to make such an object (см. Раздел 7.2.11.3 [Network Socket Address], страница 562). The return value is unspecified.

Generally a socket is only explicitly bound to a particular address when making a server, i.e. to listen on a particular port. For an outgoing connection the system will assign a local address automatically, if not already bound.

```
(bind sock AF_INET INADDR_ANY 12345)
(bind sock (make-socket-address AF_INET INADDR_ANY 12345))
```

`listen` *sock backlog* [Scheme Procedure]
`scm_listen` (*sock, backlog*) [C Function]

Enable *sock* to accept connection requests. *backlog* is an integer specifying the maximum length of the queue for pending connections. If the queue fills, new clients will fail to connect until the server calls `accept` to accept a connection from the queue.

The return value is unspecified.

`accept` *sock* [*flags*] [Scheme Procedure]
`scm_accept` (*sock*) [C Function]

Accept a connection from socket port *sock* which has been enabled for listening with `listen` above.

If there are no incoming connections in the queue, there are two possible behaviors, depending on whether *sock* has been configured for non-blocking operation or not:

- If there is no connection waiting and the socket was set to non-blocking mode with the `O_NONBLOCK` port option (см. Раздел 7.2.2 [Ports and File Descriptors], страница 521), return `#f` directly.
- Otherwise wait until a connection is available.

The return value is a pair. The `car` is a new socket port, connected and ready to communicate. The `cdr` is a socket address object (см. Раздел 7.2.11.3 [Network Socket Address], страница 562) which is where the remote connection is from (like `getpeername` below).

flags, if given, may include `SOCK_CLOEXEC` or `SOCK_NONBLOCK`, which like `O_CLOEXEC` and `O_NONBLOCK` apply to the newly accepted socket.

All communication takes place using the new socket returned. The given *sock* remains bound and listening, and `accept` may be called on it again to get another incoming connection when desired.

`getsockname` *sock* [Scheme Procedure]
`scm_getsockname` (*sock*) [C Function]

Return a socket address object which is the where *sock* is bound locally. *sock* may have obtained its local address from `bind` (above), or if a `connect` is done with an otherwise unbound socket (which is usual) then the system will have assigned an address.

Note that on many systems the address of a socket in the `AF_UNIX` namespace cannot be read.

`getpeername` *sock* [Scheme Procedure]
`scm_getpeername` (*sock*) [C Function]

Return a socket address object which is where *sock* is connected to, i.e. the remote endpoint.

Note that on many systems the address of a socket in the `AF_UNIX` namespace cannot be read.

`recv!` *sock buf* [*flags*] [Scheme Procedure]
`scm_recv` (*sock, buf, flags*) [C Function]
Receive data from a socket port. *sock* must already be bound to the address from which data is to be received. *buf* is a bytevector into which the data will be written. The size of *buf* limits the amount of data which can be received: in the case of packet protocols, if a packet larger than this limit is encountered then some data will be irrevocably lost.

The optional *flags* argument is a value or bitwise OR of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

The value returned is the number of bytes read from the socket.

Note that the data is read directly from the socket file descriptor: any unread buffered port data is ignored.

`send` *sock message* [*flags*] [Scheme Procedure]
`scm_send` (*sock, message, flags*) [C Function]
Transmit bytevector *message* on socket port *sock*. *sock* must already be bound to a destination address. The value returned is the number of bytes transmitted—it's possible for this to be less than the length of *message* if the socket is set to be non-blocking. The optional *flags* argument is a value or bitwise OR of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

Note that the data is written directly to the socket file descriptor: any unflushed buffered port data is ignored.

`recvfrom!` *sock buf* [*flags* [*start* [*end*]]] [Scheme Procedure]
`scm_recvfrom` (*sock, buf, flags, start, end*) [C Function]
Receive data from socket port *sock*, returning the originating address as well as the data. This function is usually for datagram sockets, but can be used on stream-oriented sockets too.

The data received is stored in bytevector *buf*, using either the whole bytevector or just the region between the optional *start* and *end* positions. The size of *buf* limits the amount of data that can be received. For datagram protocols if a packet larger than this is received then excess bytes are irrevocably lost.

The return value is a pair. The `car` is the number of bytes read. The `cdr` is a socket address object (см. Раздел 7.2.11.3 [Network Socket Address], страница 562) which is where the data came from, or `#f` if the origin is unknown.

The optional *flags* argument is a or bitwise-OR (`logior`) of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

Data is read directly from the socket file descriptor, any buffered port data is ignored.

On a GNU/Linux system `recvfrom!` is not multi-threading, all threads stop while a `recvfrom!` call is in progress. An application may need to use `select`, `O_NONBLOCK` or `MSG_DONTWAIT` to avoid this.

`sendto` *sock message sockaddr* [*flags*] [Scheme Procedure]
`sendto` *sock message AF_INET ipv4addr port* [*flags*] [Scheme Procedure]
`sendto` *sock message AF_INET6 ipv6addr port* [*flowinfo* [*scopeid* [*flags*]]] [Scheme Procedure]
`sendto` *sock message AF_UNIX path* [*flags*] [Scheme Procedure]
`scm_sendto` (*sock, message, fam, address, args_and_flags*) [C Function]

Transmit bytevector *message* as a datagram socket port *sock*. The destination is specified either as a socket address object, or as arguments the same as would be taken by `make-socket-address` to create such an object (см. Раздел 7.2.11.3 [Network Socket Address], страница 562).

The destination address may be followed by an optional *flags* argument which is a `logior` (см. Раздел 6.6.2.13 [Bitwise Operations], страница 134) of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

The value returned is the number of bytes transmitted – it's possible for this to be less than the length of *message* if the socket is set to be non-blocking. Note that the data is written directly to the socket file descriptor: any unflushed buffered port data is ignored.

7.2.11.5 Network Socket Examples

The following give examples of how to use network sockets.

Internet Socket Client Example

The following example demonstrates an Internet socket client. It connects to the HTTP daemon running on the local machine and returns the contents of the root index URL.

```
(let ((s (socket PF_INET SOCK_STREAM 0)))
  (connect s AF_INET (inet-pton AF_INET "127.0.0.1") 80)
  (display "GET / HTTP/1.0\r\n\r\n" s)

  (do ((line (read-line s) (read-line s)))
      ((eof-object? line))
    (display line)
    (newline)))
```

Internet Socket Server Example

The following example shows a simple Internet server which listens on port 2904 for incoming connections and sends a greeting back to the client.

```
(let ((s (socket PF_INET SOCK_STREAM 0)))
  (setsockopt s SOL_SOCKET SO_REUSEADDR 1)
  ;; Specific address?
  ;; (bind s AF_INET (inet-pton AF_INET "127.0.0.1") 2904)
  (bind s AF_INET INADDR_ANY 2904)
  (listen s 5)

  (simple-format #t "Listening for clients in pid: ~S" (getpid))
  (newline)
```

```
(while #t
  (let* ((client-connection (accept s))
         (client-details (cdr client-connection))
         (client (car client-connection)))
    (simple-format #t "Got new client connection: ~S"
                   client-details)
    (newline)
    (simple-format #t "Client address: ~S"
                   (gethostbyaddr
                    (sockaddr:addr client-details)))
    (newline)
    ;; Send back the greeting to the client port
    (display "Hello client\r\n" client)
    (close client))))
```

7.2.12 System Identification

This section lists the various procedures Guile provides for accessing information about the system it runs on.

`uname` [Scheme Procedure]
`scm_uname` () [C Function]

Return an object with some information about the computer system the program is running on.

The following procedures accept an object as returned by `uname` and return a selected component (all of which are strings).

`utsname:sysname` *un* [Scheme Procedure]

The name of the operating system.

`utsname:nodename` *un* [Scheme Procedure]

The network name of the computer.

`utsname:release` *un* [Scheme Procedure]

The current release level of the operating system implementation.

`utsname:version` *un* [Scheme Procedure]

The current version level within the release of the operating system.

`utsname:machine` *un* [Scheme Procedure]

A description of the hardware.

`gethostname` [Scheme Procedure]
`scm_gethostname` () [C Function]

Return the host name of the current processor.

`sethostname` *name* [Scheme Procedure]
`scm_sethostname` (*name*) [C Function]

Set the host name of the current processor to *name*. May only be used by the superuser. The return value is not specified.

7.2.13 Locales

`setlocale` *category* [*locale*] [Scheme Procedure]
`scm_setlocale` (*category*, *locale*) [C Function]
Get or set the current locale, used for various internationalizations. Locales are strings, such as '`sv_SE`'.

If *locale* is given then the locale for the given *category* is set and the new value returned. If *locale* is not given then the current value is returned. *category* should be one of the following values (см. Раздел "Locale Categories" в *The GNU C Library Reference Manual*):

`LC_ALL` [Переменная]
`LC_COLLATE` [Переменная]
`LC_CTYPE` [Переменная]
`LC_MESSAGES` [Переменная]
`LC_MONETARY` [Переменная]
`LC_NUMERIC` [Переменная]
`LC_TIME` [Переменная]
A common usage is '`(setlocale LC_ALL "")`', which initializes all categories based on standard environment variables (`LANG` etc). For full details on categories and locale names см. Раздел "Locales and Internationalization" в *The GNU C Library Reference Manual*.

Note that `setlocale` affects locale settings for the whole process. См. Раздел 6.25.1 [i18n Introduction], страница 486, for a thread-safe alternative.

7.2.14 Encryption

Please note that the procedures in this section are not suited for strong encryption, they are only interfaces to the well-known and common system library functions of the same name. They are just as good (or bad) as the underlying functions, so you should refer to your system documentation before using them (см. Раздел "Encrypting Passwords" в *The GNU C Library Reference Manual*).

`crypt` *key salt* [Scheme Procedure]
`scm_crypt` (*key*, *salt*) [C Function]
Encrypt *key*, with the addition of *salt* (both strings), using the `crypt` C library call.

Although `getpass` is not an encryption procedure per se, it appears here because it is often used in combination with `crypt`:

`getpass` *prompt* [Scheme Procedure]
`scm_getpass` (*prompt*) [C Function]
Display *prompt* to the standard error output and read a password from `/dev/tty`. If this file is not accessible, it reads from standard input. The password may be up to 127 characters in length. Additional characters and the terminating newline character are discarded. While reading the password, echoing and the generation of signals by special characters is disabled.

7.3 HTTP, the Web, and All That

It has always been possible to connect computers together and share information between them, but the rise of the World Wide Web over the last couple of decades has made it much easier to do so. The result is a richly connected network of computation, in which Guile forms a part.

By "the web", we mean the HTTP protocol[2] as handled by servers, clients, proxies, caches, and the various kinds of messages and message components that can be sent and received by that protocol, notably HTML.

On one level, the web is text in motion: the protocols themselves are textual (though the payload may be binary), and it's possible to create a socket and speak text to the web. But such an approach is obviously primitive. This section details the higher-level data types and operations provided by Guile: URIs, HTTP request and response records, and a conventional web server implementation.

The material in this section is arranged in ascending order, in which later concepts build on previous ones. If you prefer to start with the highest-level perspective, см. Раздел 7.3.10 [Web Examples], страница 597, and work your way back.

7.3.1 Types and the Web

It is a truth universally acknowledged, that a program with good use of data types, will be free from many common bugs. Unfortunately, the common practice in web programming seems to ignore this maxim. This subsection makes the case for expressive data types in web programming.

By "expressive data types", we mean that the data types *say* something about how a program solves a problem. For example, if we choose to represent dates using SRFI 19 date records (см. Раздел 7.5.16 [SRFI-19], страница 637), this indicates that there is a part of the program that will always have valid dates. Error handling for a number of basic cases, like invalid dates, occurs on the boundary in which we produce a SRFI 19 date record from other types, like strings.

With regards to the web, data types are helpful in the two broad phases of HTTP messages: parsing and generation.

Consider a server, which has to parse a request, and produce a response. Guile will parse the request into an HTTP request object (см. Раздел 7.3.6 [Requests], страница 588), with each header parsed into an appropriate Scheme data type. This transition from an incoming stream of characters to typed data is a state change in a program—the strings might parse, or they might not, and something has to happen if they do not. (Guile throws an error in this case.) But after you have the parsed request, "client" code (code built on top of the Guile web framework) will not have to check for syntactic validity. The types already make this information manifest.

This state change on the parsing boundary makes programs more robust, as they themselves are freed from the need to do a number of common error checks, and they can use normal Scheme procedures to handle a request instead of ad-hoc string parsers.

[2] Yes, the P is for protocol, but this phrase appears repeatedly in RFC 2616.

The need for types on the response generation side (in a server) is more subtle, though not less important. Consider the example of a POST handler, which prints out the text that a user submits from a form. Such a handler might include a procedure like this:

```
;; First, a helper procedure
(define (para . contents)
  (string-append "<p>" (string-concatenate contents) "</p>"))

;; Now the meat of our simple web application
(define (you-said text)
  (para "You said: " text))

(display (you-said "Hi!"))
⊣ <p>You said: Hi!</p>
```

This is a perfectly valid implementation, provided that the incoming text does not contain the special HTML characters '<', '>', or '&'. But this provision of a restricted character set is not reflected anywhere in the program itself: we must *assume* that the programmer understands this, and performs the check elsewhere.

Unfortunately, the short history of the practice of programming does not bear out this assumption. A *cross-site scripting* (XSS) vulnerability is just such a common error in which unfiltered user input is allowed into the output. A user could submit a crafted comment to your web site which results in visitors running malicious Javascript, within the security context of your domain:

```
(display (you-said "<script src=\"http://bad.com/nasty.js\" />"))
⊣ <p>You said: <script src="http://bad.com/nasty.js" /></p>
```

The fundamental problem here is that both user data and the program template are represented using strings. This identity means that types can't help the programmer to make a distinction between these two, so they get confused.

There are a number of possible solutions, but perhaps the best is to treat HTML not as strings, but as native s-expressions: as SXML. The basic idea is that HTML is either text, represented by a string, or an element, represented as a tagged list. So '`foo`' becomes '`"foo"`', and '`<b>foo</b>`' becomes '`(b "foo")`'. Attributes, if present, go in a tagged list headed by '`@`', like '`(img (@ (src "http://example.com/foo.png")))`'. См. Раздел 7.20 [SXML], страница 761, for more information.

The good thing about SXML is that HTML elements cannot be confused with text. Let's make a new definition of `para`:

```
(define (para . contents)
  `(p ,@contents))

(use-modules (sxml simple))
(sxml->xml (you-said "Hi!"))
⊣ <p>You said: Hi!</p>

(sxml->xml (you-said "<i>Rats, foiled again!</i>"))
⊣ <p>You said: &lt;i&gt;Rats, foiled again!&lt;/i&gt;</p>
```

So we see in the second example that HTML elements cannot be unwittingly introduced into the output. However it is now perfectly acceptable to pass SXML to `you-said`; in fact, that is the big advantage of SXML over everything-as-a-string.

```
(sxml->xml (you-said (you-said "<Hi!>")))
⊣ <p>You said: <p>You said: &lt;Hi!&gt;</p></p>
```

The SXML types allow procedures to *compose*. The types make manifest which parts are HTML elements, and which are text. So you needn't worry about escaping user input; the type transition back to a string handles that for you. XSS vulnerabilities are a thing of the past.

Well. That's all very nice and opinionated and such, but how do I use the thing? Read on!

7.3.2 Universal Resource Identifiers

Guile provides a standard data type for Universal Resource Identifiers (URIs), as defined in RFC 3986.

The generic URI syntax is as follows:

```
URI-reference := [scheme ":"] ["//" [userinfo "@"] host [":" port]] path \
                 [ "?" query ] [ "#" fragment ]
```

For example, in the URI, '`http://www.gnu.org/help/`', the scheme is `http`, the host is `www.gnu.org`, the path is `/help/`, and there is no userinfo, port, query, or fragment.

Userinfo is something of an abstraction, as some legacy URI schemes allowed userinfo of the form *`username:passwd`*. But since passwords do not belong in URIs, the RFC does not want to condone this practice, so it calls anything before the `@` sign *userinfo*.

```
(use-modules (web uri))
```

The following procedures can be found in the `(web uri)` module. Load it into your Guile, using a form like the above, to have access to them.

The most common way to build a URI from Scheme is with the `build-uri` function.

`build-uri` *scheme* **[***#:userinfo=***`#f`**]* **[***#:host=***`#f`**]* **[***#:port=***`#f`**]** [Scheme Procedure]
[*#:path=***`""`**]* **[***#:query=***`#f`**]* **[***#:fragment=***`#f`**]* **[***#:validate?=***`#t`**]**

Construct a URI. *scheme* should be a symbol, *port* either a positive, exact integer or `#f`, and the rest of the fields are either strings or `#f`. If *validate?* is true, also run some consistency checks to make sure that the constructed URI is valid.

`uri?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a URI.

Guile, URIs are represented as URI records, with a number of associated accessors.

`uri-scheme` *uri* [Scheme Procedure]
`uri-userinfo` *uri* [Scheme Procedure]
`uri-host` *uri* [Scheme Procedure]
`uri-port` *uri* [Scheme Procedure]
`uri-path` *uri* [Scheme Procedure]
`uri-query` *uri* [Scheme Procedure]

`uri-fragment` *uri* [Scheme Procedure]

Field accessors for the URI record type. The URI scheme will be a symbol, or `#f` if the object is a relative-ref (see below). The port will be either a positive, exact integer or `#f`, and the rest of the fields will be either strings or `#f` if not present.

`string->uri` *string* [Scheme Procedure]

Parse *string* into a URI object. Return `#f` if the string could not be parsed.

`uri->string` *uri* **[***#:include-fragment?=***`#t`**]* [Scheme Procedure]

Serialize *uri* to a string. If the URI has a port that is the default port for its scheme, the port is not included in the serialization. If *include-fragment?* is given as false, the resulting string will omit the fragment (if any).

`declare-default-port!` *scheme port* [Scheme Procedure]

Declare a default port for the given URI scheme.

`uri-decode` *str* **[***#:encoding=***`"utf-8"`**] **[***#:decode-plus-to-space? #t***]** [Scheme Procedure]

Percent-decode the given *str*, according to *encoding*, which should be the name of a character encoding.

Note that this function should not generally be applied to a full URI string. For paths, use `split-and-decode-uri-path` instead. For query strings, split the query on `&` and `=` boundaries, and decode the components separately.

Note also that percent-encoded strings encode *bytes*, not characters. There is no guarantee that a given byte sequence is a valid string encoding. Therefore this routine may signal an error if the decoded bytes are not valid for the given encoding. Pass `#f` for *encoding* if you want decoded bytes as a bytevector directly. См. Раздел 6.14.1 [Ports], страница 352, for more information on character encodings.

If *decode-plus-to-space?* is true, which is the default, also replace instances of the plus character '`+`' with a space character. This is needed when parsing `application/x-www-form-urlencoded` data.

Returns a string of the decoded characters, or a bytevector if *encoding* was `#f`.

`uri-encode` *str* **[***#:encoding=***`"utf-8"`**] **[***#:unescaped-chars***]** [Scheme Procedure]

Percent-encode any character not in the character set, *unescaped-chars*.

The default character set includes alphanumerics from ASCII, as well as the special characters '`-`', '`.`', '`_`', and '`~`'. Any other character will be percent-encoded, by writing out the character to a bytevector within the given *encoding*, then encoding each byte as *`%HH`*, where *HH* is the hexadecimal representation of the byte.

`split-and-decode-uri-path` *path* [Scheme Procedure]

Split *path* into its components, and decode each component, removing empty components.

For example, `"/foo/bar%20baz/"` decodes to the two-element list, `("foo" "bar baz")`.

`encode-and-join-uri-path` *parts* [Scheme Procedure]
URI-encode each element of *parts*, which should be a list of strings, and join the parts together with `/` as a delimiter.

For example, the list `("scrambled eggs" "biscuits&gravy")` encodes as `"scrambled%20eggs/biscuits%26gravy"`.

Subtypes of URI

As we noted above, not all URI objects have a scheme. You might have noted in the "generic URI syntax" example that the left-hand side of that grammar definition was URI-reference, not URI. A *URI-reference* is a generalization of a URI where the scheme is optional. If no scheme is specified, it is taken to be relative to some other related URI. A common use of URI references is when you want to be vague regarding the choice of HTTP or HTTPS – serving a web page referring to `/foo.css` will use HTTPS if loaded over HTTPS, or HTTP otherwise.

`build-uri-reference` **[***#:scheme=***#f]** **[***#:userinfo=***#f]** [Scheme Procedure]
[*#:host=***#f]** **[***#:port=***#f]** **[***#:path=***""]** **[***#:query=***#f]** **[***#:fragment=***#f]**
[*#:validate?=***#t]**
Like `build-uri`, but with an optional scheme.

`uri-reference?` *obj* [Scheme Procedure]
Return `#t` if *obj* is a URI-reference. This is the most general URI predicate, as it includes not only full URIs that have schemes (those that match `uri?`) but also URIs without schemes.

It's also possible to build a *relative-ref*: a URI-reference that explicitly lacks a scheme.

`build-relative-ref` **[***#:userinfo=***#f]** **[***#:host=***#f]** **[***#:port=***#f]** [Scheme Procedure]
[*#:path=***""]** **[***#:query=***#f]** **[***#:fragment=***#f]** **[***#:validate?=***#t]**
Like `build-uri`, but with no scheme.

`relative-ref?` *obj* [Scheme Procedure]
Return `#t` if *obj* is a "relative-ref": a URI-reference that has no scheme. Every URI-reference will either match `uri?` or `relative-ref?` (but not both).

In case it's not clear from the above, the most general of these URI types is the URI-reference, with `build-uri-reference` as the most general constructor. `build-uri` and `build-relative-ref` enforce enforce specific restrictions on the URI-reference. The most generic URI parser is then `string->uri-reference`, and there is also a parser for when you know that you want a relative-ref.

`string->uri-reference` *string* [Scheme Procedure]
Parse *string* into a URI object, while not requiring a scheme. Return `#f` if the string could not be parsed.

`string->relative-ref` *string* [Scheme Procedure]
Parse *string* into a URI object, while asserting that no scheme is present. Return `#f` if the string could not be parsed.

For compatibility reasons, note that `uri?` will return `#t` for all URI objects, even relative-refs. In contrast, `build-uri` and `string->uri` require that the resulting URI not be a relative-ref. As a predicate to distinguish relative-refs from proper URIs (in the language of RFC 3986), use something like `(and (uri-reference? x) (not (relative-ref? x)))`.

7.3.3 The Hyper-Text Transfer Protocol

The initial motivation for including web functionality in Guile, rather than rely on an external package, was to establish a standard base on which people can share code. To that end, we continue the focus on data types by providing a number of low-level parsers and unparsers for elements of the HTTP protocol.

If you are want to skip the low-level details for now and move on to web pages, см. Раздел 7.3.8 [Web Client], страница 592, and см. Раздел 7.3.9 [Web Server], страница 594. Otherwise, load the HTTP module, and read on.

```
(use-modules (web http))
```

The focus of the `(web http)` module is to parse and unparse standard HTTP headers, representing them to Guile as native data structures. For example, a `Date:` header will be represented as a SRFI-19 date record (см. Раздел 7.5.16 [SRFI-19], страница 637), rather than as a string.

Guile tries to follow RFCs fairly strictly—the road to perdition being paved with compatibility hacks—though some allowances are made for not-too-divergent texts.

Header names are represented as lower-case symbols.

`string->header` *name* [Scheme Procedure]
: Parse *name* to a symbolic header name.

`header->string` *sym* [Scheme Procedure]
: Return the string form for the header named *sym*.

For example:

```
(string->header "Content-Length")
⇒ content-length
(header->string 'content-length)
⇒ "Content-Length"

(string->header "FOO")
⇒ foo
(header->string 'foo)
⇒ "Foo"
```

Guile keeps a registry of known headers, their string names, and some parsing and serialization procedures. If a header is unknown, its string name is simply its symbol name in title-case.

`known-header?` *sym* [Scheme Procedure]
: Return `#t` if *sym* is a known header, with associated parsers and serialization procedures, or `#f` otherwise.

`header-parser` *sym* [Scheme Procedure]
Return the value parser for headers named *sym*. The result is a procedure that takes one argument, a string, and returns the parsed value. If the header isn't known to Guile, a default parser is returned that passes through the string unchanged.

`header-validator` *sym* [Scheme Procedure]
Return a predicate which returns `#t` if the given value is valid for headers named *sym*. The default validator for unknown headers is `string?`.

`header-writer` *sym* [Scheme Procedure]
Return a procedure that writes values for headers named *sym* to a port. The resulting procedure takes two arguments: a value and a port. The default writer is `display`.

For more on the set of headers that Guile knows about out of the box, см. Раздел 7.3.4 [HTTP Headers], страница 579. To add your own, use the `declare-header!` procedure:

`declare-header!` *name parser validator writer* **[***#:multiple?=***`#f`]** [Scheme Procedure]
Declare a parser, validator, and writer for a given header.

For example, let's say you are running a web server behind some sort of proxy, and your proxy adds an `X-Client-Address` header, indicating the IPv4 address of the original client. You would like for the HTTP request record to parse out this header to a Scheme value, instead of leaving it as a string. You could register this header with Guile's HTTP stack like this:

```
(declare-header! "X-Client-Address"
  (lambda (str)
    (inet-aton str))
  (lambda (ip)
    (and (integer? ip) (exact? ip) (<= 0 ip #xffffffff)))
  (lambda (ip port)
    (display (inet-ntoa ip) port)))
```

`declare-opaque-header!` *name* [Scheme Procedure]
A specialised version of `declare-header!` for the case in which you want a header's value to be returned/written "as-is".

`valid-header?` *sym val* [Scheme Procedure]
Return a true value if *val* is a valid Scheme value for the header with name *sym*, or `#f` otherwise.

Now that we have a generic interface for reading and writing headers, we do just that.

`read-header` *port* [Scheme Procedure]
Read one HTTP header from *port*. Return two values: the header name and the parsed Scheme value. May raise an exception if the header was known but the value was invalid.

Returns the end-of-file object for both values if the end of the message body was reached (i.e., a blank line).

`parse-header` *name val* [Scheme Procedure]
Parse *val*, a string, with the parser for the header named *name*. Returns the parsed value.

`write-header` *name val port* [Scheme Procedure]
Write the given header name and value to *port*, using the writer from `header-writer`.

`read-headers` *port* [Scheme Procedure]
Read the headers of an HTTP message from *port*, returning them as an ordered alist.

`write-headers` *headers port* [Scheme Procedure]
Write the given header alist to *port*. Doesn't write the final '`\r\n`', as the user might want to add another header.

The `(web http)` module also has some utility procedures to read and write request and response lines.

`parse-http-method` *str* [*start*] [*end*] [Scheme Procedure]
Parse an HTTP method from *str*. The result is an upper-case symbol, like `GET`.

`parse-http-version` *str* [*start*] [*end*] [Scheme Procedure]
Parse an HTTP version from *str*, returning it as a major–minor pair. For example, `HTTP/1.1` parses as the pair of integers, `(1 . 1)`.

`parse-request-uri` *str* [*start*] [*end*] [Scheme Procedure]
Parse a URI from an HTTP request line. Note that URIs in requests do not have to have a scheme or host name. The result is a URI object.

`read-request-line` *port* [Scheme Procedure]
Read the first line of an HTTP request from *port*, returning three values: the method, the URI, and the version.

`write-request-line` *method uri version port* [Scheme Procedure]
Write the first line of an HTTP request to *port*.

`read-response-line` *port* [Scheme Procedure]
Read the first line of an HTTP response from *port*, returning three values: the HTTP version, the response code, and the "reason phrase".

`write-response-line` *version code reason-phrase port* [Scheme Procedure]
Write the first line of an HTTP response to *port*.

7.3.4 HTTP Headers

In addition to defining the infrastructure to parse headers, the `(web http)` module defines specific parsers and unparsers for all headers defined in the HTTP/1.1 standard.

For example, if you receive a header named '`Accept-Language`' with a value '`en, es;q=0.8`', Guile parses it as a quality list (defined below):

```
(parse-header 'accept-language "en, es;q=0.8")
⇒ ((1000 . "en") (800 . "es"))
```

The format of the value for '`Accept-Language`' headers is defined below, along with all other headers defined in the HTTP standard. (If the header were unknown, the value would have been returned as a string.)

For brevity, the header definitions below are given in the form, *Type* ***name***, indicating that values for the header ***name*** will be of the given *Type*. Since Guile internally treats header names in lower case, in this document we give types title-cased names. A short description of the each header's purpose and an example follow.

For full details on the meanings of all of these headers, see the HTTP 1.1 standard, RFC 2616.

7.3.4.1 HTTP Header Types

Here we define the types that are used below, when defining headers.

`Date` [HTTP Header Type]

A SRFI-19 date.

`KVList` [HTTP Header Type]

A list whose elements are keys or key-value pairs. Keys are parsed to symbols. Values are strings by default. Non-string values are the exception, and are mentioned explicitly below, as appropriate.

`SList` [HTTP Header Type]

A list of strings.

`Quality` [HTTP Header Type]

An exact integer between 0 and 1000. Qualities are used to express preference, given multiple options. An option with a quality of 870, for example, is preferred over an option with quality 500.

(Qualities are written out over the wire as numbers between 0.0 and 1.0, but since the standard only allows three digits after the decimal, it's equivalent to integers between 0 and 1000, so that's what Guile uses.)

`QList` [HTTP Header Type]

A quality list: a list of pairs, the car of which is a quality, and the cdr a string. Used to express a list of options, along with their qualities.

`ETag` [HTTP Header Type]

An entity tag, represented as a pair. The car of the pair is an opaque string, and the cdr is `#t` if the entity tag is a "strong" entity tag, and `#f` otherwise.

7.3.4.2 General Headers

General HTTP headers may be present in any HTTP message.

`KVList cache-control` [HTTP Header]

A key-value list of cache-control directives. See RFC 2616, for more details.

If present, parameters to `max-age`, `max-stale`, `min-fresh`, and `s-maxage` are all parsed as non-negative integers.

If present, parameters to `private` and `no-cache` are parsed as lists of header names, as symbols.

```
(parse-header 'cache-control "no-cache,no-store"
⇒ (no-cache no-store)
(parse-header 'cache-control "no-cache=\"Authorization,Date\",no-store"
⇒ ((no-cache . (authorization date)) no-store)
(parse-header 'cache-control "no-cache=\"Authorization,Date\",max-age=10"
⇒ ((no-cache . (authorization date)) (max-age . 10))
```

`List connection` [HTTP Header]

A list of header names that apply only to this HTTP connection, as symbols. Additionally, the symbol '`close`' may be present, to indicate that the server should close the connection after responding to the request.

```
(parse-header 'connection "close")
⇒ (close)
```

`Date date` [HTTP Header]

The date that a given HTTP message was originated.

```
(parse-header 'date "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

`KVList pragma` [HTTP Header]

A key-value list of implementation-specific directives.

```
(parse-header 'pragma "no-cache, broccoli=tasty")
⇒ (no-cache (broccoli . "tasty"))
```

`List trailer` [HTTP Header]

A list of header names which will appear after the message body, instead of with the message headers.

```
(parse-header 'trailer "ETag")
⇒ (etag)
```

`List transfer-encoding` [HTTP Header]

A list of transfer codings, expressed as key-value lists. The only transfer coding defined by the specification is `chunked`.

```
(parse-header 'transfer-encoding "chunked")
⇒ ((chunked))
```

`List upgrade` [HTTP Header]

A list of strings, indicating additional protocols that a server could use in response to a request.

```
(parse-header 'upgrade "WebSocket")
⇒ ("WebSocket")
```

FIXME: parse out more fully?

`List via` [HTTP Header]

A list of strings, indicating the protocol versions and hosts of intermediate servers and proxies. There may be multiple `via` headers in one message.

```
(parse-header 'via "1.0 venus, 1.1 mars")
⇒ ("1.0 venus" "1.1 mars")
```

`List warning` [HTTP Header]

A list of warnings given by a server or intermediate proxy. Each warning is a itself a list of four elements: a code, as an exact integer between 0 and 1000, a host as a string, the warning text as a string, and either `#f` or a SRFI-19 date.

There may be multiple `warning` headers in one message.

```
(parse-header 'warning "123 foo \"core breach imminent\"")
⇒ ((123 "foo" "core-breach imminent" #f))
```

7.3.4.3 Entity Headers

Entity headers may be present in any HTTP message, and refer to the resource referenced in the HTTP request or response.

`List allow` [HTTP Header]

A list of allowed methods on a given resource, as symbols.

```
(parse-header 'allow "GET, HEAD")
⇒ (GET HEAD)
```

`List content-encoding` [HTTP Header]

A list of content codings, as symbols.

```
(parse-header 'content-encoding "gzip")
⇒ (gzip)
```

`List content-language` [HTTP Header]

The languages that a resource is in, as strings.

```
(parse-header 'content-language "en")
⇒ ("en")
```

`UInt content-length` [HTTP Header]

The number of bytes in a resource, as an exact, non-negative integer.

```
(parse-header 'content-length "300")
⇒ 300
```

`URI content-location` [HTTP Header]

The canonical URI for a resource, in the case that it is also accessible from a different URI.

```
(parse-header 'content-location "http://example.com/foo")
⇒ #<<uri> ...>
```

`String content-md5` [HTTP Header]

The MD5 digest of a resource.

```
(parse-header 'content-md5 "ffaea1a79810785575e29e2bd45e2fa5")
⇒ "ffaea1a79810785575e29e2bd45e2fa5"
```

`List content-range` [HTTP Header]
A range specification, as a list of three elements: the symbol `bytes`, either the symbol `*` or a pair of integers, indicating the byte rage, and either `*` or an integer, for the instance length. Used to indicate that a response only includes part of a resource.

```
(parse-header 'content-range "bytes 10-20/*")
⇒ (bytes (10 . 20) *)
```

`List content-type` [HTTP Header]
The MIME type of a resource, as a symbol, along with any parameters.

```
(parse-header 'content-type "text/plain")
⇒ (text/plain)
(parse-header 'content-type "text/plain;charset=utf-8")
⇒ (text/plain (charset . "utf-8"))
```

Note that the `charset` parameter is something is a misnomer, and the HTTP specification admits this. It specifies the *encoding* of the characters, not the character set.

`Date expires` [HTTP Header]
The date/time after which the resource given in a response is considered stale.

```
(parse-header 'expires "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

`Date last-modified` [HTTP Header]
The date/time on which the resource given in a response was last modified.

```
(parse-header 'expires "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

7.3.4.4 Request Headers

Request headers may only appear in an HTTP request, not in a response.

`List accept` [HTTP Header]
A list of preferred media types for a response. Each element of the list is itself a list, in the same format as `content-type`.

```
(parse-header 'accept "text/html,text/plain;charset=utf-8")
⇒ ((text/html) (text/plain (charset . "utf-8")))
```

Preference is expressed with quality values:

```
(parse-header 'accept "text/html;q=0.8,text/plain;q=0.6")
⇒ ((text/html (q . 800)) (text/plain (q . 600)))
```

`QList accept-charset` [HTTP Header]
A quality list of acceptable charsets. Note again that what HTTP calls a “charset” is what Guile calls a “character encoding”.

```
(parse-header 'accept-charset "iso-8859-5, unicode-1-1;q=0.8")
⇒ ((1000 . "iso-8859-5") (800 . "unicode-1-1"))
```

`QList accept-encoding` [HTTP Header]

A quality list of acceptable content codings.

```
(parse-header 'accept-encoding "gzip,identity=0.8")
⇒ ((1000 . "gzip") (800 . "identity"))
```

`QList accept-language` [HTTP Header]

A quality list of acceptable languages.

```
(parse-header 'accept-language "cn,en=0.75")
⇒ ((1000 . "cn") (750 . "en"))
```

`Pair authorization` [HTTP Header]

Authorization credentials. The car of the pair indicates the authentication scheme, like `basic`. For basic authentication, the cdr of the pair will be the base64-encoded '*user*:*pass*' string. For other authentication schemes, like `digest`, the cdr will be a key-value list of credentials.

```
(parse-header 'authorization "Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ=="
⇒ (basic . "QWxhZGRpbjpvcGVuIHNlc2FtZQ==")
```

`List expect` [HTTP Header]

A list of expectations that a client has of a server. The expectations are key-value lists.

```
(parse-header 'expect "100-continue")
⇒ ((100-continue))
```

`String from` [HTTP Header]

The email address of a user making an HTTP request.

```
(parse-header 'from "bob@example.com")
⇒ "bob@example.com"
```

`Pair host` [HTTP Header]

The host for the resource being requested, as a hostname-port pair. If no port is given, the port is `#f`.

```
(parse-header 'host "gnu.org:80")
⇒ ("gnu.org" . 80)
(parse-header 'host "gnu.org")
⇒ ("gnu.org" . #f)
```

`*|List if-match` [HTTP Header]

A set of etags, indicating that the request should proceed if and only if the etag of the resource is in that set. Either the symbol `*`, indicating any etag, or a list of entity tags.

```
(parse-header 'if-match "*")
⇒ *
(parse-header 'if-match "asdfadf")
⇒ (("asdfadf" . #t))
(parse-header 'if-match W/"asdfadf")
⇒ (("asdfadf" . #f))
```

`Date if-modified-since` [HTTP Header]
Indicates that a response should proceed if and only if the resource has been modified since the given date.

```
(parse-header 'if-modified-since "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

`*|List if-none-match` [HTTP Header]
A set of etags, indicating that the request should proceed if and only if the etag of the resource is not in the set. Either the symbol `*`, indicating any etag, or a list of entity tags.

```
(parse-header 'if-none-match "*")
⇒ *
```

`ETag|Date if-range` [HTTP Header]
Indicates that the range request should proceed if and only if the resource matches a modification date or an etag. Either an entity tag, or a SRFI-19 date.

```
(parse-header 'if-range "\"original-etag\"")
⇒ ("original-etag" . #t)
```

`Date if-unmodified-since` [HTTP Header]
Indicates that a response should proceed if and only if the resource has not been modified since the given date.

```
(parse-header 'if-not-modified-since "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

`UInt max-forwards` [HTTP Header]
The maximum number of proxy or gateway hops that a request should be subject to.

```
(parse-header 'max-forwards "10")
⇒ 10
```

`Pair proxy-authorization` [HTTP Header]
Authorization credentials for a proxy connection. See the documentation for `authorization` above for more information on the format.

```
(parse-header 'proxy-authorization "Digest foo=bar,baz=qux"
⇒ (digest (foo . "bar") (baz . "qux"))
```

`Pair range` [HTTP Header]
A range request, indicating that the client wants only part of a resource. The car of the pair is the symbol `bytes`, and the cdr is a list of pairs. Each element of the cdr indicates a range; the car is the first byte position and the cdr is the last byte position, as integers, or `#f` if not given.

```
(parse-header 'range "bytes=10-30,50-")
⇒ (bytes (10 . 30) (50 . #f))
```

`URI referer` [HTTP Header]
The URI of the resource that referred the user to this resource. The name of the header is a misspelling, but we are stuck with it.

```
(parse-header 'referer "http://www.gnu.org/")
⇒ #<uri ...>
```

`List te` [HTTP Header]

A list of transfer codings, expressed as key-value lists. A common transfer coding is `trailers`.

```
(parse-header 'te "trailers")
⇒ ((trailers))
```

`String user-agent` [HTTP Header]

A string indicating the user agent making the request. The specification defines a structured format for this header, but it is widely disregarded, so Guile does not attempt to parse strictly.

```
(parse-header 'user-agent "Mozilla/5.0")
⇒ "Mozilla/5.0"
```

7.3.4.5 Response Headers

`List accept-ranges` [HTTP Header]

A list of range units that the server supports, as symbols.

```
(parse-header 'accept-ranges "bytes")
⇒ (bytes)
```

`UInt age` [HTTP Header]

The age of a cached response, in seconds.

```
(parse-header 'age "3600")
⇒ 3600
```

`ETag etag` [HTTP Header]

The entity-tag of the resource.

```
(parse-header 'etag "\"foo\"")
⇒ ("foo" . #t)
```

`URI-reference location` [HTTP Header]

A URI reference on which a request may be completed. Used in combination with a redirecting status code to perform client-side redirection.

```
(parse-header 'location "http://example.com/other")
⇒ #<uri ...>
```

`List proxy-authenticate` [HTTP Header]

A list of challenges to a proxy, indicating the need for authentication.

```
(parse-header 'proxy-authenticate "Basic realm=\"foo\"")
⇒ ((basic (realm . "foo")))
```

`UInt|Date retry-after` [HTTP Header]

Used in combination with a server-busy status code, like 503, to indicate that a client should retry later. Either a number of seconds, or a date.

```
(parse-header 'retry-after "60")
⇒ 60
```

`String server` [HTTP Header]

A string identifying the server.

```
(parse-header 'server "My first web server")
⇒ "My first web server"
```

`*|List vary` [HTTP Header]

A set of request headers that were used in computing this response. Used to indicate that server-side content negotiation was performed, for example in response to the `accept-language` header. Can also be the symbol `*`, indicating that all headers were considered.

```
(parse-header 'vary "Accept-Language, Accept")
⇒ (accept-language accept)
```

`List www-authenticate` [HTTP Header]

A list of challenges to a user, indicating the need for authentication.

```
(parse-header 'www-authenticate "Basic realm=\"foo\"")
⇒ ((basic (realm . "foo")))
```

7.3.5 Transfer Codings

HTTP 1.1 allows for various transfer codings to be applied to message bodies. These include various types of compression, and HTTP chunked encoding. Currently, only chunked encoding is supported by guile.

Chunked coding is an optional coding that may be applied to message bodies, to allow messages whose length is not known beforehand to be returned. Such messages can be split into chunks, terminated by a final zero length chunk.

In order to make dealing with encodings more simple, guile provides procedures to create ports that "wrap" existing ports, applying transformations transparently under the hood.

These procedures are in the `(web http)` module.

```
(use-modules (web http))
```

`make-chunked-input-port` *port* [*#:keep-alive?=#f*] [Scheme Procedure]

Returns a new port, that transparently reads and decodes chunk-encoded data from *port*. If no more chunk-encoded data is available, it returns the end-of-file object. When the port is closed, *port* will also be closed, unless *keep-alive?* is true.

```
(use-modules (ice-9 rdelim))

(define s "5\r\nFirst\r\nA\r\n line\n Sec\r\n8\r\nond line\r\n0\r\n")
(define p (make-chunked-input-port (open-input-string s)))
(read-line s)
⇒ "First line"
(read-line s)
⇒ "Second line"
```

`make-chunked-output-port` *port* [*#:keep-alive?=#f*] [Scheme Procedure]

Returns a new port, which transparently encodes data as chunk-encoded before writing it to *port*. Whenever a write occurs on this port, it buffers it, until the port

is flushed, at which point it writes a chunk containing all the data written so far. When the port is closed, the data remaining is written to *port*, as is the terminating zero chunk. It also causes *port* to be closed, unless *keep-alive?* is true.

Note. Forcing a chunked output port when there is no data is buffered does not write a zero chunk, as this would cause the data to be interpreted incorrectly by the client.

```
(call-with-output-string
  (lambda (out)
    (define out* (make-chunked-output-port out #:keep-alive? #t))
    (display "first chunk" out*)
    (force-output out*)
    (force-output out*) ; note this does not write a zero chunk
    (display "second chunk" out*)
    (close-port out*)))
⇒ "b\r\nfirst chunk\r\nc\r\nsecond chunk\r\n0\r\n"
```

7.3.6 HTTP Requests

```
(use-modules (web request))
```

The request module contains a data type for HTTP requests.

7.3.6.1 An Important Note on Character Sets

HTTP requests consist of two parts: the request proper, consisting of a request line and a set of headers, and (optionally) a body. The body might have a binary content-type, and even in the textual case its length is specified in bytes, not characters.

Therefore, HTTP is a fundamentally binary protocol. However the request line and headers are specified to be in a subset of ASCII, so they can be treated as text, provided that the port's encoding is set to an ASCII-compatible one-byte-per-character encoding. ISO-8859-1 (latin-1) is just such an encoding, and happens to be very efficient for Guile.

So what Guile does when reading requests from the wire, or writing them out, is to set the port's encoding to latin-1, and treating the request headers as text.

The request body is another issue. For binary data, the data is probably in a bytevector, so we use the R6RS binary output procedures to write out the binary payload. Textual data usually has to be written out to some character encoding, usually UTF-8, and then the resulting bytevector is written out to the port.

In summary, Guile reads and writes HTTP over latin-1 sockets, without any loss of generality.

7.3.6.2 Request API

`request?` *obj* [Scheme Procedure]
`request-method` *request* [Scheme Procedure]
`request-uri` *request* [Scheme Procedure]
`request-version` *request* [Scheme Procedure]
`request-headers` *request* [Scheme Procedure]
`request-meta` *request* [Scheme Procedure]

`request-port` *request* [Scheme Procedure]

A predicate and field accessors for the request type. The fields are as follows:

`method` The HTTP method, for example, `GET`.

`uri` The URI as a URI record.

`version` The HTTP version pair, like `(1 . 1)`.

`headers` The request headers, as an alist of parsed values.

`meta` An arbitrary alist of other data, for example information returned in the `sockaddr` from `accept` (см. Раздел 7.2.11.4 [Network Sockets and Communication], страница 564).

`port` The port on which to read or write a request body, if any.

`read-request` *port* [*meta='()*] [Scheme Procedure]

Read an HTTP request from *port*, optionally attaching the given metadata, *meta*.

As a side effect, sets the encoding on *port* to ISO-8859-1 (latin-1), so that reading one character reads one byte. See the discussion of character sets above, for more information.

Note that the body is not part of the request. Once you have read a request, you may read the body separately, and likewise for writing requests.

`build-request` *uri* [*#:method='GET*] [*#:version='(1 . 1)*] [*#:headers='()*] [*#:port=#f*] [*#:meta='()*] [*#:validate-headers?=#t*] [Scheme Procedure]

Construct an HTTP request object. If *validate-headers?* is true, the headers are each run through their respective validators.

`write-request` *r port* [Scheme Procedure]

Write the given HTTP request to *port*.

Return a new request, whose `request-port` will continue writing on *port*, perhaps using some transfer encoding.

`read-request-body` *r* [Scheme Procedure]

Reads the request body from *r*, as a bytevector. Return `#f` if there was no request body.

`write-request-body` *r bv* [Scheme Procedure]

Write *bv*, a bytevector, to the port corresponding to the HTTP request *r*.

The various headers that are typically associated with HTTP requests may be accessed with these dedicated accessors. См. Раздел 7.3.4 [HTTP Headers], страница 579, for more information on the format of parsed headers.

`request-accept` *request* [*default='()*] [Scheme Procedure]
`request-accept-charset` *request* [*default='()*] [Scheme Procedure]
`request-accept-encoding` *request* [*default='()*] [Scheme Procedure]
`request-accept-language` *request* [*default='()*] [Scheme Procedure]
`request-allow` *request* [*default='()*] [Scheme Procedure]
`request-authorization` *request* [*default=#f*] [Scheme Procedure]

`request-cache-control` *request* [*default='()*] [Scheme Procedure]
`request-connection` *request* [*default='()*] [Scheme Procedure]
`request-content-encoding` *request* [*default='()*] [Scheme Procedure]
`request-content-language` *request* [*default='()*] [Scheme Procedure]
`request-content-length` *request* [*default=#f*] [Scheme Procedure]
`request-content-location` *request* [*default=#f*] [Scheme Procedure]
`request-content-md5` *request* [*default=#f*] [Scheme Procedure]
`request-content-range` *request* [*default=#f*] [Scheme Procedure]
`request-content-type` *request* [*default=#f*] [Scheme Procedure]
`request-date` *request* [*default=#f*] [Scheme Procedure]
`request-expect` *request* [*default='()*] [Scheme Procedure]
`request-expires` *request* [*default=#f*] [Scheme Procedure]
`request-from` *request* [*default=#f*] [Scheme Procedure]
`request-host` *request* [*default=#f*] [Scheme Procedure]
`request-if-match` *request* [*default=#f*] [Scheme Procedure]
`request-if-modified-since` *request* [*default=#f*] [Scheme Procedure]
`request-if-none-match` *request* [*default=#f*] [Scheme Procedure]
`request-if-range` *request* [*default=#f*] [Scheme Procedure]
`request-if-unmodified-since` *request* [*default=#f*] [Scheme Procedure]
`request-last-modified` *request* [*default=#f*] [Scheme Procedure]
`request-max-forwards` *request* [*default=#f*] [Scheme Procedure]
`request-pragma` *request* [*default='()*] [Scheme Procedure]
`request-proxy-authorization` *request* [*default=#f*] [Scheme Procedure]
`request-range` *request* [*default=#f*] [Scheme Procedure]
`request-referer` *request* [*default=#f*] [Scheme Procedure]
`request-te` *request* [*default=#f*] [Scheme Procedure]
`request-trailer` *request* [*default='()*] [Scheme Procedure]
`request-transfer-encoding` *request* [*default='()*] [Scheme Procedure]
`request-upgrade` *request* [*default='()*] [Scheme Procedure]
`request-user-agent` *request* [*default=#f*] [Scheme Procedure]
`request-via` *request* [*default='()*] [Scheme Procedure]
`request-warning` *request* [*default='()*] [Scheme Procedure]

Return the given request header, or *default* if none was present.

`request-absolute-uri` *r* [*default-host=#f*] [*default-port=#f*] [*default-scheme=#f*] [Scheme Procedure]

A helper routine to determine the absolute URI of a request, using the `host` header and the default scheme, host and port. If there is no default scheme and the URI is not itself absolute, an error is signalled.

7.3.7 HTTP Responses

```
(use-modules (web response))
```

As with requests (см. Раздел 7.3.6 [Requests], страница 588), Guile offers a data type for HTTP responses. Again, the body is represented separately from the request.

`response?` *obj* [Scheme Procedure]
`response-version` *response* [Scheme Procedure]

`response-code` *response* [Scheme Procedure]
`response-reason-phrase` *response* [Scheme Procedure]
`response-headers` *response* [Scheme Procedure]
`response-port` *response* [Scheme Procedure]

A predicate and field accessors for the response type. The fields are as follows:

`version` The HTTP version pair, like `(1 . 1)`.

`code` The HTTP response code, like `200`.

`reason-phrase`
The reason phrase, or the standard reason phrase for the response's code.

`headers` The response headers, as an alist of parsed values.

`port` The port on which to read or write a response body, if any.

`read-response` *port* [Scheme Procedure]

Read an HTTP response from *port*.

As a side effect, sets the encoding on *port* to ISO-8859-1 (latin-1), so that reading one character reads one byte. See the discussion of character sets in Раздел 7.3.7 [Responses], страница 590, for more information.

`build-response` [*#:version='(1 . 1)*] [*#:code=200*] [Scheme Procedure]
[*#:reason-phrase=#f*] [*#:headers='()*] [*#:port=#f*] [*#:validate-headers?=#t*]

Construct an HTTP response object. If *validate-headers?* is true, the headers are each run through their respective validators.

`adapt-response-version` *response version* [Scheme Procedure]

Adapt the given response to a different HTTP version. Return a new HTTP response.

The idea is that many applications might just build a response for the default HTTP version, and this method could handle a number of programmatic transformations to respond to older HTTP versions (0.9 and 1.0). But currently this function is a bit heavy-handed, just updating the version field.

`write-response` *r port* [Scheme Procedure]

Write the given HTTP response to *port*.

Return a new response, whose `response-port` will continue writing on *port*, perhaps using some transfer encoding.

`response-must-not-include-body?` *r* [Scheme Procedure]

Some responses, like those with status code 304, are specified as never having bodies. This predicate returns `#t` for those responses.

Note also, though, that responses to `HEAD` requests must also not have a body.

`response-body-port` *r* [*#:decode?=#t*] [*#:keep-alive?=#t*] [Scheme Procedure]

Return an input port from which the body of *r* can be read. The encoding of the returned port is set according to *r*'s `content-type` header, when it's textual, except if *decode?* is `#f`. Return `#f` when no body is available.

When *keep-alive?* is `#f`, closing the returned port also closes *r*'s response port.

`read-response-body` *r* [Scheme Procedure]
Read the response body from *r*, as a bytevector. Returns `#f` if there was no response body.

`write-response-body` *r bv* [Scheme Procedure]
Write *bv*, a bytevector, to the port corresponding to the HTTP response *r*.

As with requests, the various headers that are typically associated with HTTP responses may be accessed with these dedicated accessors. См. Раздел 7.3.4 [HTTP Headers], страница 579, for more information on the format of parsed headers.

`response-accept-ranges` *response* [*default*=*#f*] [Scheme Procedure]
`response-age` *response* [*default*=*'()*] [Scheme Procedure]
`response-allow` *response* [*default*=*'()*] [Scheme Procedure]
`response-cache-control` *response* [*default*=*'()*] [Scheme Procedure]
`response-connection` *response* [*default*=*'()*] [Scheme Procedure]
`response-content-encoding` *response* [*default*=*'()*] [Scheme Procedure]
`response-content-language` *response* [*default*=*'()*] [Scheme Procedure]
`response-content-length` *response* [*default*=*#f*] [Scheme Procedure]
`response-content-location` *response* [*default*=*#f*] [Scheme Procedure]
`response-content-md5` *response* [*default*=*#f*] [Scheme Procedure]
`response-content-range` *response* [*default*=*#f*] [Scheme Procedure]
`response-content-type` *response* [*default*=*#f*] [Scheme Procedure]
`response-date` *response* [*default*=*#f*] [Scheme Procedure]
`response-etag` *response* [*default*=*#f*] [Scheme Procedure]
`response-expires` *response* [*default*=*#f*] [Scheme Procedure]
`response-last-modified` *response* [*default*=*#f*] [Scheme Procedure]
`response-location` *response* [*default*=*#f*] [Scheme Procedure]
`response-pragma` *response* [*default*=*'()*] [Scheme Procedure]
`response-proxy-authenticate` *response* [*default*=*#f*] [Scheme Procedure]
`response-retry-after` *response* [*default*=*#f*] [Scheme Procedure]
`response-server` *response* [*default*=*#f*] [Scheme Procedure]
`response-trailer` *response* [*default*=*'()*] [Scheme Procedure]
`response-transfer-encoding` *response* [*default*=*'()*] [Scheme Procedure]
`response-upgrade` *response* [*default*=*'()*] [Scheme Procedure]
`response-vary` *response* [*default*=*'()*] [Scheme Procedure]
`response-via` *response* [*default*=*'()*] [Scheme Procedure]
`response-warning` *response* [*default*=*'()*] [Scheme Procedure]
`response-www-authenticate` *response* [*default*=*#f*] [Scheme Procedure]
Return the given response header, or *default* if none was present.

`text-content-type?` *type* [Scheme Procedure]
Return `#t` if *type*, a symbol as returned by `response-content-type`, represents a textual type such as `text/plain`.

7.3.8 Web Client

`(web client)` provides a simple, synchronous HTTP client, built on the lower-level HTTP, request, and response modules.

```
(use-modules (web client))
```

`open-socket-for-uri` *uri* [Scheme Procedure]

Return an open input/output port for a connection to URI. Guile dynamically loads GnuTLS for HTTPS support. См. Раздел “Guile Preparations” в *GnuTLS-Guile*, for more information.

`http-get` *uri arg...* [Scheme Procedure]
`http-head` *uri arg...* [Scheme Procedure]
`http-post` *uri arg...* [Scheme Procedure]
`http-put` *uri arg...* [Scheme Procedure]
`http-delete` *uri arg...* [Scheme Procedure]
`http-trace` *uri arg...* [Scheme Procedure]
`http-options` *uri arg...* [Scheme Procedure]

Connect to the server corresponding to *uri* and make a request over HTTP, using the appropriate method (`GET`, `HEAD`, etc.).

All of these procedures have the same prototype: a URI followed by an optional sequence of keyword arguments. These keyword arguments allow you to modify the requests in various ways, for example attaching a body to the request, or setting specific headers. The following table lists the keyword arguments and their default values.

```
#:body #f

#:port (open-socket-for-uri uri)]
#:version '(1 . 1)
#:keep-alive? #f
#:headers '()
#:decode-body? #t
#:streaming? #f
```

If you already have a port open, pass it as *port*. Otherwise, a connection will be opened to the server corresponding to *uri*. Any extra headers in the alist *headers* will be added to the request.

If *body* is not `#f`, a message body will also be sent with the HTTP request. If *body* is a string, it is encoded according to the content-type in *headers*, defaulting to UTF-8. Otherwise *body* should be a bytevector, or `#f` for no body. Although a message body may be sent with any request, usually only `POST` and `PUT` requests have bodies.

If *decode-body?* is true, as is the default, the body of the response will be decoded to string, if it is a textual content-type. Otherwise it will be returned as a bytevector.

However, if *streaming?* is true, instead of eagerly reading the response body from the server, this function only reads off the headers. The response body will be returned as a port on which the data may be read.

Unless *keep-alive?* is true, the port will be closed after the full response body has been read.

Returns two values: the response read from the server, and the response body as a string, bytevector, #f value, or as a port (if *streaming?* is true).

`http-get` is useful for making one-off requests to web sites. If you are writing a web spider or some other client that needs to handle a number of requests in parallel, it’s

better to build an event-driven URL fetcher, similar in structure to the web server (см. Раздел 7.3.9 [Web Server], страница 594).

Another option, good but not as performant, would be to use threads, possibly via par-map or futures.

`current-http-proxy` [Scheme Parameter]

Either `#f` or a non-empty string containing the URL of the HTTP proxy server to be used by the procedures in the `(web client)` module, including `open-socket-for-uri`. Its initial value is based on the `http_proxy` environment variable.

```
(current-http-proxy) ⇒ "http://localhost:8123/"
(parameterize ((current-http-proxy #f))
  (http-get "http://example.com/"))  ; temporarily bypass proxy
(current-http-proxy) ⇒ "http://localhost:8123/"
```

7.3.9 Web Server

`(web server)` is a generic web server interface, along with a main loop implementation for web servers controlled by Guile.

```
(use-modules (web server))
```

The lowest layer is the `<server-impl>` object, which defines a set of hooks to open a server, read a request from a client, write a response to a client, and close a server. These hooks – `open`, `read`, `write`, and `close`, respectively – are bound together in a `<server-impl>` object. Procedures in this module take a `<server-impl>` object, if needed.

A `<server-impl>` may also be looked up by name. If you pass the `http` symbol to `run-server`, Guile looks for a variable named `http` in the `(web server http)` module, which should be bound to a `<server-impl>` object. Such a binding is made by instantiation of the `define-server-impl` syntax. In this way the run-server loop can automatically load other backends if available.

The life cycle of a server goes as follows:

1. The `open` hook is called, to open the server. `open` takes zero or more arguments, depending on the backend, and returns an opaque server socket object, or signals an error.
2. The `read` hook is called, to read a request from a new client. The `read` hook takes one argument, the server socket. It should return three values: an opaque client socket, the request, and the request body. The request should be a `<request>` object, from `(web request)`. The body should be a string or a bytevector, or `#f` if there is no body.

   If the read failed, the `read` hook may return #f for the client socket, request, and body.
3. A user-provided handler procedure is called, with the request and body as its arguments. The handler should return two values: the response, as a `<response>` record from `(web response)`, and the response body as bytevector, or `#f` if not present.

   The respose and response body are run through `sanitize-response`, documented below. This allows the handler writer to take some convenient shortcuts: for example, instead of a `<response>`, the handler can simply return an alist of headers, in which case a default response object is constructed with those headers. Instead of a

bytevector for the body, the handler can return a string, which will be serialized into an appropriate encoding; or it can return a procedure, which will be called on a port to write out the data. See the `sanitize-response` documentation, for more.

4. The `write` hook is called with three arguments: the client socket, the response, and the body. The `write` hook returns no values.
5. At this point the request handling is complete. For a loop, we loop back and try to read a new request.
6. If the user interrupts the loop, the `close` hook is called on the server socket.

A user may define a server implementation with the following form:

`define-server-impl` *name open read write close* [Scheme Syntax]
Make a `<server-impl>` object with the hooks *open*, *read*, *write*, and *close*, and bind it to the symbol *name* in the current module.

`lookup-server-impl` *impl* [Scheme Procedure]
Look up a server implementation. If *impl* is a server implementation already, it is returned directly. If it is a symbol, the binding named *impl* in the `(web server` *`impl`*`)` module is looked up. Otherwise an error is signaled.

Currently a server implementation is a somewhat opaque type, useful only for passing to other procedures in this module, like `read-client`.

The `(web server)` module defines a number of routines that use `<server-impl>` objects to implement parts of a web server. Given that we don't expose the accessors for the various fields of a `<server-impl>`, indeed these routines are the only procedures with any access to the impl objects.

`open-server` *impl open-params* [Scheme Procedure]
Open a server for the given implementation. Return one value, the new server object. The implementation's `open` procedure is applied to *open-params*, which should be a list.

`read-client` *impl server* [Scheme Procedure]
Read a new client from *server*, by applying the implementation's `read` procedure to the server. If successful, return three values: an object corresponding to the client, a request object, and the request body. If any exception occurs, return `#f` for all three values.

`handle-request` *handler request body state* [Scheme Procedure]
Handle a given request, returning the response and body.

The response and response body are produced by calling the given *handler* with *request* and *body* as arguments.

The elements of *state* are also passed to *handler* as arguments, and may be returned as additional values. The new *state*, collected from the *handler*'s return values, is then returned as a list. The idea is that a server loop receives a handler from the user, along with whatever state values the user is interested in, allowing the user's handler to explicitly manage its state.

`sanitize-response` *request response body* [Scheme Procedure]
"Sanitize" the given response and body, making them appropriate for the given request.

As a convenience to web handler authors, *response* may be given as an alist of headers, in which case it is used to construct a default response. Ensures that the response version corresponds to the request version. If *body* is a string, encodes the string to a bytevector, in an encoding appropriate for *response*. Adds a `content-length` and `content-type` header, as necessary.

If *body* is a procedure, it is called with a port as an argument, and the output collected as a bytevector. In the future we might try to instead use a compressing, chunk-encoded port, and call this procedure later, in the write-client procedure. Authors are advised not to rely on the procedure being called at any particular time.

`write-client` *impl server client response body* [Scheme Procedure]
Write an HTTP response and body to *client*. If the server and client support persistent connections, it is the implementation's responsibility to keep track of the client thereafter, presumably by attaching it to the *server* argument somehow.

`close-server` *impl server* [Scheme Procedure]
Release resources allocated by a previous invocation of `open-server`.

Given the procedures above, it is a small matter to make a web server:

`serve-one-client` *handler impl server state* [Scheme Procedure]
Read one request from *server*, call *handler* on the request and body, and write the response to the client. Return the new state produced by the handler procedure.

`run-server` *handler* **[***impl='http***]** **[***open-params='()***]** *arg ...* [Scheme Procedure]
Run Guile's built-in web server.

handler should be a procedure that takes two or more arguments, the HTTP request and request body, and returns two or more values, the response and response body.

For examples, skip ahead to the next section, Раздел 7.3.10 [Web Examples], страница 597.

The response and body will be run through `sanitize-response` before sending back to the client.

Additional arguments to *handler* are taken from *arg ...*. These arguments comprise a *state*. Additional return values are accumulated into a new state, which will be used for subsequent requests. In this way a handler can explicitly manage its state.

The default web server implementation is `http`, which binds to a socket, listening for request on that port.

`http` **[***#:host=#f***]** **[***#:family=AF_INET***]** [HTTP Implementation]
[*#:addr=INADDR_LOOPBACK***]** **[***#:port 8080***]** **[***#:socket***]**
The default HTTP implementation. We document it as a function with keyword arguments, because that is precisely the way that it is – all of the *open-params* to `run-server` get passed to the implementation's open function.

```
;; The defaults: localhost:8080
```

```
(run-server handler)
;; Same thing
(run-server handler 'http '())
;; On a different port
(run-server handler 'http '(#:port 8081))
;; IPv6
(run-server handler 'http '(#:family AF_INET6 #:port 8081))
;; Custom socket
(run-server handler 'http `(#:socket ,(sudo-make-me-a-socket)))
```

7.3.10 Web Examples

Well, enough about the tedious internals. Let's make a web application!

7.3.10.1 Hello, World!

The first program we have to write, of course, is "Hello, World!". This means that we have to implement a web handler that does what we want.

Now we define a handler, a function of two arguments and two return values:

```
(define (handler request request-body)
  (values response response-body))
```

In this first example, we take advantage of a short-cut, returning an alist of headers instead of a proper response object. The response body is our payload:

```
(define (hello-world-handler request request-body)
  (values '((content-type . (text/plain)))
          "Hello World!"))
```

Now let's test it, by running a server with this handler. Load up the web server module if you haven't yet done so, and run a server with this handler:

```
(use-modules (web server))
(run-server hello-world-handler)
```

By default, the web server listens for requests on `localhost:8080`. Visit that address in your web browser to test. If you see the string, `Hello World!`, sweet!

7.3.10.2 Inspecting the Request

The Hello World program above is a general greeter, responding to all URIs. To make a more exclusive greeter, we need to inspect the request object, and conditionally produce different results. So let's load up the request, response, and URI modules, and do just that.

```
(use-modules (web server)) ; you probably did this already
(use-modules (web request)
             (web response)
             (web uri))

(define (request-path-components request)
  (split-and-decode-uri-path (uri-path (request-uri request))))

(define (hello-hacker-handler request body)
  (if (equal? (request-path-components request)
```

```
             '("hacker"))
        (values '((content-type . (text/plain)))
                "Hello hacker!")
        (not-found request)))

(run-server hello-hacker-handler)
```

Here we see that we have defined a helper to return the components of the URI path as a list of strings, and used that to check for a request to `/hacker/`. Then the success case is just as before – visit `http://localhost:8080/hacker/` in your browser to check.

You should always match against URI path components as decoded by `split-and-decode-uri-path`. The above example will work for `/hacker/`, `//hacker///`, and `/h%61ck%65r`.

But we forgot to define `not-found`! If you are pasting these examples into a REPL, accessing any other URI in your web browser will drop your Guile console into the debugger:

```
<unnamed port>:38:7: In procedure module-lookup:
<unnamed port>:38:7: Unbound variable: not-found

Entering a new prompt.  Type `,bt' for a backtrace or `,q' to continue.
scheme@(guile-user) [1]>
```

So let's define the function, right there in the debugger. As you probably know, we'll want to return a 404 response.

```
;; Paste this in your REPL
(define (not-found request)
  (values (build-response #:code 404)
          (string-append "Resource not found: "
                         (uri->string (request-uri request)))))

;; Now paste this to let the web server keep going:
,continue
```

Now if you access `http://localhost/foo/`, you get this error message. (Note that some popular web browsers won't show server-generated 404 messages, showing their own instead, unless the 404 message body is long enough.)

7.3.10.3 Higher-Level Interfaces

The web handler interface is a common baseline that all kinds of Guile web applications can use. You will usually want to build something on top of it, however, especially when producing HTML. Here is a simple example that builds up HTML output using SXML (см. Раздел 7.20 [SXML], страница 761).

First, load up the modules:

```
(use-modules (web server)
             (web request)
             (web response)
             (sxml simple))
```

Now we define a simple templating function that takes a list of HTML body elements, as SXML, and puts them in our super template:

```
(define (templatize title body)
  `(html (head (title ,title))
         (body ,@body)))
```

For example, the simplest Hello HTML can be produced like this:

```
(sxml->xml (templatize "Hello!" '((b "Hi!"))))
⊣
<html><head><title>Hello!</title></head><body><b>Hi!</b></body></html>
```

Much better to work with Scheme data types than to work with HTML as strings. Now we define a little response helper:

```
(define* (respond #:optional body #:key
                  (status 200)
                  (title "Hello hello!")
                  (doctype "<!DOCTYPE html>\n")
                  (content-type-params '((charset . "utf-8")))
                  (content-type 'text/html)
                  (extra-headers '())
                  (sxml (and body (templatize title body))))
  (values (build-response
           #:code status
           #:headers `((content-type
                        . (,content-type ,@content-type-params))
                       ,@extra-headers))
          (lambda (port)
            (if sxml
                (begin
                  (if doctype (display doctype port))
                  (sxml->xml sxml port))))))
```

Here we see the power of keyword arguments with default initializers. By the time the arguments are fully parsed, the `sxml` local variable will hold the templated SXML, ready for sending out to the client.

Also, instead of returning the body as a string, `respond` gives a procedure, which will be called by the web server to write out the response to the client.

Now, a simple example using this responder, which lays out the incoming headers in an HTML table.

```
(define (debug-page request body)
  (respond
   `((h1 "hello world!")
     (table
      (tr (th "header") (th "value"))
      ,@(map (lambda (pair)
               `(tr (td (tt ,(with-output-to-string
                               (lambda () (display (car pair))))))
                    (td (tt ,(with-output-to-string
                               (lambda ()
                                 (write (cdr pair)))))))))
```

```
                (request-headers request))))))

  (run-server debug-page)
```

Now if you visit any local address in your web browser, we actually see some HTML, finally.

7.3.10.4 Conclusion

Well, this is about as far as Guile's built-in web support goes, for now. There are many ways to make a web application, but hopefully by standardizing the most fundamental data types, users will be able to choose the approach that suits them best, while also being able to switch between implementations of the server. This is a relatively new part of Guile, so if you have feedback, let us know, and we can take it into account. Happy hacking on the web!

7.4 The (ice-9 getopt-long) Module

The `(ice-9 getopt-long)` facility is designed to help parse arguments that are passed to Guile programs on the command line, and is modelled after the C library's facility of the same name (см. Раздел "Getopt" в *The GNU C Library Reference Manual*). For a more low-level interface to command-line argument parsing, См. Раздел 7.5.25 [SRFI-37], страница 651.

The `(ice-9 getopt-long)` module exports two procedures: `getopt-long` and `option-ref`.

- `getopt-long` takes a list of strings — the command line arguments — an *option specification*, and some optional keyword parameters. It parses the command line arguments according to the option specification and keyword parameters, and returns a data structure that encapsulates the results of the parsing.
- `option-ref` then takes the parsed data structure and a specific option's name, and returns information about that option in particular.

To make these procedures available to your Guile script, include the expression `(use-modules (ice-9 getopt-long))` somewhere near the top, before the first usage of `getopt-long` or `option-ref`.

7.4.1 A Short getopt-long Example

This section illustrates how `getopt-long` is used by presenting and dissecting a simple example. The first thing that we need is an *option specification* that tells `getopt-long` how to parse the command line. This specification is an association list with the long option name as the key. Here is how such a specification might look:

```
(define option-spec
  '((version (single-char #\v) (value #f))
    (help    (single-char #\h) (value #f))))
```

This alist tells `getopt-long` that it should accept two long options, called *version* and *help*, and that these options can also be selected by the single-letter abbreviations *v* and *h*, respectively. The `(value #f)` clauses indicate that neither of the options accepts a value.

With this specification we can use `getopt-long` to parse a given command line:

```
(define options (getopt-long (command-line) option-spec))
```

After this call, `options` contains the parsed command line and is ready to be examined by `option-ref`. `option-ref` is called like this:

```
(option-ref options 'help #f)
```

It expects the parsed command line, a symbol indicating the option to examine, and a default value. The default value is returned if the option was not present in the command line, or if the option was present but without a value; otherwise the value from the command line is returned. Usually `option-ref` is called once for each possible option that a script supports.

The following example shows a main program which puts all this together to parse its command line and figure out what the user wanted.

```
(define (main args)
  (let* ((option-spec '((version (single-char #\v) (value #f))
                        (help    (single-char #\h) (value #f))))
         (options (getopt-long args option-spec))
         (help-wanted (option-ref options 'help #f))
         (version-wanted (option-ref options 'version #f)))
    (if (or version-wanted help-wanted)
        (begin
          (if version-wanted
              (display "getopt-long-example version 0.3\n"))
          (if help-wanted
              (display "\
getopt-long-example [options]
  -v, --version    Display version
  -h, --help       Display this help
")))
        (begin
          (display "Hello, World!") (newline)))))
```

7.4.2 How to Write an Option Specification

An option specification is an association list (см. Раздел 6.6.20 [Association Lists], страница 243) with one list element for each supported option. The key of each list element is a symbol that names the option, while the value is a list of option properties:

```
OPTION-SPEC ::=  '( (OPT-NAME1 (PROP-NAME PROP-VALUE) ...)
                    (OPT-NAME2 (PROP-NAME PROP-VALUE) ...)
                    (OPT-NAME3 (PROP-NAME PROP-VALUE) ...)
                    ...
                  )
```

Each *opt-name* specifies the long option name for that option. For example, a list element with *opt-name* `background` specifies an option that can be specified on the command line using the long option `--background`. Further information about the option — whether it takes a value, whether it is required to be present in the command line, and so on — is specified by the option properties.

In the example of the preceding section, we already saw that a long option name can have a equivalent *short option* character. The equivalent short option character can be set for an option by specifying a `single-char` property in that option's property list. For example, a list element like `'(output (single-char #\o) ...)` specifies an option with long name `--output` that can also be specified by the equivalent short name `-o`.

The `value` property specifies whether an option requires or accepts a value. If the `value` property is set to `#t`, the option requires a value: `getopt-long` will signal an error if the option name is present without a corresponding value. If set to `#f`, the option does not take a value; in this case, a non-option word that follows the option name in the command line will be treated as a non-option argument. If set to the symbol `optional`, the option accepts a value but does not require one: a non-option word that follows the option name in the command line will be interpreted as that option's value. If the option name for an option with `'(value optional)` is immediately followed in the command line by *another* option name, the value for the first option is implicitly `#t`.

The `required?` property indicates whether an option is required to be present in the command line. If the `required?` property is set to `#t`, `getopt-long` will signal an error if the option is not specified.

Finally, the `predicate` property can be used to constrain the possible values of an option. If used, the `predicate` property should be set to a procedure that takes one argument — the proposed option value as a string — and returns either `#t` or `#f` according as the proposed value is or is not acceptable. If the predicate procedure returns `#f`, `getopt-long` will signal an error.

By default, options do not have single-character equivalents, are not required, and do not take values. Where the list element for an option includes a `value` property but no `predicate` property, the option values are unconstrained.

7.4.3 Expected Command Line Format

In order for `getopt-long` to correctly parse a command line, that command line must conform to a standard set of rules for how command line options are specified. This section explains what those rules are.

`getopt-long` splits a given command line into several pieces. All elements of the argument list are classified to be either options or normal arguments. Options consist of two dashes and an option name (so-called *long* options), or of one dash followed by a single letter (*short* options).

Options can behave as switches, when they are given without a value, or they can be used to pass a value to the program. The value for an option may be specified using an equals sign, or else is simply the next word in the command line, so the following two invocations are equivalent:

```
$ ./foo.scm --output=bar.txt
$ ./foo.scm --output bar.txt
```

Short options can be used instead of their long equivalents and can be grouped together after a single dash. For example, the following commands are equivalent.

```
$ ./foo.scm --version --help
$ ./foo.scm -v --help
$ ./foo.scm -vh
```

If an option requires a value, it can only be grouped together with other short options if it is the last option in the group; the value is the next argument. So, for example, with the following option specification —

```
((apples    (single-char #\a))
 (blimps    (single-char #\b) (value #t))
 (catalexis (single-char #\c) (value #t)))
```

— the following command lines would all be acceptable:

```
$ ./foo.scm -a -b bang -c couth
$ ./foo.scm -ab bang -c couth
$ ./foo.scm -ac couth -b bang
```

But the next command line is an error, because `-b` is not the last option in its combination, and because a group of short options cannot include two options that both require values:

```
$ ./foo.scm -abc couth bang
```

If an option's value is optional, `getopt-long` decides whether the option has a value by looking at what follows it in the argument list. If the next element is a string, and it does not appear to be an option itself, then that string is the option's value.

If the option `--` appears in the argument list, argument parsing stops there and subsequent arguments are returned as ordinary arguments, even if they resemble options. So, with the command line

```
$ ./foo.scm --apples "Granny Smith" -- --blimp Goodyear
```

`getopt-long` will recognize the `--apples` option as having the value `"Granny Smith"`, but will not treat `--blimp` as an option. The strings `--blimp` and `Goodyear` will be returned as ordinary argument strings.

7.4.4 Reference Documentation for `getopt-long`

`getopt-long` *args grammar* **[***#:stop-at-first-non-option #t***]** [Scheme Procedure]

Parse the command line given in *args* (which must be a list of strings) according to the option specification *grammar*.

The *grammar* argument is expected to be a list of this form:

`((`*option* `(`*property value*`) ...) ...)`

where each *option* is a symbol denoting the long option, but without the two leading dashes (e.g. `version` if the option is called `--version`).

For each option, there may be list of arbitrarily many property/value pairs. The order of the pairs is not important, but every property may only appear once in the property list. The following table lists the possible properties:

`(single-char` *char*`)`
: Accept `-`*char* as a single-character equivalent to `--`*option*. This is how to specify traditional Unix-style flags.

`(required?` *bool*`)`
: If *bool* is true, the option is required. `getopt-long` will raise an error if it is not found in *args*.

`(value` *`bool`*`)`
: If *bool* is `#t`, the option accepts a value; if it is `#f`, it does not; and if it is the symbol `optional`, the option may appear in *args* with or without a value.

`(predicate` *`func`*`)`
: If the option accepts a value (i.e. you specified `(value #t)` for this option), then `getopt-long` will apply *func* to the value, and throw an exception if it returns `#f`. *func* should be a procedure which accepts a string and returns a boolean value; you may need to use quasiquotes to get it into *grammar*.

The `#:stop-at-first-non-option` keyword, if specified with any true value, tells `getopt-long` to stop when it gets to the first non-option in the command line. That is, at the first word which is neither an option itself, nor the value of an option. Everything in the command line from that word onwards will be returned as non-option arguments.

`getopt-long`'s *args* parameter is expected to be a list of strings like the one returned by `command-line`, with the first element being the name of the command. Therefore `getopt-long` ignores the first element in *args* and starts argument interpretation with the second element.

`getopt-long` signals an error if any of the following conditions hold.

- The option grammar has an invalid syntax.
- One of the options in the argument list was not specified by the grammar.
- A required option is omitted.
- An option which requires an argument did not get one.
- An option that doesn't accept an argument does get one (this can only happen using the long option `--opt=`*`value`* syntax).
- An option predicate fails.

`#:stop-at-first-non-option` is useful for command line invocations like `guild [--help | --version] [script [script-options]]` and `cvs [general-options] command [command-options]`, where there are options at two levels: some generic and understood by the outer command, and some that are specific to the particular script or command being invoked. To use `getopt-long` in such cases, you would call it twice: firstly with `#:stop-at-first-non-option #t`, so as to parse any generic options and identify the wanted script or sub-command; secondly, and after trimming off the initial generic command words, with a script- or sub-command-specific option grammar, so as to process those specific options.

7.4.5 Reference Documentation for `option-ref`

`option-ref` *options key default* [Scheme Procedure]
: Search *options* for a command line option named *key* and return its value, if found. If the option has no value, but was given, return `#t`. If the option was not given, return *default*. *options* must be the result of a call to `getopt-long`.

`option-ref` always succeeds, either by returning the requested option value from the command line, or the default value.

The special key `'()` can be used to get a list of all non-option arguments.

7.5 SRFI Support Modules

SRFI is an acronym for Scheme Request For Implementation. The SRFI documents define a lot of syntactic and procedure extensions to standard Scheme as defined in R5RS.

Guile has support for a number of SRFIs. This chapter gives an overview over the available SRFIs and some usage hints. For complete documentation, design rationales and further examples, we advise you to get the relevant SRFI documents from the SRFI home page `http://srfi.schemers.org/`.

7.5.1 About SRFI Usage

SRFI support in Guile is currently implemented partly in the core library, and partly as add-on modules. That means that some SRFIs are automatically available when the interpreter is started, whereas the other SRFIs require you to use the appropriate support module explicitly.

There are several reasons for this inconsistency. First, the feature checking syntactic form `cond-expand` (см. Раздел 7.5.2 [SRFI-0], страница 605) must be available immediately, because it must be there when the user wants to check for the Scheme implementation, that is, before she can know that it is safe to use `use-modules` to load SRFI support modules. The second reason is that some features defined in SRFIs had been implemented in Guile before the developers started to add SRFI implementations as modules (for example SRFI-13 (см. Раздел 7.5.11 [SRFI-13], страница 632)). In the future, it is possible that SRFIs in the core library might be factored out into separate modules, requiring explicit module loading when they are needed. So you should be prepared to have to use `use-modules` someday in the future to access SRFI-13 bindings. If you want, you can do that already. We have included the module `(srfi srfi-13)` in the distribution, which currently does nothing, but ensures that you can write future-safe code.

Generally, support for a specific SRFI is made available by using modules named `(srfi srfi-`*number*`)`, where *number* is the number of the SRFI needed. Another possibility is to use the command line option `--use-srfi`, which will load the necessary modules automatically (см. Раздел 4.2 [Invoking Guile], страница 37).

7.5.2 SRFI-0 - cond-expand

This SRFI lets a portable Scheme program test for the presence of certain features, and adapt itself by using different blocks of code, or fail if the necessary features are not available. There's no module to load, this is in the Guile core.

A program designed only for Guile will generally not need this mechanism, such a program can of course directly use the various documented parts of Guile.

`cond-expand` (*feature body*...) ... [syntax]

Expand to the *body* of the first clause whose *feature* specification is satisfied. It is an error if no *feature* is satisfied.

Features are symbols such as `srfi-1`, and a feature specification can use `and`, `or` and `not` forms to test combinations. The last clause can be an `else`, to be used if no other passes.

For example, define a private version of `alist-cons` if SRFI-1 is not available.

```
(cond-expand (srfi-1
              )
             (else
              (define (alist-cons key val alist)
                (cons (cons key val) alist))))
```

Or demand a certain set of SRFIs (list operations, string ports, `receive` and string operations), failing if they're not available.

```
(cond-expand ((and srfi-1 srfi-6 srfi-8 srfi-13)
              ))
```

The Guile core has the following features,

```
guile
guile-2  ;; starting from Guile 2.x
guile-2.2  ;; starting from Guile 2.2
r5rs
srfi-0
srfi-4
srfi-6
srfi-13
srfi-14
srfi-16
srfi-23
srfi-30
srfi-39
srfi-46
srfi-55
srfi-61
srfi-62
srfi-87
srfi-105
```

Other SRFI feature symbols are defined once their code has been loaded with `use-modules`, since only then are their bindings available.

The '`--use-srfi`' command line option (см. Раздел 4.2 [Invoking Guile], страница 37) is a good way to load SRFIs to satisfy `cond-expand` when running a portable program.

Testing the `guile` feature allows a program to adapt itself to the Guile module system, but still run on other Scheme systems. For example the following demands SRFI-8 (`receive`), but also knows how to load it with the Guile mechanism.

```
(cond-expand (srfi-8
              )
             (guile
              (use-modules (srfi srfi-8))))
```

Likewise, testing the `guile-2` feature allows code to be portable between Guile 2.*x* and previous versions of Guile. For instance, it makes it possible to write code that accounts for Guile 2.*x*'s compiler, yet be correctly interpreted on 1.8 and earlier versions:

```
(cond-expand (guile-2 (eval-when (compile)
                        ;; This must be evaluated at compile time.
                        (fluid-set! current-reader my-reader)))
             (guile
                      ;; Earlier versions of Guile do not have a
                      ;; separate compilation phase.
                      (fluid-set! current-reader my-reader)))
```

It should be noted that `cond-expand` is separate from the `*features*` mechanism (см. Раздел 6.23.2 [Feature Tracking], страница 479), feature symbols in one are unrelated to those in the other.

7.5.3 SRFI-1 - List library

The list library defined in SRFI-1 contains a lot of useful list processing procedures for construction, examining, destructuring and manipulating lists and pairs.

Since SRFI-1 also defines some procedures which are already contained in R5RS and thus are supported by the Guile core library, some list and pair procedures which appear in the SRFI-1 document may not appear in this section. So when looking for a particular list/pair processing procedure, you should also have a look at the sections Раздел 6.6.9 [Lists], страница 192, and Раздел 6.6.8 [Pairs], страница 189.

7.5.3.1 Constructors

New lists can be constructed by calling one of the following procedures.

`xcons` *d a* [Scheme Procedure]

Like `cons`, but with interchanged arguments. Useful mostly when passed to higher-order procedures.

`list-tabulate` *n init-proc* [Scheme Procedure]

Return an *n*-element list, where each list element is produced by applying the procedure *init-proc* to the corresponding list index. The order in which *init-proc* is applied to the indices is not specified.

`list-copy` *lst* [Scheme Procedure]

Return a new list containing the elements of the list *lst*.

This function differs from the core `list-copy` (см. Раздел 6.6.9.3 [List Constructors], страница 193) in accepting improper lists too. And if *lst* is not a pair at all then it's treated as the final tail of an improper list and simply returned.

`circular-list` *elt1 elt2 ...* [Scheme Procedure]

Return a circular list containing the given arguments *elt1 elt2 ...*.

`iota` *count* [*start step*] [Scheme Procedure]

Return a list containing *count* numbers, starting from *start* and adding *step* each time. The default *start* is 0, the default *step* is 1. For example,

```
(iota 6)        ⇒ (0 1 2 3 4 5)
```

```
(iota 4 2.5 -2) ⇒ (2.5 0.5 -1.5 -3.5)
```

This function takes its name from the corresponding primitive in the APL language.

7.5.3.2 Predicates

The procedures in this section test specific properties of lists.

`proper-list?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a proper list, or `#f` otherwise. This is the same as the core `list?` (см. Раздел 6.6.9.2 [List Predicates], страница 193).

A proper list is a list which ends with the empty list `()` in the usual way. The empty list `()` itself is a proper list too.

```
(proper-list? '(1 2 3))  ⇒ #t
(proper-list? '())       ⇒ #t
```

`circular-list?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a circular list, or `#f` otherwise.

A circular list is a list where at some point the `cdr` refers back to a previous pair in the list (either the start or some later point), so that following the `cdr`s takes you around in a circle, with no end.

```
(define x (list 1 2 3 4))
(set-cdr! (last-pair x) (cddr x))
x ⇒ (1 2 3 4 3 4 3 4 ...)
(circular-list? x)  ⇒ #t
```

`dotted-list?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a dotted list, or `#f` otherwise.

A dotted list is a list where the `cdr` of the last pair is not the empty list `()`. Any non-pair *obj* is also considered a dotted list, with length zero.

```
(dotted-list? '(1 2 . 3))  ⇒ #t
(dotted-list? 99)          ⇒ #t
```

It will be noted that any Scheme object passes exactly one of the above three tests `proper-list?`, `circular-list?` and `dotted-list?`. Non-lists are `dotted-list?`, finite lists are either `proper-list?` or `dotted-list?`, and infinite lists are `circular-list?`.

`null-list?` *lst* [Scheme Procedure]

Return `#t` if *lst* is the empty list `()`, `#f` otherwise. If something else than a proper or circular list is passed as *lst*, an error is signalled. This procedure is recommended for checking for the end of a list in contexts where dotted lists are not allowed.

`not-pair?` *obj* [Scheme Procedure]

Return `#t` is *obj* is not a pair, `#f` otherwise. This is shorthand notation `(not (pair? obj))` and is supposed to be used for end-of-list checking in contexts where dotted lists are allowed.

`list=` *elt= list1 ...* [Scheme Procedure]

Return `#t` if all argument lists are equal, `#f` otherwise. List equality is determined by testing whether all lists have the same length and the corresponding elements are equal in the sense of the equality predicate *elt=*. If no or only one list is given, `#t` is returned.

7.5.3.3 Selectors

`first` *pair* [Scheme Procedure]
`second` *pair* [Scheme Procedure]
`third` *pair* [Scheme Procedure]
`fourth` *pair* [Scheme Procedure]
`fifth` *pair* [Scheme Procedure]
`sixth` *pair* [Scheme Procedure]
`seventh` *pair* [Scheme Procedure]
`eighth` *pair* [Scheme Procedure]
`ninth` *pair* [Scheme Procedure]
`tenth` *pair* [Scheme Procedure]

These are synonyms for `car`, `cadr`, `caddr`, ...

`car+cdr` *pair* [Scheme Procedure]

Return two values, the car and the cdr of *pair*.

`take` *lst i* [Scheme Procedure]
`take!` *lst i* [Scheme Procedure]

Return a list containing the first *i* elements of *lst*.

`take!` may modify the structure of the argument list *lst* in order to produce the result.

`drop` *lst i* [Scheme Procedure]

Return a list containing all but the first *i* elements of *lst*.

`take-right` *lst i* [Scheme Procedure]

Return a list containing the *i* last elements of *lst*. The return shares a common tail with *lst*.

`drop-right` *lst i* [Scheme Procedure]
`drop-right!` *lst i* [Scheme Procedure]

Return a list containing all but the *i* last elements of *lst*.

`drop-right` always returns a new list, even when *i* is zero. `drop-right!` may modify the structure of the argument list *lst* in order to produce the result.

`split-at` *lst i* [Scheme Procedure]
`split-at!` *lst i* [Scheme Procedure]

Return two values, a list containing the first *i* elements of the list *lst* and a list containing the remaining elements.

`split-at!` may modify the structure of the argument list *lst* in order to produce the result.

`last` *lst* [Scheme Procedure]
Return the last element of the non-empty, finite list *lst*.

7.5.3.4 Length, Append, Concatenate, etc.

`length+` *lst* [Scheme Procedure]
Return the length of the argument list *lst*. When *lst* is a circular list, `#f` is returned.

`concatenate` *list-of-lists* [Scheme Procedure]
`concatenate!` *list-of-lists* [Scheme Procedure]
Construct a list by appending all lists in *list-of-lists*.

`concatenate!` may modify the structure of the given lists in order to produce the result.

`concatenate` is the same as `(apply append` *`list-of-lists`*`)`. It exists because some Scheme implementations have a limit on the number of arguments a function takes, which the `apply` might exceed. In Guile there is no such limit.

`append-reverse` *rev-head tail* [Scheme Procedure]
`append-reverse!` *rev-head tail* [Scheme Procedure]
Reverse *rev-head*, append *tail* to it, and return the result. This is equivalent to `(append (reverse` *`rev-head`*`)` *`tail`*`)`, but its implementation is more efficient.

```
(append-reverse '(1 2 3) '(4 5 6)) ⇒ (3 2 1 4 5 6)
```

`append-reverse!` may modify *rev-head* in order to produce the result.

`zip` *lst1 lst2 ...* [Scheme Procedure]
Return a list as long as the shortest of the argument lists, where each element is a list. The first list contains the first elements of the argument lists, the second list contains the second elements, and so on.

`unzip1` *lst* [Scheme Procedure]
`unzip2` *lst* [Scheme Procedure]
`unzip3` *lst* [Scheme Procedure]
`unzip4` *lst* [Scheme Procedure]
`unzip5` *lst* [Scheme Procedure]
`unzip1` takes a list of lists, and returns a list containing the first elements of each list, `unzip2` returns two lists, the first containing the first elements of each lists and the second containing the second elements of each lists, and so on.

`count` *pred lst1 lst2 ...* [Scheme Procedure]
Return a count of the number of times *pred* returns true when called on elements from the given lists.

pred is called with *N* parameters `(`*`pred elem1 ... elemN`*`)`, each element being from the corresponding list. The first call is with the first element of each list, the second with the second element from each, and so on.

Counting stops when the end of the shortest list is reached. At least one list must be non-circular.

7.5.3.5 Fold, Unfold & Map

`fold` *proc init lst1 lst2 ...* [Scheme Procedure]
`fold-right` *proc init lst1 lst2 ...* [Scheme Procedure]

Apply *proc* to the elements of *lst1 lst2 ...* to build a result, and return that result.

Each *proc* call is (*proc elem1 elem2 ... previous*), where *elem1* is from *lst1*, *elem2* is from *lst2*, and so on. *previous* is the return from the previous call to *proc*, or the given *init* for the first call. If any list is empty, just *init* is returned.

`fold` works through the list elements from first to last. The following shows a list reversal and the calls it makes,

```
(fold cons '() '(1 2 3))

(cons 1 '())
(cons 2 '(1))
(cons 3 '(2 1)
⇒ (3 2 1)
```

`fold-right` works through the list elements from last to first, ie. from the right. So for example the following finds the longest string, and the last among equal longest,

```
(fold-right (lambda (str prev)
              (if (> (string-length str) (string-length prev))
                  str
                  prev))
            ""
            '("x" "abc" "xyz" "jk"))
⇒ "xyz"
```

If *lst1 lst2 ...* have different lengths, `fold` stops when the end of the shortest is reached; `fold-right` commences at the last element of the shortest. Ie. elements past the length of the shortest are ignored in the other *lst*s. At least one *lst* must be non-circular.

`fold` should be preferred over `fold-right` if the order of processing doesn't matter, or can be arranged either way, since `fold` is a little more efficient.

The way `fold` builds a result from iterating is quite general, it can do more than other iterations like say `map` or `filter`. The following for example removes adjacent duplicate elements from a list,

```
(define (delete-adjacent-duplicates lst)
  (fold-right (lambda (elem ret)
                (if (equal? elem (first ret))
                    ret
                    (cons elem ret)))
              (list (last lst))
              lst))
(delete-adjacent-duplicates '(1 2 3 3 4 4 4 5))
⇒ (1 2 3 4 5)
```

Clearly the same sort of thing can be done with a `for-each` and a variable in which to build the result, but a self-contained *proc* can be re-used in multiple contexts, where a `for-each` would have to be written out each time.

`pair-fold` *proc init lst1 lst2 ...* [Scheme Procedure]
`pair-fold-right` *proc init lst1 lst2 ...* [Scheme Procedure]
The same as `fold` and `fold-right`, but apply *proc* to the pairs of the lists instead of the list elements.

`reduce` *proc default lst* [Scheme Procedure]
`reduce-right` *proc default lst* [Scheme Procedure]
`reduce` is a variant of `fold`, where the first call to *proc* is on two elements from *lst*, rather than one element and a given initial value.

If *lst* is empty, `reduce` returns *default* (this is the only use for *default*). If *lst* has just one element then that's the return value. Otherwise *proc* is called on the elements of *lst*.

Each *proc* call is `(`*`proc elem previous`*`)`, where *elem* is from *lst* (the second and subsequent elements of *lst*), and *previous* is the return from the previous call to *proc*. The first element of *lst* is the *previous* for the first call to *proc*.

For example, the following adds a list of numbers, the calls made to `+` are shown. (Of course `+` accepts multiple arguments and can add a list directly, with `apply`.)

```
(reduce + 0 '(5 6 7)) ⇒ 18

(+ 6 5)  ⇒ 11
(+ 7 11) ⇒ 18
```

`reduce` can be used instead of `fold` where the *init* value is an "identity", meaning a value which under *proc* doesn't change the result, in this case 0 is an identity since `(+ 5 0)` is just 5. `reduce` avoids that unnecessary call.

`reduce-right` is a similar variation on `fold-right`, working from the end (ie. the right) of *lst*. The last element of *lst* is the *previous* for the first call to *proc*, and the *elem* values go from the second last.

`reduce` should be preferred over `reduce-right` if the order of processing doesn't matter, or can be arranged either way, since `reduce` is a little more efficient.

`unfold` *p f g seed* **[***tail-gen***]** [Scheme Procedure]
`unfold` is defined as follows:

```
(unfold p f g seed) =
   (if (p seed) (tail-gen seed)
       (cons (f seed)
             (unfold p f g (g seed))))
```

p Determines when to stop unfolding.

f Maps each seed value to the corresponding list element.

g Maps each seed value to next seed value.

seed The state value for the unfold.

tail-gen Creates the tail of the list; defaults to `(lambda (x) '())`.

g produces a series of seed values, which are mapped to list elements by *f*. These elements are put into a list in left-to-right order, and *p* tells when to stop unfolding.

`unfold-right` *p f g seed* [*tail*] [Scheme Procedure]

Construct a list with the following loop.

```
(let lp ((seed seed) (lis tail))
   (if (p seed) lis
       (lp (g seed)
           (cons (f seed) lis))))
```

p Determines when to stop unfolding.

f Maps each seed value to the corresponding list element.

g Maps each seed value to next seed value.

seed The state value for the unfold.

tail The tail of the list; defaults to `'()`.

`map` *f lst1 lst2 ...* [Scheme Procedure]

Map the procedure over the list(s) *lst1*, *lst2*, ... and return a list containing the results of the procedure applications. This procedure is extended with respect to R5RS, because the argument lists may have different lengths. The result list will have the same length as the shortest argument lists. The order in which *f* will be applied to the list element(s) is not specified.

`for-each` *f lst1 lst2 ...* [Scheme Procedure]

Apply the procedure *f* to each pair of corresponding elements of the list(s) *lst1*, *lst2*, ... The return value is not specified. This procedure is extended with respect to R5RS, because the argument lists may have different lengths. The shortest argument list determines the number of times *f* is called. *f* will be applied to the list elements in left-to-right order.

`append-map` *f lst1 lst2 ...* [Scheme Procedure]
`append-map!` *f lst1 lst2 ...* [Scheme Procedure]

Equivalent to

```
(apply append (map f clist1 clist2 ...))
```

and

```
(apply append! (map f clist1 clist2 ...))
```

Map *f* over the elements of the lists, just as in the `map` function. However, the results of the applications are appended together to make the final result. `append-map` uses `append` to append the results together; `append-map!` uses `append!`.

The dynamic order in which the various applications of *f* are made is not specified.

`map!` *f lst1 lst2 ...* [Scheme Procedure]

Linear-update variant of `map` – `map!` is allowed, but not required, to alter the cons cells of *lst1* to construct the result list.

The dynamic order in which the various applications of *f* are made is not specified. In the n-ary case, *lst2*, *lst3*, ... must have at least as many elements as *lst1*.

`pair-for-each` *f lst1 lst2 ...* [Scheme Procedure]
Like `for-each`, but applies the procedure *f* to the pairs from which the argument lists are constructed, instead of the list elements. The return value is not specified.

`filter-map` *f lst1 lst2 ...* [Scheme Procedure]
Like `map`, but only results from the applications of *f* which are true are saved in the result list.

7.5.3.6 Filtering and Partitioning

Filtering means to collect all elements from a list which satisfy a specific condition. Partitioning a list means to make two groups of list elements, one which contains the elements satisfying a condition, and the other for the elements which don't.

The `filter` and `filter!` functions are implemented in the Guile core, См. Раздел 6.6.9.6 [List Modification], страница 195.

`partition` *pred lst* [Scheme Procedure]
`partition!` *pred lst* [Scheme Procedure]
Split *lst* into those elements which do and don't satisfy the predicate *pred*.

The return is two values (см. Раздел 6.13.7 [Multiple Values], страница 330), the first being a list of all elements from *lst* which satisfy *pred*, the second a list of those which do not.

The elements in the result lists are in the same order as in *lst* but the order in which the calls (*pred* `elem`) are made on the list elements is unspecified.

`partition` does not change *lst*, but one of the returned lists may share a tail with it. `partition!` may modify *lst* to construct its return.

`remove` *pred lst* [Scheme Procedure]
`remove!` *pred lst* [Scheme Procedure]
Return a list containing all elements from *lst* which do not satisfy the predicate *pred*. The elements in the result list have the same order as in *lst*. The order in which *pred* is applied to the list elements is not specified.

`remove!` is allowed, but not required to modify the structure of the input list.

7.5.3.7 Searching

The procedures for searching elements in lists either accept a predicate or a comparison object for determining which elements are to be searched.

`find` *pred lst* [Scheme Procedure]
Return the first element of *lst* which satisfies the predicate *pred* and `#f` if no such element is found.

`find-tail` *pred lst* [Scheme Procedure]
Return the first pair of *lst* whose car satisfies the predicate *pred* and `#f` if no such element is found.

`take-while` *pred lst* [Scheme Procedure]
`take-while!` *pred lst* [Scheme Procedure]
Return the longest initial prefix of *lst* whose elements all satisfy the predicate *pred*.

`take-while!` is allowed, but not required to modify the input list while producing the result.

`drop-while` *pred lst* [Scheme Procedure]
Drop the longest initial prefix of *lst* whose elements all satisfy the predicate *pred*.

`span` *pred lst* [Scheme Procedure]
`span!` *pred lst* [Scheme Procedure]
`break` *pred lst* [Scheme Procedure]
`break!` *pred lst* [Scheme Procedure]
`span` splits the list *lst* into the longest initial prefix whose elements all satisfy the predicate *pred*, and the remaining tail. `break` inverts the sense of the predicate.

`span!` and `break!` are allowed, but not required to modify the structure of the input list *lst* in order to produce the result.

Note that the name `break` conflicts with the `break` binding established by `while` (см. Раздел 6.13.4 [while do], страница 321). Applications wanting to use `break` from within a `while` loop will need to make a new define under a different name.

`any` *pred lst1 lst2 ...* [Scheme Procedure]
Test whether any set of elements from *lst1 lst2 ...* satisfies *pred*. If so, the return value is the return value from the successful *pred* call, or if not, the return value is `#f`.

If there are n list arguments, then *pred* must be a predicate taking n arguments. Each *pred* call is `(`*`pred elem1 elem2 ...`*` )` taking an element from each *lst*. The calls are made successively for the first, second, etc. elements of the lists, stopping when *pred* returns non-`#f`, or when the end of the shortest list is reached.

The *pred* call on the last set of elements (i.e., when the end of the shortest list has been reached), if that point is reached, is a tail call.

`every` *pred lst1 lst2 ...* [Scheme Procedure]
Test whether every set of elements from *lst1 lst2 ...* satisfies *pred*. If so, the return value is the return from the final *pred* call, or if not, the return value is `#f`.

If there are n list arguments, then *pred* must be a predicate taking n arguments. Each *pred* call is `(`*`pred elem1 elem2 ...`*`)` taking an element from each *lst*. The calls are made successively for the first, second, etc. elements of the lists, stopping if *pred* returns `#f`, or when the end of any of the lists is reached.

The *pred* call on the last set of elements (i.e., when the end of the shortest list has been reached) is a tail call.

If one of *lst1 lst2 ...*is empty then no calls to *pred* are made, and the return value is `#t`.

`list-index` *pred lst1 lst2 ...* [Scheme Procedure]
Return the index of the first set of elements, one from each of *lst1 lst2 ...*, which satisfies *pred*.

pred is called as `(`*`elem1 elem2 ...`*`)`. Searching stops when the end of the shortest *lst* is reached. The return index starts from 0 for the first set of elements. If no set of elements pass, then the return value is `#f`.

```
(list-index odd? '(2 4 6 9))      ⇒ 3
```

```
(list-index = '(1 2 3) '(3 1 2))  ⇒ #f
```

`member` *x lst* [*=*] [Scheme Procedure]

Return the first sublist of *lst* whose car is equal to *x*. If *x* does not appear in *lst*, return `#f`.

Equality is determined by `equal?`, or by the equality predicate = if given. = is called `(= x elem)`, ie. with the given *x* first, so for example to find the first element greater than 5,

```
(member 5 '(3 5 1 7 2 9) <) ⇒ (7 2 9)
```

This version of `member` extends the core `member` (см. Раздел 6.6.9.7 [List Searching], страница 197) by accepting an equality predicate.

7.5.3.8 Deleting

`delete` *x lst* [*=*] [Scheme Procedure]
`delete!` *x lst* [*=*] [Scheme Procedure]

Return a list containing the elements of *lst* but with those equal to *x* deleted. The returned elements will be in the same order as they were in *lst*.

Equality is determined by the = predicate, or `equal?` if not given. An equality call is made just once for each element, but the order in which the calls are made on the elements is unspecified.

The equality calls are always `(= x elem)`, ie. the given *x* is first. This means for instance elements greater than 5 can be deleted with `(delete 5 lst <)`.

`delete` does not modify *lst*, but the return might share a common tail with *lst*. `delete!` may modify the structure of *lst* to construct its return.

These functions extend the core `delete` and `delete!` (см. Раздел 6.6.9.6 [List Modification], страница 195) in accepting an equality predicate. See also `lset-difference` (см. Раздел 7.5.3.10 [SRFI-1 Set Operations], страница 617) for deleting multiple elements from a list.

`delete-duplicates` *lst* [*=*] [Scheme Procedure]
`delete-duplicates!` *lst* [*=*] [Scheme Procedure]

Return a list containing the elements of *lst* but without duplicates.

When elements are equal, only the first in *lst* is retained. Equal elements can be anywhere in *lst*, they don't have to be adjacent. The returned list will have the retained elements in the same order as they were in *lst*.

Equality is determined by the = predicate, or `equal?` if not given. Calls `(= x y)` are made with element *x* being before *y* in *lst*. A call is made at most once for each combination, but the sequence of the calls across the elements is unspecified.

`delete-duplicates` does not modify *lst*, but the return might share a common tail with *lst*. `delete-duplicates!` may modify the structure of *lst* to construct its return.

In the worst case, this is an $O(N^2)$ algorithm because it must check each element against all those preceding it. For long lists it is more efficient to sort and then compare only adjacent elements.

7.5.3.9 Association Lists

Association lists are described in detail in section Раздел 6.6.20 [Association Lists], страница 243. The present section only documents the additional procedures for dealing with association lists defined by SRFI-1.

`assoc` *key alist* **[=]** [Scheme Procedure]
: Return the pair from *alist* which matches *key*. This extends the core `assoc` (см. Раздел 6.6.20.3 [Retrieving Alist Entries], страница 245) by taking an optional = comparison procedure.

  The default comparison is `equal?`. If an = parameter is given it's called `(= key alistcar)`, i.e. the given target *key* is the first argument, and a `car` from *alist* is second.

  For example a case-insensitive string lookup,

  ```
  (assoc "yy" '(("XX" . 1) ("YY" . 2)) string-ci=?)
  ⇒ ("YY" . 2)
  ```

`alist-cons` *key datum alist* [Scheme Procedure]
: Cons a new association *key* and *datum* onto *alist* and return the result. This is equivalent to

  ```
  (cons (cons key datum) alist)
  ```

  `acons` (см. Раздел 6.6.20.2 [Adding or Setting Alist Entries], страница 243) in the Guile core does the same thing.

`alist-copy` *alist* [Scheme Procedure]
: Return a newly allocated copy of *alist*, that means that the spine of the list as well as the pairs are copied.

`alist-delete` *key alist* **[=]** [Scheme Procedure]
`alist-delete!` *key alist* **[=]** [Scheme Procedure]
: Return a list containing the elements of *alist* but with those elements whose keys are equal to *key* deleted. The returned elements will be in the same order as they were in *alist*.

  Equality is determined by the = predicate, or `equal?` if not given. The order in which elements are tested is unspecified, but each equality call is made `(= key alistkey)`, i.e. the given *key* parameter is first and the key from *alist* second. This means for instance all associations with a key greater than 5 can be removed with `(alist-delete 5 alist <)`.

  `alist-delete` does not modify *alist*, but the return might share a common tail with *alist*. `alist-delete!` may modify the list structure of *alist* to construct its return.

7.5.3.10 Set Operations on Lists

Lists can be used to represent sets of objects. The procedures in this section operate on such lists as sets.

Note that lists are not an efficient way to implement large sets. The procedures here typically take time $m \times n$ when operating on m and n element lists. Other data structures

like trees, bitsets (см. Раздел 6.6.11 [Bit Vectors], страница 202) or hash tables (см. Раздел 6.6.22 [Hash Tables], страница 251) are faster.

All these procedures take an equality predicate as the first argument. This predicate is used for testing the objects in the list sets for sameness. This predicate must be consistent with `eq?` (см. Раздел 6.11.1 [Equality], страница 301) in the sense that if two list elements are `eq?` then they must also be equal under the predicate. This simply means a given object must be equal to itself.

`lset<=` = *list* ... [Scheme Procedure]

Return `#t` if each list is a subset of the one following it. I.e., *list1* is a subset of *list2*, *list2* is a subset of *list3*, etc., for as many lists as given. If only one list or no lists are given, the return value is `#t`.

A list *x* is a subset of *y* if each element of *x* is equal to some element in *y*. Elements are compared using the given = procedure, called as `(= xelem yelem)`.

```
(lset<= eq?)                        ⇒ #t
(lset<= eqv? '(1 2 3) '(1))         ⇒ #f
(lset<= eqv? '(1 3 2) '(4 3 1 2)) ⇒ #t
```

`lset=` = *list* ... [Scheme Procedure]

Return `#t` if all argument lists are set-equal. *list1* is compared to *list2*, *list2* to *list3*, etc., for as many lists as given. If only one list or no lists are given, the return value is `#t`.

Two lists *x* and *y* are set-equal if each element of *x* is equal to some element of *y* and conversely each element of *y* is equal to some element of *x*. The order of the elements in the lists doesn't matter. Element equality is determined with the given = procedure, called as `(= xelem yelem)`, but exactly which calls are made is unspecified.

```
(lset= eq?)                        ⇒ #t
(lset= eqv? '(1 2 3) '(3 2 1))     ⇒ #t
(lset= string-ci=? '("a" "A" "b") '("B" "b" "a")) ⇒ #t
```

`lset-adjoin` = *list elem* ... [Scheme Procedure]

Add to *list* any of the given *elem*s not already in the list. *elem*s are `cons`ed onto the start of *list* (so the return value shares a common tail with *list*), but the order that the *elem*s are added is unspecified.

The given = procedure is used for comparing elements, called as `(= listelem elem)`, i.e., the second argument is one of the given *elem* parameters.

```
(lset-adjoin eqv? '(1 2 3) 4 1 5) ⇒ (5 4 1 2 3)
```

`lset-union` = *list* ... [Scheme Procedure]
`lset-union!` = *list* ... [Scheme Procedure]

Return the union of the argument list sets. The result is built by taking the union of *list1* and *list2*, then the union of that with *list3*, etc., for as many lists as given. For one list argument that list itself is the result, for no list arguments the result is the empty list.

The union of two lists *x* and *y* is formed as follows. If *x* is empty then the result is *y*. Otherwise start with *x* as the result and consider each *y* element (from first to last). A *y* element not equal to something already in the result is `cons`ed onto the result.

The given = procedure is used for comparing elements, called as `(= relem yelem)`. The first argument is from the result accumulated so far, and the second is from the list being union-ed in. But exactly which calls are made is otherwise unspecified.

Notice that duplicate elements in *list1* (or the first non-empty list) are preserved, but that repeated elements in subsequent lists are only added once.

```
(lset-union eqv?)                          ⇒ ()
(lset-union eqv? '(1 2 3))                 ⇒ (1 2 3)
(lset-union eqv? '(1 2 1 3) '(2 4 5) '(5)) ⇒ (5 4 1 2 1 3)
```

`lset-union` doesn't change the given lists but the result may share a tail with the first non-empty list. `lset-union!` can modify all of the given lists to form the result.

`lset-intersection` = *list1 list2 ...* [Scheme Procedure]
`lset-intersection!` = *list1 list2 ...* [Scheme Procedure]

Return the intersection of *list1* with the other argument lists, meaning those elements of *list1* which are also in all of *list2* etc. For one list argument, just that list is returned.

The test for an element of *list1* to be in the return is simply that it's equal to some element in each of *list2* etc. Notice this means an element appearing twice in *list1* but only once in each of *list2* etc will go into the return twice. The return has its elements in the same order as they were in *list1*.

The given = procedure is used for comparing elements, called as `(= elem1 elemN)`. The first argument is from *list1* and the second is from one of the subsequent lists. But exactly which calls are made and in what order is unspecified.

```
(lset-intersection eqv? '(x y))                      ⇒ (x y)
(lset-intersection eqv? '(1 2 3) '(4 3 2))           ⇒ (2 3)
(lset-intersection eqv? '(1 1 2 2) '(1 2) '(2 1) '(2)) ⇒ (2 2)
```

The return from `lset-intersection` may share a tail with *list1*. `lset-intersection!` may modify *list1* to form its result.

`lset-difference` = *list1 list2 ...* [Scheme Procedure]
`lset-difference!` = *list1 list2 ...* [Scheme Procedure]

Return *list1* with any elements in *list2*, *list3* etc removed (ie. subtracted). For one list argument, just that list is returned.

The given = procedure is used for comparing elements, called as `(= elem1 elemN)`. The first argument is from *list1* and the second from one of the subsequent lists. But exactly which calls are made and in what order is unspecified.

```
(lset-difference eqv? '(x y))             ⇒ (x y)
(lset-difference eqv? '(1 2 3) '(3 1))    ⇒ (2)
(lset-difference eqv? '(1 2 3) '(3) '(2)) ⇒ (1)
```

The return from `lset-difference` may share a tail with *list1*. `lset-difference!` may modify *list1* to form its result.

`lset-diff+intersection` = *list1 list2 ...* [Scheme Procedure]
`lset-diff+intersection!` = *list1 list2 ...* [Scheme Procedure]

Return two values (см. Раздел 6.13.7 [Multiple Values], страница 330), the difference and intersection of the argument lists as per `lset-difference` and `lset-intersection` above.

For two list arguments this partitions *list1* into those elements of *list1* which are in *list2* and not in *list2*. (But for more than two arguments there can be elements of *list1* which are neither part of the difference nor the intersection.)

One of the return values from `lset-diff+intersection` may share a tail with *list1*. `lset-diff+intersection!` may modify *list1* to form its results.

`lset-xor` = *list ...* [Scheme Procedure]
`lset-xor!` = *list ...* [Scheme Procedure]

Return an XOR of the argument lists. For two lists this means those elements which are in exactly one of the lists. For more than two lists it means those elements which appear in an odd number of the lists.

To be precise, the XOR of two lists *x* and *y* is formed by taking those elements of *x* not equal to any element of *y*, plus those elements of *y* not equal to any element of *x*. Equality is determined with the given = procedure, called as `(= e1 e2)`. One argument is from *x* and the other from *y*, but which way around is unspecified. Exactly which calls are made is also unspecified, as is the order of the elements in the result.

```
(lset-xor eqv? '(x y))             ⇒ (x y)
(lset-xor eqv? '(1 2 3) '(4 3 2))  ⇒ (4 1)
```

The return from `lset-xor` may share a tail with one of the list arguments. `lset-xor!` may modify *list1* to form its result.

7.5.4 SRFI-2 - and-let*

The following syntax can be obtained with

```
(use-modules (srfi srfi-2))
```

or alternatively

```
(use-modules (ice-9 and-let-star))
```

`and-let*` (*clause ...*) *body ...* [library syntax]

A combination of `and` and `let*`.

Each *clause* is evaluated in turn, and if `#f` is obtained then evaluation stops and `#f` is returned. If all are non-`#f` then *body* is evaluated and the last form gives the return value, or if *body* is empty then the result is `#t`. Each *clause* should be one of the following,

`(symbol expr)`
: Evaluate *expr*, check for `#f`, and bind it to *symbol*. Like `let*`, that binding is available to subsequent clauses.

`(expr)`
: Evaluate *expr* and check for `#f`.

`symbol`
: Get the value bound to *symbol* and check for `#f`.

Notice that `(expr)` has an "extra" pair of parentheses, for instance `((eq? x y))`. One way to remember this is to imagine the `symbol` in `(symbol expr)` is omitted.

`and-let*` is good for calculations where a `#f` value means termination, but where a non-`#f` value is going to be needed in subsequent expressions.

The following illustrates this, it returns text between brackets '`[...]`' in a string, or `#f` if there are no such brackets (ie. either `string-index` gives `#f`).

```
(define (extract-brackets str)
  (and-let* ((start (string-index str #\[))
             (end   (string-index str #\] start)))
    (substring str (1+ start) end)))
```

The following shows plain variables and expressions tested too. `diagnostic-levels` is taken to be an alist associating a diagnostic type with a level. `str` is printed only if the type is known and its level is high enough.

```
(define (show-diagnostic type str)
  (and-let* (want-diagnostics
             (level (assq-ref diagnostic-levels type))
             ((>= level current-diagnostic-level)))
    (display str)))
```

The advantage of `and-let*` is that an extended sequence of expressions and tests doesn't require lots of nesting as would arise from separate `and` and `let*`, or from `cond` with `=>`.

7.5.5 SRFI-4 - Homogeneous numeric vector datatypes

SRFI-4 provides an interface to uniform numeric vectors: vectors whose elements are all of a single numeric type. Guile offers uniform numeric vectors for signed and unsigned 8-bit, 16-bit, 32-bit, and 64-bit integers, two sizes of floating point values, and, as an extension to SRFI-4, complex floating-point numbers of these two sizes.

The standard SRFI-4 procedures and data types may be included via loading the appropriate module:

```
(use-modules (srfi srfi-4))
```

This module is currently a part of the default Guile environment, but it is a good practice to explicitly import the module. In the future, using SRFI-4 procedures without importing the SRFI-4 module will cause a deprecation message to be printed. (Of course, one may call the C functions at any time. Would that C had modules!)

7.5.5.1 SRFI-4 - Overview

Uniform numeric vectors can be useful since they consume less memory than the non-uniform, general vectors. Also, since the types they can store correspond directly to C types, it is easier to work with them efficiently on a low level. Consider image processing as an example, where you want to apply a filter to some image. While you could store the pixels of an image in a general vector and write a general convolution function, things are much more efficient with uniform vectors: the convolution function knows that all pixels are unsigned 8-bit values (say), and can use a very tight inner loop.

This is implemented in Scheme by having the compiler notice calls to the SRFI-4 accessors, and inline them to appropriate compiled code. From C you have access to

the raw array; functions for efficiently working with uniform numeric vectors from C are listed at the end of this section.

Uniform numeric vectors are the special case of one dimensional uniform numeric arrays.

There are 12 standard kinds of uniform numeric vectors, and they all have their own complement of constructors, accessors, and so on. Procedures that operate on a specific kind of uniform numeric vector have a "tag" in their name, indicating the element type.

`u8` unsigned 8-bit integers

`s8` signed 8-bit integers

`u16` unsigned 16-bit integers

`s16` signed 16-bit integers

`u32` unsigned 32-bit integers

`s32` signed 32-bit integers

`u64` unsigned 64-bit integers

`s64` signed 64-bit integers

`f32` the C type `float`

`f64` the C type `double`

In addition, Guile supports uniform arrays of complex numbers, with the nonstandard tags:

`c32` complex numbers in rectangular form with the real and imaginary part being a `float`

`c64` complex numbers in rectangular form with the real and imaginary part being a `double`

The external representation (ie. read syntax) for these vectors is similar to normal Scheme vectors, but with an additional tag from the tables above indicating the vector's type. For example,

```
#u16(1 2 3)
#f64(3.1415 2.71)
```

Note that the read syntax for floating-point here conflicts with `#f` for false. In Standard Scheme one can write `(1 #f3)` for a three element list `(1 #f 3)`, but for Guile `(1 #f3)` is invalid. `(1 #f 3)` is almost certainly what one should write anyway to make the intention clear, so this is rarely a problem.

7.5.5.2 SRFI-4 - API

Note that the `c32` and `c64` functions are only available from `(srfi srfi-4 gnu)`.

`u8vector?` *obj* [Scheme Procedure]
`s8vector?` *obj* [Scheme Procedure]
`u16vector?` *obj* [Scheme Procedure]
`s16vector?` *obj* [Scheme Procedure]
`u32vector?` *obj* [Scheme Procedure]

`s32vector?` *obj* [Scheme Procedure]
`u64vector?` *obj* [Scheme Procedure]
`s64vector?` *obj* [Scheme Procedure]
`f32vector?` *obj* [Scheme Procedure]
`f64vector?` *obj* [Scheme Procedure]
`c32vector?` *obj* [Scheme Procedure]
`c64vector?` *obj* [Scheme Procedure]
`scm_u8vector_p` (*obj*) [C Function]
`scm_s8vector_p` (*obj*) [C Function]
`scm_u16vector_p` (*obj*) [C Function]
`scm_s16vector_p` (*obj*) [C Function]
`scm_u32vector_p` (*obj*) [C Function]
`scm_s32vector_p` (*obj*) [C Function]
`scm_u64vector_p` (*obj*) [C Function]
`scm_s64vector_p` (*obj*) [C Function]
`scm_f32vector_p` (*obj*) [C Function]
`scm_f64vector_p` (*obj*) [C Function]
`scm_c32vector_p` (*obj*) [C Function]
`scm_c64vector_p` (*obj*) [C Function]

Return `#t` if *obj* is a homogeneous numeric vector of the indicated type.

`make-u8vector` *n* [*value*] [Scheme Procedure]
`make-s8vector` *n* [*value*] [Scheme Procedure]
`make-u16vector` *n* [*value*] [Scheme Procedure]
`make-s16vector` *n* [*value*] [Scheme Procedure]
`make-u32vector` *n* [*value*] [Scheme Procedure]
`make-s32vector` *n* [*value*] [Scheme Procedure]
`make-u64vector` *n* [*value*] [Scheme Procedure]
`make-s64vector` *n* [*value*] [Scheme Procedure]
`make-f32vector` *n* [*value*] [Scheme Procedure]
`make-f64vector` *n* [*value*] [Scheme Procedure]
`make-c32vector` *n* [*value*] [Scheme Procedure]
`make-c64vector` *n* [*value*] [Scheme Procedure]
`scm_make_u8vector` (*n*, *value*) [C Function]
`scm_make_s8vector` (*n*, *value*) [C Function]
`scm_make_u16vector` (*n*, *value*) [C Function]
`scm_make_s16vector` (*n*, *value*) [C Function]
`scm_make_u32vector` (*n*, *value*) [C Function]
`scm_make_s32vector` (*n*, *value*) [C Function]
`scm_make_u64vector` (*n*, *value*) [C Function]
`scm_make_s64vector` (*n*, *value*) [C Function]
`scm_make_f32vector` (*n*, *value*) [C Function]
`scm_make_f64vector` (*n*, *value*) [C Function]
`scm_make_c32vector` (*n*, *value*) [C Function]

`scm_make_c64vector` (*n*, *value*) [C Function]

Return a newly allocated homogeneous numeric vector holding *n* elements of the indicated type. If *value* is given, the vector is initialized with that value, otherwise the contents are unspecified.

`u8vector` *value ...* [Scheme Procedure]
`s8vector` *value ...* [Scheme Procedure]
`u16vector` *value ...* [Scheme Procedure]
`s16vector` *value ...* [Scheme Procedure]
`u32vector` *value ...* [Scheme Procedure]
`s32vector` *value ...* [Scheme Procedure]
`u64vector` *value ...* [Scheme Procedure]
`s64vector` *value ...* [Scheme Procedure]
`f32vector` *value ...* [Scheme Procedure]
`f64vector` *value ...* [Scheme Procedure]
`c32vector` *value ...* [Scheme Procedure]
`c64vector` *value ...* [Scheme Procedure]
`scm_u8vector` (*values*) [C Function]
`scm_s8vector` (*values*) [C Function]
`scm_u16vector` (*values*) [C Function]
`scm_s16vector` (*values*) [C Function]
`scm_u32vector` (*values*) [C Function]
`scm_s32vector` (*values*) [C Function]
`scm_u64vector` (*values*) [C Function]
`scm_s64vector` (*values*) [C Function]
`scm_f32vector` (*values*) [C Function]
`scm_f64vector` (*values*) [C Function]
`scm_c32vector` (*values*) [C Function]
`scm_c64vector` (*values*) [C Function]

Return a newly allocated homogeneous numeric vector of the indicated type, holding the given parameter *value*s. The vector length is the number of parameters given.

`u8vector-length` *vec* [Scheme Procedure]
`s8vector-length` *vec* [Scheme Procedure]
`u16vector-length` *vec* [Scheme Procedure]
`s16vector-length` *vec* [Scheme Procedure]
`u32vector-length` *vec* [Scheme Procedure]
`s32vector-length` *vec* [Scheme Procedure]
`u64vector-length` *vec* [Scheme Procedure]
`s64vector-length` *vec* [Scheme Procedure]
`f32vector-length` *vec* [Scheme Procedure]
`f64vector-length` *vec* [Scheme Procedure]
`c32vector-length` *vec* [Scheme Procedure]
`c64vector-length` *vec* [Scheme Procedure]
`scm_u8vector_length` (*vec*) [C Function]
`scm_s8vector_length` (*vec*) [C Function]
`scm_u16vector_length` (*vec*) [C Function]
`scm_s16vector_length` (*vec*) [C Function]

`scm_u32vector_length` (*vec*) [C Function]
`scm_s32vector_length` (*vec*) [C Function]
`scm_u64vector_length` (*vec*) [C Function]
`scm_s64vector_length` (*vec*) [C Function]
`scm_f32vector_length` (*vec*) [C Function]
`scm_f64vector_length` (*vec*) [C Function]
`scm_c32vector_length` (*vec*) [C Function]
`scm_c64vector_length` (*vec*) [C Function]

Return the number of elements in *vec*.

`u8vector-ref` *vec i* [Scheme Procedure]
`s8vector-ref` *vec i* [Scheme Procedure]
`u16vector-ref` *vec i* [Scheme Procedure]
`s16vector-ref` *vec i* [Scheme Procedure]
`u32vector-ref` *vec i* [Scheme Procedure]
`s32vector-ref` *vec i* [Scheme Procedure]
`u64vector-ref` *vec i* [Scheme Procedure]
`s64vector-ref` *vec i* [Scheme Procedure]
`f32vector-ref` *vec i* [Scheme Procedure]
`f64vector-ref` *vec i* [Scheme Procedure]
`c32vector-ref` *vec i* [Scheme Procedure]
`c64vector-ref` *vec i* [Scheme Procedure]
`scm_u8vector_ref` (*vec*, *i*) [C Function]
`scm_s8vector_ref` (*vec*, *i*) [C Function]
`scm_u16vector_ref` (*vec*, *i*) [C Function]
`scm_s16vector_ref` (*vec*, *i*) [C Function]
`scm_u32vector_ref` (*vec*, *i*) [C Function]
`scm_s32vector_ref` (*vec*, *i*) [C Function]
`scm_u64vector_ref` (*vec*, *i*) [C Function]
`scm_s64vector_ref` (*vec*, *i*) [C Function]
`scm_f32vector_ref` (*vec*, *i*) [C Function]
`scm_f64vector_ref` (*vec*, *i*) [C Function]
`scm_c32vector_ref` (*vec*, *i*) [C Function]
`scm_c64vector_ref` (*vec*, *i*) [C Function]

Return the element at index *i* in *vec*. The first element in *vec* is index 0.

`u8vector-set!` *vec i value* [Scheme Procedure]
`s8vector-set!` *vec i value* [Scheme Procedure]
`u16vector-set!` *vec i value* [Scheme Procedure]
`s16vector-set!` *vec i value* [Scheme Procedure]
`u32vector-set!` *vec i value* [Scheme Procedure]
`s32vector-set!` *vec i value* [Scheme Procedure]
`u64vector-set!` *vec i value* [Scheme Procedure]
`s64vector-set!` *vec i value* [Scheme Procedure]
`f32vector-set!` *vec i value* [Scheme Procedure]
`f64vector-set!` *vec i value* [Scheme Procedure]
`c32vector-set!` *vec i value* [Scheme Procedure]
`c64vector-set!` *vec i value* [Scheme Procedure]

`scm_u8vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_s8vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_u16vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_s16vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_u32vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_s32vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_u64vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_s64vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_f32vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_f64vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_c32vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_c64vector_set_x` (*vec*, *i*, *value*) [C Function]

Set the element at index *i* in *vec* to *value*. The first element in *vec* is index 0. The return value is unspecified.

`u8vector->list` *vec* [Scheme Procedure]
`s8vector->list` *vec* [Scheme Procedure]
`u16vector->list` *vec* [Scheme Procedure]
`s16vector->list` *vec* [Scheme Procedure]
`u32vector->list` *vec* [Scheme Procedure]
`s32vector->list` *vec* [Scheme Procedure]
`u64vector->list` *vec* [Scheme Procedure]
`s64vector->list` *vec* [Scheme Procedure]
`f32vector->list` *vec* [Scheme Procedure]
`f64vector->list` *vec* [Scheme Procedure]
`c32vector->list` *vec* [Scheme Procedure]
`c64vector->list` *vec* [Scheme Procedure]
`scm_u8vector_to_list` (*vec*) [C Function]
`scm_s8vector_to_list` (*vec*) [C Function]
`scm_u16vector_to_list` (*vec*) [C Function]
`scm_s16vector_to_list` (*vec*) [C Function]
`scm_u32vector_to_list` (*vec*) [C Function]
`scm_s32vector_to_list` (*vec*) [C Function]
`scm_u64vector_to_list` (*vec*) [C Function]
`scm_s64vector_to_list` (*vec*) [C Function]
`scm_f32vector_to_list` (*vec*) [C Function]
`scm_f64vector_to_list` (*vec*) [C Function]
`scm_c32vector_to_list` (*vec*) [C Function]
`scm_c64vector_to_list` (*vec*) [C Function]

Return a newly allocated list holding all elements of *vec*.

`list->u8vector` *lst* [Scheme Procedure]
`list->s8vector` *lst* [Scheme Procedure]
`list->u16vector` *lst* [Scheme Procedure]
`list->s16vector` *lst* [Scheme Procedure]
`list->u32vector` *lst* [Scheme Procedure]
`list->s32vector` *lst* [Scheme Procedure]
`list->u64vector` *lst* [Scheme Procedure]

`list->s64vector` *lst* [Scheme Procedure]
`list->f32vector` *lst* [Scheme Procedure]
`list->f64vector` *lst* [Scheme Procedure]
`list->c32vector` *lst* [Scheme Procedure]
`list->c64vector` *lst* [Scheme Procedure]
`scm_list_to_u8vector` (*lst*) [C Function]
`scm_list_to_s8vector` (*lst*) [C Function]
`scm_list_to_u16vector` (*lst*) [C Function]
`scm_list_to_s16vector` (*lst*) [C Function]
`scm_list_to_u32vector` (*lst*) [C Function]
`scm_list_to_s32vector` (*lst*) [C Function]
`scm_list_to_u64vector` (*lst*) [C Function]
`scm_list_to_s64vector` (*lst*) [C Function]
`scm_list_to_f32vector` (*lst*) [C Function]
`scm_list_to_f64vector` (*lst*) [C Function]
`scm_list_to_c32vector` (*lst*) [C Function]
`scm_list_to_c64vector` (*lst*) [C Function]

Return a newly allocated homogeneous numeric vector of the indicated type, initialized with the elements of the list *lst*.

`SCM scm_take_u8vector` (*const scm_t_uint8 *data, size_t len*) [C Function]
`SCM scm_take_s8vector` (*const scm_t_int8 *data, size_t len*) [C Function]
`SCM scm_take_u16vector` (*const scm_t_uint16 *data, size_t len*) [C Function]
`SCM scm_take_s16vector` (*const scm_t_int16 *data, size_t len*) [C Function]
`SCM scm_take_u32vector` (*const scm_t_uint32 *data, size_t len*) [C Function]
`SCM scm_take_s32vector` (*const scm_t_int32 *data, size_t len*) [C Function]
`SCM scm_take_u64vector` (*const scm_t_uint64 *data, size_t len*) [C Function]
`SCM scm_take_s64vector` (*const scm_t_int64 *data, size_t len*) [C Function]
`SCM scm_take_f32vector` (*const float *data, size_t len*) [C Function]
`SCM scm_take_f64vector` (*const double *data, size_t len*) [C Function]
`SCM scm_take_c32vector` (*const float *data, size_t len*) [C Function]
`SCM scm_take_c64vector` (*const double *data, size_t len*) [C Function]

Return a new uniform numeric vector of the indicated type and length that uses the memory pointed to by *data* to store its elements. This memory will eventually be freed with `free`. The argument *len* specifies the number of elements in *data*, not its size in bytes.

The `c32` and `c64` variants take a pointer to a C array of `float`s or `double`s. The real parts of the complex numbers are at even indices in that array, the corresponding imaginary parts are at the following odd index.

`const scm_t_uint8 * scm_u8vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`const scm_t_int8 * scm_s8vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`const scm_t_uint16 * scm_u16vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`const scm_t_int16 * scm_s16vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`const scm_t_uint32 * scm_u32vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`const scm_t_int32 * scm_s32vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`const scm_t_uint64 * scm_u64vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`const scm_t_int64 * scm_s64vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`const float * scm_f32vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`const double * scm_f64vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`const float * scm_c32vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`const double * scm_c64vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

Like `scm_vector_elements` (см. Раздел 6.6.10.4 [Vector Accessing from C], страница 201), but returns a pointer to the elements of a uniform numeric vector of the indicated kind.

`scm_t_uint8 * scm_u8vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`scm_t_int8 * scm_s8vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`scm_t_uint16 * scm_u16vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`scm_t_int16 * scm_s16vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`scm_t_uint32 * scm_u32vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`scm_t_int32 * scm_s32vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`scm_t_uint64 * scm_u64vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`scm_t_int64 * scm_s64vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`float * scm_f32vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`double * scm_f64vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`float * scm_c32vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`double * scm_c64vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_vector_writable_elements` (см. Раздел 6.6.10.4 [Vector Accessing from C], страница 201), but returns a pointer to the elements of a uniform numeric vector of the indicated kind.

7.5.5.3 SRFI-4 - Relation to bytevectors

Guile implements SRFI-4 vectors using bytevectors (см. Раздел 6.6.12 [Bytevectors], страница 205). Often when you have a numeric vector, you end up wanting to write its bytes somewhere, or have access to the underlying bytes, or read in bytes from somewhere else. Bytevectors are very good at this sort of thing. But the SRFI-4 APIs are nicer to use when doing number-crunching, because they are addressed by element and not by byte.

So as a compromise, Guile allows all bytevector functions to operate on numeric vectors. They address the underlying bytes in the native endianness, as one would expect.

Following the same reasoning, that it's just bytes underneath, Guile also allows uniform vectors of a given type to be accessed as if they were of any type. One can fill a `u32vector`, and access its elements with `u8vector-ref`. One can use `f64vector-ref` on bytevectors. It's all the same to Guile.

In this way, uniform numeric vectors may be written to and read from input/output ports using the procedures that operate on bytevectors.

См. Раздел 6.6.12 [Bytevectors], страница 205, for more information.

7.5.5.4 SRFI-4 - Guile extensions

Guile defines some useful extensions to SRFI-4, which are not available in the default Guile environment. They may be imported by loading the extensions module:

```
(use-modules (srfi srfi-4 gnu))
```

`any->u8vector` *obj* [Scheme Procedure]
`any->s8vector` *obj* [Scheme Procedure]
`any->u16vector` *obj* [Scheme Procedure]
`any->s16vector` *obj* [Scheme Procedure]
`any->u32vector` *obj* [Scheme Procedure]
`any->s32vector` *obj* [Scheme Procedure]
`any->u64vector` *obj* [Scheme Procedure]
`any->s64vector` *obj* [Scheme Procedure]
`any->f32vector` *obj* [Scheme Procedure]
`any->f64vector` *obj* [Scheme Procedure]
`any->c32vector` *obj* [Scheme Procedure]
`any->c64vector` *obj* [Scheme Procedure]
`scm_any_to_u8vector` (*obj*) [C Function]
`scm_any_to_s8vector` (*obj*) [C Function]
`scm_any_to_u16vector` (*obj*) [C Function]
`scm_any_to_s16vector` (*obj*) [C Function]
`scm_any_to_u32vector` (*obj*) [C Function]
`scm_any_to_s32vector` (*obj*) [C Function]
`scm_any_to_u64vector` (*obj*) [C Function]

`scm_any_to_s64vector` (*obj*) [C Function]
`scm_any_to_f32vector` (*obj*) [C Function]
`scm_any_to_f64vector` (*obj*) [C Function]
`scm_any_to_c32vector` (*obj*) [C Function]
`scm_any_to_c64vector` (*obj*) [C Function]

Return a (maybe newly allocated) uniform numeric vector of the indicated type, initialized with the elements of *obj*, which must be a list, a vector, or a uniform vector. When *obj* is already a suitable uniform numeric vector, it is returned unchanged.

7.5.6 SRFI-6 - Basic String Ports

SRFI-6 defines the procedures `open-input-string`, `open-output-string` and `get-output-string`. These procedures are included in the Guile core, so using this module does not make any difference at the moment. But it is possible that support for SRFI-6 will be factored out of the core library in the future, so using this module does not hurt, after all.

7.5.7 SRFI-8 - receive

`receive` is a syntax for making the handling of multiple-value procedures easier. It is documented in См. Раздел 6.13.7 [Multiple Values], страница 330.

7.5.8 SRFI-9 - define-record-type

This SRFI is a syntax for defining new record types and creating predicate, constructor, and field getter and setter functions. It is documented in the "Data Types" section of the manual (см. Раздел 6.6.16 [SRFI-9 Records], страница 231).

7.5.9 SRFI-10 - Hash-Comma Reader Extension

This SRFI implements a reader extension `#,()` called hash-comma. It allows the reader to give new kinds of objects, for use both in data and as constants or literals in source code. This feature is available with

```
(use-modules (srfi srfi-10))
```

The new read syntax is of the form

```
#,(tag arg...)
```

where *tag* is a symbol and the *arg*s are objects taken as parameters. *tag*s are registered with the following procedure.

`define-reader-ctor` *tag proc* [Scheme Procedure]

Register *proc* as the constructor for a hash-comma read syntax starting with symbol *tag*, i.e. `#,(`*tag* `arg...)`. *proc* is called with the given arguments `(`*proc* `arg...)` and the object it returns is the result of the read.

For example, a syntax giving a list of *N* copies of an object.

```
(define-reader-ctor 'repeat
  (lambda (obj reps)
    (make-list reps obj)))

(display '#,(repeat 99 3))
```

```
⊣ (99 99 99)
```

Notice the quote ' when the `#,( )` is used. The `repeat` handler returns a list and the program must quote to use it literally, the same as any other list. Ie.

```
(display '#,(repeat 99 3))
⇒
(display '(99 99 99))
```

When a handler returns an object which is self-evaluating, like a number or a string, then there's no need for quoting, just as there's no need when giving those directly as literals. For example an addition,

```
(define-reader-ctor 'sum
  (lambda (x y)
    (+ x y)))
(display #,(sum 123 456)) ⊣ 579
```

Once `(srfi srfi-10)` has loaded, `#,()` is available globally, there's no need to use `(srfi srfi-10)` in later modules. Similarly the tags registered are global and can be used anywhere once registered.

We do not recommend `#,()` reader extensions, however, and for three reasons.

First of all, this SRFI is not modular: the tag is matched by name, not as an identifier within a scope. Defining a reader extension in one part of a program can thus affect unrelated parts of a program because the tag is not scoped.

Secondly, reader extensions can be hard to manage from a time perspective: when does the reader extension take effect? См. Раздел 6.10.8 [Eval When], страница 296, for more discussion.

Finally, reader extensions can easily produce objects that can't be reified to an object file by the compiler. For example if you define a reader extension that makes a hash table (см. Раздел 6.6.22 [Hash Tables], страница 251), then it will work fine when run with the interpreter, and you think you have a neat hack. But then if you try to compile your program, after wrangling with the `eval-when` concerns mentioned above, the compiler will carp that it doesn't know how to serialize a hash table to disk.

In the specific case of hash tables, it would be possible for Guile to know how to pack hash tables into compiled files, but this doesn't work in general. What if the object you produce is an instance of a record type? Guile would then have to serialize the record type to disk too, and then what happens if the program independently loads the code that defines the record type? Does it define the same type or a different type? Guile's record types are nominal, not structural, so the answer is not clear at all.

For all of these reasons we recommend macros over reader extensions. Macros fulfill many of the same needs while preserving modular composition, and their interaction with `eval-when` is well-known. If you need brevity, instead use `read-hash-extend` and make your reader extension expand to a macro invocation. In that way we preserve scoping as much as possible. См. Раздел 6.18.1.6 [Reader Extensions], страница 407.

7.5.10 SRFI-11 - let-values

This module implements the binding forms for multiple values `let-values` and `let*-values`. These forms are similar to `let` and `let*` (см. Раздел 6.12.2 [Local

Bindings], страница 314), but they support binding of the values returned by multiple-valued expressions.

Write `(use-modules (srfi srfi-11))` to make the bindings available.

```
(let-values (((x y) (values 1 2))
             ((z f) (values 3 4)))
   (+ x y z f))
⇒
10
```

`let-values` performs all bindings simultaneously, which means that no expression in the binding clauses may refer to variables bound in the same clause list. `let*-values`, on the other hand, performs the bindings sequentially, just like `let*` does for single-valued expressions.

7.5.11 SRFI-13 - String Library

The SRFI-13 procedures are always available, См. Раздел 6.6.5 [Strings], страница 151.

7.5.12 SRFI-14 - Character-set Library

The SRFI-14 data type and procedures are always available, См. Раздел 6.6.4 [Character Sets], страница 143.

7.5.13 SRFI-16 - case-lambda

SRFI-16 defines a variable-arity `lambda` form, `case-lambda`. This form is available in the default Guile environment. См. Раздел 6.9.5 [Case-lambda], страница 270, for more information.

7.5.14 SRFI-17 - Generalized set!

This SRFI implements a generalized `set!`, allowing some "referencing" functions to be used as the target location of a `set!`. This feature is available from

```
(use-modules (srfi srfi-17))
```

For example `vector-ref` is extended so that

```
(set! (vector-ref vec idx) new-value)
```

is equivalent to

```
(vector-set! vec idx new-value)
```

The idea is that a `vector-ref` expression identifies a location, which may be either fetched or stored. The same form is used for the location in both cases, encouraging visual clarity. This is similar to the idea of an "lvalue" in C.

The mechanism for this kind of `set!` is in the Guile core (см. Раздел 6.9.8 [Procedures with Setters], страница 274). This module adds definitions of the following functions as procedures with setters, allowing them to be targets of a `set!`,

`car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `caaaar`, `caaadr`, `caadar`, `caaddr`, `cadaar`, `cadadr`, `caddar`, `cadddr`, `cdaaar`, `cdaadr`, `cdadar`, `cdaddr`, `cddaar`, `cddadr`, `cdddar`, `cddddr`

`string-ref`, `vector-ref`

The SRFI specifies `setter` (см. Раздел 6.9.8 [Procedures with Setters], страница 274) as a procedure with setter, allowing the setter for a procedure to be changed, eg. `(set! (setter foo) my-new-setter-handler)`. Currently Guile does not implement this, a setter can only be specified on creation (`getter-with-setter` below).

`getter-with-setter` [Функция]
: The same as the Guile core `make-procedure-with-setter` (см. Раздел 6.9.8 [Procedures with Setters], страница 274).

7.5.15 SRFI-18 - Multithreading support

This is an implementation of the SRFI-18 threading and synchronization library. The functions and variables described here are provided by

```
(use-modules (srfi srfi-18))
```

SRFI-18 defines facilities for threads, mutexes, condition variables, time, and exception handling. Because these facilities are at a higher level than Guile's primitives, they are implemented as a layer on top of what Guile provides. In particular this means that a Guile mutex is not a SRFI-18 mutex, and a Guile thread is not a SRFI-18 thread, and so on. Guile provides a set of primitives and SRFI-18 is one of the systems built in terms of those primitives.

7.5.15.1 SRFI-18 Threads

Threads created by SRFI-18 differ in two ways from threads created by Guile's built-in thread functions. First, a thread created by SRFI-18 `make-thread` begins in a blocked state and will not start execution until `thread-start!` is called on it. Second, SRFI-18 threads are constructed with a top-level exception handler that captures any exceptions that are thrown on thread exit.

SRFI-18 threads are disjoint from Guile's primitive threads. См. Раздел 6.22.1 [Threads], страница 463, for more on Guile's primitive facility.

`current-thread` [Функция]
: Returns the thread that called this function. This is the same procedure as the same-named built-in procedure `current-thread` (см. Раздел 6.22.1 [Threads], страница 463).

`thread?` *obj* [Функция]
: Returns `#t` if *obj* is a thread, `#f` otherwise. This is the same procedure as the same-named built-in procedure `thread?` (см. Раздел 6.22.1 [Threads], страница 463).

`make-thread` *thunk* [*name*] [Функция]
: Call `thunk` in a new thread and with a new dynamic state, returning the new thread and optionally assigning it the object name *name*, which may be any Scheme object.

  Note that the name `make-thread` conflicts with the `(ice-9 threads)` function `make-thread`. Applications wanting to use both of these functions will need to refer to them by different names.

`thread-name` *thread* [Функция]
: Returns the name assigned to *thread* at the time of its creation, or `#f` if it was not given a name.

`thread-specific` *thread* [Функция]
`thread-specific-set!` *thread obj* [Функция]
Get or set the "object-specific" property of *thread*. In Guile's implementation of SRFI-18, this value is stored as an object property, and will be `#f` if not set.

`thread-start!` *thread* [Функция]
Unblocks *thread* and allows it to begin execution if it has not done so already.

`thread-yield!` [Функция]
If one or more threads are waiting to execute, calling `thread-yield!` forces an immediate context switch to one of them. Otherwise, `thread-yield!` has no effect. `thread-yield!` behaves identically to the Guile built-in function `yield`.

`thread-sleep!` *timeout* [Функция]
The current thread waits until the point specified by the time object *timeout* is reached (см. Раздел 7.5.15.4 [SRFI-18 Time], страница 636). This blocks the thread only if *timeout* represents a point in the future. it is an error for *timeout* to be `#f`.

`thread-terminate!` *thread* [Функция]
Causes an abnormal termination of *thread*. If *thread* is not already terminated, all mutexes owned by *thread* become unlocked/abandoned. If *thread* is the current thread, `thread-terminate!` does not return. Otherwise `thread-terminate!` returns an unspecified value; the termination of *thread* will occur before `thread-terminate!` returns. Subsequent attempts to join on *thread* will cause a "terminated thread exception" to be raised.

`thread-terminate!` is compatible with the thread cancellation procedures in the core threads API (см. Раздел 6.22.1 [Threads], страница 463) in that if a cleanup handler has been installed for the target thread, it will be called before the thread exits and its return value (or exception, if any) will be stored for later retrieval via a call to `thread-join!`.

`thread-join!` *thread* [*timeout* [*timeout-val*]] [Функция]
Wait for *thread* to terminate and return its exit value. When a time value *timeout* is given, it specifies a point in time where the waiting should be aborted. When the waiting is aborted, *timeout-val* is returned if it is specified; otherwise, a `join-timeout-exception` exception is raised (см. Раздел 7.5.15.5 [SRFI-18 Exceptions], страница 636). Exceptions may also be raised if the thread was terminated by a call to `thread-terminate!` (`terminated-thread-exception` will be raised) or if the thread exited by raising an exception that was handled by the top-level exception handler (`uncaught-exception` will be raised; the original exception can be retrieved using `uncaught-exception-reason`).

7.5.15.2 SRFI-18 Mutexes

SRFI-18 mutexes are disjoint from Guile's primitive mutexes. См. Раздел 6.22.5 [Mutexes and Condition Variables], страница 469, for more on Guile's primitive facility.

`make-mutex` [*name*] [Функция]
Returns a new mutex, optionally assigning it the object name *name*, which may be any Scheme object. The returned mutex will be created with the configuration described above.

`mutex-name` *mutex* [Функция]
Returns the name assigned to *mutex* at the time of its creation, or `#f` if it was not given a name.

`mutex-specific` *mutex* [Функция]
Return the "object-specific" property of *mutex*, or `#f` if none is set.

`mutex-specific-set!` *mutex obj* [Функция]
Set the "object-specific" property of *mutex*.

`mutex-state` *mutex* [Функция]
Returns information about the state of *mutex*. Possible values are:

- thread *t*: the mutex is in the locked/owned state and thread *t* is the owner of the mutex
- symbol `not-owned`: the mutex is in the locked/not-owned state
- symbol `abandoned`: the mutex is in the unlocked/abandoned state
- symbol `not-abandoned`: the mutex is in the unlocked/not-abandoned state

`mutex-lock!` *mutex* [*timeout* [*thread*]] [Функция]
Lock *mutex*, optionally specifying a time object *timeout* after which to abort the lock attempt and a thread *thread* giving a new owner for *mutex* different than the current thread.

`mutex-unlock!` *mutex* [*condition-variable* [*timeout*]] [Функция]
Unlock *mutex*, optionally specifying a condition variable *condition-variable* on which to wait, either indefinitely or, optionally, until the time object *timeout* has passed, to be signalled.

7.5.15.3 SRFI-18 Condition variables

SRFI-18 does not specify a "wait" function for condition variables. Waiting on a condition variable can be simulated using the SRFI-18 `mutex-unlock!` function described in the previous section.

SRFI-18 condition variables are disjoint from Guile's primitive condition variables. См. Раздел 6.22.5 [Mutexes and Condition Variables], страница 469, for more on Guile's primitive facility.

`condition-variable?` *obj* [Функция]
Returns `#t` if *obj* is a condition variable, `#f` otherwise.

`make-condition-variable` [*name*] [Функция]
Returns a new condition variable, optionally assigning it the object name *name*, which may be any Scheme object.

`condition-variable-name` *condition-variable* [Функция]
Returns the name assigned to *condition-variable* at the time of its creation, or `#f` if it was not given a name.

`condition-variable-specific` *condition-variable* [Функция]
Return the "object-specific" property of *condition-variable*, or `#f` if none is set.

`condition-variable-specific-set!` *condition-variable obj* [Функция]
Set the "object-specific" property of *condition-variable*.

`condition-variable-signal!` *condition-variable* [Функция]
`condition-variable-broadcast!` *condition-variable* [Функция]
Wake up one thread that is waiting for *condition-variable*, in the case of `condition-variable-signal!`, or all threads waiting for it, in the case of `condition-variable-broadcast!`.

7.5.15.4 SRFI-18 Time

The SRFI-18 time functions manipulate time in two formats: a "time object" type that represents an absolute point in time in some implementation-specific way; and the number of seconds since some unspecified "epoch". In Guile's implementation, the epoch is the Unix epoch, 00:00:00 UTC, January 1, 1970.

`current-time` [Функция]
Return the current time as a time object. This procedure replaces the procedure of the same name in the core library, which returns the current time in seconds since the epoch.

`time?` *obj* [Функция]
Returns `#t` if *obj* is a time object, `#f` otherwise.

`time->seconds` *time* [Функция]
`seconds->time` *seconds* [Функция]
Convert between time objects and numerical values representing the number of seconds since the epoch. When converting from a time object to seconds, the return value is the number of seconds between *time* and the epoch. When converting from seconds to a time object, the return value is a time object that represents a time *seconds* seconds after the epoch.

7.5.15.5 SRFI-18 Exceptions

SRFI-18 exceptions are identical to the exceptions provided by Guile's implementation of SRFI-34. The behavior of exception handlers invoked to handle exceptions thrown from SRFI-18 functions, however, differs from the conventional behavior of SRFI-34 in that the continuation of the handler is the same as that of the call to the function. Handlers are called in a tail-recursive manner; the exceptions do not "bubble up".

`current-exception-handler` [Функция]
Returns the current exception handler.

`with-exception-handler` *handler thunk* [Функция]
Installs *handler* as the current exception handler and calls the procedure *thunk* with no arguments, returning its value as the value of the exception. *handler* must be a procedure that accepts a single argument. The current exception handler at the time this procedure is called will be restored after the call returns.

`raise` *obj* [Функция]
Raise *obj* as an exception. This is the same procedure as the same-named procedure defined in SRFI 34.

`join-timeout-exception?` *obj* [Функция]
: Returns `#t` if *obj* is an exception raised as the result of performing a timed join on a thread that does not exit within the specified timeout, `#f` otherwise.

`abandoned-mutex-exception?` *obj* [Функция]
: Returns `#t` if *obj* is an exception raised as the result of attempting to lock a mutex that has been abandoned by its owner thread, `#f` otherwise.

`terminated-thread-exception?` *obj* [Функция]
: Returns `#t` if *obj* is an exception raised as the result of joining on a thread that exited as the result of a call to `thread-terminate!`.

`uncaught-exception?` *obj* [Функция]
`uncaught-exception-reason` *exc* [Функция]
: `uncaught-exception?` returns `#t` if *obj* is an exception thrown as the result of joining a thread that exited by raising an exception that was handled by the top-level exception handler installed by `make-thread`. When this occurs, the original exception is preserved as part of the exception thrown by `thread-join!` and can be accessed by calling `uncaught-exception-reason` on that exception. Note that because this exception-preservation mechanism is a side-effect of `make-thread`, joining on threads that exited as described above but were created by other means will not raise this `uncaught-exception` error.

7.5.16 SRFI-19 - Time/Date Library

This is an implementation of the SRFI-19 time/date library. The functions and variables described here are provided by

```
(use-modules (srfi srfi-19))
```

7.5.16.1 SRFI-19 Introduction

This module implements time and date representations and calculations, in various time systems, including universal time (UTC) and atomic time (TAI).

For those not familiar with these time systems, TAI is based on a fixed length second derived from oscillations of certain atoms. UTC differs from TAI by an integral number of seconds, which is increased or decreased at announced times to keep UTC aligned to a mean solar day (the orbit and rotation of the earth are not quite constant).

So far, only increases in the TAI ↔ UTC difference have been needed. Such an increase is a "leap second", an extra second of TAI introduced at the end of a UTC day. When working entirely within UTC this is never seen, every day simply has 86400 seconds. But when converting from TAI to a UTC date, an extra 23:59:60 is present, where normally a day would end at 23:59:59. Effectively the UTC second from 23:59:59 to 00:00:00 has taken two TAI seconds.

In the current implementation, the system clock is assumed to be UTC, and a table of leap seconds in the code converts to TAI. See comments in `srfi-19.scm` for how to update this table.

Also, for those not familiar with the terminology, a *Julian Day* is a real number which is a count of days and fraction of a day, in UTC, starting from -4713-01-01T12:00:00Z,

ie. midday Monday 1 Jan 4713 B.C. A *Modified Julian Day* is the same, but starting from 1858-11-17T00:00:00Z, ie. midnight 17 November 1858 UTC. That time is julian day 2400000.5.

7.5.16.2 SRFI-19 Time

A *time* object has type, seconds and nanoseconds fields representing a point in time starting from some epoch. This is an arbitrary point in time, not just a time of day. Although times are represented in nanoseconds, the actual resolution may be lower.

The following variables hold the possible time types. For instance `(current-time time-process)` would give the current CPU process time.

`time-utc` [Переменная]
Universal Coordinated Time (UTC).

`time-tai` [Переменная]
International Atomic Time (TAI).

`time-monotonic` [Переменная]
Monotonic time, meaning a monotonically increasing time starting from an unspecified epoch.

Note that in the current implementation `time-monotonic` is the same as `time-tai`, and unfortunately is therefore affected by adjustments to the system clock. Perhaps this will change in the future.

`time-duration` [Переменная]
A duration, meaning simply a difference between two times.

`time-process` [Переменная]
CPU time spent in the current process, starting from when the process began.

`time-thread` [Переменная]
CPU time spent in the current thread. Not currently implemented.

`time?` *obj* [Функция]
Return `#t` if *obj* is a time object, or `#f` if not.

`make-time` *type nanoseconds seconds* [Функция]
Create a time object with the given *type*, *seconds* and *nanoseconds*.

`time-type` *time* [Функция]
`time-nanosecond` *time* [Функция]
`time-second` *time* [Функция]
`set-time-type!` *time type* [Функция]
`set-time-nanosecond!` *time nsec* [Функция]
`set-time-second!` *time sec* [Функция]
Get or set the type, seconds or nanoseconds fields of a time object.

`set-time-type!` merely changes the field, it doesn't convert the time value. For conversions, see Раздел 7.5.16.4 [SRFI-19 Time/Date conversions], страница 641.

`copy-time` *time* [Функция]
: Return a new time object, which is a copy of the given *time*.

`current-time` [*type*] [Функция]
: Return the current time of the given *type*. The default *type* is `time-utc`.

  Note that the name `current-time` conflicts with the Guile core `current-time` function (см. Раздел 7.2.5 [Time], страница 537) as well as the SRFI-18 `current-time` function (см. Раздел 7.5.15.4 [SRFI-18 Time], страница 636). Applications wanting to use more than one of these functions will need to refer to them by different names.

`time-resolution` [*type*] [Функция]
: Return the resolution, in nanoseconds, of the given time *type*. The default *type* is `time-utc`.

`time<=?` *t1 t2* [Функция]
`time<?` *t1 t2* [Функция]
`time=?` *t1 t2* [Функция]
`time>=?` *t1 t2* [Функция]
`time>?` *t1 t2* [Функция]
: Return `#t` or `#f` according to the respective relation between time objects *t1* and *t2*. *t1* and *t2* must be the same time type.

`time-difference` *t1 t2* [Функция]
`time-difference!` *t1 t2* [Функция]
: Return a time object of type `time-duration` representing the period between *t1* and *t2*. *t1* and *t2* must be the same time type.

  `time-difference` returns a new time object, `time-difference!` may modify *t1* to form its return.

`add-duration` *time duration* [Функция]
`add-duration!` *time duration* [Функция]
`subtract-duration` *time duration* [Функция]
`subtract-duration!` *time duration* [Функция]
: Return a time object which is *time* with the given *duration* added or subtracted. *duration* must be a time object of type `time-duration`.

  `add-duration` and `subtract-duration` return a new time object. `add-duration!` and `subtract-duration!` may modify the given *time* to form their return.

7.5.16.3 SRFI-19 Date

A *date* object represents a date in the Gregorian calendar and a time of day on that date in some timezone.

The fields are year, month, day, hour, minute, second, nanoseconds and timezone. A date object is immutable, its fields can be read but they cannot be modified once the object is created.

Historically, the Gregorian calendar was only used from the latter part of the year 1582 onwards, and not until even later in many countries. Prior to that most countries used

the Julian calendar. SRFI-19 does not deal with the Julian calendar at all, and so does not reflect this historical calendar reform. Instead it projects the Gregorian calendar back proleptically as far as necessary. When dealing with historical data, especially prior to the British Empire's adoption of the Gregorian calendar in 1752, one should be mindful of which calendar is used in each context, and apply non-SRFI-19 facilities to convert where necessary.

`date?` *obj* [Функция]
Return `#t` if *obj* is a date object, or `#f` if not.

`make-date` *nsecs seconds minutes hours date month year zone-offset* [Функция]
Create a new date object.

`date-nanosecond` *date* [Функция]
Nanoseconds, 0 to 999999999.

`date-second` *date* [Функция]
Seconds, 0 to 59, or 60 for a leap second. 60 is never seen when working entirely within UTC, it's only when converting to or from TAI.

`date-minute` *date* [Функция]
Minutes, 0 to 59.

`date-hour` *date* [Функция]
Hour, 0 to 23.

`date-day` *date* [Функция]
Day of the month, 1 to 31 (or less, according to the month).

`date-month` *date* [Функция]
Month, 1 to 12.

`date-year` *date* [Функция]
Year, eg. 2003. Dates B.C. are negative, eg. −46 is 46 B.C. There is no year 0, year −1 is followed by year 1.

`date-zone-offset` *date* [Функция]
Time zone, an integer number of seconds east of Greenwich.

`date-year-day` *date* [Функция]
Day of the year, starting from 1 for 1st January.

`date-week-day` *date* [Функция]
Day of the week, starting from 0 for Sunday.

`date-week-number` *date dstartw* [Функция]
Week of the year, ignoring a first partial week. *dstartw* is the day of the week which is taken to start a week, 0 for Sunday, 1 for Monday, etc.

`current-date` [*tz-offset*] [Функция]
Return a date object representing the current date/time, in UTC offset by *tz-offset*. *tz-offset* is seconds east of Greenwich and defaults to the local timezone.

`current-julian-day` [Функция]
Return the current Julian Day.

`current-modified-julian-day` [Функция]
Return the current Modified Julian Day.

7.5.16.4 SRFI-19 Time/Date conversions

`date->julian-day` *date* [Функция]
`date->modified-julian-day` *date* [Функция]
`date->time-monotonic` *date* [Функция]
`date->time-tai` *date* [Функция]
`date->time-utc` *date* [Функция]

`julian-day->date` *jdn* [*tz-offset*] [Функция]
`julian-day->time-monotonic` *jdn* [Функция]
`julian-day->time-tai` *jdn* [Функция]
`julian-day->time-utc` *jdn* [Функция]

`modified-julian-day->date` *jdn* [*tz-offset*] [Функция]
`modified-julian-day->time-monotonic` *jdn* [Функция]
`modified-julian-day->time-tai` *jdn* [Функция]
`modified-julian-day->time-utc` *jdn* [Функция]

`time-monotonic->date` *time* [*tz-offset*] [Функция]
`time-monotonic->time-tai` *time* [Функция]
`time-monotonic->time-tai!` *time* [Функция]
`time-monotonic->time-utc` *time* [Функция]
`time-monotonic->time-utc!` *time* [Функция]

`time-tai->date` *time* [*tz-offset*] [Функция]
`time-tai->julian-day` *time* [Функция]
`time-tai->modified-julian-day` *time* [Функция]
`time-tai->time-monotonic` *time* [Функция]
`time-tai->time-monotonic!` *time* [Функция]
`time-tai->time-utc` *time* [Функция]
`time-tai->time-utc!` *time* [Функция]

`time-utc->date` *time* [*tz-offset*] [Функция]
`time-utc->julian-day` *time* [Функция]
`time-utc->modified-julian-day` *time* [Функция]
`time-utc->time-monotonic` *time* [Функция]
`time-utc->time-monotonic!` *time* [Функция]
`time-utc->time-tai` *time* [Функция]
`time-utc->time-tai!` *time* [Функция]

Convert between dates, times and days of the respective types. For instance `time-tai->time-utc` accepts a *time* object of type `time-tai` and returns an object of type `time-utc`.

The `!` variants may modify their *time* argument to form their return. The plain functions create a new object.

For conversions to dates, *tz-offset* is seconds east of Greenwich. The default is the local timezone, at the given time, as provided by the system, using `localtime` (см. Раздел 7.2.5 [Time], страница 537).

On 32-bit systems, `localtime` is limited to a 32-bit `time_t`, so a default *tz-offset* is only available for times between Dec 1901 and Jan 2038. For prior dates an application might like to use the value in 1902, though some locations have zone changes prior to that. For future dates an application might like to assume today's rules extend indefinitely. But for correct daylight savings transitions it will be necessary to take an offset for the same day and time but a year in range and which has the same starting weekday and same leap/non-leap (to support rules like last Sunday in October).

7.5.16.5 SRFI-19 Date to string

`date->string` *date* **[***format***]** [Функция]

Convert a date to a string under the control of a format. *format* should be a string containing '`~`' escapes, which will be expanded as per the following conversion table. The default *format* is '`~c`', a locale-dependent date and time.

Many of these conversion characters are the same as POSIX `strftime` (см. Раздел 7.2.5 [Time], страница 537), but there are some extras and some variations.

`~~`	literal `~`
`~a`	locale abbreviated weekday, eg. '`Sun`'
`~A`	locale full weekday, eg. '`Sunday`'
`~b`	locale abbreviated month, eg. '`Jan`'
`~B`	locale full month, eg. '`January`'
`~c`	locale date and time, eg. '`Fri Jul 14 20:28:42-0400 2000`'
`~d`	day of month, zero padded, '`01`' to '`31`'
`~e`	day of month, blank padded, '` 1`' to '`31`'
`~f`	seconds and fractional seconds, with locale decimal point, eg. '`5.2`'
`~h`	same as `~b`
`~H`	hour, 24-hour clock, zero padded, '`00`' to '`23`'
`~I`	hour, 12-hour clock, zero padded, '`01`' to '`12`'
`~j`	day of year, zero padded, '`001`' to '`366`'
`~k`	hour, 24-hour clock, blank padded, '` 0`' to '`23`'
`~l`	hour, 12-hour clock, blank padded, '` 1`' to '`12`'
`~m`	month, zero padded, '`01`' to '`12`'
`~M`	minute, zero padded, '`00`' to '`59`'
`~n`	newline
`~N`	nanosecond, zero padded, '`000000000`' to '`999999999`'
`~p`	locale AM or PM
`~r`	time, 12 hour clock, '`~I:~M:~S ~p`'
`~s`	number of full seconds since "the epoch" in UTC
`~S`	second, zero padded '`00`' to '`60`' (usual limit is 59, 60 is a leap second)
`~t`	horizontal tab character

~T	time, 24 hour clock, '~H:~M:~S'
~U	week of year, Sunday first day of week, '00' to '52'
~V	week of year, Monday first day of week, '01' to '53'
~w	day of week, 0 for Sunday, '0' to '6'
~W	week of year, Monday first day of week, '00' to '52'
~y	year, two digits, '00' to '99'
~Y	year, full, eg. '2003'
~z	time zone, RFC-822 style
~Z	time zone symbol (not currently implemented)
~1	ISO-8601 date, '~Y-~m-~d'
~2	ISO-8601 time+zone, '~H:~M:~S~z'
~3	ISO-8601 time, '~H:~M:~S'
~4	ISO-8601 date/time+zone, '~Y-~m-~dT~H:~M:~S~z'
~5	ISO-8601 date/time, '~Y-~m-~dT~H:~M:~S'

Conversions '~D', '~x' and '~X' are not currently described here, since the specification and reference implementation differ.

Conversion is locale-dependent on systems that support it (см. Раздел 6.25.5 [Accessing Locale Information], страница 489). См. Раздел 7.2.13 [Locales], страница 571, for information on how to change the current locale.

7.5.16.6 SRFI-19 String to date

`string->date` *input template* [Функция]

Convert an *input* string to a date under the control of a *template* string. Return a newly created date object.

Literal characters in *template* must match characters in *input* and '~' escapes must match the input forms described in the table below. "Skip to" means characters up to one of the given type are ignored, or "no skip" for no skipping. "Read" is what's then read, and "Set" is the field affected in the date object.

For example '~Y' skips input characters until a digit is reached, at which point it expects a year and stores that to the year field of the date.

	Skip to	Read	Set
~~	no skip	literal ~	nothing
~a	char-alphabetic?	locale abbreviated weekday name	nothing
~A	char-alphabetic?	locale full weekday name	nothing
~b	char-alphabetic?	locale abbreviated month name	date-month
~B	char-alphabetic?	locale full month name	date-month
~d	char-numeric?	day of month	date-day

~e	no skip	day of month, blank padded	date-day
~h	same as '~b'		
~H	char-numeric?	hour	date-hour
~k	no skip	hour, blank padded	date-hour
~m	char-numeric?	month	date-month
~M	char-numeric?	minute	date-minute
~S	char-numeric?	second	date-second
~y	no skip	2-digit year	date-year 50 years
~Y	char-numeric?	year	date-year
~z	no skip	time zone	date-zone-offs

Notice that the weekday matching forms don't affect the date object returned, instead the weekday will be derived from the day, month and year.

Conversion is locale-dependent on systems that support it (см. Раздел 6.25.5 [Accessing Locale Information], страница 489). См. Раздел 7.2.13 [Locales], страница 571, for information on how to change the current locale.

7.5.17 SRFI-23 - Error Reporting

The SRFI-23 `error` procedure is always available.

7.5.18 SRFI-26 - specializing parameters

This SRFI provides a syntax for conveniently specializing selected parameters of a function. It can be used with,

```
(use-modules (srfi srfi-26))
```

`cut` *slot1 slot2 ...* [library syntax]
`cute` *slot1 slot2 ...* [library syntax]

Return a new procedure which will make a call (*slot1 slot2 ...*) but with selected parameters specialized to given expressions.

An example will illustrate the idea. The following is a specialization of `write`, sending output to `my-output-port`,

```
(cut write <> my-output-port)
⇒
(lambda (obj) (write obj my-output-port))
```

The special symbol `<>` indicates a slot to be filled by an argument to the new procedure. `my-output-port` on the other hand is an expression to be evaluated and passed, ie. it specializes the behaviour of `write`.

`<>` A slot to be filled by an argument from the created procedure. Arguments are assigned to `<>` slots in the order they appear in the `cut` form, there's no way to re-arrange arguments.

The first argument to `cut` is usually a procedure (or expression giving a procedure), but `<>` is allowed there too. For example,

```
(cut <> 1 2 3)
⇒
(lambda (proc) (proc 1 2 3))
```

`<...>` A slot to be filled by all remaining arguments from the new procedure. This can only occur at the end of a `cut` form.

For example, a procedure taking a variable number of arguments like `max` but in addition enforcing a lower bound,

```
(define my-lower-bound 123)

(cut max my-lower-bound <...>)
⇒
(lambda arglist (apply max my-lower-bound arglist))
```

For `cut` the specializing expressions are evaluated each time the new procedure is called. For `cute` they're evaluated just once, when the new procedure is created. The name `cute` stands for "`cut` with evaluated arguments". In all cases the evaluations take place in an unspecified order.

The following illustrates the difference between `cut` and `cute`,

```
(cut format <> "the time is ~s" (current-time))
⇒
(lambda (port) (format port "the time is ~s" (current-time)))

(cute format <> "the time is ~s" (current-time))
⇒
(let ((val (current-time)))
  (lambda (port) (format port "the time is ~s" val))
```

(There's no provision for a mixture of `cut` and `cute` where some expressions would be evaluated every time but others evaluated only once.)

`cut` is really just a shorthand for the sort of `lambda` forms shown in the above examples. But notice `cut` avoids the need to name unspecialized parameters, and is more compact. Use in functional programming style or just with `map`, `for-each` or similar is typical.

```
(map (cut * 2 <>) '(1 2 3 4))

(for-each (cut write <> my-port) my-list)
```

7.5.19 SRFI-27 - Sources of Random Bits

This subsection is based on the specification of SRFI-27 (`http://srfi.schemers.org/srfi-27/srfi-27.html`) written by Sebastian Egner.

This SRFI provides access to a (pseudo) random number generator; for Guile's built-in random number facilities, which SRFI-27 is implemented upon, См. Раздел 6.6.2.14 [Random], страница 136. With SRFI-27, random numbers are obtained from a *random source*, which encapsulates a random number generation algorithm and its state.

7.5.19.1 The Default Random Source

`random-integer` *n* [Функция]
: Return a random number between zero (inclusive) and *n* (exclusive), using the default random source. The numbers returned have a uniform distribution.

`random-real` [Функция]
: Return a random number in (0,1), using the default random source. The numbers returned have a uniform distribution.

`default-random-source` [Функция]
: A random source from which `random-integer` and `random-real` have been derived using `random-source-make-integers` and `random-source-make-reals` (см. Раздел 7.5.19.3 [SRFI-27 Random Number Generators], страница 647, for those procedures). Note that an assignment to `default-random-source` does not change `random-integer` or `random-real`; it is also strongly recommended not to assign a new value.

7.5.19.2 Random Sources

`make-random-source` [Функция]
: Create a new random source. The stream of random numbers obtained from each random source created by this procedure will be identical, unless its state is changed by one of the procedures below.

`random-source?` *object* [Функция]
: Tests whether *object* is a random source. Random sources are a disjoint type.

`random-source-randomize!` *source* [Функция]
: Attempt to set the state of the random source to a truly random value. The current implementation uses a seed based on the current system time.

`random-source-pseudo-randomize!` *source i j* [Функция]
: Changes the state of the random source s into the initial state of the (*i*, *j*)-th independent random source, where *i* and *j* are non-negative integers. This procedure provides a mechanism to obtain a large number of independent random sources (usually all derived from the same backbone generator), indexed by two integers. In contrast to `random-source-randomize!`, this procedure is entirely deterministic.

The state associated with a random state can be obtained an reinstated with the following procedures:

`random-source-state-ref` *source* [Функция]
`random-source-state-set!` *source state* [Функция]
: Get and set the state of a random source. No assumptions should be made about the nature of the state object, besides it having an external representation (i.e. it can be passed to `write` and subsequently `read` back).

7.5.19.3 Obtaining random number generator procedures

`random-source-make-integers` *source* [Функция]

Obtains a procedure to generate random integers using the random source *source*. The returned procedure takes a single argument *n*, which must be a positive integer, and returns the next uniformly distributed random integer from the interval {0, ..., *n*-1} by advancing the state of *source*.

If an application obtains and uses several generators for the same random source *source*, a call to any of these generators advances the state of *source*. Hence, the generators do not produce the same sequence of random integers each but rather share a state. This also holds for all other types of generators derived from a fixed random sources.

While the SRFI text specifies that "Implementations that support concurrency make sure that the state of a generator is properly advanced", this is currently not the case in Guile's implementation of SRFI-27, as it would cause a severe performance penalty. So in multi-threaded programs, you either must perform locking on random sources shared between threads yourself, or use different random sources for multiple threads.

`random-source-make-reals` *source* [Функция]
`random-source-make-reals` *source unit* [Функция]

Obtains a procedure to generate random real numbers $0 < x < 1$ using the random source *source*. The procedure rand is called without arguments.

The optional parameter *unit* determines the type of numbers being produced by the returned procedure and the quantization of the output. *unit* must be a number such that $0 <$ *unit* < 1. The numbers created by the returned procedure are of the same numerical type as *unit* and the potential output values are spaced by at most *unit*. One can imagine rand to create numbers as x * *unit* where x is a random integer in {1, ..., floor(1/unit)-1}. Note, however, that this need not be the way the values are actually created and that the actual resolution of rand can be much higher than unit. In case *unit* is absent it defaults to a reasonably small value (related to the width of the mantissa of an efficient number format).

7.5.20 SRFI-28 - Basic Format Strings

SRFI-28 provides a basic `format` procedure that provides only the `~a`, `~s`, `~%`, and `~~` format specifiers. You can import this procedure by using:

```
(use-modules (srfi srfi-28))
```

`format` *message arg ...* [Scheme Procedure]

Returns a formatted message, using *message* as the format string, which can contain the following format specifiers:

`~a` Insert the textual representation of the next *arg*, as if printed by `display`.

`~s` Insert the textual representation of the next *arg*, as if printed by `write`.

`~%` Insert a newline.

`~~` Insert a tilde.

This procedure is the same as calling `simple-format` (см. Раздел 6.14.5 [Simple Output], страница 359) with `#f` as the destination.

7.5.21 SRFI-30 - Nested Multi-line Comments

Starting from version 2.0, Guile's `read` supports SRFI-30/R6RS nested multi-line comments by default, Раздел 6.18.1.3 [Block Comments], страница 406.

7.5.22 SRFI-31 - A special form 'rec' for recursive evaluation

SRFI-31 defines a special form that can be used to create self-referential expressions more conveniently. The syntax is as follows:

```
<rec expression> --> (rec <variable> <expression>)
<rec expression> --> (rec (<variable>+) <body>)
```

The first syntax can be used to create self-referential expressions, for example:

```
guile> (define tmp (rec ones (cons 1 (delay ones))))
```

The second syntax can be used to create anonymous recursive functions:

```
guile> (define tmp (rec (display-n item n)
                        (if (positive? n)
                            (begin (display n) (display-n (- n 1))))))
guile> (tmp 42 3)
424242
guile>
```

7.5.23 SRFI-34 - Exception handling for programs

Guile provides an implementation of SRFI-34's exception handling mechanisms (`http://srfi.schemers.org/srfi-34/srfi-34.html`) as an alternative to its own built-in mechanisms (см. Раздел 6.13.8 [Exceptions], страница 332). It can be made available as follows:

```
(use-modules (srfi srfi-34))
```

7.5.24 SRFI-35 - Conditions

SRFI-35 (`http://srfi.schemers.org/srfi-35/srfi-35.html`) implements *conditions*, a data structure akin to records designed to convey information about exceptional conditions between parts of a program. It is normally used in conjunction with SRFI-34's `raise`:

```
(raise (condition (&message
                   (message "An error occurred"))))
```

Users can define *condition types* containing arbitrary information. Condition types may inherit from one another. This allows the part of the program that handles (or "catches") conditions to get accurate information about the exceptional condition that arose.

SRFI-35 conditions are made available using:

```
(use-modules (srfi srfi-35))
```

The procedures available to manipulate condition types are the following:

`make-condition-type` *id parent field-names* [Scheme Procedure]
Return a new condition type named *id*, inheriting from *parent*, and with the fields whose names are listed in *field-names*. *field-names* must be a list of symbols and must not contain names already used by *parent* or one of its supertypes.

`condition-type?` *obj* [Scheme Procedure]
Return true if *obj* is a condition type.

Conditions can be created and accessed with the following procedures:

`make-condition` *type . field+value* [Scheme Procedure]
Return a new condition of type *type* with fields initialized as specified by *field+value*, a sequence of field names (symbols) and values as in the following example:

```
(let ((&ct (make-condition-type 'foo &condition '(a b c))))
  (make-condition &ct 'a 1 'b 2 'c 3))
```

Note that all fields of *type* and its supertypes must be specified.

`make-compound-condition` *condition1 condition2 ...* [Scheme Procedure]
Return a new compound condition composed of *condition1 condition2 ...*. The returned condition has the type of each condition of condition1 condition2 ... (per `condition-has-type?`).

`condition-has-type?` *c type* [Scheme Procedure]
Return true if condition *c* has type *type*.

`condition-ref` *c field-name* [Scheme Procedure]
Return the value of the field named *field-name* from condition *c*.

If *c* is a compound condition and several underlying condition types contain a field named *field-name*, then the value of the first such field is returned, using the order in which conditions were passed to `make-compound-condition`.

`extract-condition` *c type* [Scheme Procedure]
Return a condition of condition type *type* with the field values specified by *c*.

If *c* is a compound condition, extract the field values from the subcondition belonging to *type* that appeared first in the call to `make-compound-condition` that created the condition.

Convenience macros are also available to create condition types and conditions.

`define-condition-type` *type supertype predicate field-spec...* [library syntax]
Define a new condition type named *type* that inherits from *supertype*. In addition, bind *predicate* to a type predicate that returns true when passed a condition of type *type* or any of its subtypes. *field-spec* must have the form `(field accessor)` where *field* is the name of field of *type* and *accessor* is the name of a procedure to access field *field* in conditions of type *type*.

The example below defines condition type `&foo`, inheriting from `&condition` with fields `a`, `b` and `c`:

```
(define-condition-type &foo &condition
```

```
foo-condition?
(a  foo-a)
(b  foo-b)
(c  foo-c))
```

`condition` *type-field-binding1 type-field-binding2 ...* [library syntax]
Return a new condition or compound condition, initialized according to *type-field-binding1 type-field-binding2 ...*. Each *type-field-binding* must have the form `(type field-specs...)`, where *type* is the name of a variable bound to a condition type; each *field-spec* must have the form `(field-name value)` where *field-name* is a symbol denoting the field being initialized to *value*. As for `make-condition`, all fields must be specified.

The following example returns a simple condition:

```
(condition (&message (message "An error occurred")))
```

The one below returns a compound condition:

```
(condition (&message (message "An error occurred"))
           (&serious))
```

Finally, SRFI-35 defines a several standard condition types.

`&condition` [Переменная]
This condition type is the root of all condition types. It has no fields.

`&message` [Переменная]
A condition type that carries a message describing the nature of the condition to humans.

`message-condition?` *c* [Scheme Procedure]
Return true if *c* is of type `&message` or one of its subtypes.

`condition-message` *c* [Scheme Procedure]
Return the message associated with message condition *c*.

`&serious` [Переменная]
This type describes conditions serious enough that they cannot safely be ignored. It has no fields.

`serious-condition?` *c* [Scheme Procedure]
Return true if *c* is of type `&serious` or one of its subtypes.

`&error` [Переменная]
This condition describes errors, typically caused by something that has gone wrong in the interaction of the program with the external world or the user.

`error?` *c* [Scheme Procedure]
Return true if *c* is of type `&error` or one of its subtypes.

7.5.25 SRFI-37 - args-fold

This is a processor for GNU `getopt_long`-style program arguments. It provides an alternative, less declarative interface than `getopt-long` in `(ice-9 getopt-long)` (см. Раздел 7.4 [The (ice-9 getopt-long) Module], страница 600). Unlike `getopt-long`, it supports repeated options and any number of short and long names per option. Access it with:

```
(use-modules (srfi srfi-37))
```

SRFI-37 principally provides an `option` type and the `args-fold` function. To use the library, create a set of options with `option` and use it as a specification for invoking `args-fold`.

Here is an example of a simple argument processor for the typical '`--version`' and '`--help`' options, which returns a backwards list of files given on the command line:

```
(args-fold (cdr (program-arguments))
           (let ((display-and-exit-proc
                  (lambda (msg)
                    (lambda (opt name arg loads)
                      (display msg) (quit)))))
             (list (option '(#\v "version") #f #f
                           (display-and-exit-proc "Foo version 42.0\n"))
                   (option '(#\h "help") #f #f
                           (display-and-exit-proc
                            "Usage: foo scheme-file ..."))))
           (lambda (opt name arg loads)
             (error "Unrecognized option `~A'" name))
           (lambda (op loads) (cons op loads))
           '())
```

`option` *names required-arg? optional-arg? processor* [Scheme Procedure]

Return an object that specifies a single kind of program option.

names is a list of command-line option names, and should consist of characters for traditional `getopt` short options and strings for `getopt_long`-style long options.

required-arg? and *optional-arg?* are mutually exclusive; one or both must be `#f`. If *required-arg?*, the option must be followed by an argument on the command line, such as '`--opt=value`' for long options, or an error will be signalled. If *optional-arg?*, an argument will be taken if available.

processor is a procedure that takes at least 3 arguments, called when `args-fold` encounters the option: the containing option object, the name used on the command line, and the argument given for the option (or `#f` if none). The rest of the arguments are `args-fold` "seeds", and the *processor* should return seeds as well.

`option-names` *opt* [Scheme Procedure]
`option-required-arg?` *opt* [Scheme Procedure]
`option-optional-arg?` *opt* [Scheme Procedure]
`option-processor` *opt* [Scheme Procedure]

Return the specified field of *opt*, an option object, as described above for `option`.

`args-fold` *args options unrecognized-option-proc operand-proc seed ...* [Scheme Procedure]

Process *args*, a list of program arguments such as that returned by `(cdr (program-arguments))`, in order against *options*, a list of option objects as described above. All functions called take the "seeds", or the last multiple-values as multiple arguments, starting with *seed ...*, and must return the new seeds. Return the final seeds.

Call `unrecognized-option-proc`, which is like an option object's processor, for any options not found in *options*.

Call `operand-proc` with any items on the command line that are not named options. This includes arguments after '`--`'. It is called with the argument in question, as well as the seeds.

7.5.26 SRFI-38 - External Representation for Data With Shared Structure

This subsection is based on the specification of SRFI-38 (`http://srfi.schemers.org/srfi-38/srfi-38.html`) written by Ray Dillinger.

This SRFI creates an alternative external representation for data written and read using `write-with-shared-structure` and `read-with-shared-structure`. It is identical to the grammar for external representation for data written and read with `write` and `read` given in section 7 of R5RS, except that the single production

```
<datum> --> <simple datum> | <compound datum>
```

is replaced by the following five productions:

```
<datum> --> <defining datum> | <nondefining datum> | <defined datum>
<defining datum> -->  #<indexnum>=<nondefining datum>
<defined datum> --> #<indexnum>#
<nondefining datum> --> <simple datum> | <compound datum>
<indexnum> --> <digit 10>+
```

`write-with-shared-structure` *obj* [Scheme procedure]
`write-with-shared-structure` *obj port* [Scheme procedure]
`write-with-shared-structure` *obj port optarg* [Scheme procedure]

Writes an external representation of *obj* to the given port. Strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote characters are escaped by backslashes. Character objects are written using the `#\` notation.

Objects which denote locations rather than values (cons cells, vectors, and non-zero-length strings in R5RS scheme; also Guile's structs, bytevectors and ports and hash-tables), if they appear at more than one point in the data being written, are preceded by '`#N=`' the first time they are written and replaced by '`#N#`' all subsequent times they are written, where *N* is a natural number used to identify that particular object. If objects which denote locations occur only once in the structure, then `write-with-shared-structure` must produce the same external representation for those objects as `write`.

`write-with-shared-structure` terminates in finite time and produces a finite representation when writing finite data.

`write-with-shared-structure` returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `(current-output-port)`. The *optarg* argument may also be omitted. If present, its effects on the output and return value are unspecified but `write-with-shared-structure` must still write a representation that can be read by `read-with-shared-structure`. Some implementations may wish to use *optarg* to specify formatting conventions, numeric radixes, or return values. Guile's implementation ignores *optarg*.

For example, the code

```
(begin (define a (cons 'val1 'val2))
       (set-cdr! a a)
       (write-with-shared-structure a))
```

should produce the output `#1=(val1 . #1#)`. This shows a cons cell whose `cdr` contains itself.

`read-with-shared-structure` [Scheme procedure]
`read-with-shared-structure` *port* [Scheme procedure]

`read-with-shared-structure` converts the external representations of Scheme objects produced by `write-with-shared-structure` into Scheme objects. That is, it is a parser for the nonterminal '`<datum>`' in the augmented external representation grammar defined above. `read-with-shared-structure` returns the next object parsable from the given input port, updating *port* to point to the first character past the end of the external representation of the object.

If an end-of-file is encountered in the input before any characters are found that can begin an object, then an end-of-file object is returned. The port remains open, and further attempts to read it (by `read-with-shared-structure` or `read` will also return an end-of-file object. If an end of file is encountered after the beginning of an object's external representation, but the external representation is incomplete and therefore not parsable, an error is signalled.

The *port* argument may be omitted, in which case it defaults to the value returned by `(current-input-port)`. It is an error to read from a closed port.

7.5.27 SRFI-39 - Parameters

This SRFI adds support for dynamically-scoped parameters. SRFI 39 is implemented in the Guile core; there's no module needed to get SRFI-39 itself. Parameters are documented in Раздел 6.13.12 [Parameters], страница 347.

This module does export one extra function: `with-parameters*`. This is a Guile-specific addition to the SRFI, similar to the core `with-fluids*` (см. Раздел 6.13.11 [Fluids and Dynamic States], страница 343).

`with-parameters*` *param-list value-list thunk* [Функция]

Establish a new dynamic scope, as per `parameterize` above, taking parameters from *param-list* and corresponding values from *value-list*. A call `(thunk)` is made in the new scope and the result from that *thunk* is the return from `with-parameters*`.

7.5.28 SRFI-41 - Streams

This subsection is based on the specification of SRFI-41 (`http://srfi.schemers.org/srfi-41/srfi-41.html`) by Philip L. Bewig.

This SRFI implements streams, sometimes called lazy lists, a sequential data structure containing elements computed only on demand. A stream is either null or is a pair with a stream in its cdr. Since elements of a stream are computed only when accessed, streams can be infinite. Once computed, the value of a stream element is cached in case it is needed again. SRFI-41 can be made available with:

```
(use-modules (srfi srfi-41))
```

7.5.28.1 SRFI-41 Stream Fundamentals

SRFI-41 Streams are based on two mutually-recursive abstract data types: An object of the `stream` abstract data type is a promise that, when forced, is either `stream-null` or is an object of type `stream-pair`. An object of the `stream-pair` abstract data type contains a `stream-car` and a `stream-cdr`, which must be a `stream`. The essential feature of streams is the systematic suspensions of the recursive promises between the two data types.

The object stored in the `stream-car` of a `stream-pair` is a promise that is forced the first time the `stream-car` is accessed; its value is cached in case it is needed again. The object may have any type, and different stream elements may have different types. If the `stream-car` is never accessed, the object stored there is never evaluated. Likewise, the `stream-cdr` is a promise to return a stream, and is only forced on demand.

7.5.28.2 SRFI-41 Stream Primitives

This library provides eight operators: constructors for `stream-null` and `stream-pair`s, type predicates for streams and the two kinds of streams, accessors for both fields of a `stream-pair`, and a lambda that creates procedures that return streams.

`stream-null` [Scheme Variable]

A promise that, when forced, is a single object, distinguishable from all other objects, that represents the null stream. `stream-null` is immutable and unique.

`stream-cons` *object-expr stream-expr* [Scheme Syntax]

Creates a newly-allocated stream containing a promise that, when forced, is a `stream-pair` with *object-expr* in its `stream-car` and *stream-expr* in its `stream-cdr`. Neither *object-expr* nor *stream-expr* is evaluated when `stream-cons` is called.

Once created, a `stream-pair` is immutable; there is no `stream-set-car!` or `stream-set-cdr!` that modifies an existing stream-pair. There is no dotted-pair or improper stream as with lists.

`stream?` *object* [Scheme Procedure]

Returns true if *object* is a stream, otherwise returns false. If *object* is a stream, its promise will not be forced. If `(stream? obj)` returns true, then one of `(stream-null? obj)` or `(stream-pair? obj)` will return true and the other will return false.

`stream-null?` *object* [Scheme Procedure]

Returns true if *object* is the distinguished null stream, otherwise returns false. If *object* is a stream, its promise will be forced.

`stream-pair?` *object* [Scheme Procedure]

Returns true if *object* is a `stream-pair` constructed by `stream-cons`, otherwise returns false. If *object* is a stream, its promise will be forced.

`stream-car` *stream* [Scheme Procedure]

Returns the object stored in the `stream-car` of *stream*. An error is signalled if the argument is not a `stream-pair`. This causes the *object-expr* passed to `stream-cons` to be evaluated if it had not yet been; the value is cached in case it is needed again.

`stream-cdr` *stream* [Scheme Procedure]

Returns the stream stored in the `stream-cdr` of *stream*. An error is signalled if the argument is not a `stream-pair`.

`stream-lambda` *formals body . . .* [Scheme Syntax]

Creates a procedure that returns a promise to evaluate the *body* of the procedure. The last *body* expression to be evaluated must yield a stream. As with normal `lambda`, *formals* may be a single variable name, in which case all the formal arguments are collected into a single list, or a list of variable names, which may be null if there are no arguments, proper if there are an exact number of arguments, or dotted if a fixed number of arguments is to be followed by zero or more arguments collected into a list. *Body* must contain at least one expression, and may contain internal definitions preceding any expressions to be evaluated.

```
(define strm123
  (stream-cons 1
    (stream-cons 2
      (stream-cons 3
        stream-null))))

(stream-car strm123) ⇒ 1
(stream-car (stream-cdr strm123) ⇒ 2

(stream-pair?
  (stream-cdr
    (stream-cons (/ 1 0) stream-null))) ⇒ #f

(stream? (list 1 2 3)) ⇒ #f

(define iter
  (stream-lambda (f x)
    (stream-cons x (iter f (f x)))))

(define nats (iter (lambda (x) (+ x 1)) 0))

(stream-car (stream-cdr nats)) ⇒ 1

(define stream-add
  (stream-lambda (s1 s2)
```

```
      (stream-cons
        (+ (stream-car s1) (stream-car s2))
        (stream-add (stream-cdr s1)
                    (stream-cdr s2)))))

(define evens (stream-add nats nats))

(stream-car evens) ⇒ 0
(stream-car (stream-cdr evens)) ⇒ 2
(stream-car (stream-cdr (stream-cdr evens))) ⇒ 4
```

7.5.28.3 SRFI-41 Stream Library

`define-stream` (*name args ...*) *body ...* [Scheme Syntax]

Creates a procedure that returns a stream, and may appear anywhere a normal `define` may appear, including as an internal definition. It may contain internal definitions of its own. The defined procedure takes arguments in the same way as `stream-lambda`. `define-stream` is syntactic sugar on `stream-lambda`; see also `stream-let`, which is also a sugaring of `stream-lambda`.

A simple version of `stream-map` that takes only a single input stream calls itself recursively:

```
(define-stream (stream-map proc strm)
  (if (stream-null? strm)
      stream-null
      (stream-cons
        (proc (stream-car strm))
        (stream-map proc (stream-cdr strm)))))
```

`list->stream` *list* [Scheme Procedure]

Returns a newly-allocated stream containing the elements from *list*.

`port->stream` [*port*] [Scheme Procedure]

Returns a newly-allocated stream containing in its elements the characters on the port. If *port* is not given it defaults to the current input port. The returned stream has finite length and is terminated by `stream-null`.

It looks like one use of `port->stream` would be this:

```
(define s ;wrong!
  (with-input-from-file filename
    (lambda () (port->stream))))
```

But that fails, because `with-input-from-file` is eager, and closes the input port prematurely, before the first character is read. To read a file into a stream, say:

```
(define-stream (file->stream filename)
  (let ((p (open-input-file filename)))
    (stream-let loop ((c (read-char p)))
      (if (eof-object? c)
          (begin (close-input-port p)
                 stream-null)
```

```
(stream-cons c
  (loop (read-char p)))))))
```

`stream` *object-expr ...* [Scheme Syntax]

Creates a newly-allocated stream containing in its elements the objects, in order. The *object-expr*s are evaluated when they are accessed, not when the stream is created. If no objects are given, as in (stream), the null stream is returned. See also `list->stream`.

```
(define strm123 (stream 1 2 3))

; (/ 1 0) not evaluated when stream is created
(define s (stream 1 (/ 1 0) -1))
```

`stream->list` [*n*] *stream* [Scheme Procedure]

Returns a newly-allocated list containing in its elements the first *n* items in *stream*. If *stream* has less than *n* items, all the items in the stream will be included in the returned list. If *n* is not given it defaults to infinity, which means that unless *stream* is finite `stream->list` will never return.

```
(stream->list 10
  (stream-map (lambda (x) (* x x))
    (stream-from 0)))
  ⇒ (0 1 4 9 16 25 36 49 64 81)
```

`stream-append` *stream ...* [Scheme Procedure]

Returns a newly-allocated stream containing in its elements those elements contained in its input *stream*s, in order of input. If any of the input streams is infinite, no elements of any of the succeeding input streams will appear in the output stream. See also `stream-concat`.

`stream-concat` *stream* [Scheme Procedure]

Takes a *stream* consisting of one or more streams and returns a newly-allocated stream containing all the elements of the input streams. If any of the streams in the input *stream* is infinite, any remaining streams in the input stream will never appear in the output stream. See also `stream-append`.

`stream-constant` *object ...* [Scheme Procedure]

Returns a newly-allocated stream containing in its elements the *object*s, repeating in succession forever.

```
(stream-constant 1) ⇒ 1 1 1 ...
(stream-constant #t #f) ⇒ #t #f #t #f #t #f ...
```

`stream-drop` *n stream* [Scheme Procedure]

Returns the suffix of the input *stream* that starts at the next element after the first *n* elements. The output stream shares structure with the input *stream*; thus, promises forced in one instance of the stream are also forced in the other instance of the stream. If the input *stream* has less than *n* elements, `stream-drop` returns the null stream. See also `stream-take`.

`stream-drop-while` *pred stream* [Scheme Procedure]
Returns the suffix of the input *stream* that starts at the first element *x* for which `(pred x)` returns false. The output stream shares structure with the input *stream*. See also `stream-take-while`.

`stream-filter` *pred stream* [Scheme Procedure]
Returns a newly-allocated stream that contains only those elements *x* of the input *stream* which satisfy the predicate `pred`.

```
(stream-filter odd? (stream-from 0))
  ⇒ 1 3 5 7 9 ...
```

`stream-fold` *proc base stream* [Scheme Procedure]
Applies a binary procedure *proc* to *base* and the first element of *stream* to compute a new *base*, then applies the procedure to the new *base* and the next element of *stream* to compute a succeeding *base*, and so on, accumulating a value that is finally returned as the value of `stream-fold` when the end of the stream is reached. *stream* must be finite, or `stream-fold` will enter an infinite loop. See also `stream-scan`, which is similar to `stream-fold`, but useful for infinite streams. For readers familiar with other functional languages, this is a left-fold; there is no corresponding right-fold, since right-fold relies on finite streams that are fully-evaluated, in which case they may as well be converted to a list.

`stream-for-each` *proc stream ...* [Scheme Procedure]
Applies *proc* element-wise to corresponding elements of the input *stream*s for side-effects; it returns nothing. `stream-for-each` stops as soon as any of its input streams is exhausted.

`stream-from` *first* [*step*] [Scheme Procedure]
Creates a newly-allocated stream that contains *first* as its first element and increments each succeeding element by *step*. If *step* is not given it defaults to 1. *first* and *step* may be of any numeric type. `stream-from` is frequently useful as a generator in `stream-of` expressions. See also `stream-range` for a similar procedure that creates finite streams.

`stream-iterate` *proc base* [Scheme Procedure]
Creates a newly-allocated stream containing *base* in its first element and applies *proc* to each element in turn to determine the succeeding element. See also `stream-unfold` and `stream-unfolds`.

`stream-length` *stream* [Scheme Procedure]
Returns the number of elements in the *stream*; it does not evaluate its elements. `stream-length` may only be used on finite streams; it enters an infinite loop with infinite streams.

`stream-let` *tag* ((*var expr*) ...) *body ...* [Scheme Syntax]
Creates a local scope that binds each variable to the value of its corresponding expression. It additionally binds *tag* to a procedure which takes the bound variables as arguments and *body* as its defining expressions, binding the *tag* with `stream-lambda`. *tag* is in scope within body, and may be called recursively. When

the expanded expression defined by the `stream-let` is evaluated, `stream-let` evaluates the expressions in its *body* in an environment containing the newly-bound variables, returning the value of the last expression evaluated, which must yield a stream.

`stream-let` provides syntactic sugar on `stream-lambda`, in the same manner as normal `let` provides syntactic sugar on normal `lambda`. However, unlike normal `let`, the *tag* is required, not optional, because unnamed `stream-let` is meaningless.

For example, `stream-member` returns the first `stream-pair` of the input *strm* with a `stream-car` *x* that satisfies `(eql? obj x)`, or the null stream if *x* is not present in *strm*.

```
(define-stream (stream-member eql? obj strm)
  (stream-let loop ((strm strm))
    (cond ((stream-null? strm) strm)
          ((eql? obj (stream-car strm)) strm)
          (else (loop (stream-cdr strm))))))
```

`stream-map` *proc stream ...* [Scheme Procedure]

Applies *proc* element-wise to corresponding elements of the input *stream*s, returning a newly-allocated stream containing elements that are the results of those procedure applications. The output stream has as many elements as the minimum-length input stream, and may be infinite.

`stream-match` *stream clause ...* [Scheme Syntax]

Provides pattern-matching for streams. The input *stream* is an expression that evaluates to a stream. Clauses are of the form `(pattern [fender] expression)`, consisting of a *pattern* that matches a stream of a particular shape, an optional *fender* that must succeed if the pattern is to match, and an *expression* that is evaluated if the pattern matches. There are four types of patterns:

- () matches the null stream.
- (*pat0 pat1 ...*) matches a finite stream with length exactly equal to the number of pattern elements.
- (*pat0 pat1 ... . pat-rest*) matches an infinite stream, or a finite stream with length at least as great as the number of pattern elements before the literal dot.
- *pat* matches an entire stream. Should always appear last in the list of clauses; it's not an error to appear elsewhere, but subsequent clauses could never match.

Each pattern element may be either:

- An identifier, which matches any stream element. Additionally, the value of the stream element is bound to the variable named by the identifier, which is in scope in the *fender* and *expression* of the corresponding *clause*. Each identifier in a single pattern must be unique.
- A literal underscore (_), which matches any stream element but creates no bindings.

The *pattern*s are tested in order, left-to-right, until a matching pattern is found; if *fender* is present, it must evaluate to a true value for the match to be successful. Pattern variables are bound in the corresponding *fender* and *expression*. Once the

matching *pattern* is found, the corresponding *expression* is evaluated and returned as the result of the match. An error is signaled if no pattern matches the input *stream*.

`stream-match` is often used to distinguish null streams from non-null streams, binding *head* and *tail*:

```
(define (len strm)
  (stream-match strm
    (() 0)
    ((head . tail) (+ 1 (len tail)))))
```

Fenders can test the common case where two stream elements must be identical; the `else` pattern is an identifier bound to the entire stream, not a keyword as in `cond`.

```
(stream-match strm
  ((x y . _) (equal? x y) 'ok)
  (else 'error))
```

A more complex example uses two nested matchers to match two different stream arguments; `(stream-merge lt? . strms)` stably merges two or more streams ordered by the `lt?` predicate:

```
(define-stream (stream-merge lt? . strms)
  (define-stream (merge xx yy)
    (stream-match xx (() yy) ((x . xs)
      (stream-match yy (() xx) ((y . ys)
        (if (lt? y x)
            (stream-cons y (merge xx ys))
            (stream-cons x (merge xs yy))))))))
  (stream-let loop ((strms strms))
    (cond ((null? strms) stream-null)
          ((null? (cdr strms)) (car strms))
          (else (merge (car strms)
                       (apply stream-merge lt?
                         (cdr strms)))))))
```

`stream-of` *expr clause ...* [Scheme Syntax]

Provides the syntax of stream comprehensions, which generate streams by means of looping expressions. The result is a stream of objects of the type returned by *expr*. There are four types of clauses:

- (*var* `in` *stream-expr*) loops over the elements of *stream-expr*, in order from the start of the stream, binding each element of the stream in turn to *var*. `stream-from` and `stream-range` are frequently useful as generators for *stream-expr*.
- (*var* `is` *expr*) binds *var* to the value obtained by evaluating *expr*.
- (*pred expr*) includes in the output stream only those elements *x* which satisfy the predicate *pred*.

The scope of variables bound in the stream comprehension is the clauses to the right of the binding clause (but not the binding clause itself) plus the result expression.

When two or more generators are present, the loops are processed as if they are nested from left to right; that is, the rightmost generator varies fastest. A consequence of

this is that only the first generator may be infinite and all subsequent generators must be finite. If no generators are present, the result of a stream comprehension is a stream containing the result expression; thus, '`(stream-of 1)`' produces a finite stream containing only the element 1.

```
(stream-of (* x x)
  (x in (stream-range 0 10))
  (even? x))
  ⇒ 0 4 16 36 64

(stream-of (list a b)
  (a in (stream-range 1 4))
  (b in (stream-range 1 3)))
  ⇒ (1 1) (1 2) (2 1) (2 2) (3 1) (3 2)

(stream-of (list i j)
  (i in (stream-range 1 5))
  (j in (stream-range (+ i 1) 5)))
  ⇒ (1 2) (1 3) (1 4) (2 3) (2 4) (3 4)
```

`stream-range` *first past* [*step*] [Scheme Procedure]

Creates a newly-allocated stream that contains *first* as its first element and increments each succeeding element by *step*. The stream is finite and ends before *past*, which is not an element of the stream. If *step* is not given it defaults to 1 if *first* is less than past and -1 otherwise. *first*, *past* and *step* may be of any real numeric type. `stream-range` is frequently useful as a generator in `stream-of` expressions. See also `stream-from` for a similar procedure that creates infinite streams.

```
(stream-range 0 10) ⇒ 0 1 2 3 4 5 6 7 8 9
(stream-range 0 10 2) ⇒ 0 2 4 6 8
```

Successive elements of the stream are calculated by adding *step* to *first*, so if any of *first*, *past* or *step* are inexact, the length of the output stream may differ from `(ceiling (- (/ (- past first) step) 1)`.

`stream-ref` *stream n* [Scheme Procedure]

Returns the *n*th element of stream, counting from zero. An error is signaled if *n* is greater than or equal to the length of stream.

```
(define (fact n)
  (stream-ref
    (stream-scan * 1 (stream-from 1))
    n))
```

`stream-reverse` *stream* [Scheme Procedure]

Returns a newly-allocated stream containing the elements of the input *stream* but in reverse order. `stream-reverse` may only be used with finite streams; it enters an infinite loop with infinite streams. `stream-reverse` does not force evaluation of the elements of the stream.

`stream-scan` *proc base stream* [Scheme Procedure]

Accumulates the partial folds of an input *stream* into a newly-allocated output stream. The output stream is the *base* followed by `(stream-fold proc base (stream-take i stream))` for each of the first *i* elements of *stream*.

```
(stream-scan + 0 (stream-from 1))
  ⇒ (stream 0 1 3 6 10 15 ...)

(stream-scan * 1 (stream-from 1))
  ⇒ (stream 1 1 2 6 24 120 ...)
```

`stream-take` *n stream* [Scheme Procedure]

Returns a newly-allocated stream containing the first *n* elements of the input *stream*. If the input *stream* has less than *n* elements, so does the output stream. See also `stream-drop`.

`stream-take-while` *pred stream* [Scheme Procedure]

Takes a predicate and a `stream` and returns a newly-allocated stream containing those elements `x` that form the maximal prefix of the input stream which satisfy *pred*. See also `stream-drop-while`.

`stream-unfold` *map pred gen base* [Scheme Procedure]

The fundamental recursive stream constructor. It constructs a stream by repeatedly applying *gen* to successive values of *base*, in the manner of `stream-iterate`, then applying *map* to each of the values so generated, appending each of the mapped values to the output stream as long as `(pred? base)` returns a true value. See also `stream-iterate` and `stream-unfolds`.

The expression below creates the finite stream '`0 1 4 9 16 25 36 49 64 81`'. Initially the *base* is 0, which is less than 10, so *map* squares the *base* and the mapped value becomes the first element of the output stream. Then *gen* increments the *base* by 1, so it becomes 1; this is less than 10, so *map* squares the new *base* and 1 becomes the second element of the output stream. And so on, until the base becomes 10, when *pred* stops the recursion and stream-null ends the output stream.

```
(stream-unfold
  (lambda (x) (expt x 2)) ; map
  (lambda (x) (< x 10))   ; pred?
  (lambda (x) (+ x 1))    ; gen
  0)                      ; base
```

`stream-unfolds` *proc seed* [Scheme Procedure]

Returns *n* newly-allocated streams containing those elements produced by successive calls to the generator *proc*, which takes the current *seed* as its argument and returns *n*+1 values

(*proc seed*) ⇒ *seed result_0 ... result_n-1*

where the returned *seed* is the input *seed* to the next call to the generator and *result_i* indicates how to produce the next element of the *i*th result stream:

- (*value*): *value* is the next car of the result stream.

- `#f`: no value produced by this iteration of the generator *proc* for the result stream.
- (): the end of the result stream.

It may require multiple calls of *proc* to produce the next element of any particular result stream. See also `stream-iterate` and `stream-unfold`.

```
(define (stream-partition pred? strm)
  (stream-unfolds
    (lambda (s)
      (if (stream-null? s)
          (values s '() '())
          (let ((a (stream-car s))
                (d (stream-cdr s)))
            (if (pred? a)
                (values d (list a) #f)
                (values d #f (list a))))))
    strm))

(call-with-values
  (lambda ()
    (stream-partition odd?
      (stream-range 1 6)))
  (lambda (odds evens)
    (list (stream->list odds)
          (stream->list evens))))
  ⇒ ((1 3 5) (2 4))
```

`stream-zip` *stream ...* [Scheme Procedure]

Returns a newly-allocated stream in which each element is a list (not a stream) of the corresponding elements of the input *stream*s. The output stream is as long as the shortest input *stream*, if any of the input *stream*s is finite, or is infinite if all the input *stream*s are infinite.

7.5.29 SRFI-42 - Eager Comprehensions

See the specification of SRFI-42 (`http://srfi.schemers.org/srfi-42/srfi-42.html`).

7.5.30 SRFI-43 - Vector Library

This subsection is based on the specification of SRFI-43 (`http://srfi.schemers.org/srfi-43/srfi-43.html`) by Taylor Campbell.

SRFI-43 implements a comprehensive library of vector operations. It can be made available with:

```
(use-modules (srfi srfi-43))
```

7.5.30.1 SRFI-43 Constructors

`make-vector` *size* [*fill*] [Scheme Procedure]

Create and return a vector of size *size*, optionally filling it with *fill*. The default value of *fill* is unspecified.

```
(make-vector 5 3) ⇒ #(3 3 3 3 3)
```

`vector` *x ...* [Scheme Procedure]

Create and return a vector whose elements are *x ...*.

```
(vector 0 1 2 3 4) ⇒ #(0 1 2 3 4)
```

`vector-unfold` *f length initial-seed ...* [Scheme Procedure]

The fundamental vector constructor. Create a vector whose length is *length* and iterates across each index k from 0 up to *length* - 1, applying *f* at each iteration to the current index and current seeds, in that order, to receive n + 1 values: the element to put in the kth slot of the new vector, and n new seeds for the next iteration. It is an error for the number of seeds to vary between iterations.

```
(vector-unfold (lambda (i x) (values x (- x 1)))
               10 0)
⇒ #(0 -1 -2 -3 -4 -5 -6 -7 -8 -9)

(vector-unfold values 10)
⇒ #(0 1 2 3 4 5 6 7 8 9)
```

`vector-unfold-right` *f length initial-seed ...* [Scheme Procedure]

Like `vector-unfold`, but it uses *f* to generate elements from right-to-left, rather than left-to-right.

```
(vector-unfold-right (lambda (i x) (values x (+ x 1)))
                     10 0)
⇒ #(9 8 7 6 5 4 3 2 1 0)
```

`vector-copy` *vec* [*start* [*end* [*fill*]]] [Scheme Procedure]

Allocate a new vector whose length is *end* - *start* and fills it with elements from *vec*, taking elements from *vec* starting at index *start* and stopping at index *end*. *start* defaults to 0 and *end* defaults to the value of `(vector-length vec)`. If *end* extends beyond the length of *vec*, the slots in the new vector that obviously cannot be filled by elements from *vec* are filled with *fill*, whose default value is unspecified.

```
(vector-copy '#(a b c d e f g h i))
⇒ #(a b c d e f g h i)

(vector-copy '#(a b c d e f g h i) 6)
⇒ #(g h i)

(vector-copy '#(a b c d e f g h i) 3 6)
⇒ #(d e f)

(vector-copy '#(a b c d e f g h i) 6 12 'x)
⇒ #(g h i x x x)
```

`vector-reverse-copy` *vec* [*start* [*end*]] [Scheme Procedure]

Like `vector-copy`, but it copies the elements in the reverse order from *vec*.

```
(vector-reverse-copy '#(5 4 3 2 1 0) 1 5)
⇒ #(1 2 3 4)
```

`vector-append` *vec ...* [Scheme Procedure]
Return a newly allocated vector that contains all elements in order from the subsequent locations in *vec ...*.

```
(vector-append '#(a) '#(b c d))
⇒ #(a b c d)
```

`vector-concatenate` *list-of-vectors* [Scheme Procedure]
Append each vector in *list-of-vectors*. Equivalent to `(apply vector-append list-of-vectors)`.

```
(vector-concatenate '(#(a b) #(c d)))
⇒ #(a b c d)
```

7.5.30.2 SRFI-43 Predicates

`vector?` *obj* [Scheme Procedure]
Return true if *obj* is a vector, else return false.

`vector-empty?` *vec* [Scheme Procedure]
Return true if *vec* is empty, i.e. its length is 0, else return false.

`vector=` *elt=? vec ...* [Scheme Procedure]
Return true if the vectors *vec ...* have equal lengths and equal elements according to *elt=?*. *elt=?* is always applied to two arguments. Element comparison must be consistent with `eq?` in the following sense: if `(eq? a b)` returns true, then `(elt=? a b)` must also return true. The order in which comparisons are performed is unspecified.

7.5.30.3 SRFI-43 Selectors

`vector-ref` *vec i* [Scheme Procedure]
Return the element at index *i* in *vec*. Indexing is based on zero.

`vector-length` *vec* [Scheme Procedure]
Return the length of *vec*.

7.5.30.4 SRFI-43 Iteration

`vector-fold` *kons knil vec1 vec2 ...* [Scheme Procedure]
The fundamental vector iterator. *kons* is iterated over each index in all of the vectors, stopping at the end of the shortest; *kons* is applied as

```
(kons i state (vector-ref vec1 i) (vector-ref vec2 i) ...)
```

where *state* is the current state value, and *i* is the current index. The current state value begins with *knil*, and becomes whatever *kons* returned at the respective iteration. The iteration is strictly left-to-right.

`vector-fold-right` *kons knil vec1 vec2 ...* [Scheme Procedure]
Similar to `vector-fold`, but it iterates right-to-left instead of left-to-right.

`vector-map` *f vec1 vec2 ...* [Scheme Procedure]
Return a new vector of the shortest size of the vector arguments. Each element at index i of the new vector is mapped from the old vectors by

```
(f i (vector-ref vec1 i) (vector-ref vec2 i) ...)
```

The dynamic order of application of *f* is unspecified.

`vector-map!` *f vec1 vec2 ...* [Scheme Procedure]
Similar to `vector-map`, but rather than mapping the new elements into a new vector, the new mapped elements are destructively inserted into *vec1*. The dynamic order of application of *f* is unspecified.

`vector-for-each` *f vec1 vec2 ...* [Scheme Procedure]
Call `(f i (vector-ref vec1 i) (vector-ref vec2 i) ...)` for each index i less than the length of the shortest vector passed. The iteration is strictly left-to-right.

`vector-count` *pred? vec1 vec2 ...* [Scheme Procedure]
Count the number of parallel elements in the vectors that satisfy *pred?*, which is applied, for each index i less than the length of the smallest vector, to i and each parallel element in the vectors at that index, in order.

```
(vector-count (lambda (i elt) (even? elt))
              '#(3 1 4 1 5 9 2 5 6))
⇒ 3
(vector-count (lambda (i x y) (< x y))
              '#(1 3 6 9) '#(2 4 6 8 10 12))
⇒ 2
```

7.5.30.5 SRFI-43 Searching

`vector-index` *pred? vec1 vec2 ...* [Scheme Procedure]
Find and return the index of the first elements in *vec1 vec2 ...* that satisfy *pred?*. If no matching element is found by the end of the shortest vector, return `#f`.

```
(vector-index even? '#(3 1 4 1 5 9))
⇒ 2
(vector-index < '#(3 1 4 1 5 9 2 5 6) '#(2 7 1 8 2))
⇒ 1
(vector-index = '#(3 1 4 1 5 9 2 5 6) '#(2 7 1 8 2))
⇒ #f
```

`vector-index-right` *pred? vec1 vec2 ...* [Scheme Procedure]
Like `vector-index`, but it searches right-to-left, rather than left-to-right. Note that the SRFI 43 specification requires that all the vectors must have the same length, but both the SRFI 43 reference implementation and Guile's implementation allow vectors with unequal lengths, and start searching from the last index of the shortest vector.

`vector-skip` *pred? vec1 vec2 ...* [Scheme Procedure]
Find and return the index of the first elements in *vec1 vec2 ...* that do not satisfy *pred?*. If no matching element is found by the end of the shortest vector, return `#f`. Equivalent to `vector-index` but with the predicate inverted.

```
(vector-skip number? '#(1 2 a b 3 4 c d)) ⇒ 2
```

`vector-skip-right` *pred? vec1 vec2 . . .* [Scheme Procedure]
Like `vector-skip`, but it searches for a non-matching element right-to-left, rather than left-to-right. Note that the SRFI 43 specification requires that all the vectors must have the same length, but both the SRFI 43 reference implementation and Guile's implementation allow vectors with unequal lengths, and start searching from the last index of the shortest vector.

`vector-binary-search` *vec value cmp* [*start* [*end*]] [Scheme Procedure]
Find and return an index of *vec* between *start* and *end* whose value is *value* using a binary search. If no matching element is found, return `#f`. The default *start* is 0 and the default *end* is the length of *vec*.

cmp must be a procedure of two arguments such that `(cmp a b)` returns a negative integer if $a < b$, a positive integer if $a > b$, or zero if $a = b$. The elements of *vec* must be sorted in non-decreasing order according to *cmp*.

Note that SRFI 43 does not document the *start* and *end* arguments, but both its reference implementation and Guile's implementation support them.

```
(define (char-cmp c1 c2)
  (cond ((char<? c1 c2) -1)
        ((char>? c1 c2) 1)
        (else 0)))

(vector-binary-search '#(#\a #\b #\c #\d #\e #\f #\g #\h)
                      #\g
                      char-cmp)
⇒ 6
```

`vector-any` *pred? vec1 vec2 . . .* [Scheme Procedure]
Find the first parallel set of elements from *vec1 vec2 . . .* for which *pred?* returns a true value. If such a parallel set of elements exists, `vector-any` returns the value that *pred?* returned for that set of elements. The iteration is strictly left-to-right.

`vector-every` *pred? vec1 vec2 . . .* [Scheme Procedure]
If, for every index i between 0 and the length of the shortest vector argument, the set of elements `(vector-ref vec1 i) (vector-ref vec2 i) . . .` satisfies *pred?*, `vector-every` returns the value that *pred?* returned for the last set of elements, at the last index of the shortest vector. Otherwise it returns `#f`. The iteration is strictly left-to-right.

7.5.30.6 SRFI-43 Mutators

`vector-set!` *vec i value* [Scheme Procedure]
Assign the contents of the location at *i* in *vec* to *value*.

`vector-swap!` *vec i j* [Scheme Procedure]
Swap the values of the locations in *vec* at *i* and *j*.

`vector-fill!` *vec fill* [*start* [*end*]] [Scheme Procedure]
Assign the value of every location in *vec* between *start* and *end* to *fill*. *start* defaults to 0 and *end* defaults to the length of *vec*.

`vector-reverse!` *vec* [*start* [*end*]] [Scheme Procedure]
Destructively reverse the contents of *vec* between *start* and *end*. *start* defaults to 0 and *end* defaults to the length of *vec*.

`vector-copy!` *target tstart source* [*sstart* [*send*]] [Scheme Procedure]
Copy a block of elements from *source* to *target*, both of which must be vectors, starting in *target* at *tstart* and starting in *source* at *sstart*, ending when (*send* - *sstart*) elements have been copied. It is an error for *target* to have a length less than (*tstart* + *send* - *sstart*). *sstart* defaults to 0 and *send* defaults to the length of *source*.

`vector-reverse-copy!` *target tstart source* [*sstart* [*send*]] [Scheme Procedure]
Like `vector-copy!`, but this copies the elements in the reverse order. It is an error if *target* and *source* are identical vectors and the *target* and *source* ranges overlap; however, if *tstart* = *sstart*, `vector-reverse-copy!` behaves as `(vector-reverse! target tstart send)` would.

7.5.30.7 SRFI-43 Conversion

`vector->list` *vec* [*start* [*end*]] [Scheme Procedure]
Return a newly allocated list containing the elements in *vec* between *start* and *end*. *start* defaults to 0 and *end* defaults to the length of *vec*.

`reverse-vector->list` *vec* [*start* [*end*]] [Scheme Procedure]
Like `vector->list`, but the resulting list contains the specified range of elements of *vec* in reverse order.

`list->vector` *proper-list* [*start* [*end*]] [Scheme Procedure]
Return a newly allocated vector of the elements from *proper-list* with indices between *start* and *end*. *start* defaults to 0 and *end* defaults to the length of *proper-list*. Note that SRFI 43 does not document the *start* and *end* arguments, but both its reference implementation and Guile's implementation support them.

`reverse-list->vector` *proper-list* [*start* [*end*]] [Scheme Procedure]
Like `list->vector`, but the resulting vector contains the specified range of elements of *proper-list* in reverse order. Note that SRFI 43 does not document the *start* and *end* arguments, but both its reference implementation and Guile's implementation support them.

7.5.31 SRFI-45 - Primitives for Expressing Iterative Lazy Algorithms

This subsection is based on the specification of SRFI-45 (`http://srfi.schemers.org/srfi-45/srfi-45.html`) written by Andre van Tonder.

Lazy evaluation is traditionally simulated in Scheme using `delay` and `force`. However, these primitives are not powerful enough to express a large class of lazy algorithms that

are iterative. Indeed, it is folklore in the Scheme community that typical iterative lazy algorithms written using delay and force will often require unbounded memory.

This SRFI provides set of three operations: {`lazy`, `delay`, `force`}, which allow the programmer to succinctly express lazy algorithms while retaining bounded space behavior in cases that are properly tail-recursive. A general recipe for using these primitives is provided. An additional procedure `eager` is provided for the construction of eager promises in cases where efficiency is a concern.

Although this SRFI redefines `delay` and `force`, the extension is conservative in the sense that the semantics of the subset {`delay`, `force`} in isolation (i.e., as long as the program does not use `lazy`) agrees with that in R5RS. In other words, no program that uses the R5RS definitions of delay and force will break if those definition are replaced by the SRFI-45 definitions of delay and force.

Guile also adds `promise?` to the list of exports, which is not part of the official SRFI-45.

`promise?` *obj* [Scheme Procedure]
Return true if *obj* is an SRFI-45 promise, otherwise return false.

`delay` *expression* [Scheme Syntax]
Takes an expression of arbitrary type *a* and returns a promise of type `(Promise a)` which at some point in the future may be asked (by the `force` procedure) to evaluate the expression and deliver the resulting value.

`lazy` *expression* [Scheme Syntax]
Takes an expression of type `(Promise a)` and returns a promise of type `(Promise a)` which at some point in the future may be asked (by the `force` procedure) to evaluate the expression and deliver the resulting promise.

`force` *expression* [Scheme Procedure]
Takes an argument of type `(Promise a)` and returns a value of type *a* as follows: If a value of type *a* has been computed for the promise, this value is returned. Otherwise, the promise is first evaluated, then overwritten by the obtained promise or value, and then force is again applied (iteratively) to the promise.

`eager` *expression* [Scheme Procedure]
Takes an argument of type *a* and returns a value of type `(Promise a)`. As opposed to `delay`, the argument is evaluated eagerly. Semantically, writing `(eager expression)` is equivalent to writing

```
(let ((value expression)) (delay value)).
```

However, the former is more efficient since it does not require unnecessary creation and evaluation of thunks. We also have the equivalence

```
(delay expression) = (lazy (eager expression))
```

The following reduction rules may be helpful for reasoning about these primitives. However, they do not express the memoization and memory usage semantics specified above:

```
(force (delay expression)) -> expression
(force (lazy  expression)) -> (force expression)
(force (eager value))      -> value
```

Correct usage

We now provide a general recipe for using the primitives {`lazy`, `delay`, `force`} to express lazy algorithms in Scheme. The transformation is best described by way of an example: Consider the stream-filter algorithm, expressed in a hypothetical lazy language as

```
(define (stream-filter p? s)
  (if (null? s) '()
      (let ((h (car s))
            (t (cdr s)))
        (if (p? h)
            (cons h (stream-filter p? t))
            (stream-filter p? t)))))
```

This algorithm can be expressed as follows in Scheme:

```
(define (stream-filter p? s)
  (lazy
     (if (null? (force s)) (delay '())
         (let ((h (car (force s)))
               (t (cdr (force s))))
           (if (p? h)
               (delay (cons h (stream-filter p? t)))
               (stream-filter p? t))))))
```

In other words, we

- wrap all constructors (e.g., `'()`, `cons`) with `delay`,
- apply `force` to arguments of deconstructors (e.g., `car`, `cdr` and `null?`),
- wrap procedure bodies with `(lazy ...)`.

7.5.32 SRFI-46 Basic syntax-rules Extensions

Guile's core `syntax-rules` supports the extensions specified by SRFI-46/R7RS. Tail patterns have been supported since at least Guile 2.0, and custom ellipsis identifiers have been supported since Guile 2.0.10. См. Раздел 6.10.2 [Syntax Rules], страница 278.

7.5.33 SRFI-55 - Requiring Features

SRFI-55 provides `require-extension` which is a portable mechanism to load selected SRFI modules. This is implemented in the Guile core, there's no module needed to get SRFI-55 itself.

`require-extension` *clause1 clause2 ...* [library syntax]

Require the features of *clause1 clause2 ...* , throwing an error if any are unavailable.

A *clause* is of the form (`identifier` `arg...`). The only *identifier* currently supported is `srfi` and the arguments are SRFI numbers. For example to get SRFI-1 and SRFI-6,

```
(require-extension (srfi 1 6))
```

`require-extension` can only be used at the top-level.

A Guile-specific program can simply `use-modules` to load SRFIs not already in the core, `require-extension` is for programs designed to be portable to other Scheme implementations.

7.5.34 SRFI-60 - Integers as Bits

This SRFI provides various functions for treating integers as bits and for bitwise manipulations. These functions can be obtained with,

```
(use-modules (srfi srfi-60))
```

Integers are treated as infinite precision twos-complement, the same as in the core logical functions (см. Раздел 6.6.2.13 [Bitwise Operations], страница 134). And likewise bit indexes start from 0 for the least significant bit. The following functions in this SRFI are already in the Guile core,

`logand`, `logior`, `logxor`, `lognot`, `logtest`, `logcount`, `integer-length`, `logbit?`, `ash`

`bitwise-and` *n1 ...* [Функция]
`bitwise-ior` *n1 ...* [Функция]
`bitwise-xor` *n1 ...* [Функция]
`bitwise-not` *n* [Функция]
`any-bits-set?` *j k* [Функция]
`bit-set?` *index n* [Функция]
`arithmetic-shift` *n count* [Функция]
`bit-field` *n start end* [Функция]
`bit-count` *n* [Функция]

Aliases for `logand`, `logior`, `logxor`, `lognot`, `logtest`, `logbit?`, `ash`, `bit-extract` and `logcount` respectively.

Note that the name `bit-count` conflicts with `bit-count` in the core (см. Раздел 6.6.11 [Bit Vectors], страница 202).

`bitwise-if` *mask n1 n0* [Функция]
`bitwise-merge` *mask n1 n0* [Функция]

Return an integer with bits selected from *n1* and *n0* according to *mask*. Those bits where *mask* has 1s are taken from *n1*, and those where *mask* has 0s are taken from *n0*.

```
(bitwise-if 3 #b0101 #b1010) ⇒ 9
```

`log2-binary-factors` *n* [Функция]
`first-set-bit` *n* [Функция]

Return a count of how many factors of 2 are present in *n*. This is also the bit index of the lowest 1 bit in *n*. If *n* is 0, the return is −1.

```
(log2-binary-factors 6) ⇒ 1
(log2-binary-factors -8) ⇒ 3
```

`copy-bit` *index n newbit* [Функция]

Return *n* with the bit at *index* set according to *newbit*. *newbit* should be `#t` to set the bit to 1, or `#f` to set it to 0. Bits other than at *index* are unchanged in the return.

```
(copy-bit 1 #b0101 #t) ⇒ 7
```

`copy-bit-field` *n newbits start end* [Функция]

Return *n* with the bits from *start* (inclusive) to *end* (exclusive) changed to the value *newbits*.

The least significant bit in *newbits* goes to *start*, the next to *start* + 1, etc. Anything in *newbits* past the *end* given is ignored.

```
(copy-bit-field #b10000 #b11 1 3) ⇒ #b10110
```

`rotate-bit-field` *n count start end* [Функция]

Return *n* with the bit field from *start* (inclusive) to *end* (exclusive) rotated upwards by *count* bits.

count can be positive or negative, and it can be more than the field width (it'll be reduced modulo the width).

```
(rotate-bit-field #b0110 2 1 4) ⇒ #b1010
```

`reverse-bit-field` *n start end* [Функция]

Return *n* with the bits from *start* (inclusive) to *end* (exclusive) reversed.

```
(reverse-bit-field #b101001 2 4) ⇒ #b100101
```

`integer->list` *n* [*len*] [Функция]

Return bits from *n* in the form of a list of `#t` for 1 and `#f` for 0. The least significant *len* bits are returned, and the first list element is the most significant of those bits. If *len* is not given, the default is `(integer-length n)` (см. Раздел 6.6.2.13 [Bitwise Operations], страница 134).

```
(integer->list 6)   ⇒ (#t #t #f)
(integer->list 1 4) ⇒ (#f #f #f #t)
```

`list->integer` *lst* [Функция]

`booleans->integer` *bool...* [Функция]

Return an integer formed bitwise from the given *lst* list of booleans, or for `booleans->integer` from the *bool* arguments.

Each boolean is `#t` for a 1 and `#f` for a 0. The first element becomes the most significant bit in the return.

```
(list->integer '(#t #f #t #f)) ⇒ 10
```

7.5.35 SRFI-61 - A more general `cond` clause

This SRFI extends RnRS `cond` to support test expressions that return multiple values, as well as arbitrary definitions of test success. SRFI 61 is implemented in the Guile core; there's no module needed to get SRFI-61 itself. Extended `cond` is documented in Раздел 6.13.2 [Simple Conditional Evaluation], страница 318.

7.5.36 SRFI-62 - S-expression comments.

Starting from version 2.0, Guile's `read` supports SRFI-62/R7RS S-expression comments by default.

7.5.37 SRFI-64 - A Scheme API for test suites.

See the specification of SRFI-64 (`http://srfi.schemers.org/srfi-64/srfi-64.html`).

7.5.38 SRFI-67 - Compare procedures

See the specification of SRFI-67 (`http://srfi.schemers.org/srfi-67/srfi-67.html`).

7.5.39 SRFI-69 - Basic hash tables

This is a portable wrapper around Guile's built-in hash table and weak table support. См. Раздел 6.6.22 [Hash Tables], страница 251, for information on that built-in support. Above that, this hash-table interface provides association of equality and hash functions with tables at creation time, so variants of each function are not required, as well as a procedure that takes care of most uses for Guile hash table handles, which this SRFI does not provide as such.

Access it with:

```
(use-modules (srfi srfi-69))
```

7.5.39.1 Creating hash tables

`make-hash-table` [*equal-proc hash-proc #:weak weakness start-size*] [Scheme Procedure]

Create and answer a new hash table with *equal-proc* as the equality function and *hash-proc* as the hashing function.

By default, *equal-proc* is `equal?`. It can be any two-argument procedure, and should answer whether two keys are the same for this table's purposes.

My default *hash-proc* assumes that `equal-proc` is no coarser than `equal?` unless it is literally `string-ci=?`. If provided, *hash-proc* should be a two-argument procedure that takes a key and the current table size, and answers a reasonably good hash integer between 0 (inclusive) and the size (exclusive).

weakness should be `#f` or a symbol indicating how "weak" the hash table is:

`#f` An ordinary non-weak hash table. This is the default.

`key` When the key has no more non-weak references at GC, remove that entry.

`value` When the value has no more non-weak references at GC, remove that entry.

`key-or-value`
When either has no more non-weak references at GC, remove the association.

As a legacy of the time when Guile couldn't grow hash tables, *start-size* is an optional integer argument that specifies the approximate starting size for the hash table, which will be rounded to an algorithmically-sounder number.

By *coarser* than `equal?`, we mean that for all *x* and *y* values where `(`*`equal-proc x y`*`)`, `(equal?` *x y*`)` as well. If that does not hold for your *equal-proc*, you must provide a *hash-proc*.

In the case of weak tables, remember that *references* above always refers to `eq?`-wise references. Just because you have a reference to some string `"foo"` doesn't mean that an association with key `"foo"` in a weak-key table *won't* be collected; it only counts as a reference if the two `"foo"`s are `eq?`, regardless of *equal-proc*. As such, it is usually only sensible to use `eq?` and `hashq` as the equivalence and hash functions for a weak table. См. Раздел 6.19.3 [Weak References], страница 429, for more information on Guile's built-in weak table support.

`alist->hash-table` *alist* [*equal-proc hash-proc #:weak weakness start-size*] [Scheme Procedure]

As with `make-hash-table`, but initialize it with the associations in *alist*. Where keys are repeated in *alist*, the leftmost association takes precedence.

7.5.39.2 Accessing table items

`hash-table-ref` *table key* [*default-thunk*] [Scheme Procedure]
`hash-table-ref/default` *table key default* [Scheme Procedure]

Answer the value associated with *key* in *table*. If *key* is not present, answer the result of invoking the thunk *default-thunk*, which signals an error instead by default.

`hash-table-ref/default` is a variant that requires a third argument, *default*, and answers *default* itself instead of invoking it.

`hash-table-set!` *table key new-value* [Scheme Procedure]

Set *key* to *new-value* in *table*.

`hash-table-delete!` *table key* [Scheme Procedure]

Remove the association of *key* in *table*, if present. If absent, do nothing.

`hash-table-exists?` *table key* [Scheme Procedure]

Answer whether *key* has an association in *table*.

`hash-table-update!` *table key modifier* [*default-thunk*] [Scheme Procedure]
`hash-table-update!/default` *table key modifier default* [Scheme Procedure]

Replace *key*'s associated value in *table* by invoking *modifier* with one argument, the old value.

If *key* is not present, and *default-thunk* is provided, invoke it with no arguments to get the "old value" to be passed to *modifier* as above. If *default-thunk* is not provided in such a case, signal an error.

`hash-table-update!/default` is a variant that requires the fourth argument, which is used directly as the "old value" rather than as a thunk to be invoked to retrieve the "old value".

7.5.39.3 Table properties

`hash-table-size` *table* [Scheme Procedure]

Answer the number of associations in *table*. This is guaranteed to run in constant time for non-weak tables.

`hash-table-keys` *table* [Scheme Procedure]

Answer an unordered list of the keys in *table*.

`hash-table-values` *table* [Scheme Procedure]

Answer an unordered list of the values in *table*.

`hash-table-walk` *table proc* [Scheme Procedure]

Invoke *proc* once for each association in *table*, passing the key and value as arguments.

`hash-table-fold` *table proc init* [Scheme Procedure]
Invoke (*proc key value previous*) for each *key* and *value* in *table*, where *previous* is the result of the previous invocation, using *init* as the first *previous* value. Answer the final *proc* result.

`hash-table->alist` *table* [Scheme Procedure]
Answer an alist where each association in *table* is an association in the result.

7.5.39.4 Hash table algorithms

Each hash table carries an *equivalence function* and a *hash function*, used to implement key lookups. Beginning users should follow the rules for consistency of the default *hash-proc* specified above. Advanced users can use these to implement their own equivalence and hash functions for specialized lookup semantics.

`hash-table-equivalence-function` *hash-table* [Scheme Procedure]
`hash-table-hash-function` *hash-table* [Scheme Procedure]
Answer the equivalence and hash function of *hash-table*, respectively.

`hash` *obj* [*size*] [Scheme Procedure]
`string-hash` *obj* [*size*] [Scheme Procedure]
`string-ci-hash` *obj* [*size*] [Scheme Procedure]
`hash-by-identity` *obj* [*size*] [Scheme Procedure]
Answer a hash value appropriate for equality predicate `equal?`, `string=?`, `string-ci=?`, and `eq?`, respectively.

`hash` is a backwards-compatible replacement for Guile's built-in `hash`.

7.5.40 SRFI-87 => in case clauses

Starting from version 2.0.6, Guile's core `case` syntax supports `=>` in clauses, as specified by SRFI-87/R7RS. См. Раздел 6.13.2 [Conditionals], страница 318.

7.5.41 SRFI-88 Keyword Objects

SRFI-88 (`http://srfi.schemers.org/srfi-88/srfi-88.html`) provides *keyword objects*, which are equivalent to Guile's keywords (см. Раздел 6.6.7 [Keywords], страница 184). SRFI-88 keywords can be entered using the *postfix keyword syntax*, which consists of an identifier followed by `:` (см. Раздел 6.18.2 [Scheme Read], страница 407). SRFI-88 can be made available with:

```
(use-modules (srfi srfi-88))
```

Doing so installs the right reader option for keyword syntax, using `(read-set! keywords 'postfix)`. It also provides the procedures described below.

`keyword?` *obj* [Scheme Procedure]
Return `#t` if *obj* is a keyword. This is the same procedure as the same-named built-in procedure (см. Раздел 6.6.7.4 [Keyword Procedures], страница 187).

```
(keyword? foo:)         ⇒ #t
(keyword? 'foo:)        ⇒ #t
(keyword? "foo")        ⇒ #f
```

`keyword->string` *kw* [Scheme Procedure]

Return the name of *kw* as a string, i.e., without the trailing colon. The returned string may not be modified, e.g., with `string-set!`.

```
(keyword->string foo:)  ⇒ "foo"
```

`string->keyword` *str* [Scheme Procedure]

Return the keyword object whose name is *str*.

```
(keyword->string (string->keyword "a b c"))     ⇒ "a b c"
```

7.5.42 SRFI-98 Accessing environment variables.

This is a portable wrapper around Guile's built-in support for interacting with the current environment, См. Раздел 7.2.6 [Runtime Environment], страница 540.

`get-environment-variable` *name* [Scheme Procedure]

Returns a string containing the value of the environment variable given by the string `name`, or `#f` if the named environment variable is not found. This is equivalent to `(getenv name)`.

`get-environment-variables` [Scheme Procedure]

Returns the names and values of all the environment variables as an association list in which both the keys and the values are strings.

7.5.43 SRFI-105 Curly-infix expressions.

Guile's built-in reader includes support for SRFI-105 curly-infix expressions. See the specification of SRFI-105 (`http://srfi.schemers.org/srfi-105/srfi-105.html`). Some examples:

```
{n <= 5}                ⇒  (<= n 5)
{a + b + c}             ⇒  (+ a b c)
{a * {b + c}}           ⇒  (* a (+ b c))
{(- a) / b}             ⇒  (/ (- a) b)
{-(a) / b}              ⇒  (/ (- a) b) as well
{(f a b) + (g h)}       ⇒  (+ (f a b) (g h))
{f(a b) + g(h)}         ⇒  (+ (f a b) (g h)) as well
{f[a b] + g(h)}         ⇒  (+ ($bracket-apply$ f a b) (g h))
'{a + f(b) + x}         ⇒  '(+ a (f b) x)
{length(x) >= 6}        ⇒  (>= (length x) 6)
{n-1 + n-2}             ⇒  (+ n-1 n-2)
{n * factorial{n - 1}}  ⇒  (* n (factorial (- n 1)))
{{a > 0} and {b >= 1}}  ⇒  (and (> a 0) (>= b 1))
{f{n - 1}(x)}           ⇒  ((f (- n 1)) x)
{a . z}                 ⇒  ($nfx$ a . z)
{a + b - c}             ⇒  ($nfx$ a + b - c)
```

To enable curly-infix expressions within a file, place the reader directive `#!curly-infix` before the first use of curly-infix notation. To globally enable curly-infix expressions in Guile's reader, set the `curly-infix` read option.

Guile also implements the following non-standard extension to SRFI-105: if `curly-infix` is enabled and there is no other meaning assigned to square brackets

(i.e. the `square-brackets` read option is turned off), then lists within square brackets are read as normal lists but with the special symbol `$bracket-list$` added to the front. To enable this combination of read options within a file, use the reader directive `#!curly-infix-and-bracket-lists`. For example:

```
[a b]    ⇒  ($bracket-list$ a b)
[a . b]  ⇒  ($bracket-list$ a . b)
```

For more information on reader options, См. Раздел 6.18.2 [Scheme Read], страница 407.

7.5.44 SRFI-111 Boxes.

SRFI-111 (`http://srfi.schemers.org/srfi-111/srfi-111.html`) provides boxes: objects with a single mutable cell.

`box` *value* [Scheme Procedure]
Return a newly allocated box whose contents is initialized to *value*.

`box?` *obj* [Scheme Procedure]
Return true if *obj* is a box, otherwise return false.

`unbox` *box* [Scheme Procedure]
Return the current contents of *box*.

`set-box!` *box value* [Scheme Procedure]
Set the contents of *box* to *value*.

7.6 R6RS Support

См. Раздел 6.20.6 [R6RS Libraries], страница 439, for more information on how to define R6RS libraries, and their integration with Guile modules.

7.6.1 Incompatibilities with the R6RS

There are some incompatibilities between Guile and the R6RS. Some of them are intentional, some of them are bugs, and some are simply unimplemented features. Please let the Guile developers know if you find one that is not on this list.

- The R6RS specifies many situations in which a conforming implementation must signal a specific error. Guile doesn't really care about that too much—if a correct R6RS program would not hit that error, we don't bother checking for it.
- Multiple `library` forms in one file are not yet supported. This is because the expansion of `library` sets the current module, but does not restore it. This is a bug.
- R6RS unicode escapes within strings are disabled by default, because they conflict with Guile's already-existing escapes. The same is the case for R6RS treatment of escaped newlines in strings.

  R6RS behavior can be turned on via a reader option. См. Раздел 6.6.5.1 [String Syntax], страница 151, for more information.
- A `set!` to a variable transformer may only expand to an expression, not a definition—even if the original `set!` expression was in definition context.

- Instead of using the algorithm detailed in chapter 10 of the R6RS, expansion of toplevel forms happens sequentially.

  For example, while the expansion of the following set of toplevel definitions does the correct thing:

```
(begin
 (define even?
   (lambda (x)
     (or (= x 0) (odd? (- x 1)))))
 (define-syntax odd?
   (syntax-rules ()
     ((odd? x) (not (even? x)))))
 (even? 10))
⇒ #t
```

  The same definitions outside of the `begin` wrapper do not:

```
(define even?
  (lambda (x)
    (or (= x 0) (odd? (- x 1)))))
(define-syntax odd?
  (syntax-rules ()
    ((odd? x) (not (even? x)))))
(even? 10)
<unnamed port>:4:18: In procedure even?:
<unnamed port>:4:18: Wrong type to apply: #<syntax-transformer odd?>
```

  This is because when expanding the right-hand-side of `even?`, the reference to `odd?` is not yet marked as a syntax transformer, so it is assumed to be a function.

  This bug will only affect top-level programs, not code in `library` forms. Fixing it for toplevel forms seems doable, but tricky to implement in a backward-compatible way. Suggestions and/or patches would be appreciated.

- The `(rnrs io ports)` module is incomplete. Work is ongoing to fix this.
- Guile does not prevent use of textual I/O procedures on binary ports, or vice versa. All ports in Guile support both binary and textual I/O. См. Раздел 6.14.3 [Encoding], страница 355, for full details.
- Guile's implementation of `equal?` may fail to terminate when applied to arguments containing cycles.

7.6.2 R6RS Standard Libraries

In contrast with earlier versions of the Revised Report, the R6RS organizes the procedures and syntactic forms required of conforming implementations into a set of "standard libraries" which can be imported as necessary by user programs and libraries. Here we briefly list the libraries that have been implemented for Guile.

We do not attempt to document these libraries fully here, as most of their functionality is already available in Guile itself. The expectation is that most Guile users will use the well-known and well-documented Guile modules. These R6RS libraries are mostly useful to users who want to port their code to other R6RS systems.

The documentation in the following sections reproduces some of the content of the library section of the Report, but is mostly intended to provide supplementary information about Guile's implementation of the R6RS standard libraries. For complete documentation, design rationales and further examples, we advise you to consult the "Standard Libraries" section of the Report (см. Раздел "Standard Libraries" в *The Revised^6 Report on the Algorithmic Language Scheme*).

7.6.2.1 Library Usage

Guile implements the R6RS 'library' form as a transformation to a native Guile module definition. As a consequence of this, all of the libraries described in the following subsections, in addition to being available for use by R6RS libraries and top-level programs, can also be imported as if they were normal Guile modules—via a `use-modules` form, say. For example, the R6RS "composite" library can be imported by:

```
(import (rnrs (6)))
(use-modules ((rnrs) :version (6)))
```

For more information on Guile's library implementation, see (см. Раздел 6.20.6 [R6RS Libraries], страница 439).

7.6.2.2 rnrs base

The `(rnrs base (6))` library exports the procedures and syntactic forms described in the main section of the Report (см. Раздел "Base library" в *The Revised^6 Report on the Algorithmic Language Scheme*). They are grouped below by the existing manual sections to which they correspond.

`boolean?` *obj* [Scheme Procedure]
`not` *x* [Scheme Procedure]
См. Раздел 6.6.1 [Booleans], страница 112, for documentation.

`symbol?` *obj* [Scheme Procedure]
`symbol->string` *sym* [Scheme Procedure]
`string->symbol` *str* [Scheme Procedure]
См. Раздел 6.6.6.4 [Symbol Primitives], страница 177, for documentation.

`char?` *obj* [Scheme Procedure]
`char=?` [Scheme Procedure]
`char<?` [Scheme Procedure]
`char>?` [Scheme Procedure]
`char<=?` [Scheme Procedure]
`char>=?` [Scheme Procedure]
`integer->char` *n* [Scheme Procedure]
`char->integer` *chr* [Scheme Procedure]
См. Раздел 6.6.3 [Characters], страница 139, for documentation.

`list?` *x* [Scheme Procedure]
`null?` *x* [Scheme Procedure]
См. Раздел 6.6.9.2 [List Predicates], страница 193, for documentation.

`pair?` *x* [Scheme Procedure]
`cons` *x y* [Scheme Procedure]
`car` *pair* [Scheme Procedure]
`cdr` *pair* [Scheme Procedure]
`caar` *pair* [Scheme Procedure]
`cadr` *pair* [Scheme Procedure]
`cdar` *pair* [Scheme Procedure]
`cddr` *pair* [Scheme Procedure]
`caaar` *pair* [Scheme Procedure]
`caadr` *pair* [Scheme Procedure]
`cadar` *pair* [Scheme Procedure]
`cdaar` *pair* [Scheme Procedure]
`caddr` *pair* [Scheme Procedure]
`cdadr` *pair* [Scheme Procedure]
`cddar` *pair* [Scheme Procedure]
`cdddr` *pair* [Scheme Procedure]
`caaaar` *pair* [Scheme Procedure]
`caaadr` *pair* [Scheme Procedure]
`caadar` *pair* [Scheme Procedure]
`cadaar` *pair* [Scheme Procedure]
`cdaaar` *pair* [Scheme Procedure]
`cddaar` *pair* [Scheme Procedure]
`cdadar` *pair* [Scheme Procedure]
`cdaadr` *pair* [Scheme Procedure]
`cadadr` *pair* [Scheme Procedure]
`caaddr` *pair* [Scheme Procedure]
`caddar` *pair* [Scheme Procedure]
`cadddr` *pair* [Scheme Procedure]
`cdaddr` *pair* [Scheme Procedure]
`cddadr` *pair* [Scheme Procedure]
`cdddar` *pair* [Scheme Procedure]
`cddddr` *pair* [Scheme Procedure]

См. Раздел 6.6.8 [Pairs], страница 189, for documentation.

`number?` *obj* [Scheme Procedure]

См. Раздел 6.6.2.1 [Numerical Tower], страница 114, for documentation.

`string?` *obj* [Scheme Procedure]

См. Раздел 6.6.5.2 [String Predicates], страница 153, for documentation.

`procedure?` *obj* [Scheme Procedure]

См. Раздел 6.9.7 [Procedure Properties], страница 273, for documentation.

`define` *name value* [Scheme Syntax]
`set!` *variable-name value* [Scheme Syntax]

См. Раздел 3.1.3 [Definition], страница 16, for documentation.

`define-syntax` *keyword expression* [Scheme Syntax]
`let-syntax` ((*keyword transformer*) ...) *exp1 exp2* ... [Scheme Syntax]

`letrec-syntax` ((*keyword transformer*) ...) *exp1 exp2 ...* [Scheme Syntax]

См. Раздел 6.10.1 [Defining Macros], страница 277, for documentation.

`identifier-syntax` *exp* [Scheme Syntax]

См. Раздел 6.10.6 [Identifier Macros], страница 293, for documentation.

`syntax-rules` *literals* (*pattern template*) ... [Scheme Syntax]

См. Раздел 6.10.2 [Syntax Rules], страница 278, for documentation.

`lambda` *formals body* [Scheme Syntax]

См. Раздел 6.9.1 [Lambda], страница 262, for documentation.

`let` *bindings body* [Scheme Syntax]
`let*` *bindings body* [Scheme Syntax]
`letrec` *bindings body* [Scheme Syntax]
`letrec*` *bindings body* [Scheme Syntax]

См. Раздел 6.12.2 [Local Bindings], страница 314, for documentation.

`let-values` *bindings body* [Scheme Syntax]
`let*-values` *bindings body* [Scheme Syntax]

См. Раздел 7.5.10 [SRFI-11], страница 631, for documentation.

`begin` *expr1 expr2 ...* [Scheme Syntax]

См. Раздел 6.13.1 [begin], страница 317, for documentation.

`quote` *expr* [Scheme Syntax]
`quasiquote` *expr* [Scheme Syntax]
`unquote` *expr* [Scheme Syntax]
`unquote-splicing` *expr* [Scheme Syntax]

См. Раздел 6.18.1.1 [Expression Syntax], страница 404, for documentation.

`if` *test consequence* [*alternate*] [Scheme Syntax]
`cond` *clause1 clause2 ...* [Scheme Syntax]
`case` *key clause1 clause2 ...* [Scheme Syntax]

См. Раздел 6.13.2 [Conditionals], страница 318, for documentation.

`and` *expr ...* [Scheme Syntax]
`or` *expr ...* [Scheme Syntax]

См. Раздел 6.13.3 [and or], страница 320, for documentation.

`eq?` *x y* [Scheme Procedure]
`eqv?` *x y* [Scheme Procedure]
`equal?` *x y* [Scheme Procedure]
`symbol=?` *symbol1 symbol2 ...* [Scheme Procedure]

См. Раздел 6.11.1 [Equality], страница 301, for documentation.

`symbol=?` is identical to `eq?`.

`complex?` *z* [Scheme Procedure]

См. Раздел 6.6.2.4 [Complex Numbers], страница 122, for documentation.

`real-part` *z* [Scheme Procedure]
`imag-part` *z* [Scheme Procedure]
`make-rectangular` *real_part imaginary_part* [Scheme Procedure]
`make-polar` *x y* [Scheme Procedure]
`magnitude` *z* [Scheme Procedure]
`angle` *z* [Scheme Procedure]

См. Раздел 6.6.2.10 [Complex], страница 127, for documentation.

`sqrt` *z* [Scheme Procedure]
`exp` *z* [Scheme Procedure]
`expt` *z1 z2* [Scheme Procedure]
`log` *z* [Scheme Procedure]
`sin` *z* [Scheme Procedure]
`cos` *z* [Scheme Procedure]
`tan` *z* [Scheme Procedure]
`asin` *z* [Scheme Procedure]
`acos` *z* [Scheme Procedure]
`atan` *z* [Scheme Procedure]

См. Раздел 6.6.2.12 [Scientific], страница 133, for documentation.

`real?` *x* [Scheme Procedure]
`rational?` *x* [Scheme Procedure]
`numerator` *x* [Scheme Procedure]
`denominator` *x* [Scheme Procedure]
`rationalize` *x eps* [Scheme Procedure]

См. Раздел 6.6.2.3 [Reals and Rationals], страница 119, for documentation.

`exact?` *x* [Scheme Procedure]
`inexact?` *x* [Scheme Procedure]
`exact` *z* [Scheme Procedure]
`inexact` *z* [Scheme Procedure]

См. Раздел 6.6.2.5 [Exactness], страница 122, for documentation. The `exact` and `inexact` procedures are identical to the `inexact->exact` and `exact->inexact` procedures provided by Guile's code library.

`integer?` *x* [Scheme Procedure]

См. Раздел 6.6.2.2 [Integers], страница 115, for documentation.

`odd?` *n* [Scheme Procedure]
`even?` *n* [Scheme Procedure]
`gcd` *x ...* [Scheme Procedure]
`lcm` *x ...* [Scheme Procedure]
`exact-integer-sqrt` *k* [Scheme Procedure]

См. Раздел 6.6.2.7 [Integer Operations], страница 125, for documentation.

`=` [Scheme Procedure]
`<` [Scheme Procedure]
`>` [Scheme Procedure]
`<=` [Scheme Procedure]

`>=` [Scheme Procedure]
`zero?` *x* [Scheme Procedure]
`positive?` *x* [Scheme Procedure]
`negative?` *x* [Scheme Procedure]
См. Раздел 6.6.2.8 [Comparison], страница 126, for documentation.

`for-each` *f lst1 lst2 ...* [Scheme Procedure]
См. Раздел 7.5.3.5 [SRFI-1 Fold and Map], страница 611, for documentation.

`list` *elem ...* [Scheme Procedure]
См. Раздел 6.6.9.3 [List Constructors], страница 193, for documentation.

`length` *lst* [Scheme Procedure]
`list-ref` *lst k* [Scheme Procedure]
`list-tail` *lst k* [Scheme Procedure]
См. Раздел 6.6.9.4 [List Selection], страница 194, for documentation.

`append` *lst ... obj* [Scheme Procedure]
`append` [Scheme Procedure]
`reverse` *lst* [Scheme Procedure]
См. Раздел 6.6.9.5 [Append/Reverse], страница 194, for documentation.

`number->string` *n* [*radix*] [Scheme Procedure]
`string->number` *str* [*radix*] [Scheme Procedure]
См. Раздел 6.6.2.9 [Conversion], страница 127, for documentation.

`string` *char ...* [Scheme Procedure]
`make-string` *k* [*chr*] [Scheme Procedure]
`list->string` *lst* [Scheme Procedure]
См. Раздел 6.6.5.3 [String Constructors], страница 153, for documentation.

`string->list` *str* [*start* [*end*]] [Scheme Procedure]
См. Раздел 6.6.5.4 [List/String Conversion], страница 154, for documentation.

`string-length` *str* [Scheme Procedure]
`string-ref` *str k* [Scheme Procedure]
`string-copy` *str* [*start* [*end*]] [Scheme Procedure]
`substring` *str start* [*end*] [Scheme Procedure]
См. Раздел 6.6.5.5 [String Selection], страница 155, for documentation.

`string=?` *s1 s2 s3 ...* [Scheme Procedure]
`string<?` *s1 s2 s3 ...* [Scheme Procedure]
`string>?` *s1 s2 s3 ...* [Scheme Procedure]
`string<=?` *s1 s2 s3 ...* [Scheme Procedure]
`string>=?` *s1 s2 s3 ...* [Scheme Procedure]
См. Раздел 6.6.5.7 [String Comparison], страница 158, for documentation.

`string-append` *arg ...* [Scheme Procedure]
См. Раздел 6.6.5.10 [Reversing and Appending Strings], страница 165, for documentation.

`string-for-each` *proc s* [*start* [*end*]] [Scheme Procedure]

См. Раздел 6.6.5.11 [Mapping Folding and Unfolding], страница 166, for documentation.

`+` *z1 ...* [Scheme Procedure]
`-` *z1 z2 ...* [Scheme Procedure]
`*` *z1 ...* [Scheme Procedure]
`/` *z1 z2 ...* [Scheme Procedure]
`max` *x1 x2 ...* [Scheme Procedure]
`min` *x1 x2 ...* [Scheme Procedure]
`abs` *x* [Scheme Procedure]
`truncate` *x* [Scheme Procedure]
`floor` *x* [Scheme Procedure]
`ceiling` *x* [Scheme Procedure]
`round` *x* [Scheme Procedure]

См. Раздел 6.6.2.11 [Arithmetic], страница 128, for documentation.

`div` *x y* [Scheme Procedure]
`mod` *x y* [Scheme Procedure]
`div-and-mod` *x y* [Scheme Procedure]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `div` returns the integer q and `mod` returns the real number r such that $x = q * y + r$ and $0 <= r < abs(y)$. `div-and-mod` returns both q and r, and is more efficient than computing each separately. Note that when $y > 0$, `div` returns $floor(x/y)$, otherwise it returns $ceiling(x/y)$.

```
(div 123 10) ⇒ 12
(mod 123 10) ⇒ 3
(div-and-mod 123 10) ⇒ 12 and 3
(div-and-mod 123 -10) ⇒ -12 and 3
(div-and-mod -123 10) ⇒ -13 and 7
(div-and-mod -123 -10) ⇒ 13 and 7
(div-and-mod -123.2 -63.5) ⇒ 2.0 and 3.8
(div-and-mod 16/3 -10/7) ⇒ -3 and 22/21
```

`div0` *x y* [Scheme Procedure]
`mod0` *x y* [Scheme Procedure]
`div0-and-mod0` *x y* [Scheme Procedure]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `div0` returns the integer q and `mod0` returns the real number r such that $x = q * y + r$ and $-abs(y/2) <= r < abs(y/2)$. `div0-and-mod0` returns both q and r, and is more efficient than computing each separately.

Note that `div0` returns x/y rounded to the nearest integer. When x/y lies exactly half-way between two integers, the tie is broken according to the sign of y. If $y > 0$, ties are rounded toward positive infinity, otherwise they are rounded toward negative infinity. This is a consequence of the requirement that $-abs(y/2) <= r < abs(y/2)$.

```
(div0 123 10) ⇒ 12
(mod0 123 10) ⇒ 3
```

```
(div0-and-mod0 123 10) ⇒ 12 and 3
(div0-and-mod0 123 -10) ⇒ -12 and 3
(div0-and-mod0 -123 10) ⇒ -12 and -3
(div0-and-mod0 -123 -10) ⇒ 12 and -3
(div0-and-mod0 -123.2 -63.5) ⇒ 2.0 and 3.8
(div0-and-mod0 16/3 -10/7) ⇒ -4 and -8/21
```

`real-valued?` *obj* [Scheme Procedure]
`rational-valued?` *obj* [Scheme Procedure]
`integer-valued?` *obj* [Scheme Procedure]

These procedures return `#t` if and only if their arguments can, respectively, be coerced to a real, rational, or integer value without a loss of numerical precision.

`real-valued?` will return `#t` for complex numbers whose imaginary parts are zero.

`nan?` *x* [Scheme Procedure]
`infinite?` *x* [Scheme Procedure]
`finite?` *x* [Scheme Procedure]

`nan?` returns `#t` if *x* is a NaN value, `#f` otherwise. `infinite?` returns `#t` if *x* is an infinite value, `#f` otherwise. `finite?` returns `#t` if *x* is neither infinite nor a NaN value, otherwise it returns `#f`. Every real number satisfies exactly one of these predicates. An exception is raised if *x* is not real.

`assert` *expr* [Scheme Syntax]

Raises an `&assertion` condition if *expr* evaluates to `#f`; otherwise evaluates to the value of *expr*.

`error` *who message irritant1 ...* [Scheme Procedure]
`assertion-violation` *who message irritant1 ...* [Scheme Procedure]

These procedures raise compound conditions based on their arguments: If *who* is not `#f`, the condition will include a `&who` condition whose `who` field is set to *who*; a `&message` condition will be included with a `message` field equal to *message*; an `&irritants` condition will be included with its `irritants` list given by `irritant1 ...`.

`error` produces a compound condition with the simple conditions described above, as well as an `&error` condition; `assertion-violation` produces one that includes an `&assertion` condition.

`vector-map` *proc v* [Scheme Procedure]
`vector-for-each` *proc v* [Scheme Procedure]

These procedures implement the `map` and `for-each` contracts over vectors.

`vector` *arg ...* [Scheme Procedure]
`vector?` *obj* [Scheme Procedure]
`make-vector` *len* [Scheme Procedure]
`make-vector` *len fill* [Scheme Procedure]
`list->vector` *l* [Scheme Procedure]
`vector->list` *v* [Scheme Procedure]

См. Раздел 6.6.10.2 [Vector Creation], страница 198, for documentation.

`vector-length` *vector* [Scheme Procedure]
`vector-ref` *vector k* [Scheme Procedure]
`vector-set!` *vector k obj* [Scheme Procedure]
`vector-fill!` *v fill* [Scheme Procedure]
См. Раздел 6.6.10.3 [Vector Accessors], страница 199, for documentation.

`call-with-current-continuation` *proc* [Scheme Procedure]
`call/cc` *proc* [Scheme Procedure]
См. Раздел 6.13.6 [Continuations], страница 328, for documentation.

`values` *arg ...* [Scheme Procedure]
`call-with-values` *producer consumer* [Scheme Procedure]
См. Раздел 6.13.7 [Multiple Values], страница 330, for documentation.

`dynamic-wind` *in_guard thunk out_guard* [Scheme Procedure]
См. Раздел 6.13.10 [Dynamic Wind], страница 339, for documentation.

`apply` *proc arg ... arglst* [Scheme Procedure]
См. Раздел 6.18.4 [Fly Evaluation], страница 409, for documentation.

7.6.2.3 rnrs unicode

The `(rnrs unicode (6))` library provides procedures for manipulating Unicode characters and strings.

`char-upcase` *char* [Scheme Procedure]
`char-downcase` *char* [Scheme Procedure]
`char-titlecase` *char* [Scheme Procedure]
`char-foldcase` *char* [Scheme Procedure]
These procedures translate their arguments from one Unicode character set to another. `char-upcase`, `char-downcase`, and `char-titlecase` are identical to their counterparts in the Guile core library; См. Раздел 6.6.3 [Characters], страница 139, for documentation.

`char-foldcase` returns the result of applying `char-upcase` to its argument, followed by `char-downcase`—except in the case of the Turkic characters `U+0130` and `U+0131`, for which the procedure acts as the identity function.

`char-ci=?` *char1 char2 char3 ...* [Scheme Procedure]
`char-ci<?` *char1 char2 char3 ...* [Scheme Procedure]
`char-ci>?` *char1 char2 char3 ...* [Scheme Procedure]
`char-ci<=?` *char1 char2 char3 ...* [Scheme Procedure]
`char-ci>=?` *char1 char2 char3 ...* [Scheme Procedure]
These procedures facilitate case-insensitive comparison of Unicode characters. They are identical to the procedures provided by Guile's core library. См. Раздел 6.6.3 [Characters], страница 139, for documentation.

`char-alphabetic?` *char* [Scheme Procedure]
`char-numeric?` *char* [Scheme Procedure]
`char-whitespace?` *char* [Scheme Procedure]
`char-upper-case?` *char* [Scheme Procedure]

`char-lower-case?` *char* [Scheme Procedure]
`char-title-case?` *char* [Scheme Procedure]
These procedures implement various Unicode character set predicates. They are identical to the procedures provided by Guile's core library. См. Раздел 6.6.3 [Characters], страница 139, for documentation.

`char-general-category` *char* [Scheme Procedure]
См. Раздел 6.6.3 [Characters], страница 139, for documentation.

`string-upcase` *string* [Scheme Procedure]
`string-downcase` *string* [Scheme Procedure]
`string-titlecase` *string* [Scheme Procedure]
`string-foldcase` *string* [Scheme Procedure]
These procedures perform Unicode case folding operations on their input. См. Раздел 6.6.5.9 [Alphabetic Case Mapping], страница 164, for documentation.

`string-ci=?` *string1 string2 string3 ...* [Scheme Procedure]
`string-ci<?` *string1 string2 string3 ...* [Scheme Procedure]
`string-ci>?` *string1 string2 string3 ...* [Scheme Procedure]
`string-ci<=?` *string1 string2 string3 ...* [Scheme Procedure]
`string-ci>=?` *string1 string2 string3 ...* [Scheme Procedure]
These procedures perform case-insensitive comparison on their input. См. Раздел 6.6.5.7 [String Comparison], страница 158, for documentation.

`string-normalize-nfd` *string* [Scheme Procedure]
`string-normalize-nfkd` *string* [Scheme Procedure]
`string-normalize-nfc` *string* [Scheme Procedure]
`string-normalize-nfkc` *string* [Scheme Procedure]
These procedures perform Unicode string normalization operations on their input. См. Раздел 6.6.5.7 [String Comparison], страница 158, for documentation.

7.6.2.4 rnrs bytevectors

The `(rnrs bytevectors (6))` library provides procedures for working with blocks of binary data. This functionality is documented in its own section of the manual; См. Раздел 6.6.12 [Bytevectors], страница 205.

7.6.2.5 rnrs lists

The `(rnrs lists (6))` library provides procedures additional procedures for working with lists.

`find` *proc list* [Scheme Procedure]
This procedure is identical to the one defined in Guile's SRFI-1 implementation. См. Раздел 7.5.3.7 [SRFI-1 Searching], страница 614, for documentation.

`for-all` *proc list1 list2 ...* [Scheme Procedure]
`exists` *proc list1 list2 ...* [Scheme Procedure]
The `for-all` procedure is identical to the `every` procedure defined by SRFI-1; the `exists` procedure is identical to SRFI-1's `any`. См. Раздел 7.5.3.7 [SRFI-1 Searching], страница 614, for documentation.

`filter` *proc list* [Scheme Procedure]
`partition` *proc list* [Scheme Procedure]
These procedures are identical to the ones provided by SRFI-1. См. Раздел 6.6.9.6 [List Modification], страница 195, for a description of `filter`; См. Раздел 7.5.3.6 [SRFI-1 Filtering and Partitioning], страница 614, for `partition`.

`fold-right` *combine nil list1 list2 ...* [Scheme Procedure]
This procedure is identical the `fold-right` procedure provided by SRFI-1. См. Раздел 7.5.3.5 [SRFI-1 Fold and Map], страница 611, for documentation.

`fold-left` *combine nil list1 list2 ...* [Scheme Procedure]
This procedure is like `fold` from SRFI-1, but *combine* is called with the seed as the first argument. См. Раздел 7.5.3.5 [SRFI-1 Fold and Map], страница 611, for documentation.

`remp` *proc list* [Scheme Procedure]
`remove` *obj list* [Scheme Procedure]
`remv` *obj list* [Scheme Procedure]
`remq` *obj list* [Scheme Procedure]
`remove`, `remv`, and `remq` are identical to the `delete`, `delv`, and `delq` procedures provided by Guile's core library, (см. Раздел 6.6.9.6 [List Modification], страница 195). `remp` is identical to the alternate `remove` procedure provided by SRFI-1; См. Раздел 7.5.3.8 [SRFI-1 Deleting], страница 616.

`memp` *proc list* [Scheme Procedure]
`member` *obj list* [Scheme Procedure]
`memv` *obj list* [Scheme Procedure]
`memq` *obj list* [Scheme Procedure]
`member`, `memv`, and `memq` are identical to the procedures provided by Guile's core library; См. Раздел 6.6.9.7 [List Searching], страница 197, for their documentation. `memp` uses the specified predicate function `proc` to test elements of the list *list*—it behaves similarly to `find`, except that it returns the first sublist of *list* whose `car` satisfies *proc*.

`assp` *proc alist* [Scheme Procedure]
`assoc` *obj alist* [Scheme Procedure]
`assv` *obj alist* [Scheme Procedure]
`assq` *obj alist* [Scheme Procedure]
`assoc`, `assv`, and `assq` are identical to the procedures provided by Guile's core library; См. Раздел 6.6.20.1 [Alist Key Equality], страница 243, for their documentation. `assp` uses the specified predicate function `proc` to test keys in the association list *alist*.

`cons*` *obj1 ... obj* [Scheme Procedure]
`cons*` *obj* [Scheme Procedure]
This procedure is identical to the one exported by Guile's core library. См. Раздел 6.6.9.3 [List Constructors], страница 193, for documentation.

7.6.2.6 rnrs sorting

The `(rnrs sorting (6))` library provides procedures for sorting lists and vectors.

`list-sort` *proc list* [Scheme Procedure]
`vector-sort` *proc vector* [Scheme Procedure]

These procedures return their input sorted in ascending order, without modifying the original data. *proc* must be a procedure that takes two elements from the input list or vector as arguments, and returns a true value if the first is "less" than the second, `#f` otherwise. `list-sort` returns a list; `vector-sort` returns a vector.

Both `list-sort` and `vector-sort` are implemented in terms of the `stable-sort` procedure from Guile's core library. См. Раздел 6.11.3 [Sorting], страница 304, for a discussion of the behavior of that procedure.

`vector-sort!` *proc vector* [Scheme Procedure]

Performs a destructive, "in-place" sort of *vector*, using *proc* as described above to determine an ascending ordering of elements. `vector-sort!` returns an unspecified value.

This procedure is implemented in terms of the `sort!` procedure from Guile's core library. См. Раздел 6.11.3 [Sorting], страница 304, for more information.

7.6.2.7 rnrs control

The `(rnrs control (6))` library provides syntactic forms useful for constructing conditional expressions and controlling the flow of execution.

`when` *test expression1 expression2 ...* [Scheme Syntax]
`unless` *test expression1 expression2 ...* [Scheme Syntax]

The `when` form is evaluated by evaluating the specified *test* expression; if the result is a true value, the *expression*s that follow it are evaluated in order, and the value of the final *expression* becomes the value of the entire `when` expression.

The `unless` form behaves similarly, with the exception that the specified *expression*s are only evaluated if the value of *test* is false.

`do` ((*variable init step*) ...) (*test expression ...*) *command ...* [Scheme Syntax]

This form is identical to the one provided by Guile's core library. См. Раздел 6.13.4 [while do], страница 321, for documentation.

`case-lambda` *clause ...* [Scheme Syntax]

This form is identical to the one provided by Guile's core library. См. Раздел 6.9.5 [Case-lambda], страница 270, for documentation.

7.6.2.8 R6RS Records

The manual sections below describe Guile's implementation of R6RS records, which provide support for user-defined data types. The R6RS records API provides a superset of the features provided by Guile's "native" records, as well as those of the SRFI-9 records API; См. Раздел 6.6.17 [Records], страница 234, and Раздел 6.6.16 [SRFI-9 Records], страница 231, for a description of those interfaces.

As with SRFI-9 and Guile's native records, R6RS records are constructed using a record-type descriptor that specifies attributes like the record's name, its fields, and the mutability of those fields.

R6RS records extend this framework to support single inheritance via the specification of a "parent" type for a record type at definition time. Accessors and mutator procedures for the fields of a parent type may be applied to records of a subtype of this parent. A record type may be *sealed*, in which case it cannot be used as the parent of another record type.

The inheritance mechanism for record types also informs the process of initializing the fields of a record and its parents. Constructor procedures that generate new instances of a record type are obtained from a record constructor descriptor, which encapsulates the record-type descriptor of the record to be constructed along with a *protocol* procedure that defines how constructors for record subtypes delegate to the constructors of their parent types.

A protocol is a procedure used by the record system at construction time to bind arguments to the fields of the record being constructed. The protocol procedure is passed a procedure *n* that accepts the arguments required to construct the record's parent type; this procedure, when invoked, will return a procedure *p* that accepts the arguments required to construct a new instance of the record type itself and returns a new instance of the record type.

The protocol should in turn return a procedure that uses *n* and *p* to initialize the fields of the record type and its parent type(s). This procedure will be the constructor returned by

As a trivial example, consider the hypothetical record type `pixel`, which encapsulates an x-y location on a screen, and `voxel`, which has `pixel` as its parent type and stores an additional coordinate. The following protocol produces a constructor procedure that accepts all three coordinates, uses the first two to initialize the fields of `pixel`, and binds the third to the single field of `voxel`.

```
(lambda (n)
  (lambda (x y z)
    (let ((p (n x y)))
      (p z))))
```

It may be helpful to think of protocols as "constructor factories" that produce chains of delegating constructors glued together by the helper procedure *n*.

An R6RS record type may be declared to be *nongenerative* via the use of a unique generated or user-supplied symbol—or *uid*—such that subsequent record type declarations with the same uid and attributes will return the previously-declared record-type descriptor.

R6RS record types may also be declared to be *opaque*, in which case the various predicates and introspection procedures defined in `(rnrs records introspection)` will behave as if records of this type are not records at all.

Note that while the R6RS records API shares much of its namespace with both the SRFI-9 and native Guile records APIs, it is not currently compatible with either.

7.6.2.9 rnrs records syntactic

The `(rnrs records syntactic (6))` library exports the syntactic API for working with R6RS records.

`define-record-type` *name-spec record-clause ...* [Scheme Syntax]

Defines a new record type, introducing bindings for a record-type descriptor, a record constructor descriptor, a constructor procedure, a record predicate, and accessor and mutator procedures for the new record type's fields.

name-spec must either be an identifier or must take the form `(record-name constructor-name predicate-name)`, where *record-name*, *constructor-name*, and *predicate-name* are all identifiers and specify the names to which, respectively, the record-type descriptor, constructor, and predicate procedures will be bound. If *name-spec* is only an identifier, it specifies the name to which the generated record-type descriptor will be bound.

Each *record-clause* must be one of the following:

- `(fields field-spec*)`, where each *field-spec* specifies a field of the new record type and takes one of the following forms:
  - `(immutable field-name accessor-name)`, which specifies an immutable field with the name *field-name* and binds an accessor procedure for it to the name given by *accessor-name*
  - `(mutable field-name accessor-name mutator-name)`, which specifies a mutable field with the name *field-name* and binds accessor and mutator procedures to *accessor-name* and *mutator-name*, respectively
  - `(immutable field-name)`, which specifies an immutable field with the name *field-name*; an accessor procedure for it will be created and named by appending record name and *field-name* with a hyphen separator
  - `(mutable field-name)`, which specifies a mutable field with the name *field-name*; an accessor procedure for it will be created and named as described above; a mutator procedure will also be created and named by appending `-set!` to the accessor name
  - `field-name`, which specifies an immutable field with the name *field-name*; an access procedure for it will be created and named as described above
- `(parent parent-name)`, where *parent-name* is a symbol giving the name of the record type to be used as the parent of the new record type
- `(protocol expression)`, where *expression* evaluates to a protocol procedure which behaves as described above, and is used to create a record constructor descriptor for the new record type
- `(sealed sealed?)`, where *sealed?* is a boolean value that specifies whether or not the new record type is sealed
- `(opaque opaque?)`, where *opaque?* is a boolean value that specifies whether or not the new record type is opaque
- `(nongenerative [uid])`, which specifies that the record type is nongenerative via the optional uid *uid*. If *uid* is not specified, a unique uid will be generated at expansion time

- `(parent-rtd parent-rtd parent-cd)`, a more explicit form of the `parent` form above; *parent-rtd* and *parent-cd* should evaluate to a record-type descriptor and a record constructor descriptor, respectively

`record-type-descriptor` *record-name* [Scheme Syntax]
Evaluates to the record-type descriptor associated with the type specified by *record-name*.

`record-constructor-descriptor` *record-name* [Scheme Syntax]
Evaluates to the record-constructor descriptor associated with the type specified by *record-name*.

7.6.2.10 rnrs records procedural

The `(rnrs records procedural (6))` library exports the procedural API for working with R6RS records.

`make-record-type-descriptor` *name parent uid sealed? opaque? fields* [Scheme Procedure]
Returns a new record-type descriptor with the specified characteristics: *name* must be a symbol giving the name of the new record type; *parent* must be either `#f` or a non-sealed record-type descriptor for the returned record type to extend; *uid* must be either `#f`, indicating that the record type is generative, or a symbol giving the type's nongenerative uid; *sealed?* and *opaque?* must be boolean values that specify the sealedness and opaqueness of the record type; *fields* must be a vector of zero or more field specifiers of the form `(mutable name)` or `(immutable name)`, where name is a symbol giving a name for the field.

If *uid* is not `#f`, it must be a symbol

`record-type-descriptor?` *obj* [Scheme Procedure]
Returns `#t` if *obj* is a record-type descriptor, `#f` otherwise.

`make-record-constructor-descriptor` *rtd parent-constructor-descriptor protocol* [Scheme Procedure]
Returns a new record constructor descriptor that can be used to produce constructors for the record type specified by the record-type descriptor *rtd* and whose delegation and binding behavior are specified by the protocol procedure *protocol*.

parent-constructor-descriptor specifies a record constructor descriptor for the parent type of *rtd*, if one exists. If *rtd* represents a base type, then *parent-constructor-descriptor* must be `#f`. If *rtd* is an extension of another type, *parent-constructor-descriptor* may still be `#f`, but protocol must also be `#f` in this case.

`record-constructor` *rcd* [Scheme Procedure]
Returns a record constructor procedure by invoking the protocol defined by the record-constructor descriptor *rcd*.

`record-predicate` *rtd* [Scheme Procedure]
Returns the record predicate procedure for the record-type descriptor *rtd*.

`record-accessor` *rtd k* [Scheme Procedure]
Returns the record field accessor procedure for the *k*th field of the record-type descriptor *rtd*.

`record-mutator` *rtd k* [Scheme Procedure]
Returns the record field mutator procedure for the *k*th field of the record-type descriptor *rtd*. An `&assertion` condition will be raised if this field is not mutable.

7.6.2.11 rnrs records inspection

The `(rnrs records inspection (6))` library provides procedures useful for accessing metadata about R6RS records.

`record?` *obj* [Scheme Procedure]
Return `#t` if the specified object is a non-opaque R6RS record, `#f` otherwise.

`record-rtd` *record* [Scheme Procedure]
Returns the record-type descriptor for *record*. An `&assertion` is raised if *record* is opaque.

`record-type-name` *rtd* [Scheme Procedure]
Returns the name of the record-type descriptor *rtd*.

`record-type-parent` *rtd* [Scheme Procedure]
Returns the parent of the record-type descriptor *rtd*, or `#f` if it has none.

`record-type-uid` *rtd* [Scheme Procedure]
Returns the uid of the record-type descriptor *rtd*, or `#f` if it has none.

`record-type-generative?` *rtd* [Scheme Procedure]
Returns `#t` if the record-type descriptor *rtd* is generative, `#f` otherwise.

`record-type-sealed?` *rtd* [Scheme Procedure]
Returns `#t` if the record-type descriptor *rtd* is sealed, `#f` otherwise.

`record-type-opaque?` *rtd* [Scheme Procedure]
Returns `#t` if the record-type descriptor *rtd* is opaque, `#f` otherwise.

`record-type-field-names` *rtd* [Scheme Procedure]
Returns a vector of symbols giving the names of the fields defined by the record-type descriptor *rtd* (and not any of its sub- or supertypes).

`record-field-mutable?` *rtd k* [Scheme Procedure]
Returns `#t` if the field at index *k* of the record-type descriptor *rtd* (and not any of its sub- or supertypes) is mutable.

7.6.2.12 rnrs exceptions

The `(rnrs exceptions (6))` library provides functionality related to signaling and handling exceptional situations. This functionality is similar to the exception handling systems provided by Guile's core library См. Раздел 6.13.8 [Exceptions], страница 332, and by the SRFI-18 and SRFI-34 modules—См. Раздел 7.5.15.5 [SRFI-18 Exceptions],

страница 636, and Раздел 7.5.23 [SRFI-34], страница 648, respectively—but there are some key differences in concepts and behavior.

A raised exception may be *continuable* or *non-continuable*. When an exception is raised non-continuably, another exception, with the condition type `&non-continuable`, will be raised when the exception handler returns locally. Raising an exception continuably captures the current continuation and invokes it after a local return from the exception handler.

Like SRFI-18 and SRFI-34, R6RS exceptions are implemented on top of Guile's native `throw` and `catch` forms, and use custom "throw keys" to identify their exception types. As a consequence, Guile's `catch` form can handle exceptions thrown by these APIs, but the reverse is not true: Handlers registered by the `with-exception-handler` procedure described below will only be called on exceptions thrown by the corresponding `raise` procedure.

`with-exception-handler` *handler thunk* [Scheme Procedure]

Installs *handler*, which must be a procedure taking one argument, as the current exception handler during the invocation of *thunk*, a procedure taking zero arguments. The handler in place at the time `with-exception-handler` is called is made current again once either *thunk* returns or *handler* is invoked after an exception is thrown from within *thunk*.

This procedure is similar to the `with-throw-handler` procedure provided by Guile's code library; (см. Раздел 6.13.8.3 [Throw Handlers], страница 335).

`guard` (*variable clause1 clause2 ...*) *body* [Scheme Syntax]

Evaluates the expression given by *body*, first creating an ad hoc exception handler that binds a raised exception to *variable* and then evaluates the specified *clause*s as if they were part of a `cond` expression, with the value of the first matching clause becoming the value of the `guard` expression (см. Раздел 6.13.2 [Conditionals], страница 318). If none of the clause's test expressions evaluates to `#t`, the exception is re-raised, with the exception handler that was current before the evaluation of the `guard` form.

For example, the expression

```
(guard (ex ((eq? ex 'foo) 'bar) ((eq? ex 'bar) 'baz))
  (raise 'bar))
```

evaluates to `baz`.

`raise` *obj* [Scheme Procedure]

Raises a non-continuable exception by invoking the currently-installed exception handler on *obj*. If the handler returns, a `&non-continuable` exception will be raised in the dynamic context in which the handler was installed.

`raise-continuable` *obj* [Scheme Procedure]

Raises a continuable exception by invoking currently-installed exception handler on *obj*.

7.6.2.13 rnrs conditions

The `(rnrs condition (6))` library provides forms and procedures for constructing new condition types, as well as a library of pre-defined condition types that represent a variety of common exceptional situations. Conditions are records of a subtype of the `&condition` record type, which is neither sealed nor opaque. См. Раздел 7.6.2.8 [R6RS Records], страница 689.

Conditions may be manipulated singly, as *simple conditions*, or when composed with other conditions to form *compound conditions*. Compound conditions do not "nest"—constructing a new compound condition out of existing compound conditions will "flatten" them into their component simple conditions. For example, making a new condition out of a `&message` condition and a compound condition that contains an `&assertion` condition and another `&message` condition will produce a compound condition that contains two `&message` conditions and one `&assertion` condition.

The record type predicates and field accessors described below can operate on either simple or compound conditions. In the latter case, the predicate returns `#t` if the compound condition contains a component simple condition of the appropriate type; the field accessors return the requisite fields from the first component simple condition found to be of the appropriate type.

This library is quite similar to the SRFI-35 conditions module (см. Раздел 7.5.24 [SRFI-35], страница 648). Among other minor differences, the `(rnrs conditions)` library features slightly different semantics around condition field accessors, and comes with a larger number of pre-defined condition types. The two APIs are not currently compatible, however; the `condition?` predicate from one API will return `#f` when applied to a condition object created in the other.

`&condition` [Condition Type]
`condition?` *obj* [Scheme Procedure]

The base record type for conditions.

`condition` *condition1 ...* [Scheme Procedure]
`simple-conditions` *condition* [Scheme Procedure]

The `condition` procedure creates a new compound condition out of its condition arguments, flattening any specified compound conditions into their component simple conditions as described above.

`simple-conditions` returns a list of the component simple conditions of the compound condition `condition`, in the order in which they were specified at construction time.

`condition-predicate` *rtd* [Scheme Procedure]
`condition-accessor` *rtd proc* [Scheme Procedure]

These procedures return condition predicate and accessor procedures for the specified condition record type *rtd*.

`define-condition-type` *condition-type supertype constructor predicate field-spec ...* [Scheme Syntax]

Evaluates to a new record type definition for a condition type with the name *condition-type* that has the condition type *supertype* as its parent. A default

constructor, which binds its arguments to the fields of this type and its parent types, will be bound to the identifier *constructor*; a condition predicate will be bound to *predicate*. The fields of the new type, which are immutable, are specified by the *field-spec*s, each of which must be of the form:

```
(field accessor)
```

where *field* gives the name of the field and *accessor* gives the name for a binding to an accessor procedure created for this field.

`&message` [Condition Type]
`make-message-condition` *message* [Scheme Procedure]
`message-condition?` *obj* [Scheme Procedure]
`condition-message` *condition* [Scheme Procedure]

A type that includes a message describing the condition that occurred.

`&warning` [Condition Type]
`make-warning` [Scheme Procedure]
`warning?` *obj* [Scheme Procedure]

A base type for representing non-fatal conditions during execution.

`&serious` [Condition Type]
`make-serious-condition` [Scheme Procedure]
`serious-condition?` *obj* [Scheme Procedure]

A base type for conditions representing errors serious enough that cannot be ignored.

`&error` [Condition Type]
`make-error` [Scheme Procedure]
`error?` *obj* [Scheme Procedure]

A base type for conditions representing errors.

`&violation` [Condition Type]
`make-violation` [Scheme Procedure]
`violation?` [Scheme Procedure]

A subtype of `&serious` that can be used to represent violations of a language or library standard.

`&assertion` [Condition Type]
`make-assertion-violation` [Scheme Procedure]
`assertion-violation?` *obj* [Scheme Procedure]

A subtype of `&violation` that indicates an invalid call to a procedure.

`&irritants` [Condition Type]
`make-irritants-condition` *irritants* [Scheme Procedure]
`irritants-condition?` *obj* [Scheme Procedure]
`condition-irritants` *condition* [Scheme Procedure]

A base type used for storing information about the causes of another condition in a compound condition.

`&who` [Condition Type]
`make-who-condition` *who* [Scheme Procedure]

`who-condition?` *obj* [Scheme Procedure]
`condition-who` *condition* [Scheme Procedure]
A base type used for storing the identity, a string or symbol, of the entity responsible for another condition in a compound condition.

`&non-continuable` [Condition Type]
`make-non-continuable-violation` [Scheme Procedure]
`non-continuable-violation?` *obj* [Scheme Procedure]
A subtype of `&violation` used to indicate that an exception handler invoked by `raise` has returned locally.

`&implementation-restriction` [Condition Type]
`make-implementation-restriction-violation` [Scheme Procedure]
`implementation-restriction-violation?` *obj* [Scheme Procedure]
A subtype of `&violation` used to indicate a violation of an implementation restriction.

`&lexical` [Condition Type]
`make-lexical-violation` [Scheme Procedure]
`lexical-violation?` *obj* [Scheme Procedure]
A subtype of `&violation` used to indicate a syntax violation at the level of the datum syntax.

`&syntax` [Condition Type]
`make-syntax-violation` *form subform* [Scheme Procedure]
`syntax-violation?` *obj* [Scheme Procedure]
`syntax-violation-form` *condition* [Scheme Procedure]
`syntax-violation-subform` *condition* [Scheme Procedure]
A subtype of `&violation` that indicates a syntax violation. The *form* and *subform* fields, which must be datum values, indicate the syntactic form responsible for the condition.

`&undefined` [Condition Type]
`make-undefined-violation` [Scheme Procedure]
`undefined-violation?` *obj* [Scheme Procedure]
A subtype of `&violation` that indicates a reference to an unbound identifier.

7.6.2.14 I/O Conditions

These condition types are exported by both the `(rnrs io ports (6))` and `(rnrs io simple (6))` libraries.

`&i/o` [Condition Type]
`make-i/o-error` [Scheme Procedure]
`i/o-error?` *obj* [Scheme Procedure]
A condition supertype for more specific I/O errors.

`&i/o-read` [Condition Type]
`make-i/o-read-error` [Scheme Procedure]
`i/o-read-error?` *obj* [Scheme Procedure]
A subtype of `&i/o`; represents read-related I/O errors.

`&i/o-write` [Condition Type]
`make-i/o-write-error` [Scheme Procedure]
`i/o-write-error?` *obj* [Scheme Procedure]
A subtype of `&i/o`; represents write-related I/O errors.

`&i/o-invalid-position` [Condition Type]
`make-i/o-invalid-position-error` *position* [Scheme Procedure]
`i/o-invalid-position-error?` *obj* [Scheme Procedure]
`i/o-error-position` *condition* [Scheme Procedure]
A subtype of `&i/o`; represents an error related to an attempt to set the file position to an invalid position.

`&i/o-filename` [Condition Type]
`make-io-filename-error` *filename* [Scheme Procedure]
`i/o-filename-error?` *obj* [Scheme Procedure]
`i/o-error-filename` *condition* [Scheme Procedure]
A subtype of `&i/o`; represents an error related to an operation on a named file.

`&i/o-file-protection` [Condition Type]
`make-i/o-file-protection-error` *filename* [Scheme Procedure]
`i/o-file-protection-error?` *obj* [Scheme Procedure]
A subtype of `&i/o-filename`; represents an error resulting from an attempt to access a named file for which the caller had insufficient permissions.

`&i/o-file-is-read-only` [Condition Type]
`make-i/o-file-is-read-only-error` *filename* [Scheme Procedure]
`i/o-file-is-read-only-error?` *obj* [Scheme Procedure]
A subtype of `&i/o-file-protection`; represents an error related to an attempt to write to a read-only file.

`&i/o-file-already-exists` [Condition Type]
`make-i/o-file-already-exists-error` *filename* [Scheme Procedure]
`i/o-file-already-exists-error?` *obj* [Scheme Procedure]
A subtype of `&i/o-filename`; represents an error related to an operation on an existing file that was assumed not to exist.

`&i/o-file-does-not-exist` [Condition Type]
`make-i/o-file-does-not-exist-error` [Scheme Procedure]
`i/o-file-does-not-exist-error?` *obj* [Scheme Procedure]
A subtype of `&i/o-filename`; represents an error related to an operation on a non-existent file that was assumed to exist.

`&i/o-port` [Condition Type]
`make-i/o-port-error` *port* [Scheme Procedure]
`i/o-port-error?` *obj* [Scheme Procedure]
`i/o-error-port` *condition* [Scheme Procedure]
A subtype of `&i/o`; represents an error related to an operation on the port *port*.

7.6.2.15 Transcoders

The transcoder facilities are exported by `(rnrs io ports)`.

Several different Unicode encoding schemes describe standard ways to encode characters and strings as byte sequences and to decode those sequences. Within this document, a *codec* is an immutable Scheme object that represents a Unicode or similar encoding scheme.

An *end-of-line style* is a symbol that, if it is not `none`, describes how a textual port transcodes representations of line endings.

A *transcoder* is an immutable Scheme object that combines a codec with an end-of-line style and a method for handling decoding errors. Each transcoder represents some specific bidirectional (but not necessarily lossless), possibly stateful translation between byte sequences and Unicode characters and strings. Every transcoder can operate in the input direction (bytes to characters) or in the output direction (characters to bytes). A *transcoder* parameter name means that the corresponding argument must be a transcoder.

A *binary port* is a port that supports binary I/O, does not have an associated transcoder and does not support textual I/O. A *textual port* is a port that supports textual I/O, and does not support binary I/O. A textual port may or may not have an associated transcoder.

`latin-1-codec` [Scheme Procedure]
`utf-8-codec` [Scheme Procedure]
`utf-16-codec` [Scheme Procedure]

These are predefined codecs for the ISO 8859-1, UTF-8, and UTF-16 encoding schemes.

A call to any of these procedures returns a value that is equal in the sense of `eqv?` to the result of any other call to the same procedure.

`eol-style` *`eol-style-symbol`* [Scheme Syntax]

eol-style-symbol should be a symbol whose name is one of `lf`, `cr`, `crlf`, `nel`, `crnel`, `ls`, and `none`.

The form evaluates to the corresponding symbol. If the name of *eol-style-symbol* is not one of these symbols, the effect and result are implementation-dependent; in particular, the result may be an eol-style symbol acceptable as an *eol-style* argument to `make-transcoder`. Otherwise, an exception is raised.

All eol-style symbols except `none` describe a specific line-ending encoding:

`lf`	linefeed
`cr`	carriage return
`crlf`	carriage return, linefeed
`nel`	next line
`crnel`	carriage return, next line
`ls`	line separator

For a textual port with a transcoder, and whose transcoder has an eol-style symbol `none`, no conversion occurs. For a textual input port, any eol-style symbol other than `none` means that all of the above line-ending encodings are recognized and are translated into a single linefeed. For a textual output port, `none` and `lf` are

equivalent. Linefeed characters are encoded according to the specified eol-style symbol, and all other characters that participate in possible line endings are encoded as is.

Note: Only the name of *eol-style-symbol* is significant.

`native-eol-style` [Scheme Procedure]
Returns the default end-of-line style of the underlying platform, e.g., `lf` on Unix and `crlf` on Windows.

`&i/o-decoding` [Condition Type]
`make-i/o-decoding-error` *port* [Scheme Procedure]
`i/o-decoding-error?` *obj* [Scheme Procedure]
This condition type could be defined by

```
(define-condition-type &i/o-decoding &i/o-port
  make-i/o-decoding-error i/o-decoding-error?)
```

An exception with this type is raised when one of the operations for textual input from a port encounters a sequence of bytes that cannot be translated into a character or string by the input direction of the port's transcoder.

When such an exception is raised, the port's position is past the invalid encoding.

`&i/o-encoding` [Condition Type]
`make-i/o-encoding-error` *port char* [Scheme Procedure]
`i/o-encoding-error?` *obj* [Scheme Procedure]
`i/o-encoding-error-char` *condition* [Scheme Procedure]
This condition type could be defined by

```
(define-condition-type &i/o-encoding &i/o-port
  make-i/o-encoding-error i/o-encoding-error?
  (char i/o-encoding-error-char))
```

An exception with this type is raised when one of the operations for textual output to a port encounters a character that cannot be translated into bytes by the output direction of the port's transcoder. *char* is the character that could not be encoded.

`error-handling-mode` *error-handling-mode-symbol* [Scheme Syntax]
error-handling-mode-symbol should be a symbol whose name is one of `ignore`, `raise`, and `replace`. The form evaluates to the corresponding symbol. If *error-handling-mode-symbol* is not one of these identifiers, effect and result are implementation-dependent: The result may be an error-handling-mode symbol acceptable as a *handling-mode* argument to `make-transcoder`. If it is not acceptable as a *handling-mode* argument to `make-transcoder`, an exception is raised.

Note: Only the name of *error-handling-mode-symbol* is significant.

The error-handling mode of a transcoder specifies the behavior of textual I/O operations in the presence of encoding or decoding errors.

If a textual input operation encounters an invalid or incomplete character encoding, and the error-handling mode is `ignore`, an appropriate number of bytes of the invalid encoding are ignored and decoding continues with the following bytes.

If the error-handling mode is `replace`, the replacement character U+FFFD is injected into the data stream, an appropriate number of bytes are ignored, and decoding continues with the following bytes.

If the error-handling mode is `raise`, an exception with condition type `&i/o-decoding` is raised.

If a textual output operation encounters a character it cannot encode, and the error-handling mode is `ignore`, the character is ignored and encoding continues with the next character. If the error-handling mode is `replace`, a codec-specific replacement character is emitted by the transcoder, and encoding continues with the next character. The replacement character is U+FFFD for transcoders whose codec is one of the Unicode encodings, but is the `?` character for the Latin-1 encoding. If the error-handling mode is `raise`, an exception with condition type `&i/o-encoding` is raised.

`make-transcoder` *codec* [Scheme Procedure]
`make-transcoder` *codec eol-style* [Scheme Procedure]
`make-transcoder` *codec eol-style handling-mode* [Scheme Procedure]
codec must be a codec; *eol-style*, if present, an eol-style symbol; and *handling-mode*, if present, an error-handling-mode symbol.

eol-style may be omitted, in which case it defaults to the native end-of-line style of the underlying platform. *handling-mode* may be omitted, in which case it defaults to `replace`. The result is a transcoder with the behavior specified by its arguments.

`native-transcoder` [Scheme procedure]
Returns an implementation-dependent transcoder that represents a possibly locale-dependent "native" transcoding.

`transcoder-codec` *transcoder* [Scheme Procedure]
`transcoder-eol-style` *transcoder* [Scheme Procedure]
`transcoder-error-handling-mode` *transcoder* [Scheme Procedure]
These are accessors for transcoder objects; when applied to a transcoder returned by `make-transcoder`, they return the *codec*, *eol-style*, and *handling-mode* arguments, respectively.

`bytevector->string` *bytevector transcoder* [Scheme Procedure]
Returns the string that results from transcoding the *bytevector* according to the input direction of the transcoder.

`string->bytevector` *string transcoder* [Scheme Procedure]
Returns the bytevector that results from transcoding the *string* according to the output direction of the transcoder.

7.6.2.16 rnrs io ports

Guile's binary and textual port interface was heavily inspired by R6RS, so many R6RS port interfaces are documented elsewhere. Note that R6RS ports are not disjoint from Guile's native ports, so Guile-specific procedures will work on ports created using the R6RS API, and vice versa. Also note that in Guile, all ports are both textual and binary. См. Раздел 6.14 [Input and Output], страница 352, for more on Guile's core port API. The

R6RS ports module wraps Guile's I/O routines in a helper that will translate native Guile exceptions to R6RS conditions; См. Раздел 7.6.2.14 [R6RS I/O Conditions], страница 697, for more. См. Раздел 7.6.2.17 [R6RS File Ports], страница 704, for documentation on the R6RS file port interface.

Note: The implementation of this R6RS API is not complete yet.

`eof-object?` *obj* [Scheme Procedure]
См. Раздел 6.14.2 [Binary I/O], страница 353, for documentation.

`eof-object` [Scheme Procedure]
Return the end-of-file (EOF) object.

```
(eof-object? (eof-object))
⇒ #t
```

`port?` *obj* [Scheme Procedure]
`input-port?` *obj* [Scheme Procedure]
`output-port?` *obj* [Scheme Procedure]
См. Раздел 6.14.1 [Ports], страница 352, for documentation.

`port-transcoder` *port* [Scheme Procedure]
Return a transcoder associated with the encoding of *port*. См. Раздел 6.14.3 [Encoding], страница 355, and См. Раздел 7.6.2.15 [R6RS Transcoders], страница 699.

`binary-port?` *port* [Scheme Procedure]
`textual-port?` *port* [Scheme Procedure]
Return `#t`, as all ports in Guile are suitable for binary and textual I/O. См. Раздел 6.14.3 [Encoding], страница 355, for more details.

`transcoded-port` *binary-port transcoder* [Scheme Procedure]
The `transcoded-port` procedure returns a new textual port with the specified *transcoder*. Otherwise the new textual port's state is largely the same as that of *binary-port*. If *binary-port* is an input port, the new textual port will be an input port and will transcode the bytes that have not yet been read from *binary-port*. If *binary-port* is an output port, the new textual port will be an output port and will transcode output characters into bytes that are written to the byte sink represented by *binary-port*.

As a side effect, however, `transcoded-port` closes *binary-port* in a special way that allows the new textual port to continue to use the byte source or sink represented by *binary-port*, even though *binary-port* itself is closed and cannot be used by the input and output operations described in this chapter.

`port-position` *port* [Scheme Procedure]
Equivalent to `(seek` ***port*** `SEEK_CUR 0)`. См. Раздел 6.14.7 [Random Access], страница 362.

`port-has-port-position?` *port* [Scheme Procedure]
Return `#t` is *port* supports `port-position`.

`set-port-position!` *port offset* [Scheme Procedure]
Equivalent to `(seek port SEEK_SET offset)`. См. Раздел 6.14.7 [Random Access], страница 362.

`port-has-set-port-position!?` *port* [Scheme Procedure]
Return `#t` is *port* supports `set-port-position!`.

`call-with-port` *port proc* [Scheme Procedure]
Call *proc*, passing it *port* and closing *port* upon exit of *proc*. Return the return values of *proc*.

`port-eof?` *input-port* [Scheme Procedure]
Equivalent to `(eof-object? (lookahead-u8 input-port))`.

`standard-input-port` [Scheme Procedure]
`standard-output-port` [Scheme Procedure]
`standard-error-port` [Scheme Procedure]
Returns a fresh binary input port connected to standard input, or a binary output port connected to the standard output or standard error, respectively. Whether the port supports the `port-position` and `set-port-position!` operations is implementation-dependent.

`current-input-port` [Scheme Procedure]
`current-output-port` [Scheme Procedure]
`current-error-port` [Scheme Procedure]
См. Раздел 6.14.9 [Default Ports], страница 364.

`open-bytevector-input-port` *bv* [*transcoder*] [Scheme Procedure]
`open-bytevector-output-port` [*transcoder*] [Scheme Procedure]
См. Раздел 6.14.10.2 [Bytevector Ports], страница 368.

`make-custom-binary-input-port` *id read! get-position set-position! close* [Scheme Procedure]
`make-custom-binary-output-port` *id write! get-position set-position! close* [Scheme Procedure]
`make-custom-binary-input/output-port` *id read! write! get-position set-position! close* [Scheme Procedure]
См. Раздел 6.14.10.4 [Custom Ports], страница 369.

`get-u8` *port* [Scheme Procedure]
`lookahead-u8` *port* [Scheme Procedure]
`get-bytevector-n` *port count* [Scheme Procedure]
`get-bytevector-n!` *port bv start count* [Scheme Procedure]
`get-bytevector-some` *port* [Scheme Procedure]
`get-bytevector-all` *port* [Scheme Procedure]
`put-u8` *port octet* [Scheme Procedure]
`put-bytevector` *port bv* [*start* [*count*]] [Scheme Procedure]
См. Раздел 6.14.2 [Binary I/O], страница 353.

`get-char` *textual-input-port* [Scheme Procedure]
`lookahead-char` *textual-input-port* [Scheme Procedure]
`get-string-n` *textual-input-port count* [Scheme Procedure]
`get-string-n!` *textual-input-port string start count* [Scheme Procedure]
`get-string-all` *textual-input-port* [Scheme Procedure]
`get-line` *textual-input-port* [Scheme Procedure]
`put-char` *port char* [Scheme Procedure]
`put-string` *port string* **[***start* **[***count***]]** [Scheme Procedure]
См. Раздел 6.14.4 [Textual I/O], страница 357.

`get-datum` *textual-input-port count* [Scheme Procedure]
Reads an external representation from *textual-input-port* and returns the datum it represents. The `get-datum` procedure returns the next datum that can be parsed from the given *textual-input-port*, updating *textual-input-port* to point exactly past the end of the external representation of the object.

Any *interlexeme space* (comment or whitespace, см. Раздел 6.18.1 [Scheme Syntax], страница 404) in the input is first skipped. If an end of file occurs after the interlexeme space, the end-of-file object is returned.

If a character inconsistent with an external representation is encountered in the input, an exception with condition types `&lexical` and `&i/o-read` is raised. Also, if the end of file is encountered after the beginning of an external representation, but the external representation is incomplete and therefore cannot be parsed, an exception with condition types `&lexical` and `&i/o-read` is raised.

`put-datum` *textual-output-port datum* [Scheme Procedure]
datum should be a datum value. The `put-datum` procedure writes an external representation of *datum* to *textual-output-port*. The specific external representation is implementation-dependent. However, whenever possible, an implementation should produce a representation for which `get-datum`, when reading the representation, will return an object equal (in the sense of `equal?`) to *datum*.

> **Note:** Not all datums may allow producing an external representation for which `get-datum` will produce an object that is equal to the original. Specifically, NaNs contained in *datum* may make this impossible.
>
> **Note:** The `put-datum` procedure merely writes the external representation, but no trailing delimiter. If `put-datum` is used to write several subsequent external representations to an output port, care should be taken to delimit them properly so they can be read back in by subsequent calls to `get-datum`.

`flush-output-port` *port* [Scheme Procedure]
См. Раздел 6.14.6 [Buffering], страница 360, for documentation on `force-output`.

7.6.2.17 R6RS File Ports

The facilities described in this section are exported by the `(rnrs io ports)` module.

`buffer-mode` *buffer-mode-symbol* [Scheme Syntax]
buffer-mode-symbol must be a symbol whose name is one of `none`, `line`, and `block`. The result is the corresponding symbol, and specifies the associated buffer mode.

См. Раздел 6.14.6 [Buffering], страница 360, for a discussion of these different buffer modes. To control the amount of buffering, use `setvbuf` instead. Note that only the name of *buffer-mode-symbol* is significant.

См. Раздел 6.14.6 [Buffering], страница 360, for a discussion of port buffering.

`buffer-mode?` *obj* [Scheme Procedure]

Returns `#t` if the argument is a valid buffer-mode symbol, and returns `#f` otherwise.

When opening a file, the various procedures accept a `file-options` object that encapsulates flags to specify how the file is to be opened. A `file-options` object is an enum-set (см. Раздел 7.6.2.26 [rnrs enums], страница 717) over the symbols constituting valid file options.

A *file-options* parameter name means that the corresponding argument must be a file-options object.

`file-options` *`file-options-symbol` ...* [Scheme Syntax]

Each *file-options-symbol* must be a symbol.

The `file-options` syntax returns a file-options object that encapsulates the specified options.

When supplied to an operation that opens a file for output, the file-options object returned by `(file-options)` specifies that the file is created if it does not exist and an exception with condition type `&i/o-file-already-exists` is raised if it does exist. The following standard options can be included to modify the default behavior.

`no-create`
: If the file does not already exist, it is not created; instead, an exception with condition type `&i/o-file-does-not-exist` is raised. If the file already exists, the exception with condition type `&i/o-file-already-exists` is not raised and the file is truncated to zero length.

`no-fail`
: If the file already exists, the exception with condition type `&i/o-file-already-exists` is not raised, even if `no-create` is not included, and the file is truncated to zero length.

`no-truncate`
: If the file already exists and the exception with condition type `&i/o-file-already-exists` has been inhibited by inclusion of `no-create` or `no-fail`, the file is not truncated, but the port's current position is still set to the beginning of the file.

These options have no effect when a file is opened only for input. Symbols other than those listed above may be used as *file-options-symbol*s; they have implementation-specific meaning, if any.

Note: Only the name of *file-options-symbol* is significant.

`open-file-input-port` *filename* [Scheme Procedure]
`open-file-input-port` *filename file-options* [Scheme Procedure]
`open-file-input-port` *filename file-options buffer-mode* [Scheme Procedure]

`open-file-input-port` *filename file-options buffer-mode maybe-transcoder* [Scheme Procedure]

maybe-transcoder must be either a transcoder or `#f`.

The `open-file-input-port` procedure returns an input port for the named file. The *file-options* and *maybe-transcoder* arguments are optional.

The *file-options* argument, which may determine various aspects of the returned port, defaults to the value of `(file-options)`.

The *buffer-mode* argument, if supplied, must be one of the symbols that name a buffer mode. The *buffer-mode* argument defaults to `block`.

If *maybe-transcoder* is a transcoder, it becomes the transcoder associated with the returned port.

If *maybe-transcoder* is `#f` or absent, the port will be a binary port and will support the `port-position` and `set-port-position!` operations. Otherwise the port will be a textual port, and whether it supports the `port-position` and `set-port-position!` operations is implementation-dependent (and possibly transcoder-dependent).

`open-file-output-port` *filename* [Scheme Procedure]
`open-file-output-port` *filename file-options* [Scheme Procedure]
`open-file-output-port` *filename file-options buffer-mode* [Scheme Procedure]
`open-file-output-port` *filename file-options buffer-mode maybe-transcoder* [Scheme Procedure]

maybe-transcoder must be either a transcoder or `#f`.

The `open-file-output-port` procedure returns an output port for the named file.

The *file-options* argument, which may determine various aspects of the returned port, defaults to the value of `(file-options)`.

The *buffer-mode* argument, if supplied, must be one of the symbols that name a buffer mode. The *buffer-mode* argument defaults to `block`.

If *maybe-transcoder* is a transcoder, it becomes the transcoder associated with the port.

If *maybe-transcoder* is `#f` or absent, the port will be a binary port and will support the `port-position` and `set-port-position!` operations. Otherwise the port will be a textual port, and whether it supports the `port-position` and `set-port-position!` operations is implementation-dependent (and possibly transcoder-dependent).

7.6.2.18 rnrs io simple

The `(rnrs io simple (6))` library provides convenience functions for performing textual I/O on ports. This library also exports all of the condition types and associated procedures described in (см. Раздел 7.6.2.14 [R6RS I/O Conditions], страница 697). In the context of this section, when stating that a procedure behaves "identically" to the corresponding procedure in Guile's core library, this is modulo the behavior wrt. conditions: such procedures raise the appropriate R6RS conditions in case of error, but otherwise behave identically.

Note: There are still known issues regarding condition-correctness; some errors may still be thrown as native Guile exceptions instead of the appropriate R6RS conditions.

`eof-object` [Scheme Procedure]
`eof-object?` *obj* [Scheme Procedure]
These procedures are identical to the ones provided by the `(rnrs io ports (6))` library. См. Раздел 7.6.2.16 [rnrs io ports], страница 701, for documentation.

`input-port?` *obj* [Scheme Procedure]
`output-port?` *obj* [Scheme Procedure]
These procedures are identical to the ones provided by Guile's core library. См. Раздел 6.14.1 [Ports], страница 352, for documentation.

`call-with-input-file` *filename proc* [Scheme Procedure]
`call-with-output-file` *filename proc* [Scheme Procedure]
`open-input-file` *filename* [Scheme Procedure]
`open-output-file` *filename* [Scheme Procedure]
`with-input-from-file` *filename thunk* [Scheme Procedure]
`with-output-to-file` *filename thunk* [Scheme Procedure]
These procedures are identical to the ones provided by Guile's core library. См. Раздел 6.14.10.1 [File Ports], страница 365, for documentation.

`close-input-port` *input-port* [Scheme Procedure]
`close-output-port` *output-port* [Scheme Procedure]
Closes the given *input-port* or *output-port*. These are legacy interfaces; just use `close-port`.

`peek-char` [Scheme Procedure]
`peek-char` *textual-input-port* [Scheme Procedure]
`read-char` [Scheme Procedure]
`read-char` *textual-input-port* [Scheme Procedure]
These procedures are identical to the ones provided by Guile's core library. См. Раздел 6.14.11 [Venerable Port Interfaces], страница 372, for documentation.

`read` [Scheme Procedure]
`read` *textual-input-port* [Scheme Procedure]
This procedure is identical to the one provided by Guile's core library. См. Раздел 6.18.2 [Scheme Read], страница 407, for documentation.

`display` *obj* [Scheme Procedure]
`display` *obj textual-output-port* [Scheme Procedure]
`newline` [Scheme Procedure]
`newline` *textual-output-port* [Scheme Procedure]
`write` *obj* [Scheme Procedure]
`write` *obj textual-output-port* [Scheme Procedure]
`write-char` *char* [Scheme Procedure]
`write-char` *char textual-output-port* [Scheme Procedure]
These procedures are identical to the ones provided by Guile's core library. См. Раздел 6.14.11 [Venerable Port Interfaces], страница 372, and См. Раздел 6.18.3 [Scheme Write], страница 408, for documentation.

7.6.2.19 rnrs files

The `(rnrs files (6))` library provides the `file-exists?` and `delete-file` procedures, which test for the existence of a file and allow the deletion of files from the file system, respectively.

These procedures are identical to the ones provided by Guile's core library. См. Раздел 7.2.3 [File System], страница 528, for documentation.

7.6.2.20 rnrs programs

The `(rnrs programs (6))` library provides procedures for process management and introspection.

`command-line` [Scheme Procedure]

This procedure is identical to the one provided by Guile's core library. См. Раздел 7.2.6 [Runtime Environment], страница 540, for documentation.

`exit` [*status*] [Scheme Procedure]

This procedure is identical to the one provided by Guile's core library. См. Раздел 7.2.7 [Processes], страница 542, for documentation.

7.6.2.21 rnrs arithmetic fixnums

The `(rnrs arithmetic fixnums (6))` library provides procedures for performing arithmetic operations on an implementation-dependent range of exact integer values, which R6RS refers to as *fixnums*. In Guile, the size of a fixnum is determined by the size of the `SCM` type; a single SCM struct is guaranteed to be able to hold an entire fixnum, making fixnum computations particularly efficient—(см. Раздел 6.3 [The SCM Type], страница 108). On 32-bit systems, the most negative and most positive fixnum values are, respectively, -536870912 and 536870911.

Unless otherwise specified, all of the procedures below take fixnums as arguments, and will raise an `&assertion` condition if passed a non-fixnum argument or an `&implementation-restriction` condition if their result is not itself a fixnum.

`fixnum?` *obj* [Scheme Procedure]

Returns `#t` if *obj* is a fixnum, `#f` otherwise.

`fixnum-width` [Scheme Procedure]
`least-fixnum` [Scheme Procedure]
`greatest-fixnum` [Scheme Procedure]

These procedures return, respectively, the maximum number of bits necessary to represent a fixnum value in Guile, the minimum fixnum value, and the maximum fixnum value.

`fx=?` *fx1 fx2 fx3 ...* [Scheme Procedure]
`fx>?` *fx1 fx2 fx3 ...* [Scheme Procedure]
`fx<?` *fx1 fx2 fx3 ...* [Scheme Procedure]
`fx>=?` *fx1 fx2 fx3 ...* [Scheme Procedure]
`fx<=?` *fx1 fx2 fx3 ...* [Scheme Procedure]

These procedures return `#t` if their fixnum arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing; `#f` otherwise.

`fxzero?` *fx* [Scheme Procedure]
`fxpositive?` *fx* [Scheme Procedure]
`fxnegative?` *fx* [Scheme Procedure]
`fxodd?` *fx* [Scheme Procedure]
`fxeven?` *fx* [Scheme Procedure]

These numerical predicates return `#t` if *fx* is, respectively, zero, greater than zero, less than zero, odd, or even; `#f` otherwise.

`fxmax` *fx1 fx2 ...* [Scheme Procedure]
`fxmin` *fx1 fx2 ...* [Scheme Procedure]

These procedures return the maximum or minimum of their arguments.

`fx+` *fx1 fx2* [Scheme Procedure]
`fx*` *fx1 fx2* [Scheme Procedure]

These procedures return the sum or product of their arguments.

`fx-` *fx1 fx2* [Scheme Procedure]
`fx-` *fx* [Scheme Procedure]

Returns the difference of *fx1* and *fx2*, or the negation of *fx*, if called with a single argument.

An `&assertion` condition is raised if the result is not itself a fixnum.

`fxdiv-and-mod` *fx1 fx2* [Scheme Procedure]
`fxdiv` *fx1 fx2* [Scheme Procedure]
`fxmod` *fx1 fx2* [Scheme Procedure]
`fxdiv0-and-mod0` *fx1 fx2* [Scheme Procedure]
`fxdiv0` *fx1 fx2* [Scheme Procedure]
`fxmod0` *fx1 fx2* [Scheme Procedure]

These procedures implement number-theoretic division on fixnums; См. ⟨undefined⟩ [(rnrs base)], страница ⟨undefined⟩, for a description of their semantics.

`fx+/carry` *fx1 fx2 fx3* [Scheme Procedure]

Returns the two fixnum results of the following computation:

```
(let* ((s (+ fx1 fx2 fx3))
       (s0 (mod0 s (expt 2 (fixnum-width))))
       (s1 (div0 s (expt 2 (fixnum-width)))))
  (values s0 s1))
```

`fx-/carry` *fx1 fx2 fx3* [Scheme Procedure]

Returns the two fixnum results of the following computation:

```
(let* ((d (- fx1 fx2 fx3))
       (d0 (mod0 d (expt 2 (fixnum-width))))
       (d1 (div0 d (expt 2 (fixnum-width)))))
  (values d0 d1))
```

`fx*/carry` *fx1 fx2 fx3* [Scheme Procedure]

```
Returns the two fixnum results of the following computation:
(let* ((s (+ (* fx1 fx2) fx3))
```

```
        (s0 (mod0 s (expt 2 (fixnum-width))))
        (s1 (div0 s (expt 2 (fixnum-width)))))
    (values s0 s1))
```

`fxnot` *fx* [Scheme Procedure]
`fxand` *fx1 ...* [Scheme Procedure]
`fxior` *fx1 ...* [Scheme Procedure]
`fxxor` *fx1 ...* [Scheme Procedure]

These procedures are identical to the `lognot`, `logand`, `logior`, and `logxor` procedures provided by Guile's core library. См. Раздел 6.6.2.13 [Bitwise Operations], страница 134, for documentation.

`fxif` *fx1 fx2 fx3* [Scheme Procedure]

Returns the bitwise "if" of its fixnum arguments. The bit at position `i` in the return value will be the `i`th bit from *fx2* if the `i`th bit of *fx1* is 1, the `i`th bit from *fx3*.

`fxbit-count` *fx* [Scheme Procedure]

Returns the number of 1 bits in the two's complement representation of *fx*.

`fxlength` *fx* [Scheme Procedure]

Returns the number of bits necessary to represent *fx*.

`fxfirst-bit-set` *fx* [Scheme Procedure]

Returns the index of the least significant 1 bit in the two's complement representation of *fx*.

`fxbit-set?` *fx1 fx2* [Scheme Procedure]

Returns `#t` if the *fx2*th bit in the two's complement representation of *fx1* is 1, `#f` otherwise.

`fxcopy-bit` *fx1 fx2 fx3* [Scheme Procedure]

Returns the result of setting the *fx2*th bit of *fx1* to the *fx2*th bit of *fx3*.

`fxbit-field` *fx1 fx2 fx3* [Scheme Procedure]

Returns the integer representation of the contiguous sequence of bits in *fx1* that starts at position *fx2* (inclusive) and ends at position *fx3* (exclusive).

`fxcopy-bit-field` *fx1 fx2 fx3 fx4* [Scheme Procedure]

Returns the result of replacing the bit field in *fx1* with start and end positions *fx2* and *fx3* with the corresponding bit field from *fx4*.

`fxarithmetic-shift` *fx1 fx2* [Scheme Procedure]
`fxarithmetic-shift-left` *fx1 fx2* [Scheme Procedure]
`fxarithmetic-shift-right` *fx1 fx2* [Scheme Procedure]

Returns the result of shifting the bits of *fx1* right or left by the *fx2* positions. `fxarithmetic-shift` is identical to `fxarithmetic-shift-left`.

`fxrotate-bit-field` *fx1 fx2 fx3 fx4* [Scheme Procedure]

Returns the result of cyclically permuting the bit field in *fx1* with start and end positions *fx2* and *fx3* by *fx4* bits in the direction of more significant bits.

`fxreverse-bit-field` *fx1 fx2 fx3* [Scheme Procedure]
Returns the result of reversing the order of the bits of *fx1* between position *fx2* (inclusive) and position *fx3* (exclusive).

7.6.2.22 rnrs arithmetic flonums

The `(rnrs arithmetic flonums (6))` library provides procedures for performing arithmetic operations on inexact representations of real numbers, which R6RS refers to as *flonums*.

Unless otherwise specified, all of the procedures below take flonums as arguments, and will raise an `&assertion` condition if passed a non-flonum argument.

`flonum?` *obj* [Scheme Procedure]
Returns `#t` if *obj* is a flonum, `#f` otherwise.

`real->flonum` *x* [Scheme Procedure]
Returns the flonum that is numerically closest to the real number *x*.

`fl=?` *fl1 fl2 fl3 ...* [Scheme Procedure]
`fl<?` *fl1 fl2 fl3 ...* [Scheme Procedure]
`fl<=?` *fl1 fl2 fl3 ...* [Scheme Procedure]
`fl>?` *fl1 fl2 fl3 ...* [Scheme Procedure]
`fl>=?` *fl1 fl2 fl3 ...* [Scheme Procedure]
These procedures return `#t` if their flonum arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing; `#f` otherwise.

`flinteger?` *fl* [Scheme Procedure]
`flzero?` *fl* [Scheme Procedure]
`flpositive?` *fl* [Scheme Procedure]
`flnegative?` *fl* [Scheme Procedure]
`flodd?` *fl* [Scheme Procedure]
`fleven?` *fl* [Scheme Procedure]
These numerical predicates return `#t` if *fl* is, respectively, an integer, zero, greater than zero, less than zero, odd, even, `#f` otherwise. In the case of `flodd?` and `fleven?`, *fl* must be an integer-valued flonum.

`flfinite?` *fl* [Scheme Procedure]
`flinfinite?` *fl* [Scheme Procedure]
`flnan?` *fl* [Scheme Procedure]
These numerical predicates return `#t` if *fl* is, respectively, not infinite, infinite, or a `NaN` value.

`flmax` *fl1 fl2 ...* [Scheme Procedure]
`flmin` *fl1 fl2 ...* [Scheme Procedure]
These procedures return the maximum or minimum of their arguments.

`fl+` *fl1 ...* [Scheme Procedure]
`fl*` *fl ...* [Scheme Procedure]
These procedures return the sum or product of their arguments.

`fl-` *fl1 fl2 ...* [Scheme Procedure]
`fl-` *fl* [Scheme Procedure]
`fl/` *fl1 fl2 ...* [Scheme Procedure]
`fl/` *fl* [Scheme Procedure]
These procedures return, respectively, the difference or quotient of their arguments when called with two arguments; when called with a single argument, they return the additive or multiplicative inverse of *fl*.

`flabs` *fl* [Scheme Procedure]
Returns the absolute value of *fl*.

`fldiv-and-mod` *fl1 fl2* [Scheme Procedure]
`fldiv` *fl1 fl2* [Scheme Procedure]
`fldmod` *fl1 fl2* [Scheme Procedure]
`fldiv0-and-mod0` *fl1 fl2* [Scheme Procedure]
`fldiv0` *fl1 fl2* [Scheme Procedure]
`flmod0` *fl1 fl2* [Scheme Procedure]
These procedures implement number-theoretic division on flonums; См. ⟨undefined⟩ [(rnrs base)], страница ⟨undefined⟩, for a description for their semantics.

`flnumerator` *fl* [Scheme Procedure]
`fldenominator` *fl* [Scheme Procedure]
These procedures return the numerator or denominator of *fl* as a flonum.

`flfloor` *fl1* [Scheme Procedure]
`flceiling` *fl* [Scheme Procedure]
`fltruncate` *fl* [Scheme Procedure]
`flround` *fl* [Scheme Procedure]
These procedures are identical to the `floor`, `ceiling`, `truncate`, and `round` procedures provided by Guile's core library. См. Раздел 6.6.2.11 [Arithmetic], страница 128, for documentation.

`flexp` *fl* [Scheme Procedure]
`fllog` *fl* [Scheme Procedure]
`fllog` *fl1 fl2* [Scheme Procedure]
`flsin` *fl* [Scheme Procedure]
`flcos` *fl* [Scheme Procedure]
`fltan` *fl* [Scheme Procedure]
`flasin` *fl* [Scheme Procedure]
`flacos` *fl* [Scheme Procedure]
`flatan` *fl* [Scheme Procedure]
`flatan` *fl1 fl2* [Scheme Procedure]
These procedures, which compute the usual transcendental functions, are the flonum variants of the procedures provided by the R6RS base library (см. ⟨undefined⟩ [(rnrs base)], страница ⟨undefined⟩).

`flsqrt` *fl* [Scheme Procedure]
Returns the square root of *fl*. If *fl* is `-0.0`, *-0.0* is returned; for other negative values, a `NaN` value is returned.

`flexpt` *fl1 fl2* [Scheme Procedure]
Returns the value of *fl1* raised to the power of *fl2*.

The following condition types are provided to allow Scheme implementations that do not support infinities or `NaN` values to indicate that a computation resulted in such a value. Guile supports both of these, so these conditions will never be raised by Guile's standard libraries implementation.

`&no-infinities` [Condition Type]
`make-no-infinities-violation` *obj* [Scheme Procedure]
`no-infinities-violation?` [Scheme Procedure]
A condition type indicating that a computation resulted in an infinite value on a Scheme implementation incapable of representing infinities.

`&no-nans` [Condition Type]
`make-no-nans-violation` *obj* [Scheme Procedure]
`no-nans-violation?` *obj* [Scheme Procedure]
A condition type indicating that a computation resulted in a `NaN` value on a Scheme implementation incapable of representing `NaN`s.

`fixnum->flonum` *fx* [Scheme Procedure]
Returns the flonum that is numerically closest to the fixnum *fx*.

7.6.2.23 rnrs arithmetic bitwise

The `(rnrs arithmetic bitwise (6))` library provides procedures for performing bitwise arithmetic operations on the two's complement representations of fixnums.

This library and the procedures it exports share functionality with SRFI-60, which provides support for bitwise manipulation of integers (см. Раздел 7.5.34 [SRFI-60], страница 671).

`bitwise-not` *ei* [Scheme Procedure]
`bitwise-and` *ei1 ...* [Scheme Procedure]
`bitwise-ior` *ei1 ...* [Scheme Procedure]
`bitwise-xor` *ei1 ...* [Scheme Procedure]
These procedures are identical to the `lognot`, `logand`, `logior`, and `logxor` procedures provided by Guile's core library. См. Раздел 6.6.2.13 [Bitwise Operations], страница 134, for documentation.

`bitwise-if` *ei1 ei2 ei3* [Scheme Procedure]
Returns the bitwise "if" of its arguments. The bit at position `i` in the return value will be the `i`th bit from *ei2* if the `i`th bit of *ei1* is 1, the `i`th bit from *ei3*.

`bitwise-bit-count` *ei* [Scheme Procedure]
Returns the number of 1 bits in the two's complement representation of *ei*.

`bitwise-length` *ei* [Scheme Procedure]
Returns the number of bits necessary to represent *ei*.

`bitwise-first-bit-set` *ei* [Scheme Procedure]
Returns the index of the least significant 1 bit in the two's complement representation of *ei*.

`bitwise-bit-set?` *ei1 ei2* [Scheme Procedure]
Returns `#t` if the *ei2*th bit in the two's complement representation of *ei1* is 1, `#f` otherwise.

`bitwise-copy-bit` *ei1 ei2 ei3* [Scheme Procedure]
Returns the result of setting the *ei2*th bit of *ei1* to the *ei2*th bit of *ei3*.

`bitwise-bit-field` *ei1 ei2 ei3* [Scheme Procedure]
Returns the integer representation of the contiguous sequence of bits in *ei1* that starts at position *ei2* (inclusive) and ends at position *ei3* (exclusive).

`bitwise-copy-bit-field` *ei1 ei2 ei3 ei4* [Scheme Procedure]
Returns the result of replacing the bit field in *ei1* with start and end positions *ei2* and *ei3* with the corresponding bit field from *ei4*.

`bitwise-arithmetic-shift` *ei1 ei2* [Scheme Procedure]
`bitwise-arithmetic-shift-left` *ei1 ei2* [Scheme Procedure]
`bitwise-arithmetic-shift-right` *ei1 ei2* [Scheme Procedure]
Returns the result of shifting the bits of *ei1* right or left by the *ei2* positions. `bitwise-arithmetic-shift` is identical to `bitwise-arithmetic-shift-left`.

`bitwise-rotate-bit-field` *ei1 ei2 ei3 ei4* [Scheme Procedure]
Returns the result of cyclically permuting the bit field in *ei1* with start and end positions *ei2* and *ei3* by *ei4* bits in the direction of more significant bits.

`bitwise-reverse-bit-field` *ei1 ei2 ei3* [Scheme Procedure]
Returns the result of reversing the order of the bits of *ei1* between position *ei2* (inclusive) and position *ei3* (exclusive).

7.6.2.24 rnrs syntax-case

The `(rnrs syntax-case (6))` library provides access to the `syntax-case` system for writing hygienic macros. With one exception, all of the forms and procedures exported by this library are "re-exports" of Guile's native support for `syntax-case`; См. Раздел 6.10.3 [Syntax Case], страница 283, for documentation, examples, and rationale.

`make-variable-transformer` *proc* [Scheme Procedure]
Creates a new variable transformer out of *proc*, a procedure that takes a syntax object as input and returns a syntax object. If an identifier to which the result of this procedure is bound appears on the left-hand side of a `set!` expression, *proc* will be called with a syntax object representing the entire `set!` expression, and its return value will replace that `set!` expression.

`syntax-case` *expression* (*literal ...*) *clause ...* [Scheme Syntax]
The `syntax-case` pattern matching form.

`syntax` *template* [Scheme Syntax]
`quasisyntax` *template* [Scheme Syntax]
`unsyntax` *template* [Scheme Syntax]
`unsyntax-splicing` *template* [Scheme Syntax]
These forms allow references to be made in the body of a syntax-case output expression subform to datum and non-datum values. They are identical to the forms

provided by Guile's core library; См. Раздел 6.10.3 [Syntax Case], страница 283, for documentation.

`identifier?` *obj* [Scheme Procedure]
`bound-identifier=?` *id1 id2* [Scheme Procedure]
`free-identifier=?` *id1 id2* [Scheme Procedure]
These predicate procedures operate on syntax objects representing Scheme identifiers. `identifier?` returns `#t` if *obj* represents an identifier, `#f` otherwise. `bound-identifier=?` returns `#t` if and only if a binding for *id1* would capture a reference to *id2* in the transformer's output, or vice-versa. `free-identifier=?` returns `#t` if and only *id1* and *id2* would refer to the same binding in the output of the transformer, independent of any bindings introduced by the transformer.

`generate-temporaries` *l* [Scheme Procedure]
Returns a list, of the same length as *l*, which must be a list or a syntax object representing a list, of globally unique symbols.

`syntax->datum` *syntax-object* [Scheme Procedure]
`datum->syntax` *template-id datum* [Scheme Procedure]
These procedures convert wrapped syntax objects to and from Scheme datum values. The syntax object returned by `datum->syntax` shares contextual information with the syntax object *template-id*.

`syntax-violation` *whom message form* [Scheme Procedure]
`syntax-violation` *whom message form subform* [Scheme Procedure]
Constructs a new compound condition that includes the following simple conditions:

- If *whom* is not `#f`, a `&who` condition with the *whom* as its field
- A `&message` condition with the specified *message*
- A `&syntax` condition with the specified *form* and optional *subform* fields

7.6.2.25 rnrs hashtables

The `(rnrs hashtables (6))` library provides structures and procedures for creating and accessing hash tables. The hash tables API defined by R6RS is substantially similar to both Guile's native hash tables implementation as well as the one provided by SRFI-69; См. Раздел 6.6.22 [Hash Tables], страница 251, and Раздел 7.5.39 [SRFI-69], страница 673, respectively. Note that you can write portable R6RS library code that manipulates SRFI-69 hash tables (by importing the `(srfi :69)` library); however, hash tables created by one API cannot be used by another.

Like SRFI-69 hash tables—and unlike Guile's native ones—R6RS hash tables associate hash and equality functions with a hash table at the time of its creation. Additionally, R6RS allows for the creation (via `hashtable-copy`; see below) of immutable hash tables.

`make-eq-hashtable` [Scheme Procedure]
`make-eq-hashtable` *k* [Scheme Procedure]
Returns a new hash table that uses `eq?` to compare keys and Guile's `hashq` procedure as a hash function. If *k* is given, it specifies the initial capacity of the hash table.

`make-eqv-hashtable` [Scheme Procedure]
`make-eqv-hashtable` *k* [Scheme Procedure]
Returns a new hash table that uses `eqv?` to compare keys and Guile's `hashv` procedure as a hash function. If *k* is given, it specifies the initial capacity of the hash table.

`make-hashtable` *hash-function equiv* [Scheme Procedure]
`make-hashtable` *hash-function equiv k* [Scheme Procedure]
Returns a new hash table that uses *equiv* to compare keys and *hash-function* as a hash function. *equiv* must be a procedure that accepts two arguments and returns a true value if they are equivalent, `#f` otherwise; *hash-function* must be a procedure that accepts one argument and returns a non-negative integer.

If *k* is given, it specifies the initial capacity of the hash table.

`hashtable?` *obj* [Scheme Procedure]
Returns `#t` if *obj* is an R6RS hash table, `#f` otherwise.

`hashtable-size` *hashtable* [Scheme Procedure]
Returns the number of keys currently in the hash table *hashtable*.

`hashtable-ref` *hashtable key default* [Scheme Procedure]
Returns the value associated with *key* in the hash table *hashtable*, or *default* if none is found.

`hashtable-set!` *hashtable key obj* [Scheme Procedure]
Associates the key *key* with the value *obj* in the hash table *hashtable*, and returns an unspecified value. An `&assertion` condition is raised if *hashtable* is immutable.

`hashtable-delete!` *hashtable key* [Scheme Procedure]
Removes any association found for the key *key* in the hash table *hashtable*, and returns an unspecified value. An `&assertion` condition is raised if *hashtable* is immutable.

`hashtable-contains?` *hashtable key* [Scheme Procedure]
Returns `#t` if the hash table *hashtable* contains an association for the key *key*, `#f` otherwise.

`hashtable-update!` *hashtable key proc default* [Scheme Procedure]
Associates with *key* in the hash table *hashtable* the result of calling *proc*, which must be a procedure that takes one argument, on the value currently associated *key* in *hashtable*—or on *default* if no such association exists. An `&assertion` condition is raised if *hashtable* is immutable.

`hashtable-copy` *hashtable* [Scheme Procedure]
`hashtable-copy` *hashtable mutable* [Scheme Procedure]
Returns a copy of the hash table *hashtable*. If the optional argument *mutable* is provided and is a true value, the new hash table will be mutable.

`hashtable-clear!` *hashtable* [Scheme Procedure]
`hashtable-clear!` *hashtable k* [Scheme Procedure]
Removes all of the associations from the hash table *hashtable*. The optional argument *k*, which specifies a new capacity for the hash table, is accepted by Guile's `(rnrs hashtables)` implementation, but is ignored.

`hashtable-keys` *hashtable* [Scheme Procedure]
Returns a vector of the keys with associations in the hash table *hashtable*, in an unspecified order.

`hashtable-entries` *hashtable* [Scheme Procedure]
Return two values—a vector of the keys with associations in the hash table *hashtable*, and a vector of the values to which these keys are mapped, in corresponding but unspecified order.

`hashtable-equivalence-function` *hashtable* [Scheme Procedure]
Returns the equivalence predicated use by *hashtable*. This procedure returns `eq?` and `eqv?`, respectively, for hash tables created by `make-eq-hashtable` and `make-eqv-hashtable`.

`hashtable-hash-function` *hashtable* [Scheme Procedure]
Returns the hash function used by *hashtable*. For hash tables created by `make-eq-hashtable` or `make-eqv-hashtable`, `#f` is returned.

`hashtable-mutable?` *hashtable* [Scheme Procedure]
Returns `#t` if *hashtable* is mutable, `#f` otherwise.

A number of hash functions are provided for convenience:

`equal-hash` *obj* [Scheme Procedure]
Returns an integer hash value for *obj*, based on its structure and current contents. This hash function is suitable for use with `equal?` as an equivalence function.

`string-hash` *string* [Scheme Procedure]
`symbol-hash` *symbol* [Scheme Procedure]
These procedures are identical to the ones provided by Guile's core library. См. Раздел 6.6.22.2 [Hash Table Reference], страница 253, for documentation.

`string-ci-hash` *string* [Scheme Procedure]
Returns an integer hash value for *string* based on its contents, ignoring case. This hash function is suitable for use with `string-ci=?` as an equivalence function.

7.6.2.26 rnrs enums

The `(rnrs enums (6))` library provides structures and procedures for working with enumerable sets of symbols. Guile's implementation defines an *enum-set* record type that encapsulates a finite set of distinct symbols, the *universe*, and a subset of these symbols, which define the enumeration set.

The SRFI-1 list library provides a number of procedures for performing set operations on lists; Guile's `(rnrs enums)` implementation makes use of several of them. См. Раздел 7.5.3.10 [SRFI-1 Set Operations], страница 617, for more information.

`make-enumeration` *symbol-list* [Scheme Procedure]
Returns a new enum-set whose universe and enumeration set are both equal to *symbol-list*, a list of symbols.

`enum-set-universe` *enum-set* [Scheme Procedure]
Returns an enum-set representing the universe of *enum-set*, an enum-set.

`enum-set-indexer` *enum-set* [Scheme Procedure]
Returns a procedure that takes a single argument and returns the zero-indexed position of that argument in the universe of *enum-set*, or `#f` if its argument is not a member of that universe.

`enum-set-constructor` *enum-set* [Scheme Procedure]
Returns a procedure that takes a single argument, a list of symbols from the universe of *enum-set*, an enum-set, and returns a new enum-set with the same universe that represents a subset containing the specified symbols.

`enum-set->list` *enum-set* [Scheme Procedure]
Returns a list containing the symbols of the set represented by *enum-set*, an enum-set, in the order that they appear in the universe of *enum-set*.

`enum-set-member?` *symbol enum-set* [Scheme Procedure]
`enum-set-subset?` *enum-set1 enum-set2* [Scheme Procedure]
`enum-set=?` *enum-set1 enum-set2* [Scheme Procedure]
These procedures test for membership of symbols and enum-sets in other enum-sets. `enum-set-member?` returns `#t` if and only if *symbol* is a member of the subset specified by *enum-set*. `enum-set-subset?` returns `#t` if and only if the universe of *enum-set1* is a subset of the universe of *enum-set2* and every symbol in *enum-set1* is present in *enum-set2*. `enum-set=?` returns `#t` if and only if *enum-set1* is a subset, as per `enum-set-subset?` of *enum-set2* and vice versa.

`enum-set-union` *enum-set1 enum-set2* [Scheme Procedure]
`enum-set-intersection` *enum-set1 enum-set2* [Scheme Procedure]
`enum-set-difference` *enum-set1 enum-set2* [Scheme Procedure]
These procedures return, respectively, the union, intersection, and difference of their enum-set arguments.

`enum-set-complement` *enum-set* [Scheme Procedure]
Returns *enum-set*'s complement (an enum-set), with regard to its universe.

`enum-set-projection` *enum-set1 enum-set2* [Scheme Procedure]
Returns the projection of the enum-set *enum-set1* onto the universe of the enum-set *enum-set2*.

`define-enumeration` *type-name* (*symbol ...*) *constructor-syntax* [Scheme Syntax]
Evaluates to two new definitions: A constructor bound to *constructor-syntax* that behaves similarly to constructors created by `enum-set-constructor`, above, and creates new *enum-set*s in the universe specified by `(symbol ...)`; and a "predicate macro" bound to *type-name*, which has the following form:

```
(type-name sym)
```

If *sym* is a member of the universe specified by the *symbol*s above, this form evaluates to *sym*. Otherwise, a `&syntax` condition is raised.

7.6.2.27 rnrs

The `(rnrs (6))` library is a composite of all of the other R6RS standard libraries—it imports and re-exports all of their exported procedures and syntactic forms—with the exception of the following libraries:

- `(rnrs eval (6))`
- `(rnrs mutable-pairs (6))`
- `(rnrs mutable-strings (6))`
- `(rnrs r5rs (6))`

7.6.2.28 rnrs eval

The `(rnrs eval (6)` library provides procedures for performing "on-the-fly" evaluation of expressions.

`eval` *expression environment* [Scheme Procedure]
Evaluates *expression*, which must be a datum representation of a valid Scheme expression, in the environment specified by *environment*. This procedure is identical to the one provided by Guile's code library; См. Раздел 6.18.4 [Fly Evaluation], страница 409, for documentation.

`environment` *import-spec ...* [Scheme Procedure]
Constructs and returns a new environment based on the specified *import-spec*s, which must be datum representations of the import specifications used with the `import` form. См. Раздел 6.20.6 [R6RS Libraries], страница 439, for documentation.

7.6.2.29 rnrs mutable-pairs

The `(rnrs mutable-pairs (6))` library provides the `set-car!` and `set-cdr!` procedures, which allow the `car` and `cdr` fields of a pair to be modified.

These procedures are identical to the ones provide by Guile's core library. См. Раздел 6.6.8 [Pairs], страница 189, for documentation. All pairs in Guile are mutable; consequently, these procedures will never throw the `&assertion` condition described in the R6RS libraries specification.

7.6.2.30 rnrs mutable-strings

The `(rnrs mutable-strings (6))` library provides the `string-set!` and `string-fill!` procedures, which allow the content of strings to be modified "in-place."

These procedures are identical to the ones provided by Guile's core library. См. Раздел 6.6.5.6 [String Modification], страница 157, for documentation. All strings in Guile are mutable; consequently, these procedures will never throw the `&assertion` condition described in the R6RS libraries specification.

7.6.2.31 rnrs r5rs

The `(rnrs r5rs (6))` library exports bindings for some procedures present in R5RS but omitted from the R6RS base library specification.

`exact->inexact` *z* [Scheme Procedure]
`inexact->exact` *z* [Scheme Procedure]
These procedures are identical to the ones provided by Guile's core library. См. Раздел 6.6.2.5 [Exactness], страница 122, for documentation.

`quotient` *n1 n2* [Scheme Procedure]
`remainder` *n1 n2* [Scheme Procedure]
`modulo` *n1 n2* [Scheme Procedure]
These procedures are identical to the ones provided by Guile's core library. См. Раздел 6.6.2.7 [Integer Operations], страница 125, for documentation.

`delay` *expr* [Scheme Syntax]
`force` *promise* [Scheme Procedure]
The `delay` form and the `force` procedure are identical to their counterparts in Guile's core library. См. Раздел 6.18.9 [Delayed Evaluation], страница 418, for documentation.

`null-environment` *n* [Scheme Procedure]
`scheme-report-environment` *n* [Scheme Procedure]
These procedures are identical to the ones provided by the `(ice-9 r5rs)` Guile module. См. Раздел 6.20.11 [Environments], страница 446, for documentation.

7.7 Pattern Matching

The `(ice-9 match)` module provides a *pattern matcher*, written by Alex Shinn, and compatible with Andrew K. Wright's pattern matcher found in many Scheme implementations.

A pattern matcher can match an object against several patterns and extract the elements that make it up. Patterns can represent any Scheme object: lists, strings, symbols, records, etc. They can optionally contain *pattern variables*. When a matching pattern is found, an expression associated with the pattern is evaluated, optionally with all pattern variables bound to the corresponding elements of the object:

```
(let ((l '(hello (world))))
  (match l           ;; <- the input object
    (('hello (who))  ;; <- the pattern
     who)))          ;; <- the expression evaluated upon matching
⇒ world
```

In this example, list *l* matches the pattern `('hello (who))`, because it is a two-element list whose first element is the symbol `hello` and whose second element is a one-element list. Here *who* is a pattern variable. `match`, the pattern matcher, locally binds *who* to the value contained in this one-element list—i.e., the symbol `world`. An error would be raised if *l* did not match the pattern.

The same object can be matched against a simpler pattern:

```
(let ((l '(hello (world))))
  (match l
    ((x y)
     (values x y))))
```

```
⇒ hello
⇒ (world)
```

Here pattern `(x y)` matches any two-element list, regardless of the types of these elements. Pattern variables *x* and *y* are bound to, respectively, the first and second element of *l*.

Patterns can be composed, and nested. For instance, `...` (ellipsis) means that the previous pattern may be matched zero or more times in a list:

```
(match lst
  (((heads tails ...) ...)
   heads))
```

This expression returns the first element of each list within *lst*. For proper lists of proper lists, it is equivalent to `(map car lst)`. However, it performs additional checks to make sure that *lst* and the lists therein are proper lists, as prescribed by the pattern, raising an error if they are not.

Compared to hand-written code, pattern matching noticeably improves clarity and conciseness—no need to resort to series of `car` and `cdr` calls when matching lists, for instance. It also improves robustness, by making sure the input *completely* matches the pattern—conversely, hand-written code often trades robustness for conciseness. And of course, `match` is a macro, and the code it expands to is just as efficient as equivalent hand-written code.

The pattern matcher is defined as follows:

`match` *exp clause1 clause2 ...* [Scheme Syntax]

Match object *exp* against the patterns in *clause1 clause2 ...* in the order in which they appear. Return the value produced by the first matching clause. If no clause matches, throw an exception with key `match-error`.

Each clause has the form `(pattern body1 body2 ...)`. Each *pattern* must follow the syntax described below. Each body is an arbitrary Scheme expression, possibly referring to pattern variables of *pattern*.

The syntax and interpretation of patterns is as follows:

```
        patterns:                          matches:

pat ::= identifier                         anything, and binds identifier
      | _                                  anything
      | ()                                 the empty list
      | #t                                 #t
      | #f                                 #f
      | string                             a string
      | number                             a number
      | character                          a character
      | 'sexp                              an s-expression
      | 'symbol                            a symbol (special case of s-expr)
      | (pat_1 ... pat_n)                  list of n elements
      | (pat_1 ... pat_n . pat_{n+1})      list of n or more
      | (pat_1 ... pat_n pat_n+1 ooo)      list of n or more, each element
```

```
                                        of remainder must match pat_n+1
      | #(pat_1 ... pat_n)              vector of n elements
      | #(pat_1 ... pat_n pat_n+1 ooo)  vector of n or more, each element
                                        of remainder must match pat_n+1
      | #&pat                           box
      | ($ record-name pat_1 ... pat_n) a record
      | (= field pat)                   a ``field'' of an object
      | (and pat_1 ... pat_n)           if all of pat_1 thru pat_n match
      | (or pat_1 ... pat_n)            if any of pat_1 thru pat_n match
      | (not pat_1 ... pat_n)           if all pat_1 thru pat_n don't match
      | (? predicate pat_1 ... pat_n)   if predicate true and all of
                                        pat_1 thru pat_n match
      | (set! identifier)               anything, and binds setter
      | (get! identifier)               anything, and binds getter
      | `qp                             a quasi-pattern
      | (identifier *** pat)            matches pat in a tree and binds
                                        identifier to the path leading
                                        to the object that matches pat

ooo ::= ...                             zero or more
      | ___                             zero or more
      | ..1                             1 or more

        quasi-patterns:                 matches:

qp  ::= ()                              the empty list
      | #t                              #t
      | #f                              #f
      | string                          a string
      | number                          a number
      | character                       a character
      | identifier                      a symbol
      | (qp_1 ... qp_n)                 list of n elements
      | (qp_1 ... qp_n . qp_{n+1})      list of n or more
      | (qp_1 ... qp_n qp_n+1 ooo)      list of n or more, each element
                                        of remainder must match qp_n+1
      | #(qp_1 ... qp_n)                vector of n elements
      | #(qp_1 ... qp_n qp_n+1 ooo)     vector of n or more, each element
                                        of remainder must match qp_n+1
      | #&qp                            box
      | ,pat                            a pattern
      | ,@pat                           a pattern
```

The names `quote`, `quasiquote`, `unquote`, `unquote-splicing`, `?`, `_`, `$`, `and`, `or`, `not`, `set!`, `get!`, `...`, and `___` cannot be used as pattern variables.

Here is a more complex example:

```
(use-modules (srfi srfi-9))
```

```
(let ()
  (define-record-type person
    (make-person name friends)
    person?
    (name    person-name)
    (friends person-friends))

  (letrec ((alice (make-person "Alice" (delay (list bob))))
           (bob   (make-person "Bob" (delay (list alice)))))
    (match alice
      (($ person name (= force (($ person "Bob"))))
       (list 'friend-of-bob name))
      (_ #f))))

⇒ (friend-of-bob "Alice")
```

Here the `$` pattern is used to match a SRFI-9 record of type *person* containing two or more slots. The value of the first slot is bound to *name*. The `=` pattern is used to apply `force` on the second slot, and then checking that the result matches the given pattern. In other words, the complete pattern matches any *person* whose second slot is a promise that evaluates to a one-element list containing a *person* whose first slot is `"Bob"`.

Please refer to the `ice-9/match.upstream.scm` file in your Guile installation for more details.

Guile also comes with a pattern matcher specifically tailored to SXML trees, См. Раздел 7.16 [sxml-match], страница 751.

7.8 Readline Support

Guile comes with an interface module to the readline library (см. *GNU Readline Library*). This makes interactive use much more convenient, because of the command-line editing features of readline. Using `(ice-9 readline)`, you can navigate through the current input line with the cursor keys, retrieve older command lines from the input history and even search through the history entries.

7.8.1 Loading Readline Support

The module is not loaded by default and so has to be loaded and activated explicitly. This is done with two simple lines of code:

```
(use-modules (ice-9 readline))
(activate-readline)
```

The first line will load the necessary code, and the second will activate readline's features for the REPL. If you plan to use this module often, you should save these to lines to your `.guile` personal startup file.

You will notice that the REPL's behaviour changes a bit when you have loaded the readline module. For example, when you press Enter before typing in the closing parentheses of a list, you will see the *continuation* prompt, three dots: `...` This gives you a nice visual feedback when trying to match parentheses. To make this even easier,

bouncing parentheses are implemented. That means that when you type in a closing parentheses, the cursor will jump to the corresponding opening parenthesis for a short time, making it trivial to make them match.

Once the readline module is activated, all lines entered interactively will be stored in a history and can be recalled later using the cursor-up and -down keys. Readline also understands the Emacs keys for navigating through the command line and history.

When you quit your Guile session by evaluating `(quit)` or pressing Ctrl-D, the history will be saved to the file `.guile_history` and read in when you start Guile for the next time. Thus you can start a new Guile session and still have the (probably long-winded) definition expressions available.

You can specify a different history file by setting the environment variable `GUILE_HISTORY`. And you can make Guile specific customizations to your `.inputrc` by testing for application '`Guile`' (см. Раздел "Conditional Init Constructs" в *GNU Readline Library*). For instance to define a key inserting a matched pair of parentheses,

```
$if Guile
  "\C-o": "()\C-b"
$endif
```

7.8.2 Readline Options

The readline interface module can be tweaked in a few ways to better suit the user's needs. Configuration is done via the readline module's options interface, in a similar way to the evaluator and debugging options (см. Раздел 6.23.3 [Runtime Options], страница 481).

`readline-options` [Scheme Procedure]
`readline-enable` *option-name* [Scheme Procedure]
`readline-disable` *option-name* [Scheme Procedure]
`readline-set!` *option-name value* [Scheme Syntax]

Accessors for the readline options. Note that unlike the enable/disable procedures, `readline-set!` is syntax, which expects an unquoted option name.

Here is the list of readline options generated by typing `(readline-options 'help)` in Guile. You can also see the default values.

```
history-file    yes     Use history file.
history-length  200     History length.
bounce-parens   500     Time (ms) to show matching opening parenthesis
                        (0 = off).
bracketed-paste yes     Disable interpretation of control characters
                        in pastes.
```

The readline options interface can only be used *after* loading the readline module, because it is defined in that module.

7.8.3 Readline Functions

The following functions are provided by

```
(use-modules (ice-9 readline))
```

There are two ways to use readline from Scheme code, either make calls to `readline` directly to get line by line input, or use the readline port below with all the usual reading functions.

`readline` [*prompt*] [Функция]

Read a line of input from the user and return it as a string (without a newline at the end). *prompt* is the prompt to show, or the default is the string set in `set-readline-prompt!` below.

```
(readline "Type something: ") ⇒ "hello"
```

`set-readline-input-port!` *port* [Функция]
`set-readline-output-port!` *port* [Функция]

Set the input and output port the readline function should read from and write to. *port* must be a file port (см. Раздел 6.14.10.1 [File Ports], страница 365), and should usually be a terminal.

The default is the `current-input-port` and `current-output-port` (см. Раздел 6.14.9 [Default Ports], страница 364) when `(ice-9 readline)` loads, which in an interactive user session means the Unix "standard input" and "standard output".

7.8.3.1 Readline Port

`readline-port` [Функция]

Return a buffered input port (см. Раздел 7.14 [Buffered Input], страница 748) which calls the `readline` function above to get input. This port can be used with all the usual reading functions (`read`, `read-char`, etc), and the user gets the interactive editing features of readline.

There's only a single readline port created. `readline-port` creates it when first called, and on subsequent calls just returns what it previously made.

`activate-readline` [Функция]

If the `current-input-port` is a terminal (см. Раздел 7.2.9 [`isatty?`], страница 552) then enable readline for all reading from `current-input-port` (см. Раздел 6.14.9 [Default Ports], страница 364) and enable readline features in the interactive REPL (см. Раздел 3.3.3 [The REPL], страница 26).

```
(activate-readline)
(read-char)
```

`activate-readline` enables readline on `current-input-port` simply by a `set-current-input-port` to the `readline-port` above. An application can do that directly if the extra REPL features that `activate-readline` adds are not wanted.

`set-readline-prompt!` *prompt1* [*prompt2*] [Функция]

Set the prompt string to print when reading input. This is used when reading through `readline-port`, and is also the default prompt for the `readline` function above.

prompt1 is the initial prompt shown. If a user might enter an expression across multiple lines, then *prompt2* is a different prompt to show further input required. In the Guile REPL for instance this is an ellipsis ('`...`').

See `set-buffered-input-continuation?!` (см. Раздел 7.14 [Buffered Input], страница 748) for an application to indicate the boundaries of logical expressions (assuming of course an application has such a notion).

7.8.3.2 Completion

`with-readline-completion-function` *completer thunk* [Функция]
Call (*thunk*) with *completer* as the readline tab completion function to be used in any readline calls within that *thunk*. *completer* can be `#f` for no completion.

completer will be called as (*completer* `text state`), as described in (см. Раздел "How Completing Works" в *GNU Readline Library*). *text* is a partial word to be completed, and each *completer* call should return a possible completion string or `#f` when no more. *state* is `#f` for the first call asking about a new *text* then `#t` while getting further completions of that *text*.

Here's an example *completer* for user login names from the password file (см. Раздел 7.2.4 [User Information], страница 535), much like readline's own `rl_username_completion_function`,

```
(define (username-completer-function text state)
  (if (not state)
      (setpwent))  ;; new, go to start of database
  (let more ((pw (getpwent)))
    (if pw
        (if (string-prefix? text (passwd:name pw))
            (passwd:name pw)     ;; this name matches, return it
            (more (getpwent)))   ;; doesn't match, look at next
        (begin
          ;; end of database, close it and return #f
          (endpwent)
          #f))))
```

`apropos-completion-function` *text state* [Функция]
A completion function offering completions for Guile functions and variables (all `define`s). This is the default completion function.

`filename-completion-function` *text state* [Функция]
A completion function offering filename completions. This is readline's `rl_filename_completion_function` (см. Раздел "Completion Functions" в *GNU Readline Library*).

`make-completion-function` *string-list* [Функция]
Return a completion function which offers completions from the possibilities in *string-list*. Matching is case-sensitive.

7.9 Pretty Printing

The module `(ice-9 pretty-print)` provides the procedure `pretty-print`, which provides nicely formatted output of Scheme objects. This is especially useful for deeply nested or complex data structures, such as lists and vectors.

The module is loaded by entering the following:

```
(use-modules (ice-9 pretty-print))
```

This makes the procedure `pretty-print` available. As an example how `pretty-print` will format the output, see the following:

```
(pretty-print '(define (foo) (lambda (x)
(cond ((zero? x) #t) ((negative? x) -x) (else
(if (= x 1) 2 (* x x x)))))))
⊣
(define (foo)
  (lambda (x)
    (cond ((zero? x) #t)
          ((negative? x) -x)
          (else (if (= x 1) 2 (* x x x))))))
```

`pretty-print` *obj* **[***port***]** **[***keyword-options***]** [Scheme Procedure]

Print the textual representation of the Scheme object *obj* to *port*. *port* defaults to the current output port, if not given.

The further *keyword-options* are keywords and parameters as follows,

`#:display?` *flag*
: If *flag* is true then print using `display`. The default is `#f` which means use `write` style. См. Раздел 6.18.3 [Scheme Write], страница 408.

`#:per-line-prefix` *string*
: Print the given *string* as a prefix on each line. The default is no prefix.

`#:width` *columns*
: Print within the given *columns*. The default is 79.

`#:max-expr-width` *columns*
: The maximum width of an expression. The default is 50.

Also exported by the `(ice-9 pretty-print)` module is `truncated-print`, a procedure to print Scheme datums, truncating the output to a certain number of characters. This is useful when you need to present an arbitrary datum to the user, but you only have one line in which to do so.

```
(define exp '(a b #(c d e) f . g))
(truncated-print exp #:width 10) (newline)
⊣ (a b . #)
(truncated-print exp #:width 15) (newline)
⊣ (a b # f . g)
(truncated-print exp #:width 18) (newline)
⊣ (a b #(c ...) . #)
(truncated-print exp #:width 20) (newline)
⊣ (a b #(c d e) f . g)
(truncated-print "The quick brown fox" #:width 20) (newline)
⊣ "The quick brown..."
(truncated-print (current-module) #:width 20) (newline)
⊣ #<directory (gui...>
```

`truncated-print` will not output a trailing newline. If an expression does not fit in the given width, it will be truncated – possibly ellipsized[3], or in the worst case, displayed as `#`.

`truncated-print` *obj* [*port*] [*keyword-options*] [Scheme Procedure]

Print *obj*, truncating the output, if necessary, to make it fit into *width* characters. By default, *obj* will be printed using `write`, though that behavior can be overridden via the *display?* keyword argument.

The default behaviour is to print depth-first, meaning that the entire remaining width will be available to each sub-expression of *obj* – e.g., if *obj* is a vector, each member of *obj*. One can attempt to "ration" the available width, trying to allocate it equally to each sub-expression, via the *breadth-first?* keyword argument.

The further *keyword-options* are keywords and parameters as follows,

`#:display?` *flag*
: If *flag* is true then print using `display`. The default is `#f` which means use `write` style. см. Раздел 6.18.3 [Scheme Write], страница 408.

`#:width` *columns*
: Print within the given *columns*. The default is 79.

`#:breadth-first?` *flag*
: If *flag* is true, then allocate the available width breadth-first among elements of a compound data structure (list, vector, pair, etc.). The default is `#f` which means that any element is allowed to consume all of the available width.

7.10 Formatted Output

The `format` function is a powerful way to print numbers, strings and other objects together with literal text under the control of a format string. This function is available from

```
(use-modules (ice-9 format))
```

A format string is generally more compact and easier than using just the standard procedures like `display`, `write` and `newline`. Parameters in the output string allow various output styles, and parameters can be taken from the arguments for runtime flexibility.

`format` is similar to the Common Lisp procedure of the same name, but it's not identical and doesn't have quite all the features found in Common Lisp.

C programmers will note the similarity between `format` and `printf`, though escape sequences are marked with `~` instead of `%`, and are more powerful.

`format` *dest fmt arg ...* [Scheme Procedure]

Write output specified by the *fmt* string to *dest*. *dest* can be an output port, `#t` for `current-output-port` (см. Раздел 6.14.9 [Default Ports], страница 364), or `#f` to return the output as a string.

fmt can contain literal text to be output, and `~` escapes. Each escape has the form

```
~ [param [, param...] [:] [@] code
```

[3] On Unicode-capable ports, the ellipsis is represented by character 'HORIZONTAL ELLIPSIS' (U+2026), otherwise it is represented by three dots.

`code` is a character determining the escape sequence. The `:` and `@` characters are optional modifiers, one or both of which change the way various codes operate. Optional parameters are accepted by some codes too. Parameters have the following forms,

`[+/-]number`
: An integer, with optional `+` or `-`.

`'` (apostrophe)
: The following character in the format string, for instance `'z` for `z`.

`v`
: The next function argument as the parameter. `v` stands for "variable", a parameter can be calculated at runtime and included in the arguments. Upper case `V` can be used too.

`#`
: The number of arguments remaining. (See `~*` below for some usages.)

Parameters are separated by commas (`,`). A parameter can be left empty to keep its default value when supplying later parameters.

The following escapes are available. The code letters are not case-sensitive, upper and lower case are the same.

`~a`
`~s`
: Object output. Parameters: *minwidth*, *padinc*, *minpad*, *padchar*.

  `~a` outputs an argument like `display`, `~s` outputs an argument like `write` (см. Раздел 6.18.3 [Scheme Write], страница 408).

```
(format #t "~a" "foo") ⊣ foo
(format #t "~s" "foo") ⊣ "foo"
```

  `~:a` and `~:s` put objects that don't have an external representation in quotes like a string.

```
(format #t "~:a" car) ⊣ "#<primitive-procedure car>"
```

  If the output is less than *minwidth* characters (default 0), it's padded on the right with *padchar* (default space). `~@a` and `~@s` put the padding on the left instead.

```
(format #f "~5a" 'abc)       ⇒ "abc  "
(format #f "~5,,,'-@a" 'abc) ⇒ "--abc"
```

  minpad is a minimum for the padding then plus a multiple of *padinc*. Ie. the padding is $minpad+N*padinc$, where n is the smallest integer making the total object plus padding greater than or equal to *minwidth*. The default *minpad* is 0 and the default *padinc* is 1 (imposing no minimum or multiple).

```
(format #f "~5,1,4a" 'abc) ⇒ "abc    "
```

`~c`
: Character. Parameter: *charnum*.

  Output a character. The default is to simply output, as per `write-char` (см. Раздел 6.14.11 [Venerable Port Interfaces], страница 372). `~@c`

prints in `write` style. `~:c` prints control characters (ASCII 0 to 31) in `^X` form.

```
(format #t "~c" #\z)        ⊣ z
(format #t "~@c" #\z)       ⊣ #\z
(format #t "~:c" #\newline) ⊣ ^J
```

If the *charnum* parameter is given then an argument is not taken but instead the character is `(integer->char` *`charnum`*`)` (см. Раздел 6.6.3 [Characters], страница 139). This can be used for instance to output characters given by their ASCII code.

```
(format #t "~65c")  ⊣ A
```

`~d`
`~x`
`~o`
`~b` Integer. Parameters: *minwidth*, *padchar*, *commachar*, *commawidth*.

Output an integer argument as a decimal, hexadecimal, octal or binary integer (respectively), in a locale-independent way.

```
(format #t "~d" 123) ⊣ 123
```

`~@d` etc shows a `+` sign is shown on positive numbers.

```
(format #t "~@b" 12) ⊣ +1100
```

If the output is less than the *minwidth* parameter (default no minimum), it's padded on the left with the *padchar* parameter (default space).

```
(format #t "~5,'*d" 12)   ⊣ ***12
(format #t "~5,'0d" 12)   ⊣ 00012
(format #t "~3d"    1234) ⊣ 1234
```

`~:d` adds commas (or the *commachar* parameter) every three digits (or the *commawidth* parameter many). However, when your intent is to write numbers in a way that follows typographical conventions, using `~h` is recommended.

```
(format #t "~:d" 1234567)          ⊣ 1,234,567
(format #t "~10,'*,'/,2:d" 12345) ⊣ ***1/23/45
```

Hexadecimal `~x` output is in lower case, but the `~(` and `~)` case conversion directives described below can be used to get upper case.

```
(format #t "~x"       65261) ⊣ feed
(format #t "~:@(~x~)" 65261) ⊣ FEED
```

`~r` Integer in words, roman numerals, or a specified radix. Parameters: *radix*, *minwidth*, *padchar*, *commachar*, *commawidth*.

With no parameters output is in words as a cardinal like "ten", or `~:r` prints an ordinal like "tenth".

```
(format #t "~r" 9)  ⊣ nine        ;; cardinal
(format #t "~r" -9) ⊣ minus nine  ;; cardinal
(format #t "~:r" 9) ⊣ ninth       ;; ordinal
```

And also with no parameters, `~@r` gives roman numerals and `~:@r` gives old roman numerals. In old roman numerals there's no "subtraction", so

9 is `VIIII` instead of `IX`. In both cases only positive numbers can be output.

```
(format #t "~@r" 89)   ⊣ LXXXIX     ;; roman
(format #t "~:@r" 89) ⊣ LXXXVIIII  ;; old roman
```

When a parameter is given it means numeric output in the specified *radix*. The modifiers and parameters following the radix are the same as described for `~d` etc above.

```
(format #f "~3r" 27)   ⇒ "1000"    ;; base 3
(format #f "~3,5r" 26) ⇒ "  222"   ;; base 3 width 5
```

`~f` Fixed-point float. Parameters: *width*, *decimals*, *scale*, *overflowchar*, *padchar*.

Output a number or number string in fixed-point format, ie. with a decimal point.

```
(format #t "~f" 5)      ⊣ 5.0
(format #t "~f" "123")  ⊣ 123.0
(format #t "~f" "1e-1") ⊣ 0.1
```

`~@f` prints a `+` sign on positive numbers (including zero).

```
(format #t "~@f" 0) ⊣ +0.0
```

If the output is less than *width* characters it's padded on the left with *padchar* (space by default). If the output equals or exceeds *width* then there's no padding. The default for *width* is no padding.

```
(format #f "~6f" -1.5)      ⇒ "  -1.5"
(format #f "~6,,,'*f" 23)  ⇒ "**23.0"
(format #f "~6f" 1234567.0) ⇒ "1234567.0"
```

decimals is how many digits to print after the decimal point, with the value rounded or padded with zeros as necessary. (The default is to output as many decimals as required.)

```
(format #t "~1,2f" 3.125) ⊣ 3.13
(format #t "~1,2f" 1.5)   ⊣ 1.50
```

scale is a power of 10 applied to the value, moving the decimal point that many places. A positive *scale* increases the value shown, a negative decreases it.

```
(format #t "~,,2f" 1234)  ⊣ 123400.0
(format #t "~,,-2f" 1234) ⊣ 12.34
```

If *overflowchar* and *width* are both given and if the output would exceed *width*, then that many *overflowchar*s are printed instead of the value.

```
(format #t "~6,,,'xf" 12345) ⊣ 12345.
(format #t "~5,,,'xf" 12345) ⊣ xxxxx
```

`~h` Localized number[4]. Parameters: *width*, *decimals*, *padchar*.

Like `~f`, output an exact or floating point number, but do so according to the current locale, or according to the given locale object when the `:`

[4] The `~h` format specifier first appeared in Guile version 2.0.6.

modifier is used (см. Раздел 6.25.4 [Number Input and Output], страница 489).

```
(format #t "~h" 12345.5678)  ; with "C" as the current locale
⊣ 12345.5678

(format #t "~14,,'*:h" 12345.5678
        (make-locale LC_ALL "en_US"))
⊣ ***12,345.5678

(format #t "~,2:h" 12345.5678
        (make-locale LC_NUMERIC "fr_FR"))
⊣ 12 345,56
```

`~e` Exponential float. Parameters: *width*, *mantdigits*, *expdigits*, *intdigits*, *overflowchar*, *padchar*, *expchar*.

Output a number or number string in exponential notation.

```
(format #t "~e" 5000.25) ⊣ 5.00025E+3
(format #t "~e" "123.4") ⊣ 1.234E+2
(format #t "~e" "1e4")   ⊣ 1.0E+4
```

`~@e` prints a `+` sign on positive numbers (including zero). (This is for the mantissa, a `+` or `-` sign is always shown on the exponent.)

```
(format #t "~@e" 5000.0) ⊣ +5.0E+3
```

If the output is less than *width* characters it's padded on the left with *padchar* (space by default). The default for *width* is to output with no padding.

```
(format #f "~10e" 1234.0)     ⇒ "  1.234E+3"
(format #f "~10,,,,,'*e" 0.5) ⇒ "****5.0E-1"
```

mantdigits is the number of digits shown in the mantissa after the decimal point. The value is rounded or trailing zeros are added as necessary. The default *mantdigits* is to show as much as needed by the value.

```
(format #f "~,3e" 11111.0) ⇒ "1.111E+4"
(format #f "~,8e" 123.0)   ⇒ "1.23000000E+2"
```

expdigits is the minimum number of digits shown for the exponent, with leading zeros added if necessary. The default for *expdigits* is to show only as many digits as required. At least 1 digit is always shown.

```
(format #f "~,,1e" 1.0e99) ⇒ "1.0E+99"
(format #f "~,,6e" 1.0e99) ⇒ "1.0E+000099"
```

intdigits (default 1) is the number of digits to show before the decimal point in the mantissa. *intdigits* can be zero, in which case the integer part is a single `0`, or it can be negative, in which case leading zeros are shown after the decimal point.

```
(format #t "~,,,3e" 12345.0)  ⊣ 123.45E+2
(format #t "~,,,0e" 12345.0)  ⊣ 0.12345E+5
(format #t "~,,,-3e" 12345.0) ⊣ 0.00012345E+8
```

If *overflowchar* is given then *width* is a hard limit. If the output would exceed *width* then instead that many *overflowchar*s are printed.

```
(format #f "~6,,,'xe" 100.0) ⇒ "1.0E+2"
(format #f "~3,,,'xe" 100.0) ⇒ "xxx"
```

expchar is the exponent marker character (default `E`).

```
(format #t "~,,,,,,'ee" 100.0) ⊣ 1.0e+2
```

`~g` General float. Parameters: *width*, *mantdigits*, *expdigits*, *intdigits*, *overflowchar*, *padchar*, *expchar*.

Output a number or number string in either exponential format the same as `~e`, or fixed-point format like `~f` but aligned where the mantissa would have been and followed by padding where the exponent would have been.

Fixed-point is used when the absolute value is 0.1 or more and it takes no more space than the mantissa in exponential format, ie. basically up to *mantdigits* digits.

```
(format #f "~12,4,2g" 999.0)    ⇒ "   999.0    "
(format #f "~12,4,2g" "100000") ⇒ "  1.0000E+05"
```

The parameters are interpreted as per `~e` above. When fixed-point is used, the *decimals* parameter to `~f` is established from *mantdigits*, so as to give a total *mantdigits* + 1 figures.

`~$` Monetary style fixed-point float. Parameters: *decimals*, *intdigits*, *width*, *padchar*.

Output a number or number string in fixed-point format, ie. with a decimal point. *decimals* is the number of decimal places to show, default 2.

```
(format #t "~$" 5)        ⊣ 5.00
(format #t "~4$" "2.25") ⊣ 2.2500
(format #t "~4$" "1e-2") ⊣ 0.0100
```

`~@$` prints a `+` sign on positive numbers (including zero).

```
(format #t "~@$" 0) ⊣ +0.00
```

intdigits is a minimum number of digits to show in the integer part of the value (default 1).

```
(format #t "~,3$" 9.5)   ⊣ 009.50
(format #t "~,0$" 0.125) ⊣ .13
```

If the output is less than *width* characters (default 0), it's padded on the left with *padchar* (default space). `~:$` puts the padding after the sign.

```
(format #f "~,,8$" -1.5)   ⇒ "   -1.50"
(format #f "~,,8:$" -1.5)  ⇒ "-   1.50"
(format #f "~,,8,'.:@$" 3) ⇒ "+...3.00"
```

Note that floating point for dollar amounts is generally not a good idea, because a cent 0.01 cannot be represented exactly in the binary floating point Guile uses, which leads to slowly accumulating rounding errors. Keeping values as cents (or fractions of a cent) in integers then printing with the scale option in `~f` may be a better approach.

`~i` Complex fixed-point float. Parameters: *width*, *decimals*, *scale*, *overflowchar*, *padchar*.

Output the argument as a complex number, with both real and imaginary part shown (even if one or both are zero).

The parameters and modifiers are the same as for fixed-point `~f` described above. The real and imaginary parts are both output with the same given parameters and modifiers, except that for the imaginary part the `@` modifier is always enabled, so as to print a `+` sign between the real and imaginary parts.

```
(format #t "~i" 1)  ⊣ 1.0+0.0i
```

`~p` Plural. No parameters.

Output nothing if the argument is 1, or '`s`' for any other value.

```
(format #t "enter name~p" 1) ⊣ enter name
(format #t "enter name~p" 2) ⊣ enter names
```

`~@p` prints '`y`' for 1 or '`ies`' otherwise.

```
(format #t "pupp~@p" 1) ⊣ puppy
(format #t "pupp~@p" 2) ⊣ puppies
```

`~:p` re-uses the preceding argument instead of taking a new one, which can be convenient when printing some sort of count.

```
(format #t "~d cat~:p" 9)    ⊣ 9 cats
(format #t "~d pupp~:@p" 5) ⊣ 5 puppies
```

`~p` is designed for English plurals and there's no attempt to support other languages. `~[` conditionals (below) may be able to help. When using `gettext` to translate messages `ngettext` is probably best though (см. Раздел 6.25 [Internationalization], страница 486).

`~y` Structured printing. Parameters: *width*.

`~y` outputs an argument using `pretty-print` (см. Раздел 7.9 [Pretty Printing], страница 726). The result will be formatted to fit within *width* columns (79 by default), consuming multiple lines if necessary.

`~@y` outputs an argument using `truncated-print` (см. Раздел 7.9 [Pretty Printing], страница 726). The resulting code will be formatted to fit within *width* columns (79 by default), on a single line. The output will be truncated if necessary.

`~:@y` is like `~@y`, except the *width* parameter is interpreted to be the maximum column to which to output. That is to say, if you are at column 10, and `~60:@y` is seen, the datum will be truncated to 50 columns.

`~?`
`~k` Sub-format. No parameters.

Take a format string argument and a second argument which is a list of arguments for that string, and output the result.

```
(format #t "~?" "~d ~d" '(1 2))    ⊣ 1 2
```

`~@?` takes arguments for the sub-format directly rather than in a list.

```
(format #t "~@? ~s" "~d ~d" 1 2 "foo") ⊣ 1 2 "foo"
```

`~?` and `~k` are the same, `~k` is provided for T-Scheme compatibility.

`~*` Argument jumping. Parameter: *N*.

Move forward *N* arguments (default 1) in the argument list. `~:*` moves backwards. (*N* cannot be negative.)

```
(format #f "~d ~2*~d" 1 2 3 4) ⇒ "1 4"
(format #f "~d ~:*~d" 6)       ⇒ "6 6"
```

`~@*` moves to argument number *N*. The first argument is number 0 (and that's the default for *N*).

```
(format #f "~d~d again ~@*~d~d" 1 2) ⇒ "12 again 12"
(format #f "~d~d~d ~1@*~d~d" 1 2 3)  ⇒ "123 23"
```

A `#` move to the end followed by a `:` modifier move back can be used for an absolute position relative to the end of the argument list, a reverse of what the `@` modifier does.

```
(format #t "~#*~2:*~a" 'a 'b 'c 'd)   ⊣ c
```

At the end of the format string the current argument position doesn't matter, any further arguments are ignored.

`~t` Advance to a column position. Parameters: *colnum*, *colinc*, *padchar*.

Output *padchar* (space by default) to move to the given *colnum* column. The start of the line is column 0, the default for *colnum* is 1.

```
(format #f "~tX")  ⇒ " X"
(format #f "~3tX") ⇒ "   X"
```

If the current column is already past *colnum*, then the move is to there plus a multiple of *colinc*, ie. column $colnum + N * colinc$ for the smallest N which makes that value greater than or equal to the current column. The default *colinc* is 1 (which means no further move).

```
(format #f "abcd~2,5,'.tx") ⇒ "abcd...x"
```

`~@t` takes *colnum* as an offset from the current column. *colnum* many pad characters are output, then further padding to make the current column a multiple of *colinc*, if it isn't already so.

```
(format #f "a~3,5'*@tx") ⇒ "a****x"
```

`~t` is implemented using `port-column` (см. Раздел 6.14.4 [Textual I/O], страница 357), so it works even there has been other output before `format`.

`~~` Tilde character. Parameter: *n*.

Output a tilde character `~`, or *n* many if a parameter is given. Normally `~` introduces an escape sequence, `~~` is the way to output a literal tilde.

`~%` Newline. Parameter: *n*.

Output a newline character, or *n* many if a parameter is given. A newline (or a few newlines) can of course be output just by including them in the format string.

`~&` Start a new line. Parameter: *n*.

Output a newline if not already at the start of a line. With a parameter, output that many newlines, but with the first only if not already at the start of a line. So for instance 3 would be a newline if not already at the start of a line, and 2 further newlines.

`~_` Space character. Parameter: *n*.

Output a space character, or *n* many if a parameter is given.

With a variable parameter this is one way to insert runtime calculated padding (`~t` or the various field widths can do similar things).

```
(format #f "~v_foo" 4) ⇒ "    foo"
```

`~/` Tab character. Parameter: *n*.

Output a tab character, or *n* many if a parameter is given.

`~|` Formfeed character. Parameter: *n*.

Output a formfeed character, or *n* many if a parameter is given.

`~!` Force output. No parameters.

At the end of output, call `force-output` to flush any buffers on the destination (см. Раздел 6.14.6 [Buffering], страница 360). `~!` can occur anywhere in the format string, but the force is done at the end of output.

When output is to a string (destination `#f`), `~!` does nothing.

`~newline` (ie. newline character)

Continuation line. No parameters.

Skip this newline and any following whitespace in the format string, ie. don't send it to the output. This can be used to break up a long format string for readability, but not print the extra whitespace.

```
(format #f "abc~
            ~d def~
            ~d" 1 2) ⇒ "abc1 def2"
```

`~:newline` skips the newline but leaves any further whitespace to be printed normally.

`~@newline` prints the newline then skips following whitespace.

`~( ~)` Case conversion. No parameters.

Between `~(` and `~)` the case of all output is changed. The modifiers on `~(` control the conversion.

`~(` — lower case.

`~:@(` — upper case.

For example,

```
(format #t "~(Hello~)")    ⊣ hello
(format #t "~:@(Hello~)") ⊣ HELLO
```

In the future it's intended the modifiers `:` and `@` alone will capitalize the first letters of words, as per Common Lisp `format`, but the current implementation of this is flawed and not recommended for use.

Case conversions do not nest, currently. This might change in the future, but if it does then it will be to Common Lisp style where the outermost conversion has priority, overriding inner ones (making those fairly pointless).

`~{ ~}` Iteration. Parameter: *maxreps* (for `~{`).

The format between `~{` and `~}` is iterated. The modifiers to `~{` determine how arguments are taken. The default is a list argument with each iteration successively consuming elements from it. This is a convenient way to output a whole list.

```
(format #t "~{~d~}"     '(1 2 3))        ⊣ 123
(format #t "~{~s=~d ~}" '("x" 1 "y" 2)) ⊣ "x"=1 "y"=2
```

`~:{` takes a single argument which is a list of lists, each of those contained lists gives the arguments for the iterated format.

```
(format #t "~:{~dx~d ~}" '((1 2) (3 4) (5 6)))
⊣ 1x2 3x4 5x6
```

`~@{` takes arguments directly, with each iteration successively consuming arguments.

```
(format #t "~@{~d~}"     1 2 3)        ⊣ 123
(format #t "~@{~s=~d ~}" "x" 1 "y" 2) ⊣ "x"=1 "y"=2
```

`~:@{` takes list arguments, one argument for each iteration, using that list for the format.

```
(format #t "~:@{~dx~d ~}" '(1 2) '(3 4) '(5 6))
⊣ 1x2 3x4 5x6
```

Iterating stops when there are no more arguments or when the *maxreps* parameter to `~{` is reached (default no maximum).

```
(format #t "~2{~d~}" '(1 2 3 4)) ⊣ 12
```

If the format between `~{` and `~}` is empty, then a format string argument is taken (before iteration argument(s)) and used instead. This allows a sub-format (like `~?` above) to be iterated.

```
(format #t "~{~}" "~d" '(1 2 3)) ⊣ 123
```

Iterations can be nested, an inner iteration operates in the same way as described, but of course on the arguments the outer iteration provides it. This can be used to work into nested list structures. For example in the following the inner `~{~d~}x` is applied to `(1 2)` then `(3 4 5)` etc.

```
(format #t "~{~{~d~}x~}" '((1 2) (3 4 5))) ⊣ 12x345x
```

See also `~^` below for escaping from iteration.

`~[ ~; ~]` Conditional. Parameter: *selector*.

A conditional block is delimited by `~[` and `~]`, and `~;` separates clauses within the block. `~[` takes an integer argument and that number clause is used. The first clause is number 0.

```
(format #f "~[peach~;banana~;mango~]" 1)  ⇒ "banana"
```

The *selector* parameter can be used for the clause number, instead of taking an argument.

```
(format #f "~2[peach~;banana~;mango~]") ⇒ "mango"
```

If the clause number is out of range then nothing is output. Or the last clause can be `~:;` to use that for a number out of range.

```
(format #f "~[banana~;mango~]"         99) ⇒ ""
(format #f "~[banana~;mango~:;fruit~]" 99) ⇒ "fruit"
```

`~:[` treats the argument as a flag, and expects two clauses. The first is used if the argument is `#f` or the second otherwise.

```
(format #f "~:[false~;not false~]" #f)   ⇒ "false"
(format #f "~:[false~;not false~]" 'abc) ⇒ "not false"

(let ((n 3))
  (format #t "~d gnu~:[s are~; is~] here" n (= 1 n)))
⊣ 3 gnus are here
```

`~@[` also treats the argument as a flag, and expects one clause. If the argument is `#f` then no output is produced and the argument is consumed, otherwise the clause is used and the argument is not consumed, it's left for the clause. This can be used for instance to suppress output if `#f` means something not available.

```
(format #f "~@[temperature=~d~]" 27) ⇒ "temperature=27"
(format #f "~@[temperature=~d~]" #f) ⇒ ""
```

`~^` Escape. Parameters: *val1*, *val2*, *val3*.

Stop formatting if there are no more arguments. This can be used for instance to have a format string adapt to a variable number of arguments.

```
(format #t "~d~^ ~d" 1)   ⊣ 1
(format #t "~d~^ ~d" 1 2) ⊣ 1 2
```

Within a `~{ ~}` iteration, `~^` stops the current iteration step if there are no more arguments to that step, but continuing with possible further steps and the rest of the format. This can be used for instance to avoid a separator on the last iteration, or to adapt to variable length argument lists.

```
(format #f "~{~d~^/~} go"    '(1 2 3))     ⇒ "1/2/3 go"
(format #f "~:{ ~d~^~d~} go" '((1) (2 3))) ⇒ " 1 23 go"
```

Within a `~?` sub-format, `~^` operates just on that sub-format. If it terminates the sub-format then the originating format will still continue.

```
(format #t "~? items" "~d~^ ~d" '(1))   ⊣ 1 items
(format #t "~? items" "~d~^ ~d" '(1 2)) ⊣ 1 2 items
```

The parameters to `~^` (which are numbers) change the condition used to terminate. For a single parameter, termination is when that value is zero (notice this makes plain `~^` equivalent to `~#^`). For two parameters, termination is when those two are equal. For three parameters, termination is when *val1* $\leq$ *val2* and *val2* $\leq$ *val3*.

`~q` Inquiry message. Insert a copyright message into the output. `~:q` inserts the format implementation version.

It's an error if there are not enough arguments for the escapes in the format string, but any excess arguments are ignored.

Iterations `~{ ~}` and conditionals `~[ ~; ~]` can be nested, but must be properly nested, meaning the inner form must be entirely within the outer form. So it's not possible, for instance, to try to conditionalize the endpoint of an iteration.

```
(format #t "~{ ~[ ... ~] ~}" ...)       ;; good
(format #t "~{ ~[ ... ~} ... ~]" ...)   ;; bad
```

The same applies to case conversions `~( ~)`, they must properly nest with respect to iterations and conditionals (though currently a case conversion cannot nest within another case conversion).

When a sub-format (`~?`) is used, that sub-format string must be self-contained. It cannot for instance give a `~{` to begin an iteration form and have the `~}` up in the originating format, or similar.

Guile contains a `format` procedure even when the module `(ice-9 format)` is not loaded. The default `format` is `simple-format` (см. Раздел 6.14.5 [Simple Output], страница 359), it doesn't support all escape sequences documented in this section, and will signal an error if you try to use one of them. The reason for two versions is that the full `format` is fairly large and requires some time to load. `simple-format` is often adequate too.

7.11 File Tree Walk

The functions in this section traverse a tree of files and directories. They come in two flavors: the first one is a high-level functional interface, and the second one is similar to the C `ftw` and `nftw` routines (см. Раздел "Working with Directory Trees" в *GNU C Library Reference Manual*).

```
(use-modules (ice-9 ftw))
```

`file-system-tree` *file-name* [*enter?* [*stat*]] [Scheme Procedure]

Return a tree of the form `(`*`file-name stat children ...`*`)` where *stat* is the result of `(`*`stat file-name`*`)` and *children* are similar structures for each file contained in *file-name* when it designates a directory.

The optional *enter?* predicate is invoked as `(`*`enter? name stat`*`)` and should return true to allow recursion into directory *name*; the default value is a procedure that always returns `#t`. When a directory does not match *enter?*, it nonetheless appears in the resulting tree, only with zero children.

The *stat* argument is optional and defaults to `lstat`, as for `file-system-fold` (see below.)

The example below shows how to obtain a hierarchical listing of the files under the `module/language` directory in the Guile source tree, discarding their `stat` info:

```
(use-modules (ice-9 match))
```

```
(define remove-stat
  ;; Remove the `stat' object the `file-system-tree' provides
  ;; for each file in the tree.
  (match-lambda
    ((name stat)              ; flat file
     name)
    ((name stat children ...) ; directory
     (list name (map remove-stat children)))))

(let ((dir (string-append (assq-ref %guile-build-info 'top_srcdir)
                          "/module/language")))
  (remove-stat (file-system-tree dir)))

⇒
("language"
 (("value" ("spec.go" "spec.scm"))
  ("scheme"
   ("spec.go"
    "spec.scm"
    "compile-tree-il.scm"
    "decompile-tree-il.scm"
    "decompile-tree-il.go"
    "compile-tree-il.go"))
  ("tree-il"
   ("spec.go"
    "fix-letrec.go"
    "inline.go"
    "fix-letrec.scm"
    "compile-glil.go"
    "spec.scm"
    "optimize.scm"
    "primitives.scm"
    ...))
  ...))
```

It is often desirable to process directories entries directly, rather than building up a tree of entries in memory, like `file-system-tree` does. The following procedure, a *combinator*, is designed to allow directory entries to be processed directly as a directory tree is traversed; in fact, `file-system-tree` is implemented in terms of it.

`file-system-fold` *enter? leaf down up skip error init file-name* [*stat*] [Scheme Procedure]

Traverse the directory at *file-name*, recursively, and return the result of the successive applications of the *leaf*, *down*, *up*, and *skip* procedures as described below.

Enter sub-directories only when `(`*`enter? path stat result`*`)` returns true. When a sub-directory is entered, call `(`*`down path stat result`*`)`, where *path* is the path of

the sub-directory and *stat* the result of `(false-if-exception (stat path))`; when it is left, call `(up path stat result)`.

For each file in a directory, call `(leaf path stat result)`.

When *enter?* returns `#f`, or when an unreadable directory is encountered, call `(skip path stat result)`.

When *file-name* names a flat file, `(leaf path stat init)` is returned.

When an `opendir` or *stat* call fails, call `(error path stat errno result)`, with *errno* being the operating system error number that was raised—e.g., `EACCES`—and *stat* either `#f` or the result of the *stat* call for that entry, when available.

The special `.` and `..` entries are not passed to these procedures. The *path* argument to the procedures is a full file name—e.g., `"../foo/bar/gnu"`; if *file-name* is an absolute file name, then *path* is also an absolute file name. Files and directories, as identified by their device/inode number pair, are traversed only once.

The optional *stat* argument defaults to `lstat`, which means that symbolic links are not followed; the `stat` procedure can be used instead when symbolic links are to be followed (см. Раздел 7.2.3 [File System], страница 528).

The example below illustrates the use of `file-system-fold`:

```
(define (total-file-size file-name)
  "Return the size in bytes of the files under FILE-NAME (similar
to `du --apparent-size' with GNU Coreutils.)"

  (define (enter? name stat result)
    ;; Skip version control directories.
    (not (member (basename name) '(".git" ".svn" "CVS"))))
  (define (leaf name stat result)
    ;; Return RESULT plus the size of the file at NAME.
    (+ result (stat:size stat)))

  ;; Count zero bytes for directories.
  (define (down name stat result) result)
  (define (up name stat result) result)

  ;; Likewise for skipped directories.
  (define (skip name stat result) result)

  ;; Ignore unreadable files/directories but warn the user.
  (define (error name stat errno result)
    (format (current-error-port) "warning: ~a: ~a~%"
            name (strerror errno))
    result)

  (file-system-fold enter? leaf down up skip error
                    0  ; initial counter is zero bytes
                    file-name))
```

```
(total-file-size ".")
⇒ 8217554

(total-file-size "/dev/null")
⇒ 0
```

The alternative C-like functions are described below.

`scandir` *name* [*select?* [*entry<?*]] [Scheme Procedure]

Return the list of the names of files contained in directory *name* that match predicate *select?* (by default, all files). The returned list of file names is sorted according to *entry<?*, which defaults to `string-locale<?` such that file names are sorted in the locale's alphabetical order (см. Раздел 6.25.2 [Text Collation], страница 487). Return `#f` when *name* is unreadable or is not a directory.

This procedure is modeled after the C library function of the same name (см. Раздел "Scanning Directory Content" в *GNU C Library Reference Manual*).

`ftw` *startname proc* ['*hash-size n*] [Scheme Procedure]

Walk the file system tree descending from *startname*, calling *proc* for each file and directory.

Hard links and symbolic links are followed. A file or directory is reported to *proc* only once, and skipped if seen again in another place. One consequence of this is that `ftw` is safe against circularly linked directory structures.

Each *proc* call is (*proc* `filename statinfo flag`) and it should return `#t` to continue, or any other value to stop.

filename is the item visited, being *startname* plus a further path and the name of the item. *statinfo* is the return from `stat` (см. Раздел 7.2.3 [File System], страница 528) on *filename*. *flag* is one of the following symbols,

`regular` *filename* is a file, this includes special files like devices, named pipes, etc.

`directory`
: *filename* is a directory.

`invalid-stat`
: An error occurred when calling `stat`, so nothing is known. *statinfo* is `#f` in this case.

`directory-not-readable`
: *filename* is a directory, but one which cannot be read and hence won't be recursed into.

`symlink` *filename* is a dangling symbolic link. Symbolic links are normally followed and their target reported, the link itself is reported if the target does not exist.

The return value from `ftw` is `#t` if it ran to completion, or otherwise the non-`#t` value from *proc* which caused the stop.

Optional argument symbol `hash-size` and an integer can be given to set the size of the hash table used to track items already visited. (см. Раздел 6.6.22.2 [Hash Table Reference], страница 253)

In the current implementation, returning non-`#t` from *proc* is the only valid way to terminate `ftw`. *proc* must not use `throw` or similar to escape.

`nftw` *startname proc* [*'chdir*] [*'depth*] [*'hash-size n*] [*'mount*] [*'physical*] [Scheme Procedure]

Walk the file system tree starting at *startname*, calling *proc* for each file and directory. `nftw` has extra features over the basic `ftw` described above.

Like `ftw`, hard links and symbolic links are followed. A file or directory is reported to *proc* only once, and skipped if seen again in another place. One consequence of this is that `nftw` is safe against circular linked directory structures.

Each *proc* call is `(`*`proc`* `filename statinfo flag base level)` and it should return `#t` to continue, or any other value to stop.

filename is the item visited, being *startname* plus a further path and the name of the item. *statinfo* is the return from `stat` on *filename* (см. Раздел 7.2.3 [File System], страница 528). *base* is an integer offset into *filename* which is where the basename for this item begins. *level* is an integer giving the directory nesting level, starting from 0 for the contents of *startname* (or that item itself if it's a file). *flag* is one of the following symbols,

`regular` *filename* is a file, including special files like devices, named pipes, etc.

`directory`
: *filename* is a directory.

`directory-processed`
: *filename* is a directory, and its contents have all been visited. This flag is given instead of `directory` when the `depth` option below is used.

`invalid-stat`
: An error occurred when applying `stat` to *filename*, so nothing is known about it. *statinfo* is `#f` in this case.

`directory-not-readable`
: *filename* is a directory, but one which cannot be read and hence won't be recursed into.

`stale-symlink`
: *filename* is a dangling symbolic link. Links are normally followed and their target reported, the link itself is reported if its target does not exist.

`symlink` When the `physical` option described below is used, this indicates *filename* is a symbolic link whose target exists (and is not being followed).

The following optional arguments can be given to modify the way `nftw` works. Each is passed as a symbol (and `hash-size` takes a following integer value).

`chdir` Change to the directory containing the item before calling *proc*. When `nftw` returns the original current directory is restored.

Under this option, generally the *base* parameter to each *proc* call should be used to pick out the base part of the *filename*. The *filename* is still a

path but with a changed directory it won't be valid (unless the *startname* directory was absolute).

`depth` Visit files "depth first", meaning *proc* is called for the contents of each directory before it's called for the directory itself. Normally a directory is reported first, then its contents.

Under this option, the *flag* to *proc* for a directory is `directory-processed` instead of `directory`.

`hash-size` *n*
Set the size of the hash table used to track items already visited. (см. Раздел 6.6.22.2 [Hash Table Reference], страница 253)

`mount` Don't cross a mount point, meaning only visit items on the same file system as *startname* (ie. the same `stat:dev`).

`physical` Don't follow symbolic links, instead report them to *proc* as `symlink`. Dangling links (those whose target doesn't exist) are still reported as `stale-symlink`.

The return value from `nftw` is `#t` if it ran to completion, or otherwise the non-`#t` value from *proc* which caused the stop.

In the current implementation, returning non-`#t` from *proc* is the only valid way to terminate `ftw`. *proc* must not use `throw` or similar to escape.

7.12 Queues

The functions in this section are provided by

```
(use-modules (ice-9 q))
```

This module implements queues holding arbitrary scheme objects and designed for efficient first-in / first-out operations.

`make-q` creates a queue, and objects are entered and removed with `enq!` and `deq!`. `q-push!` and `q-pop!` can be used too, treating the front of the queue like a stack.

`make-q` [Scheme Procedure]
Return a new queue.

`q?` *obj* [Scheme Procedure]
Return `#t` if *obj* is a queue, or `#f` if not.

Note that queues are not a distinct class of objects but are implemented with cons cells. For that reason certain list structures can get `#t` from `q?`.

`enq!` *q obj* [Scheme Procedure]
Add *obj* to the rear of *q*, and return *q*.

`deq!` *q* [Scheme Procedure]
`q-pop!` *q* [Scheme Procedure]
Remove and return the front element from *q*. If *q* is empty, a `q-empty` exception is thrown.

`deq!` and `q-pop!` are the same operation, the two names just let an application match `enq!` with `deq!`, or `q-push!` with `q-pop!`.

`q-push!` *q obj* [Scheme Procedure]
Add *obj* to the front of *q*, and return *q*.

`q-length` *q* [Scheme Procedure]
Return the number of elements in *q*.

`q-empty?` *q* [Scheme Procedure]
Return true if *q* is empty.

`q-empty-check` *q* [Scheme Procedure]
Throw a `q-empty` exception if *q* is empty.

`q-front` *q* [Scheme Procedure]
Return the first element of *q* (without removing it). If *q* is empty, a `q-empty` exception is thrown.

`q-rear` *q* [Scheme Procedure]
Return the last element of *q* (without removing it). If *q* is empty, a `q-empty` exception is thrown.

`q-remove!` *q obj* [Scheme Procedure]
Remove all occurrences of *obj* from *q*, and return *q*. *obj* is compared to queue elements using `eq?`.

The `q-empty` exceptions described above are thrown just as `(throw 'q-empty)`, there's no message etc like an error throw.

A queue is implemented as a cons cell, the `car` containing a list of queued elements, and the `cdr` being the last cell in that list (for ease of enqueuing).

```
(list . last-cell)
```

If the queue is empty, *list* is the empty list and *last-cell* is `#f`.

An application can directly access the queue list if desired, for instance to search the elements or to insert at a specific point.

`sync-q!` *q* [Scheme Procedure]
Recompute the *last-cell* field in *q*.

All the operations above maintain *last-cell* as described, so normally there's no need for `sync-q!`. But if an application modifies the queue *list* then it must either maintain *last-cell* similarly, or call `sync-q!` to recompute it.

7.13 Streams

This section documents Guile's legacy stream module. For a more complete and portable stream library, см. Раздел 7.5.28 [SRFI-41], страница 654.

A stream represents a sequence of values, each of which is calculated only when required. This allows large or even infinite sequences to be represented and manipulated with familiar operations like "car", "cdr", "map" or "fold". In such manipulations only as much as needed is actually held in memory at any one time. The functions in this section are available from

```
(use-modules (ice-9 streams))
```

Streams are implemented using promises (см. Раздел 6.18.9 [Delayed Evaluation], страница 418), which is how the underlying calculation of values is made only when needed, and the values then retained so the calculation is not repeated.

Here is a simple example producing a stream of all odd numbers,

```
(define odds (make-stream (lambda (state)
                            (cons state (+ state 2)))
                          1))
(stream-car odds)              ⇒ 1
(stream-car (stream-cdr odds)) ⇒ 3
```

`stream-map` could be used to derive a stream of odd squares,

```
(define (square n) (* n n))
(define oddsquares (stream-map square odds))
```

These are infinite sequences, so it's not possible to convert them to a list, but they could be printed (infinitely) with for example

```
(stream-for-each (lambda (n sq)
                   (format #t "~a squared is ~a\n" n sq))
                 odds oddsquares)
⊣
1 squared is 1
3 squared is 9
5 squared is 25
7 squared is 49
...
```

`make-stream` *proc initial-state* [Scheme Procedure]

Return a new stream, formed by calling *proc* successively.

Each call is (*proc state*), it should return a pair, the `car` being the value for the stream, and the `cdr` being the new *state* for the next call. For the first call *state* is the given *initial-state*. At the end of the stream, *proc* should return some non-pair object.

`stream-car` *stream* [Scheme Procedure]

Return the first element from *stream*. *stream* must not be empty.

`stream-cdr` *stream* [Scheme Procedure]

Return a stream which is the second and subsequent elements of *stream*. *stream* must not be empty.

`stream-null?` *stream* [Scheme Procedure]

Return true if *stream* is empty.

`list->stream` *list* [Scheme Procedure]
`vector->stream` *vector* [Scheme Procedure]

Return a stream with the contents of *list* or *vector*.

list or *vector* should not be modified subsequently, since it's unspecified whether changes there will be reflected in the stream returned.

`port->stream` *port readproc* [Scheme Procedure]
Return a stream which is the values obtained by reading from *port* using *readproc*. Each read call is (*readproc port*), and it should return an EOF object (см. Раздел 6.14.2 [Binary I/O], страница 353) at the end of input.

For example a stream of characters from a file,

```
(port->stream (open-input-file "/foo/bar.txt") read-char)
```

`stream->list` *stream* [Scheme Procedure]
Return a list which is the entire contents of *stream*.

`stream->reversed-list` *stream* [Scheme Procedure]
Return a list which is the entire contents of *stream*, but in reverse order.

`stream->list&length` *stream* [Scheme Procedure]
Return two values (см. Раздел 6.13.7 [Multiple Values], страница 330), being firstly a list which is the entire contents of *stream*, and secondly the number of elements in that list.

`stream->reversed-list&length` *stream* [Scheme Procedure]
Return two values (см. Раздел 6.13.7 [Multiple Values], страница 330) being firstly a list which is the entire contents of *stream*, but in reverse order, and secondly the number of elements in that list.

`stream->vector` *stream* [Scheme Procedure]
Return a vector which is the entire contents of *stream*.

`stream-fold` *proc init stream1 stream2 ...* [Функция]
Apply *proc* successively over the elements of the given streams, from first to last until the end of the shortest stream is reached. Return the result from the last *proc* call.

Each call is (*proc* `elem1 elem2 ... prev`), where each *elem* is from the corresponding *stream*. *prev* is the return from the previous *proc* call, or the given *init* for the first call.

`stream-for-each` *proc stream1 stream2 ...* [Функция]
Call *proc* on the elements from the given *stream*s. The return value is unspecified.

Each call is (*proc* `elem1 elem2 ...`), where each *elem* is from the corresponding *stream*. `stream-for-each` stops when it reaches the end of the shortest *stream*.

`stream-map` *proc stream1 stream2 ...* [Функция]
Return a new stream which is the results of applying *proc* to the elements of the given *stream*s.

Each call is (*proc* `elem1 elem2 ...`), where each *elem* is from the corresponding *stream*. The new stream ends when the end of the shortest given *stream* is reached.

7.14 Buffered Input

The following functions are provided by

```
(use-modules (ice-9 buffered-input))
```

A buffered input port allows a reader function to return chunks of characters which are to be handed out on reading the port. A notion of further input for an application level logical expression is maintained too, and passed through to the reader.

`make-buffered-input-port` *reader* [Scheme Procedure]

Create an input port which returns characters obtained from the given *reader* function. *reader* is called (*reader* cont), and should return a string or an EOF object.

The new port gives precisely the characters returned by *reader*, nothing is added, so if any newline characters or other separators are desired they must come from the reader function.

The *cont* parameter to *reader* is `#f` for initial input, or `#t` when continuing an expression. This is an application level notion, set with `set-buffered-input-continuation?!` below. If the user has entered a partial expression then it allows *reader* for instance to give a different prompt to show more is required.

`make-line-buffered-input-port` *reader* [Scheme Procedure]

Create an input port which returns characters obtained from the specified *reader* function, similar to `make-buffered-input-port` above, but where *reader* is expected to be a line-oriented.

reader is called (*reader* cont), and should return a string or an EOF object as above. Each string is a line of input without a newline character, the port code inserts a newline after each string.

`set-buffered-input-continuation?!` *port cont* [Scheme Procedure]

Set the input continuation flag for a given buffered input *port*.

An application uses this by calling with a *cont* flag of `#f` when beginning to read a new logical expression. For example with the Scheme `read` function (см. Раздел 6.18.2 [Scheme Read], страница 407),

```
(define my-port (make-buffered-input-port my-reader))

(set-buffered-input-continuation?! my-port #f)
(let ((obj (read my-port)))
  ...
```

7.15 Expect

The macros in this section are made available with:

```
(use-modules (ice-9 expect))
```

`expect` is a macro for selecting actions based on the output from a port. The name comes from a tool of similar functionality by Don Libes. Actions can be taken when a particular string is matched, when a timeout occurs, or when end-of-file is seen on the port. The `expect` macro is described below; `expect-strings` is a front-end to `expect` based on regexec (see the regular expression documentation).

`expect-strings` *clause ...* [Макроопределение]

By default, `expect-strings` will read from the current input port. The first term in each clause consists of an expression evaluating to a string pattern (regular expression). As characters are read one-by-one from the port, they are accumulated in a buffer string which is matched against each of the patterns. When a pattern matches, the remaining expression(s) in the clause are evaluated and the value of the last is returned. For example:

```
(with-input-from-file "/etc/passwd"
  (lambda ()
    (expect-strings
      ("^nobody" (display "Got a nobody user.\n")
                 (display "That's no problem.\n"))
      ("^daemon" (display "Got a daemon user.\n")))))
```

The regular expression is compiled with the `REG_NEWLINE` flag, so that the `^` and $ anchors will match at any newline, not just at the start and end of the string.

There are two other ways to write a clause:

The expression(s) to evaluate can be omitted, in which case the result of the regular expression match (converted to strings, as obtained from regexec with match-pick set to `""`) will be returned if the pattern matches.

The symbol `=>` can be used to indicate that the expression is a procedure which will accept the result of a successful regular expression match. E.g.,

```
("^daemon" => write)
("^d(aemon)" => (lambda args (for-each write args)))
("^da(em)on" => (lambda (all sub)
                  (write all) (newline)
                  (write sub) (newline)))
```

The order of the substrings corresponds to the order in which the opening brackets occur.

A number of variables can be used to control the behaviour of `expect` (and `expect-strings`). Most have default top-level bindings to the value `#f`, which produces the default behaviour. They can be redefined at the top level or locally bound in a form enclosing the expect expression.

`expect-port`
: A port to read characters from, instead of the current input port.

`expect-timeout`
: `expect` will terminate after this number of seconds, returning `#f` or the value returned by expect-timeout-proc.

`expect-timeout-proc`
: A procedure called if timeout occurs. The procedure takes a single argument: the accumulated string.

`expect-eof-proc`
: A procedure called if end-of-file is detected on the input port. The procedure takes a single argument: the accumulated string.

`expect-char-proc`
: A procedure to be called every time a character is read from the port. The procedure takes a single argument: the character which was read.

`expect-strings-compile-flags`
: Flags to be used when compiling a regular expression, which are passed to `make-regexp` См. Раздел 6.15.1 [Regexp Functions], страница 380. The default value is `regexp/newline`.

`expect-strings-exec-flags`
: Flags to be used when executing a regular expression, which are passed to regexp-exec См. Раздел 6.15.1 [Regexp Functions], страница 380. The default value is `regexp/noteol`, which prevents `$` from matching the end of the string while it is still accumulating, but still allows it to match after a line break or at the end of file.

Here's an example using all of the variables:

```
(let ((expect-port (open-input-file "/etc/passwd"))
      (expect-timeout 1)
      (expect-timeout-proc
        (lambda (s) (display "Times up!\n")))
      (expect-eof-proc
        (lambda (s) (display "Reached the end of the file!\n")))
      (expect-char-proc display)
      (expect-strings-compile-flags (logior regexp/newline regexp/icase))
      (expect-strings-exec-flags 0))
   (expect-strings
     ("^nobody"  (display "Got a nobody user\n"))))
```

`expect` *clause ...* [Макроопределение]
: `expect` is used in the same way as `expect-strings`, but tests are specified not as patterns, but as procedures. The procedures are called in turn after each character is read from the port, with two arguments: the value of the accumulated string and a flag to indicate whether end-of-file has been reached. The flag will usually be `#f`, but if end-of-file is reached, the procedures are called an additional time with the final accumulated string and `#t`.

  The test is successful if the procedure returns a non-false value.

  If the `=>` syntax is used, then if the test succeeds it must return a list containing the arguments to be provided to the corresponding expression.

  In the following example, a string will only be matched at the beginning of the file:

```
(let ((expect-port (open-input-file "/etc/passwd")))
  (expect
     ((lambda (s eof?) (string=? s "fnord!"))
        (display "Got a nobody user!\n"))))
```

  The control variables described for `expect-strings` also influence the behaviour of `expect`, with the exception of variables whose names begin with `expect-strings-`.

7.16 sxml-match: Pattern Matching of SXML

The (sxml match) module provides syntactic forms for pattern matching of SXML trees, in a "by example" style reminiscent of the pattern matching of the syntax-rules and syntax-case macro systems. См. Раздел 7.20 [SXML], страница 761, for more information on SXML.

The following example[5] provides a brief illustration, transforming a music album catalog language into HTML.

```
(define (album->html x)
  (sxml-match x
    [(album (@ (title ,t)) (catalog (num ,n) (fmt ,f)) ...)
     `(ul (li ,t)
          (li (b ,n) (i ,f)) ...)]))
```

Three macros are provided: sxml-match, sxml-match-let, and sxml-match-let*.

Compared to a standard s-expression pattern matcher (см. Раздел 7.7 [Pattern Matching], страница 720), sxml-match provides the following benefits:

- matching of SXML elements does not depend on any degree of normalization of the SXML;
- matching of SXML attributes (within an element) is under-ordered; the order of the attributes specified within the pattern need not match the ordering with the element being matched;
- all attributes specified in the pattern must be present in the element being matched; in the spirit that XML is 'extensible', the element being matched may include additional attributes not specified in the pattern.

The present module is a descendant of WebIt!, and was inspired by an s-expression pattern matcher developed by Erik Hilsdale, Dan Friedman, and Kent Dybvig at Indiana University.

Syntax

sxml-match provides case-like form for pattern matching of XML nodes.

sxml-match *input-expression clause1 clause2 . . .* [Scheme Syntax]
Match *input-expression*, an SXML tree, according to the given *clause*s (one or more), each consisting of a pattern and one or more expressions to be evaluated if the pattern match succeeds. Optionally, each *clause* within sxml-match may include a *guard expression*.

The pattern notation is based on that of Scheme's syntax-rules and syntax-case macro systems. The grammar for the sxml-match syntax is given below:

```
match-form ::= (sxml-match input-expression
                 clause+)
```

[5] This example is taken from a paper by Krishnamurthi et al. Their paper was the first to show the usefulness of the syntax-rules style of pattern matching for transformation of XML, though the language described, XT3D, is an XML language.

```
clause ::= [node-pattern action-expression+]
         | [node-pattern (guard expression*) action-expression+]

node-pattern ::= literal-pattern
               | pat-var-or-cata
               | element-pattern
               | list-pattern

literal-pattern ::= string
                  | character
                  | number
                  | #t
                  | #f

attr-list-pattern ::= (@ attribute-pattern*)
                    | (@ attribute-pattern* . pat-var-or-cata)

attribute-pattern ::= (tag-symbol attr-val-pattern)

attr-val-pattern ::= literal-pattern
                   | pat-var-or-cata
                   | (pat-var-or-cata default-value-expr)

element-pattern ::= (tag-symbol attr-list-pattern?)
                  | (tag-symbol attr-list-pattern? nodeset-pattern)
                  | (tag-symbol attr-list-pattern?
                                nodeset-pattern? . pat-var-or-cata)

list-pattern ::= (list nodeset-pattern)
               | (list nodeset-pattern? . pat-var-or-cata)
               | (list)

nodeset-pattern ::= node-pattern
                  | node-pattern ...
                  | node-pattern nodeset-pattern
                  | node-pattern ... nodeset-pattern

pat-var-or-cata ::= (unquote var-symbol)
                  | (unquote [var-symbol*])
                  | (unquote [cata-expression -> var-symbol*])
```

Within a list or element body pattern, ellipses may appear only once, but may be followed by zero or more node patterns.

Guard expressions cannot refer to the return values of catamorphisms.

Ellipses in the output expressions must appear only in an expression context; ellipses are not allowed in a syntactic form.

The sections below illustrate specific aspects of the `sxml-match` pattern matcher.

Matching XML Elements

The example below illustrates the pattern matching of an XML element:

```
(sxml-match '(e (@ (i 1)) 3 4 5)
  [(e (@ (i ,d)) ,a ,b ,c) (list d a b c)]
  [,otherwise #f])
```

Each clause in `sxml-match` contains two parts: a pattern and one or more expressions which are evaluated if the pattern is successfully match. The example above matches an element `e` with an attribute `i` and three children.

Pattern variables must be "unquoted" in the pattern. The above expression binds *d* to `1`, *a* to `3`, *b* to `4`, and *c* to `5`.

Ellipses in Patterns

As in `syntax-rules`, ellipses may be used to specify a repeated pattern. Note that the pattern `item ...` specifies zero-or-more matches of the pattern `item`.

The use of ellipses in a pattern is illustrated in the code fragment below, where nested ellipses are used to match the children of repeated instances of an `a` element, within an element `d`.

```
(define x '(d (a 1 2 3) (a 4 5) (a 6 7 8) (a 9 10)))

(sxml-match x
  [(d (a ,b ...) ...)
   (list (list b ...) ...)])
```

The above expression returns a value of `((1 2 3) (4 5) (6 7 8) (9 10))`.

Ellipses in Quasiquote'd Output

Within the body of an `sxml-match` form, a slightly extended version of quasiquote is provided, which allows the use of ellipses. This is illustrated in the example below.

```
(sxml-match '(e 3 4 5 6 7)
  [(e ,i ... 6 7) `("start" ,(list 'wrap i) ... "end")]
  [,otherwise #f])
```

The general pattern is that `` `(something ,i ...) `` is rewritten as `` `(something ,@i) ``.

Matching Nodesets

A nodeset pattern is designated by a list in the pattern, beginning the identifier list. The example below illustrates matching a nodeset.

```
(sxml-match '("i" "j" "k" "l" "m")
  [(list ,a ,b ,c ,d ,e)
   `((p ,a) (p ,b) (p ,c) (p ,d) (p ,e))])
```

This example wraps each nodeset item in an HTML paragraph element. This example can be rewritten and simplified through using ellipsis:

```
(sxml-match '("i" "j" "k" "l" "m")
  [(list ,i ...)
   `((p ,i) ...)])
```

This version will match nodesets of any length, and wrap each item in the nodeset in an HTML paragraph element.

Matching the "Rest" of a Nodeset

Matching the "rest" of a nodeset is achieved by using a `. rest)` pattern at the end of an element or nodeset pattern.

This is illustrated in the example below:

```
(sxml-match '(e 3 (f 4 5 6) 7)
  [(e ,a (f . ,y) ,d)
   (list a y d)])
```

The above expression returns `(3 (4 5 6) 7)`.

Matching the Unmatched Attributes

Sometimes it is useful to bind a list of attributes present in the element being matched, but which do not appear in the pattern. This is achieved by using a `. rest)` pattern at the end of the attribute list pattern. This is illustrated in the example below:

```
(sxml-match '(a (@ (z 1) (y 2) (x 3)) 4 5 6)
  [(a (@ (y ,www) . ,qqq) ,t ,u ,v)
   (list www qqq t u v)])
```

The above expression matches the attribute `y` and binds a list of the remaining attributes to the variable *qqq*. The result of the above expression is `(2 ((z 1) (x 3)) 4 5 6)`.

This type of pattern also allows the binding of all attributes:

```
(sxml-match '(a (@ (z 1) (y 2) (x 3)))
  [(a (@ . ,qqq))
   qqq])
```

Default Values in Attribute Patterns

It is possible to specify a default value for an attribute which is used if the attribute is not present in the element being matched. This is illustrated in the following example:

```
(sxml-match '(e 3 4 5)
  [(e (@ (z (,d 1))) ,a ,b ,c) (list d a b c)])
```

The value `1` is used when the attribute `z` is absent from the element `e`.

Guards in Patterns

Guards may be added to a pattern clause via the `guard` keyword. A guard expression may include zero or more expressions which are evaluated only if the pattern is matched. The body of the clause is only evaluated if the guard expressions evaluate to `#t`.

The use of guard expressions is illustrated below:

```
(sxml-match '(a 2 3)
  ((a ,n) (guard (number? n)) n)
  ((a ,m ,n) (guard (number? m) (number? n)) (+ m n)))
```

Catamorphisms

The example below illustrates the use of explicit recursion within an `sxml-match` form. This example implements a simple calculator for the basic arithmetic operations, which are represented by the XML elements `plus`, `minus`, `times`, and `div`.

```
(define simple-eval
  (lambda (x)
    (sxml-match x
      [,i (guard (integer? i)) i]
      [(plus ,x ,y) (+ (simple-eval x) (simple-eval y))]
      [(times ,x ,y) (* (simple-eval x) (simple-eval y))]
      [(minus ,x ,y) (- (simple-eval x) (simple-eval y))]
      [(div ,x ,y) (/ (simple-eval x) (simple-eval y))]
      [,otherwise (error "simple-eval: invalid expression" x)])))
```

Using the catamorphism feature of `sxml-match`, a more concise version of `simple-eval` can be written. The pattern `,[x]` recursively invokes the pattern matcher on the value bound in this position.

```
(define simple-eval
  (lambda (x)
    (sxml-match x
      [,i (guard (integer? i)) i]
      [(plus ,[x] ,[y]) (+ x y)]
      [(times ,[x] ,[y]) (* x y)]
      [(minus ,[x] ,[y]) (- x y)]
      [(div ,[x] ,[y]) (/ x y)]
      [,otherwise (error "simple-eval: invalid expression" x)])))
```

Named-Catamorphisms

It is also possible to explicitly name the operator in the "cata" position. Where `,[id*]` recurs to the top of the current `sxml-match`, `,[cata -> id*]` recurs to `cata`. `cata` must evaluate to a procedure which takes one argument, and returns as many values as there are identifiers following `->`.

Named catamorphism patterns allow processing to be split into multiple, mutually recursive procedures. This is illustrated in the example below: a transformation that formats a "TV Guide" into HTML.

```
(define (tv-guide->html g)
  (define (cast-list cl)
    (sxml-match cl
      [(CastList (CastMember (Character (Name ,ch)) (Actor (Name ,a))) ...)
       `(div (ul (li ,ch ": " ,a) ...))]))
  (define (prog p)
    (sxml-match p
      [(Program (Start ,start-time) (Duration ,dur) (Series ,series-title)
                (Description ,desc ...))
       `(div (p ,start-time
                (br) ,series-title
```

```
                  (br) ,desc ...))]
          [(Program (Start ,start-time) (Duration ,dur) (Series ,series-title)
                    (Description ,desc ...)
                    ,[cast-list -> cl])
           `(div (p ,start-time
                    (br) ,series-title
                    (br) ,desc ...)
                 ,cl)]))
      (sxml-match g
        [(TVGuide (@ (start ,start-date)
                     (end ,end-date))
                  (Channel (Name ,nm) ,[prog -> p] ...) ...)
         `(html (head (title "TV Guide"))
                (body (h1 "TV Guide")
                      (div (h2 ,nm) ,p ...) ...))]))
```

sxml-match-let and sxml-match-let*

`sxml-match-let` ((*pat expr*) ...) *expression0 expression ...* [Scheme Syntax]
`sxml-match-let*` ((*pat expr*) ...) *expression0 expression ...* [Scheme Syntax]

These forms generalize the `let` and `let*` forms of Scheme to allow an XML pattern in the binding position, rather than a simple variable.

For example, the expression below:

```
(sxml-match-let ([(a ,i ,j) '(a 1 2)])
  (+ i j))
```

binds the variables *i* and *j* to `1` and `2` in the XML value given.

7.17 The Scheme shell (scsh)

An incomplete port of the Scheme shell (scsh) was once available for Guile as a separate package. However this code has bitrotten somewhat. The pieces are available in Guile's legacy CVS repository, which may be browsed at `http://cvs.savannah.gnu.org/viewvc/guile/guile-scsh/?root=guile`.

For information about scsh see `http://www.scsh.net/`.

This bitrotting is a bit of a shame, as there is a good deal of well-written Scheme code in scsh. Adopting this code and porting it to current Guile should be an educational experience, in addition to providing something of value to Guile folks.

7.18 Curried Definitions

The macros in this section are provided by

```
(use-modules (ice-9 curried-definitions))
```

and replace those provided by default.

Prior to Guile 2.0, Guile provided a type of definition known colloquially as a "curried definition". The idea is to extend the syntax of `define` so that you can conveniently define procedures that return procedures, up to any desired depth.

For example,

```
(define ((foo x) y)
  (list x y))
```

is a convenience form of

```
(define foo
  (lambda (x)
    (lambda (y)
      (list x y))))
```

`define` (. . . (*name args* . . .) . . .) *body* . . . [Scheme Syntax]
`define*` (. . . (*name args* . . .) . . .) *body* . . . [Scheme Syntax]
`define-public` (. . . (*name args* . . .) . . .) *body* . . . [Scheme Syntax]

Create a top level variable *name* bound to the procedure with parameter list *args*. If *name* is itself a formal parameter list, then a higher order procedure is created using that formal-parameter list, and returning a procedure that has parameter list *args*. This nesting may occur to arbitrary depth.

`define*` is similar but the formal parameter lists take additional options as described in Раздел 6.9.4.1 [lambda* and define*], страница 267. For example,

```
(define* ((foo #:keys (bar 'baz) (quux 'zot)) frotz #:rest rest)
  (list bar quux frotz rest))

((foo #:quux 'foo) 1 2 3 4 5)
⇒ (baz foo 1 (2 3 4 5))
```

`define-public` is similar to `define` but it also adds *name* to the list of exported bindings of the current module.

7.19 Statprof

Statprof is a statistical profiler for Guile.

A simple use of statprof would look like this:

```
(use-modules (statprof))
(statprof (lambda ()
            (map 1+ (iota 1000000))
            #f))
```

This would run the thunk with statistical profiling, finally displaying a flat table of statistics which could look something like this:

```
%     cumulative   self
time   seconds     seconds  procedure
 57.14  39769.73      0.07  ice-9/boot-9.scm:249:5:map1
 28.57      0.04      0.04  ice-9/boot-9.scm:1165:0:iota
 14.29      0.02      0.02  1+
  0.00      0.12      0.00  <current input>:2:10
---
Sample count: 7
Total time: 0.123490713 seconds (0.201983993 seconds in GC)
```

All of the numerical data with the exception of the calls column is statistically approximate. In the following column descriptions, and in all of statprof, "time" refers to execution time (both user and system), not wall clock time.

The `% time` column indicates the percentage of the run-time time spent inside the procedure itself (not counting children). It is calculated as `self seconds`, measuring the amount of time spent in the procedure, divided by the total run-time.

`cumulative seconds` also counts time spent in children of a function. For recursive functions, this can exceed the total time, as in our example above, because each activation on the stack adds to the cumulative time.

Finally, the GC time measures the time spent in the garbage collector. On systems with multiple cores, this time can be larger than the run time, because it counts time spent in all threads, and will run the "marking" phase of GC in parallel. If GC time is a significant fraction of the run time, that means that most time in your program is spent allocating objects and cleaning up after those allocations. To speed up your program, one good place to start would be to look at how to reduce the allocation rate.

Statprof's main mode of operation is as a statistical profiler. However statprof can also run in a "precise" mode as well. Pass the `#:count-calls? #t` keyword argument to `statprof` to record all calls:

```
(use-modules (statprof))
(statprof (lambda ()
            (map 1+ (iota 1000000))
            #f)
          #:count-calls? #t)
```

The result has an additional `calls` column:

```
%     cumulative   self
time   seconds    seconds   calls   procedure
 82.26      0.73      0.73 1000000  1+
 11.29 420925.80      0.10 1000001  ice-9/boot-9.scm:249:5:map1
  4.84      0.06      0.04       1  ice-9/boot-9.scm:1165:0:iota
[...]
---
Sample count: 62
Total time: 0.893098065 seconds (1.222796536 seconds in GC)
```

As you can see, the profile is perturbed: `1+` ends up on top, whereas it was not marked as hot in the earlier profile. This is because the overhead of call-counting unfairly penalizes calls. Still, this precise mode can be useful at times to do algorithmic optimizations based on the precise call counts.

Implementation notes

The profiler works by setting the unix profiling signal `ITIMER_PROF` to go off after the interval you define in the call to `statprof-reset`. When the signal fires, a sampling routine runs which crawls up the stack, recording all instruction pointers into a buffer. After the sample is complete, the profiler resets profiling timer to fire again after the appropriate interval.

Later, when profiling stops, that log buffer is analyzed to produce the "self seconds" and "cumulative seconds" statistics. A procedure at the top of the stack counts toward "self" samples, and everything on the stack counts towards "cumulative" samples.

While the profiler is running it measures how much CPU time (system and user – which is also what `ITIMER_PROF` tracks) has elapsed while code has been executing within the profiler. Only run time counts towards the profile, not wall-clock time. For example, sleeping and waiting for input or output do not cause the timer clock to advance.

Usage

`statprof` *thunk* **[***#:loop loop=1***]** **[***#:hz hz=100***]** **[***#:port port=(current-output-port)***]** **[***#:count-calls? count-calls?=#f***]** **[***#:display-style display-style='flat***]** [Scheme Procedure]

Profile the execution of *thunk*, and return its return values.

The stack will be sampled *hz* times per second, and the thunk itself will be called *loop* times.

If *count-calls?* is true, all procedure calls will be recorded. This operation is somewhat expensive.

After the *thunk* has been profiled, print out a profile to *port*. If *display-style* is `flat`, the results will be printed as a flat profile. Otherwise if *display-style* is `tree`, print the results as a tree profile.

Note that `statprof` requires a working profiling timer. Some platforms do not support profiling timers. `(provided? 'ITIMER_PROF)` can be used to check for support of profiling timers.

Profiling can also be enabled and disabled manually.

`statprof-active?` [Scheme Procedure]

Returns `#t` if `statprof-start` has been called more times than `statprof-stop`, `#f` otherwise.

`statprof-start` [Scheme Procedure]

`statprof-stop` [Scheme Procedure]

Start or stop the profiler.

`statprof-reset` *sample-seconds sample-microseconds count-calls?* [Scheme Procedure]

Reset the profiling sample interval to *sample-seconds* and *sample-microseconds*. If *count-calls?* is true, arrange to instrument procedure calls as well as collecting statistical profiling data.

If you use the manual `statprof-start`/`statprof-stop` interface, an implicit statprof state will persist starting from the last call to `statprof-reset`, or the first call to `statprof-start`. There are a number of accessors to fetch statistics from this implicit state.

`statprof-accumulated-time` [Scheme Procedure]

Returns the time accumulated during the last statprof run.

`statprof-sample-count` [Scheme Procedure]
Returns the number of samples taken during the last statprof run.

`statprof-fold-call-data` *proc init* [Scheme Procedure]
Fold *proc* over the call-data accumulated by statprof. This procedure cannot be called while statprof is active.

proc will be called with arguments, *call-data* and *prior-result*.

`statprof-proc-call-data` *proc* [Scheme Procedure]
Returns the call-data associated with *proc*, or `#f` if none is available.

`statprof-call-data-name` *cd* [Scheme Procedure]
`statprof-call-data-calls` *cd* [Scheme Procedure]
`statprof-call-data-cum-samples` *cd* [Scheme Procedure]
`statprof-call-data-self-samples` *cd* [Scheme Procedure]
Accessors for the fields in a statprof call-data object.

`statprof-call-data->stats` *call-data* [Scheme Procedure]
Returns an object of type `statprof-stats`.

`statprof-stats-proc-name` *stats* [Scheme Procedure]
`statprof-stats-%-time-in-proc` *stats* [Scheme Procedure]
`statprof-stats-cum-secs-in-proc` *stats* [Scheme Procedure]
`statprof-stats-self-secs-in-proc` *stats* [Scheme Procedure]
`statprof-stats-calls` *stats* [Scheme Procedure]
`statprof-stats-self-secs-per-call` *stats* [Scheme Procedure]
`statprof-stats-cum-secs-per-call` *stats* [Scheme Procedure]
Accessors for the fields in a `statprof-stats` object.

`statprof-display` [*port*=(*current-output-port*)] [*#:style style*=*flat*] [Scheme Procedure]
Displays a summary of the statistics collected. Possible values for *style* include:

`flat` Display a traditional gprof-style flat profile.

`anomalies`
Find statistical anomalies in the data.

`tree` Display a tree profile.

`statprof-fetch-stacks` [Scheme Procedure]
Returns a list of stacks, as they were captured since the last call to `statprof-reset`.

`statprof-fetch-call-tree` [*#:precise precise?*=*#f*] [Scheme Procedure]
```
Return a call tree for the previous statprof run.

The return value is a list of nodes.  A node is a list of the form:
@code
 node ::= (@var{proc} @var{count} . @var{nodes})
@end code
```

The @var{proc} is a printable representation of a procedure, as a string. If @var{precise?} is false, which is the default, then a node corresponds to a procedure invocation. If it is true, then a node corresponds to a return point in a procedure. Passing @code{#:precise? #t} allows a user to distinguish different source lines in a procedure, but usually it is too much detail, so it is off by default.

`gcprof` *thunk* [*#:loop*] [Scheme Procedure]

Like the `statprof` procedure, but instead of profiling CPU time, we profile garbage collection.

The stack will be sampled soon after every garbage collection during the evaluation of *thunk*, yielding an approximate idea of what is causing allocation in your program.

Since GC does not occur very frequently, you may need to use the *loop* parameter, to cause *thunk* to be called *loop* times.

7.20 SXML

SXML is a native representation of XML in terms of standard Scheme data types: lists, symbols, and strings. For example, the simple XML fragment:

```
<parrot type="African Grey"><name>Alfie</name></parrot>
```

may be represented with the following SXML:

```
(parrot (@ (type "African Grey")) (name "Alfie"))
```

SXML is very general, and is capable of representing all of XML. Formally, this means that SXML is a conforming implementation of the http://www.w3.org/TR/xml-infoset/ (`XML Information Set`) standard.

Guile includes several facilities for working with XML and SXML: parsers, serializers, and transformers.

7.20.1 SXML Overview

(This section needs to be written; volunteers welcome.)

7.20.2 Reading and Writing XML

The `(sxml simple)` module presents a basic interface for parsing XML from a port into the Scheme SXML format, and for serializing it back to text.

```
(use-modules (sxml simple))
```

`xml->sxml` [*string-or-port*] [*#:namespaces='()*] [Scheme Procedure]
[*#:declare-namespaces?=#t*] [*#:trim-whitespace?=#f*] [*#:entities='()*]
[*#:default-entity-handler=#f*] [*#:doctype-handler=#f*]

Use SSAX to parse an XML document into SXML. Takes one optional argument, *string-or-port*, which defaults to the current input port. Returns the resulting SXML document. If *string-or-port* is a port, it will be left pointing at the next available character in the port.

As is normal in SXML, XML elements parse as tagged lists. Attributes, if any, are placed after the tag, within an `@` element. The root of the resulting XML will be contained

in a special tag, `*TOP*`. This tag will contain the root element of the XML, but also any prior processing instructions.

```
(xml->sxml "<foo/>")
⇒ (*TOP* (foo))
(xml->sxml "<foo>text</foo>")
⇒ (*TOP* (foo "text"))
(xml->sxml "<foo kind=\"bar\">text</foo>")
⇒ (*TOP* (foo (@ (kind "bar")) "text"))
(xml->sxml "<?xml version=\"1.0\"?><foo/>")
⇒ (*TOP* (*PI* xml "version=\"1.0\"") (foo))
```

All namespaces in the XML document must be declared, via `xmlns` attributes. SXML elements built from non-default namespaces will have their tags prefixed with their URI. Users can specify custom prefixes for certain namespaces with the `#:namespaces` keyword argument to `xml->sxml`.

```
(xml->sxml "<foo xmlns=\"http://example.org/ns1\">text</foo>")
⇒ (*TOP* (http://example.org/ns1:foo "text"))
(xml->sxml "<foo xmlns=\"http://example.org/ns1\">text</foo>"
           #:namespaces '((ns1 . "http://example.org/ns1")))
⇒ (*TOP* (ns1:foo "text"))
(xml->sxml "<foo xmlns:bar=\"http://example.org/ns2\"><bar:baz/></foo>"
           #:namespaces '((ns2 . "http://example.org/ns2")))
⇒ (*TOP* (foo (ns2:baz)))
```

By default, namespaces passed to `xml->sxml` are treated as if they were declared on the root element. Passing a false `#:declare-namespaces?` argument will disable this behavior, requiring in-document declarations of namespaces before use..

```
(xml->sxml "<foo><ns2:baz/></foo>"
           #:namespaces '((ns2 . "http://example.org/ns2")))
⇒ (*TOP* (foo (ns2:baz)))
(xml->sxml "<foo><ns2:baz/></foo>"
           #:namespaces '((ns2 . "http://example.org/ns2"))
           #:declare-namespaces? #f)
⇒ error: undeclared namespace: `bar'
```

By default, all whitespace in XML is significant. Passing the `#:trim-whitespace?` keyword argument to `xml->sxml` will trim whitespace in front, behind and between elements, treating it as "unsignificant". Whitespace in text fragments is left alone.

```
(xml->sxml "<foo>\n<bar> Alfie the parrot! </bar>\n</foo>")
⇒ (*TOP* (foo "\n" (bar " Alfie the parrot! ") "\n"))
(xml->sxml "<foo>\n<bar> Alfie the parrot! </bar>\n</foo>"
           #:trim-whitespace? #t)
⇒ (*TOP* (foo (bar " Alfie the parrot! ")))
```

Parsed entities may be declared with the `#:entities` keyword argument, or handled with the `#:default-entity-handler`. By default, only the standard `<`, `>`, `&`, `&apos;` and `&quot;` entities are defined, as well as the `&#N;` and `&#xN;` (decimal and hexadecimal) numeric character entities.

```
(xml->sxml "<foo>&amp;</foo>")
```

```
⇒ (*TOP* (foo "&"))
(xml->sxml "<foo> </foo>")
⇒ error: undefined entity: nbsp
(xml->sxml "<foo> </foo>")
⇒ (*TOP* (foo "\xa0"))
(xml->sxml "<foo> </foo>"
           #:entities '((nbsp . "\xa0")))
⇒ (*TOP* (foo "\xa0"))
(xml->sxml "<foo>  &foo;</foo>"
           #:default-entity-handler
           (lambda (port name)
             (case name
               ((nbsp) "\xa0")
               (else
                (format (current-warning-port)
                        "~a:~a:~a: undefined entitity: ~a\n"
                        (or (port-filename port) "<unknown file>")
                        (port-line port) (port-column port)
                        name)
                (symbol->string name)))))
⊣ <unknown file>:0:17: undefined entitity: foo
⇒ (*TOP* (foo "\xa0 foo"))
```

By default, `xml->sxml` skips over the `<!DOCTYPE>` declaration, if any. This behavior can be overridden with the `#:doctype-handler` argument, which should be a procedure of three arguments: the *docname* (a symbol), *systemid* (a string), and the internal doctype subset (as a string or `#f` if not present).

The handler should return keyword arguments as multiple values, as if it were calling its continuation with keyword arguments. The continuation accepts the `#:entities` and `#:namespaces` keyword arguments, in the same format that `xml->sxml` itself takes. These entities and namespaces will be prepended to those given to the `xml->sxml` invocation.

```
(define (handle-foo docname systemid internal-subset)
  (case docname
    ((foo)
     (values #:entities '((greets . "<i>Hello, world!</i>"))))
    (else
     (values))))

(xml->sxml "<!DOCTYPE foo><p>&greets;</p>"
           #:doctype-handler handle-foo)
⇒ (*TOP* (p (i "Hello, world!")))
```

If the document has no doctype declaration, the *doctype-handler* is invoked with `#f` for the three arguments.

In the future, the continuation may accept other keyword arguments, for example to validate the parsed SXML against the doctype.

`sxml->xml` *tree* [*port*] [Scheme Procedure]
Serialize the SXML tree *tree* as XML. The output will be written to the current output port, unless the optional argument *port* is present.

`sxml->string` *sxml* [Scheme Procedure]
Detag an sxml tree *sxml* into a string. Does not perform any formatting.

7.20.3 SSAX: A Functional XML Parsing Toolkit

Guile's XML parser is based on Oleg Kiselyov's powerful XML parsing toolkit, SSAX.

7.20.3.1 History

Back in the 1990s, when the world was young again and XML was the solution to all of its problems, there were basically two kinds of XML parsers out there: DOM parsers and SAX parsers.

A DOM parser reads through an entire XML document, building up a tree of "DOM objects" representing the document structure. They are very easy to use, but sometimes you don't actually want all of the information in a document; building an object tree is not necessary if all you want to do is to count word frequencies in a document, for example.

SAX parsers were created to give the programmer more control on the parsing process. A programmer gives the SAX parser a number of "callbacks": functions that will be called on various features of the XML stream as they are encountered. SAX parsers are more efficient, but much harder to user, as users typically have to manually maintain a stack of open elements.

Kiselyov realized that the SAX programming model could be made much simpler if the callbacks were formulated not as a linear fold across the features of the XML stream, but as a *tree fold* over the structure implicit in the XML. In this way, the user has a very convenient, functional-style interface that can still generate optimal parsers.

The `xml->sxml` interface from the `(sxml simple)` module is a DOM-style parser built using SSAX, though it returns SXML instead of DOM objects.

7.20.3.2 Implementation

`(sxml ssax)` is a package of low-to-high level lexing and parsing procedures that can be combined to yield a SAX, a DOM, a validating parser, or a parser intended for a particular document type. The procedures in the package can be used separately to tokenize or parse various pieces of XML documents. The package supports XML Namespaces, internal and external parsed entities, user-controlled handling of whitespace, and validation. This module therefore is intended to be a framework, a set of "Lego blocks" you can use to build a parser following any discipline and performing validation to any degree. As an example of the parser construction, the source file includes a semi-validating SXML parser.

SSAX has a "sequential" feel of SAX yet a "functional style" of DOM. Like a SAX parser, the framework scans the document only once and permits incremental processing. An application that handles document elements in order can run as efficiently as possible. *Unlike* a SAX parser, the framework does not require an application register stateful callbacks and surrender control to the parser. Rather, it is the application that can drive the framework – calling its functions to get the current lexical or syntax element. These functions do not maintain or mutate any state save the input port. Therefore, the

framework permits parsing of XML in a pure functional style, with the input port being a monad (or a linear, read-once parameter).

Besides the *port*, there is another monad – *seed*. Most of the middle- and high-level parsers are single-threaded through the *seed*. The functions of this framework do not process or affect the *seed* in any way: they simply pass it around as an instance of an opaque datatype. User functions, on the other hand, can use the seed to maintain user's state, to accumulate parsing results, etc. A user can freely mix their own functions with those of the framework. On the other hand, the user may wish to instantiate a high-level parser: `SSAX:make-elem-parser` or `SSAX:make-parser`. In the latter case, the user must provide functions of specific signatures, which are called at predictable moments during the parsing: to handle character data, element data, or processing instructions (PI). The functions are always given the *seed*, among other parameters, and must return the new *seed*.

From a functional point of view, XML parsing is a combined pre-post-order traversal of a "tree" that is the XML document itself. This down-and-up traversal tells the user about an element when its start tag is encountered. The user is notified about the element once more, after all element's children have been handled. The process of XML parsing therefore is a fold over the raw XML document. Unlike a fold over trees defined in [1], the parser is necessarily single-threaded – obviously as elements in a text XML document are laid down sequentially. The parser therefore is a tree fold that has been transformed to accept an accumulating parameter [1,2].

Formally, the denotational semantics of the parser can be expressed as

```
parser:: (Start-tag -> Seed -> Seed) ->
  (Start-tag -> Seed -> Seed -> Seed) ->
  (Char-Data -> Seed -> Seed) ->
  XML-text-fragment -> Seed -> Seed
parser fdown fup fchar "<elem attrs> content </elem>" seed
 = fup "<elem attrs>" seed
(parser fdown fup fchar "content" (fdown "<elem attrs>" seed))

parser fdown fup fchar "char-data content" seed
 = parser fdown fup fchar "content" (fchar "char-data" seed)

parser fdown fup fchar "elem-content content" seed
 = parser fdown fup fchar "content" (
parser fdown fup fchar "elem-content" seed)
```

Compare the last two equations with the left fold

```
fold-left kons elem:list seed = fold-left kons list (kons elem seed)
```

The real parser created by `SSAX:make-parser` is slightly more complicated, to account for processing instructions, entity references, namespaces, processing of document type declaration, etc.

The XML standard document referred to in this module is `http://www.w3.org/TR/1998/REC-xml-19980210.html`

The present file also defines a procedure that parses the text of an XML document or of a separate element into SXML, an S-expression-based model of an XML Information Set. SXML is also an Abstract Syntax Tree of an XML document. SXML is similar but not identical to DOM; SXML is particularly suitable for Scheme-based XML/HTML authoring, SXPath queries, and tree transformations. See SXML.html for more details. SXML is a

term implementation of evaluation of the XML document [3]. The other implementation is context-passing.

The present frameworks fully supports the XML Namespaces Recommendation: `http://www.w3.org/TR/REC-xml-names/`.

Other links:

[1] Jeremy Gibbons, Geraint Jones, "The Under-appreciated Unfold," Proc. ICFP'98, 1998, pp. 273-279.

[2] Richard S. Bird, The promotion and accumulation strategies in transformational programming, ACM Trans. Progr. Lang. Systems, 6(4):487-504, October 1984.

[3] Ralf Hinze, "Deriving Backtracking Monad Transformers," Functional Pearl. Proc ICFP'00, pp. 186-197.

7.20.3.3 Usage

`current-ssax-error-port` [Scheme Procedure]

`with-ssax-error-to-port` *port thunk* [Scheme Procedure]

`xml-token? _` [Scheme Procedure]

```
-- Scheme Procedure: pair? x
    Return `#t' if X is a pair; otherwise return `#f'.
```

`xml-token-kind` *token* [Scheme Syntax]

`xml-token-head` *token* [Scheme Syntax]

`make-empty-attlist` [Scheme Procedure]

`attlist-add` *attlist name-value* [Scheme Procedure]

`attlist-null?` *x* [Scheme Procedure]

Return `#t` if *x* is the empty list, else `#f`.

`attlist-remove-top` *attlist* [Scheme Procedure]

`attlist->alist` *attlist* [Scheme Procedure]

`attlist-fold` *kons knil lis1* [Scheme Procedure]

`define-parsed-entity!` *entity str* [Scheme Procedure]

Define a new parsed entity. *entity* should be a symbol.

Instances of &*entity*; in XML text will be replaced with the string *str*, which will then be parsed.

`reset-parsed-entity-definitions!` [Scheme Procedure]

Restore the set of parsed entity definitions to its initial state.

`ssax:uri-string->symbol` *uri-str* [Scheme Procedure]

`ssax:skip-internal-dtd` *port* [Scheme Procedure]

`ssax:read-pi-body-as-string` *port* [Scheme Procedure]

`ssax:reverse-collect-str-drop-ws` *fragments* [Scheme Procedure]

`ssax:read-markup-token` *port* [Scheme Procedure]

`ssax:read-cdata-body` *port str-handler seed* [Scheme Procedure]

`ssax:read-char-ref` *port* [Scheme Procedure]

`ssax:read-attributes` *port entities* [Scheme Procedure]

`ssax:complete-start-tag` *tag-head port elems entities namespaces* [Scheme Procedure]

`ssax:read-external-id` *port* [Scheme Procedure]

`ssax:read-char-data` *port expect-eof? str-handler seed* [Scheme Procedure]

`ssax:xml->sxml` *port namespace-prefix-assig* [Scheme Procedure]

`ssax:make-parser` . *kw-val-pairs* [Scheme Syntax]

`ssax:make-pi-parser` *orig-handlers* [Scheme Syntax]

`ssax:make-elem-parser` *my-new-level-seed my-finish-element my-char-data-handler my-pi-handlers* [Scheme Syntax]

7.20.4 Transforming SXML

7.20.4.1 Overview

SXML expression tree transformers

Pre-Post-order traversal of a tree and creation of a new tree

```
pre-post-order:: <tree> x <bindings> -> <new-tree>
```

where

```
 <bindings> ::= (<binding> ...)
 <binding> ::= (<trigger-symbol> *preorder* . <handler>) |
               (<trigger-symbol> *macro* . <handler>) |
(<trigger-symbol> <new-bindings> . <handler>) |
(<trigger-symbol> . <handler>)
 <trigger-symbol> ::= XMLname | *text* | *default*
 <handler> :: <trigger-symbol> x [<tree>] -> <new-tree>
```

The pre-post-order function visits the nodes and nodelists pre-post-order (depth-first). For each `<Node>` of the form `(`*name* `<Node> ...)`, it looks up an association with the given *name* among its <*bindings*>. If failed, `pre-post-order` tries to locate a `*default*` binding. It's an error if the latter attempt fails as well. Having found a binding, the `pre-post-order` function first checks to see if the binding is of the form

```
(<trigger-symbol> *preorder* . <handler>)
```

If it is, the handler is 'applied' to the current node. Otherwise, the pre-post-order function first calls itself recursively for each child of the current node, with <*new-bindings*> prepended to the <*bindings*> in effect. The result of these calls is passed to the <*handler*> (along with the head of the current <*Node*>). To be more precise, the handler is _applied_ to the head of the current node and its processed children. The result of the handler, which

should also be a `<tree>`, replaces the current <*Node*>. If the current <*Node*> is a text string or other atom, a special binding with a symbol `*text*` is looked up.

A binding can also be of a form

```
(<trigger-symbol> *macro* . <handler>)
```

This is equivalent to `*preorder*` described above. However, the result is re-processed again, with the current stylesheet.

7.20.4.2 Usage

`SRV:send-reply` . *fragments* [Scheme Procedure]

Output the *fragments* to the current output port.

The fragments are a list of strings, characters, numbers, thunks, `#f`, `#t` – and other fragments. The function traverses the tree depth-first, writes out strings and characters, executes thunks, and ignores `#f` and `'()`. The function returns `#t` if anything was written at all; otherwise the result is `#f` If `#t` occurs among the fragments, it is not written out but causes the result of `SRV:send-reply` to be `#t`.

`foldts` *fdown fup fhere seed tree* [Scheme Procedure]

`post-order` *tree bindings* [Scheme Procedure]

`pre-post-order` *tree bindings* [Scheme Procedure]

`replace-range` *beg-pred end-pred forest* [Scheme Procedure]

7.20.5 SXML Tree Fold

7.20.5.1 Overview

`(sxml fold)` defines a number of variants of the *fold* algorithm for use in transforming SXML trees. Additionally it defines the layout operator, `fold-layout`, which might be described as a context-passing variant of SSAX's `pre-post-order`.

7.20.5.2 Usage

`foldt` *fup fhere tree* [Scheme Procedure]

The standard multithreaded tree fold.

fup is of type [a] -> a. *fhere* is of type object -> a.

`foldts` *fdown fup fhere seed tree* [Scheme Procedure]

The single-threaded tree fold originally defined in SSAX. См. Раздел 7.20.3 [SSAX], страница 764, for more information.

`foldts*` *fdown fup fhere seed tree* [Scheme Procedure]

A variant of `foldts` that allows pre-order tree rewrites. Originally defined in Andy Wingo's 2007 paper, *Applications of fold to XML transformation*.

`fold-values` *proc list . seeds* [Scheme Procedure]

A variant of `fold` that allows multi-valued seeds. Note that the order of the arguments differs from that of `fold`. См. Раздел 7.5.3.5 [SRFI-1 Fold and Map], страница 611.

`foldts*-values` *fdown fup fhere tree . seeds* [Scheme Procedure]
A variant of `foldts*` that allows multi-valued seeds. Originally defined in Andy Wingo's 2007 paper, *Applications of fold to XML transformation*.

`fold-layout` *tree bindings params layout stylesheet* [Scheme Procedure]
A traversal combinator in the spirit of `pre-post-order`. См. Раздел 7.20.4 [Transforming SXML], страница 767.

`fold-layout` was originally presented in Andy Wingo's 2007 paper, *Applications of fold to XML transformation*.

```
bindings := (<binding>...)
binding  := (<tag> <handler-pair>...)
          | (*default* . <post-handler>)
          | (*text* . <text-handler>)
tag      := <symbol>
handler-pair := (pre-layout . <pre-layout-handler>)
          | (post . <post-handler>)
          | (bindings . <bindings>)
          | (pre . <pre-handler>)
          | (macro . <macro-handler>)
```

pre-layout-handler
A function of three arguments:

kids the kids of the current node, before traversal

params the params of the current node

layout the layout coming into this node

pre-layout-handler is expected to use this information to return a layout to pass to the kids. The default implementation returns the layout given in the arguments.

post-handler
A function of five arguments:

tag the current tag being processed

params the params of the current node

layout the layout coming into the current node, before any kids were processed

klayout the layout after processing all of the children

kids the already-processed child nodes

post-handler should return two values, the layout to pass to the next node and the final tree.

text-handler
text-handler is a function of three arguments:

text the string

params the current params

layout the current layout

text-handler should return two values, the layout to pass to the next node and the value to which the string should transform.

7.20.6 SXPath

7.20.6.1 Overview

SXPath: SXML Query Language

SXPath is a query language for SXML, an instance of XML Information set (Infoset) in the form of s-expressions. See `(sxml ssax)` for the definition of SXML and more details. SXPath is also a translation into Scheme of an XML Path Language, XPath (`http://www.w3.org/TR/xpath`). XPath and SXPath describe means of selecting a set of Infoset's items or their properties.

To facilitate queries, XPath maps the XML Infoset into an explicit tree, and introduces important notions of a location path and a current, context node. A location path denotes a selection of a set of nodes relative to a context node. Any XPath tree has a distinguished, root node – which serves as the context node for absolute location paths. Location path is recursively defined as a location step joined with a location path. A location step is a simple query of the database relative to a context node. A step may include expressions that further filter the selected set. Each node in the resulting set is used as a context node for the adjoining location path. The result of the step is a union of the sets returned by the latter location paths.

The SXML representation of the XML Infoset (see SSAX.scm) is rather suitable for querying as it is. Bowing to the XPath specification, we will refer to SXML information items as 'Nodes':

```
<Node> ::= <Element> | <attributes-coll> | <attrib>
   | "text string" | <PI>
```

This production can also be described as

```
<Node> ::= (name . <Nodeset>) | "text string"
```

An (ordered) set of nodes is just a list of the constituent nodes:

```
<Nodeset> ::= (<Node> ...)
```

Nodesets, and Nodes other than text strings are both lists. A <Nodeset> however is either an empty list, or a list whose head is not a symbol. A symbol at the head of a node is either an XML name (in which case it's a tag of an XML element), or an administrative name such as '@'. This uniform list representation makes processing rather simple and elegant, while avoiding confusion. The multi-branch tree structure formed by the mutually-recursive datatypes <Node> and <Nodeset> lends itself well to processing by functional languages.

A location path is in fact a composite query over an XPath tree or its branch. A singe step is a combination of a projection, selection or a transitive closure. Multiple steps are combined via join and union operations. This insight allows us to *elegantly* implement XPath as a sequence of projection and filtering primitives – converters – joined

by *combinators*. Each converter takes a node and returns a nodeset which is the result of the corresponding query relative to that node. A converter can also be called on a set of nodes. In that case it returns a union of the corresponding queries over each node in the set. The union is easily implemented as a list append operation as all nodes in a SXML tree are considered distinct, by XPath conventions. We also preserve the order of the members in the union. Query combinators are high-order functions: they take converter(s) (which is a Node|Nodeset -> Nodeset function) and compose or otherwise combine them. We will be concerned with only relative location paths [XPath]: an absolute location path is a relative path applied to the root node.

Similarly to XPath, SXPath defines full and abbreviated notations for location paths. In both cases, the abbreviated notation can be mechanically expanded into the full form by simple rewriting rules. In the case of SXPath the corresponding rules are given in the documentation of the `sxpath` procedure. См. [SXPath procedure documentation], страница 774.

The regression test suite at the end of the file `SXPATH-old.scm` shows a representative sample of SXPaths in both notations, juxtaposed with the corresponding XPath expressions. Most of the samples are borrowed literally from the XPath specification.

Much of the following material is taken from the SXPath sources by Oleg Kiselyov et al.

7.20.6.2 Basic Converters and Applicators

A converter is a function mapping a nodeset (or a single node) to another nodeset. Its type can be represented like this:

```
type Converter = Node|Nodeset -> Nodeset
```

A converter can also play the role of a predicate: in that case, if a converter, applied to a node or a nodeset, yields a non-empty nodeset, the converter-predicate is deemed satisfied. Likewise, an empty nodeset is equivalent to `#f` in denoting failure.

`nodeset?` *x* [Scheme Procedure]

Return `#t` if *x* is a nodeset.

`node-typeof?` *crit* [Scheme Procedure]

This function implements a 'Node test' as defined in Sec. 2.3 of the XPath document. A node test is one of the components of a location step. It is also a converter-predicate in SXPath.

The function `node-typeof?` takes a type criterion and returns a function, which, when applied to a node, will tell if the node satisfies the test.

The criterion *crit* is a symbol, one of the following:

`id`	tests if the node has the right name (id)
`@`	tests if the node is an <attributes-coll>
`*`	tests if the node is an <Element>
`*text*`	tests if the node is a text node
`*PI*`	tests if the node is a PI (processing instruction) node
`*any*`	`#t` for any type of node

`node-eq?` *other* [Scheme Procedure]

A curried equivalence converter predicate that takes a node *other* and returns a function that takes another node. The two nodes are compared using `eq?`.

`node-equal?` *other* [Scheme Procedure]

A curried equivalence converter predicate that takes a node *other* and returns a function that takes another node. The two nodes are compared using `equal?`.

`node-pos` *n* [Scheme Procedure]

Select the *n*'th element of a nodeset and return as a singular nodeset. If the *n*'th element does not exist, return an empty nodeset. If *n* is a negative number the node is picked from the tail of the list.

```
((node-pos 1) nodeset)  ; return the the head of the nodeset (if exists)
((node-pos 2) nodeset)  ; return the node after that (if exists)
((node-pos -1) nodeset) ; selects the last node of a non-empty nodeset
((node-pos -2) nodeset) ; selects the last but one node, if exists.
```

`filter` *pred?* [Scheme Procedure]

A filter applicator, which introduces a filtering context. The argument converter *pred?* is considered a predicate, with either `#f` or `nil` meaning failure.

`take-until` *pred?* [Scheme Procedure]

```
take-until:: Converter -> Converter, or
take-until:: Pred -> Node|Nodeset -> Nodeset
```

Given a converter-predicate *pred?* and a nodeset, apply the predicate to each element of the nodeset, until the predicate yields anything but `#f` or `nil`. Return the elements of the input nodeset that have been processed until that moment (that is, which fail the predicate).

`take-until` is a variation of the `filter` above: `take-until` passes elements of an ordered input set up to (but not including) the first element that satisfies the predicate. The nodeset returned by `((take-until (not pred)) nset)` is a subset – to be more precise, a prefix – of the nodeset returned by `((filter pred) nset)`.

`take-after` *pred?* [Scheme Procedure]

```
take-after:: Converter -> Converter, or
take-after:: Pred -> Node|Nodeset -> Nodeset
```

Given a converter-predicate *pred?* and a nodeset, apply the predicate to each element of the nodeset, until the predicate yields anything but `#f` or `nil`. Return the elements of the input nodeset that have not been processed: that is, return the elements of the input nodeset that follow the first element that satisfied the predicate.

`take-after` along with `take-until` partition an input nodeset into three parts: the first element that satisfies a predicate, all preceding elements and all following elements.

`map-union` *proc lst* [Scheme Procedure]

Apply *proc* to each element of *lst* and return the list of results. If *proc* returns a nodeset, splice it into the result

From another point of view, `map-union` is a function `Converter->Converter`, which places an argument-converter in a joining context.

`node-reverse` *node-or-nodeset* [Scheme Procedure]

```
node-reverse :: Converter, or
node-reverse:: Node|Nodeset -> Nodeset
```

Reverses the order of nodes in the nodeset. This basic converter is needed to implement a reverse document order (see the XPath Recommendation).

`node-trace` *title* [Scheme Procedure]

```
node-trace:: String -> Converter
```

`(node-trace title)` is an identity converter. In addition it prints out the node or nodeset it is applied to, prefixed with the *title*. This converter is very useful for debugging.

7.20.6.3 Converter Combinators

Combinators are higher-order functions that transmogrify a converter or glue a sequence of converters into a single, non-trivial converter. The goal is to arrive at converters that correspond to XPath location paths.

From a different point of view, a combinator is a fixed, named *pattern* of applying converters. Given below is a complete set of such patterns that together implement XPath location path specification. As it turns out, all these combinators can be built from a small number of basic blocks: regular functional composition, `map-union` and `filter` applicators, and the nodeset union.

`select-kids` *test-pred?* [Scheme Procedure]

`select-kids` takes a converter (or a predicate) as an argument and returns another converter. The resulting converter applied to a nodeset returns an ordered subset of its children that satisfy the predicate *test-pred?*.

`node-self` *pred?* [Scheme Procedure]

Similar to `select-kids` except that the predicate *pred?* is applied to the node itself rather than to its children. The resulting nodeset will contain either one component, or will be empty if the node failed the predicate.

`node-join` . *selectors* [Scheme Procedure]

```
node-join:: [LocPath] -> Node|Nodeset -> Nodeset, or
node-join:: [Converter] -> Converter
```

Join the sequence of location steps or paths as described above.

`node-reduce` . *converters* [Scheme Procedure]

```
node-reduce:: [LocPath] -> Node|Nodeset -> Nodeset, or
node-reduce:: [Converter] -> Converter
```

A regular functional composition of converters. From a different point of view, `((apply node-reduce converters) nodeset)` is equivalent to `(foldl apply nodeset converters)`, i.e., folding, or reducing, a list of converters with the nodeset as a seed.

`node-or` . *converters* [Scheme Procedure]

```
node-or:: [Converter] -> Converter
```

This combinator applies all converters to a given node and produces the union of their results. This combinator corresponds to a union (| operation) for XPath location paths.

`node-closure` *test-pred?* [Scheme Procedure]

```
node-closure:: Converter -> Converter
```

Select all *descendants* of a node that satisfy a converter-predicate *test-pred?*. This combinator is similar to `select-kids` but applies to grand... children as well. This combinator implements the `descendant::` XPath axis. Conceptually, this combinator can be expressed as

```
(define (node-closure f)
  (node-or
    (select-kids f)
    (node-reduce (select-kids (node-typeof? '*)) (node-closure f))))
```

This definition, as written, looks somewhat like a fixpoint, and it will run forever. It is obvious however that sooner or later `(select-kids (node-typeof? '*))` will return an empty nodeset. At this point further iterations will no longer affect the result and can be stopped.

`node-parent` *rootnode* [Scheme Procedure]

```
node-parent:: RootNode -> Converter
```

`(node-parent rootnode)` yields a converter that returns a parent of a node it is applied to. If applied to a nodeset, it returns the list of parents of nodes in the nodeset. The *rootnode* does not have to be the root node of the whole SXML tree – it may be a root node of a branch of interest.

Given the notation of Philip Wadler's paper on semantics of XSLT,

```
parent(x) = { y | y=subnode*(root), x=subnode(y) }
```

Therefore, `node-parent` is not the fundamental converter: it can be expressed through the existing ones. Yet `node-parent` is a rather convenient converter. It corresponds to a `parent::` axis of SXPath. Note that the `parent::` axis can be used with an attribute node as well.

`sxpath` *path* [Scheme Procedure]

Evaluate an abbreviated SXPath.

```
sxpath:: AbbrPath -> Converter, or
sxpath:: AbbrPath -> Node|Nodeset -> Nodeset
```

path is a list. It is translated to the full SXPath according to the following rewriting rules:

```
(sxpath '())
⇒ (node-join)

(sxpath '(path-component ...))
⇒ (node-join (sxpath1 path-component) (sxpath '(...)))

(sxpath1 '//)
⇒ (node-or
```

```
   (node-self (node-typeof? '*any*))
   (node-closure (node-typeof? '*any*)))

(sxpath1 '(equal? x))
⇒ (select-kids (node-equal? x))

(sxpath1 '(eq? x))
⇒ (select-kids (node-eq? x))

(sxpath1 ?symbol)
⇒ (select-kids (node-typeof? ?symbol)

(sxpath1 procedure)
⇒ procedure

(sxpath1 '(?symbol ...))
⇒ (sxpath1 '((?symbol) ...))

(sxpath1 '(path reducer ...))
⇒ (node-reduce (sxpath path) (sxpathr reducer) ...)

(sxpathr number)
⇒ (node-pos number)

(sxpathr path-filter)
⇒ (filter (sxpath path-filter))
```

7.20.7 (sxml ssax input-parse)

7.20.7.1 Overview

A simple lexer.

The procedures in this module surprisingly often suffice to parse an input stream. They either skip, or build and return tokens, according to inclusion or delimiting semantics. The list of characters to expect, include, or to break at may vary from one invocation of a function to another. This allows the functions to easily parse even context-sensitive languages.

EOF is generally frowned on, and thrown up upon if encountered. Exceptions are mentioned specifically. The list of expected characters (characters to skip until, or break-characters) may include an EOF "character", which is to be coded as the symbol, `*eof*`.

The input stream to parse is specified as a *port*, which is usually the last (and optional) argument. It defaults to the current input port if omitted.

If the parser encounters an error, it will throw an exception to the key `parser-error`. The arguments will be of the form (*port message specialising-msg**).

The first argument is a port, which typically points to the offending character or its neighborhood. You can then use `port-column` and `port-line` to query the current position.

message is the description of the error. Other arguments supply more details about the problem.

7.20.7.2 Usage

`peek-next-char` [*port*] [Scheme Procedure]

`assert-curr-char` *expected-chars comment* [*port*] [Scheme Procedure]

`skip-until` *arg* [*port*] [Scheme Procedure]

`skip-while` *skip-chars* [*port*] [Scheme Procedure]

`next-token` *prefix-skipped-chars break-chars* [*comment*] [*port*] [Scheme Procedure]

`next-token-of` *incl-list/pred* [*port*] [Scheme Procedure]

`read-text-line` [*port*] [Scheme Procedure]

`read-string` *n* [*port*] [Scheme Procedure]

`find-string-from-port?` _ _ . _ [Scheme Procedure]

Looks for *str* in <*input-port*>, optionally within the first *max-no-char* characters.

7.20.8 (sxml apply-templates)

7.20.8.1 Overview

Pre-order traversal of a tree and creation of a new tree:

```
apply-templates:: tree x <templates> -> <new-tree>
```

where

```
<templates> ::= (<template> ...)
<template>  ::= (<node-test> <node-test> ... <node-test> . <handler>)
<node-test> ::= an argument to node-typeof? above
<handler>   ::= <tree> -> <new-tree>
```

This procedure does a *normal*, pre-order traversal of an SXML tree. It walks the tree, checking at each node against the list of matching templates.

If the match is found (which must be unique, i.e., unambiguous), the corresponding handler is invoked and given the current node as an argument. The result from the handler, which must be a `<tree>`, takes place of the current node in the resulting tree. The name of the function is not accidental: it resembles rather closely an `apply-templates` function of XSLT.

7.20.8.2 Usage

`apply-templates` *tree templates* [Scheme Procedure]

7.21 Texinfo Processing

7.21.1 (texinfo)

7.21.1.1 Overview

Texinfo processing in scheme

This module parses texinfo into SXML. TeX will always be the processor of choice for print output, of course. However, although `makeinfo` works well for info, its output in other formats is not very customizable, and the program is not extensible as a whole. This module aims to provide an extensible framework for texinfo processing that integrates texinfo into the constellation of SXML processing tools.

Notes on the SXML vocabulary

Consider the following texinfo fragment:

```
@deffn Primitive set-car! pair value
This function...
@end deffn
```

Logically, the category (Primitive), name (set-car!), and arguments (pair value) are "attributes" of the deffn, with the description as the content. However, texinfo allows for @-commands within the arguments to an environment, like `@deffn`, which means that texinfo "attributes" are PCDATA. XML attributes, on the other hand, are CDATA. For this reason, "attributes" of texinfo @-commands are called "arguments", and are grouped under the special element, '%'.

Because '%' is not a valid NCName, stexinfo is a superset of SXML. In the interests of interoperability, this module provides a conversion function to replace the '%' with 'texinfo-arguments'.

7.21.1.2 Usage

`call-with-file-and-dir` *filename proc* [Функция]

Call the one-argument procedure *proc* with an input port that reads from *filename*. During the dynamic extent of *proc*'s execution, the current directory will be `(dirname` *`filename`*`)`. This is useful for parsing documents that can include files by relative path name.

`texi-command-specs` [Переменная]

`texi-command-depth` *command max-depth* [Функция]

Given the texinfo command *command*, return its nesting level, or `#f` if it nests too deep for *max-depth*.

Examples:

```
(texi-command-depth 'chapter 4)        ⇒ 1
(texi-command-depth 'top 4)            ⇒ 0
(texi-command-depth 'subsection 4)     ⇒ 3
(texi-command-depth 'appendixsubsec 4) ⇒ 3
(texi-command-depth 'subsection 2)     ⇒ #f
```

`texi-fragment->stexi` *string-or-port* [Функция]

Parse the texinfo commands in *string-or-port*, and return the resultant stexi tree. The head of the tree will be the special command, `*fragment*`.

`texi->stexi` *port* [Функция]
Read a full texinfo document from *port* and return the parsed stexi tree. The parsing will start at the `@settitle` and end at `@bye` or EOF.

`stexi->sxml` *tree* [Функция]
Transform the stexi tree *tree* into sxml. This involves replacing the `%` element that keeps the texinfo arguments with an element for each argument.

FIXME: right now it just changes % to `texinfo-arguments` – that doesn't hang with the idea of making a dtd at some point

7.21.2 (texinfo docbook)

7.21.2.1 Overview

This module exports procedures for transforming a limited subset of the SXML representation of docbook into stexi. It is not complete by any means. The intention is to gather a number of routines and stylesheets so that external modules can parse specific subsets of docbook, for example that set generated by certain tools.

7.21.2.2 Usage

`*sdocbook->stexi-rules*` [Переменная]

`*sdocbook-block-commands*` [Переменная]

`sdocbook-flatten` *sdocbook* [Функция]
"Flatten" a fragment of sdocbook so that block elements do not nest inside each other.

Docbook is a nested format, where e.g. a `refsect2` normally appears inside a `refsect1`. Logical divisions in the document are represented via the tree topology; a `refsect2` element *contains* all of the elements in its section.

On the contrary, texinfo is a flat format, in which sections are marked off by standalone section headers like `@subsection`, and block elements do not nest inside each other.

This function takes a nested sdocbook fragment *sdocbook* and flattens all of the sections, such that e.g.

```
(refsect1 (refsect2 (para "Hello")))
```

becomes

```
((refsect1) (refsect2) (para "Hello"))
```

Oftentimes (always?) sectioning elements have `<title>` as their first element child; users interested in processing the `refsect*` elements into proper sectioning elements like `chapter` might be interested in `replace-titles` and `filter-empty-elements`. См. [replace-titles], страница 779, and [filter-empty-elements], страница 778.

Returns a nodeset; that is to say, an untagged list of stexi elements. См. Раздел 7.20.6 [SXPath], страница 770, for the definition of a nodeset.

`filter-empty-elements` *sdocbook* [Функция]
Filters out empty elements in an sdocbook nodeset. Mostly useful after running `sdocbook-flatten`.

`replace-titles` *sdocbook-fragment* [Функция]
Iterate over the sdocbook nodeset *sdocbook-fragment*, transforming contiguous `refsect` and `title` elements into the appropriate texinfo sectioning command. Most useful after having run `sdocbook-flatten`.

For example:

```
(replace-titles '((refsect1) (title "Foo") (para "Bar.")))
   ⇒ '((chapter "Foo") (para "Bar."))
```

7.21.3 (texinfo html)

7.21.3.1 Overview

This module implements transformation from `stexi` to HTML. Note that the output of `stexi->shtml` is actually SXML with the HTML vocabulary. This means that the output can be further processed, and that it must eventually be serialized by `sxml->xml`. См. Раздел 7.20.2 [Reading and Writing XML], страница 761.

References (i.e., the `@ref` family of commands) are resolved by a *ref-resolver*. См. [texinfo html add-ref-resolver!], страница 779.

7.21.3.2 Usage

`add-ref-resolver!` *proc* [Функция]
Add *proc* to the head of the list of ref-resolvers. *proc* will be expected to take the name of a node and the name of a manual and return the URL of the referent, or `#f` to pass control to the next ref-resolver in the list.

The default ref-resolver will return the concatenation of the manual name, `#`, and the node name.

`stexi->shtml` *tree* [Функция]
Transform the stexi *tree* into shtml, resolving references via ref-resolvers. See the module commentary for more details.

`urlify` *str* [Функция]

7.21.4 (texinfo indexing)

7.21.4.1 Overview

Given a piece of stexi, return an index of a specified variety.

Note that currently, `stexi-extract-index` doesn't differentiate between different kinds of index entries. That's a bug ;)

7.21.4.2 Usage

`stexi-extract-index` *tree manual-name kind* [Функция]
Given an stexi tree *tree*, index all of the entries of type *kind*. *kind* can be one of the predefined texinfo indices (`concept`, `variable`, `function`, `key`, `program`, `type`) or one of the special symbols `auto` or `all`. `auto` will scan the stext for a `(printindex)` statement, and `all` will generate an index from all entries, regardless of type.

The returned index is a list of pairs, the car of which is the entry (a string) and the cdr of which is a node name (a string).

7.21.5 (texinfo string-utils)

7.21.5.1 Overview

Module '`(texinfo string-utils)`' provides various string-related functions useful to Guile's texinfo support.

7.21.5.2 Usage

`escape-special-chars` *str special-chars escape-char* [Функция]

Returns a copy of *str* with all given special characters preceded by the given *escape-char*.

special-chars can either be a single character, or a string consisting of all the special characters.

```
;; make a string regexp-safe...
 (escape-special-chars "***(Example String)***"
                       "[]()/*."
                       #\\)
=> "\\*\\*\\*\\(Example String\\)\\*\\*\\*"

;; also can escape a singe char...
 (escape-special-chars "richardt@vzavenue.net"
                       #\@
                       #\@)
=> "richardt@@vzavenue.net"
```

`transform-string` *str match? replace* [*start*] [*end*] [Функция]

Uses *match?* against each character in *str*, and performs a replacement on each character for which matches are found.

match? may either be a function, a character, a string, or `#t`. If *match?* is a function, then it takes a single character as input, and should return '`#t`' for matches. *match?* is a character, it is compared to each string character using `char=?`. If *match?* is a string, then any character in that string will be considered a match. `#t` will cause every character to be a match.

If *replace* is a function, it is called with the matched character as an argument, and the returned value is sent to the output string via '`display`'. If *replace* is anything else, it is sent through the output string via '`display`'.

Note that the replacement for the matched characters does not need to be a single character. That is what differentiates this function from '`string-map`', and what makes it useful for applications such as converting '`#\&`' to '`"&"`' in web page text. Some other functions in this module are just wrappers around common uses of '`transform-string`'. Transformations not possible with this function should probably be done with regular expressions.

If *start* and *end* are given, they control which portion of the string undergoes transformation. The entire input string is still output, though. So, if *start* is '5', then the first five characters of *str* will still appear in the returned string.

```
; these two are equivalent...
 (transform-string str #\space #\-) ; change all spaces to -'s
 (transform-string str (lambda (c) (char=? #\space c)) #\-)
```

`expand-tabs` *str* [*tab-size*] [Функция]

Returns a copy of *str* with all tabs expanded to spaces. *tab-size* defaults to 8.

Assuming tab size of 8, this is equivalent to:

```
(transform-string str #\tab "        ")
```

`center-string` *str* [*width*] [*chr*] [*rchr*] [Функция]

Returns a copy of *str* centered in a field of *width* characters. Any needed padding is done by character *chr*, which defaults to '`#\space`'. If *rchr* is provided, then the padding to the right will use it instead. See the examples below. left and *rchr* on the right. The default *width* is 80. The default *chr* and *rchr* is '`#\space`'. The string is never truncated.

```
 (center-string "Richard Todd" 24)
=> "      Richard Todd      "

 (center-string " Richard Todd " 24 #\=)
=> "===== Richard Todd ====="

 (center-string " Richard Todd " 24 #\< #\>)
=> "<<<<< Richard Todd >>>>>"
```

`left-justify-string` *str* [*width*] [*chr*] [Функция]

`left-justify-string str [width chr]`. Returns a copy of *str* padded with *chr* such that it is left justified in a field of *width* characters. The default *width* is 80. Unlike '`string-pad`' from srfi-13, the string is never truncated.

`right-justify-string` *str* [*width*] [*chr*] [Функция]

Returns a copy of *str* padded with *chr* such that it is right justified in a field of *width* characters. The default *width* is 80. The default *chr* is '`#\space`'. Unlike '`string-pad`' from srfi-13, the string is never truncated.

`collapse-repeated-chars` *str* [*chr*] [*num*] [Функция]

Returns a copy of *str* with all repeated instances of *chr* collapsed down to at most *num* instances. The default value for *chr* is '`#\space`', and the default value for *num* is 1.

```
 (collapse-repeated-chars "H  e  l  l  o")
=> "H e l l o"
 (collapse-repeated-chars "H--e--l--l--o" #\-)
=> "H-e-l-l-o"
 (collapse-repeated-chars "H-e--l---l----o" #\- 2)
=> "H-e--l--l--o"
```

`make-text-wrapper` [*#:line-width*] [*#:expand-tabs?*] [*#:tab-width*] [*#:collapse-whitespace?*] [*#:subsequent-indent*] [*#:initial-indent*] [*#:break-long-words?*] [Функция]

Returns a procedure that will split a string into lines according to the given parameters.

`#:line-width`
: This is the target length used when deciding where to wrap lines. Default is 80.

`#:expand-tabs?`
: Boolean describing whether tabs in the input should be expanded. Default is #t.

`#:tab-width`
: If tabs are expanded, this will be the number of spaces to which they expand. Default is 8.

`#:collapse-whitespace?`
: Boolean describing whether the whitespace inside the existing text should be removed or not. Default is #t.

  If text is already well-formatted, and is just being wrapped to fit in a different width, then set this to '`#f`'. This way, many common text conventions (such as two spaces between sentences) can be preserved if in the original text. If the input text spacing cannot be trusted, then leave this setting at the default, and all repeated whitespace will be collapsed down to a single space.

`#:initial-indent`
: Defines a string that will be put in front of the first line of wrapped text. Default is the empty string, "".

`#:subsequent-indent`
: Defines a string that will be put in front of all lines of wrapped text, except the first one. Default is the empty string, "".

`#:break-long-words?`
: If a single word is too big to fit on a line, this setting tells the wrapper what to do. Defaults to #t, which will break up long words. When set to #f, the line will be allowed, even though it is longer than the defined `#:line-width`.

The return value is a procedure of one argument, the input string, which returns a list of strings, where each element of the list is one line.

`fill-string` *str . kwargs* [Функция]

Wraps the text given in string *str* according to the parameters provided in *kwargs*, or the default setting if they are not given. Returns a single string with the wrapped text. Valid keyword arguments are discussed in `make-text-wrapper`.

`string->wrapped-lines` *str . kwargs* [Функция]

`string->wrapped-lines str keywds ...` Wraps the text given in string *str* according to the parameters provided in *keywds*, or the default setting if they are

not given. Returns a list of strings representing the formatted lines. Valid keyword arguments are discussed in `make-text-wrapper`.

7.21.6 (texinfo plain-text)

7.21.6.1 Overview

Transformation from stexi to plain-text. Strives to re-create the output from `info`; comes pretty damn close.

7.21.6.2 Usage

`stexi->plain-text` *tree* [Функция]
Transform *tree* into plain text. Returns a string.

7.21.7 (texinfo serialize)

7.21.7.1 Overview

Serialization of `stexi` to plain texinfo.

7.21.7.2 Usage

`stexi->texi` *tree* [Функция]
Serialize the stexi *tree* into plain texinfo.

7.21.8 (texinfo reflection)

7.21.8.1 Overview

Routines to generare `stexi` documentation for objects and modules.

Note that in this context, an *object* is just a value associated with a location. It has nothing to do with GOOPS.

7.21.8.2 Usage

`module-stexi-documentation` *sym-name* **[***%docs-resolver***]** **[***#:docs-resolver***]** [Функция]
Return documentation for the module named *sym-name*. The documentation will be formatted as `stexi` (см. Раздел 7.21.1 [texinfo], страница 776).

`script-stexi-documentation` *scriptpath* [Функция]
Return documentation for given script. The documentation will be taken from the script's commentary, and will be returned in the `stexi` format (см. Раздел 7.21.1 [texinfo], страница 776).

`object-stexi-documentation` _ **[**_**]** **[***#:force***]** [Функция]

`package-stexi-standard-copying` *name version updated years copyright-holder permissions* [Функция]
Create a standard texinfo `copying` section.

years is a list of years (as integers) in which the modules being documented were released. All other arguments are strings.

`package-stexi-standard-titlepage` *name version updated authors* [Функция]

Create a standard GNU title page.

authors is a list of `(name . email)` pairs. All other arguments are strings.

Here is an example of the usage of this procedure:

```
(package-stexi-standard-titlepage
 "Foolib"
 "3.2"
 "26 September 2006"
 '(("Alyssa P Hacker" . "alyssa@example.com"))
 '(2004 2005 2006)
 "Free Software Foundation, Inc."
 "Standard GPL permissions blurb goes here")
```

`package-stexi-generic-menu` *name entries* [Функция]

Create a menu from a generic alist of entries, the car of which should be the node name, and the cdr the description. As an exception, an entry of `#f` will produce a separator.

`package-stexi-standard-menu` *name modules module-descriptions extra-entries* [Функция]

Create a standard top node and menu, suitable for processing by makeinfo.

`package-stexi-extended-menu` *name module-pairs script-pairs extra-entries* [Функция]

Create an "extended" menu, like the standard menu but with a section for scripts.

`package-stexi-standard-prologue` *name filename category description copying titlepage menu* [Функция]

Create a standard prologue, suitable for later serialization to texinfo and .info creation with makeinfo.

Returns a list of stexinfo forms suitable for passing to `package-stexi-documentation` as the prologue. См. [texinfo reflection package-stexi-documentation], страница 784, [texinfo reflection package-stexi-standard-titlepage], страница 784, [texinfo reflection package-stexi-standard-copying], страница 783, and [texinfo reflection package-stexi-standard-menu], страница 784.

`package-stexi-documentation` *modules name filename prologue epilogue* [Функция]
[*#:module-stexi-documentation-args***]** **[***#:scripts***]**

Create stexi documentation for a *package*, where a package is a set of modules that is released together.

modules is expected to be a list of module names, where a module name is a list of symbols. The stexi that is returned will be titled *name* and a texinfo filename of *filename*.

prologue and *epilogue* are lists of stexi forms that will be spliced into the output document before and after the generated modules documentation, respectively. См. [texinfo reflection package-stexi-standard-prologue], страница 784, to create a conventional GNU texinfo prologue.

module-stexi-documentation-args is an optional argument that, if given, will be added to the argument list when `module-texi-documentation` is called. For example, it might be useful to define a `#:docs-resolver` argument.

`package-stexi-documentation-for-include` *modules* [Функция]
module-descriptions **[***#:module-stexi-documentation-args***]**

Create stexi documentation for a *package*, where a package is a set of modules that is released together.

modules is expected to be a list of module names, where a module name is a list of symbols. Returns an stexinfo fragment.

Unlike `package-stexi-documentation`, this function simply produces a menu and the module documentations instead of producing a full texinfo document. This can be useful if you write part of your manual by hand, and just use `@include` to pull in the automatically generated parts.

module-stexi-documentation-args is an optional argument that, if given, will be added to the argument list when `module-texi-documentation` is called. For example, it might be useful to define a `#:docs-resolver` argument.

8 GOOPS

GOOPS - это объектно-ориентированное расширение для Guile. Его реализация проистекает из STk-3.99.3 от Erick Gallesio и версии 1.3 of Gregor Kiczales' *Tiny-Clos*. Оно очень близко по духу к CLOS, Common Lisp Object System, но адаптировано для Scheme.

GOOPS - это полная объектно-ориентированная система с классами, объектами, множественным наследованием и обобщенными функциями имеющими несколько методов обработки. Кроме того, ее реализация основывается на метаобъектном протоколе - это означает, что основные операции GOOPS сами определены как методы соответствующих классов, и их можно насторить путем переопределения классов или путем переопределения этих методов.

Чтобы начать использовать GOOPS, вам сначала нужно импортировать модуль `(oop goops)`. Вы можете сделать это в Guile REPL выполнив:

```
(use-modules (oop goops))
```

8.1 Copyright Notice

The material in this chapter is partly derived from the STk Reference Manual written by Erick Gallesio, whose copyright notice is as follows.

Материал был адаптирован для использования в Guile с разрешения автора.

8.2 Определение Класса

Новый класс определяется синтаксиом `define-class`:

```
(define-class class (superclass ...)
   slot-description ...
   class-option ...)
```

class является именем определяемого класса. Список суперклассов *superclass* указывает, какие существующие классы, если таковые имеются, наследуются для слотов и свойств. *Slots* Слоты хранят per-instance[1] данные, для экземпляров класса — подобно "полям(fields)" или "переменным членам(member variables)" в других объектно ориентированных системах. Каждое *slot-description* определение дает имя слота и, возможно, некоторые "свойства" этого слота; например его начальное значение, имя функции, которая получит доступ к его значению, и так далее. Опции класса (Class options), описание слотов (slot descriptions) и наследование(inheritance) обсуждаются ниже

[1] Обычно — но также смотрите опцию `#:allocation`.

`define-class` *name* (*super ...*) *slot-definition ... class-option ...* [syntax]

Определяет класс с именем *name* который наследуется от классов *super*, со слотами определенными в определении слотов*slot-definition*s и опциями класса *class-option*. Созданный класс привязывается к переменной *name* в текущем окружении.

Каждое определение слота *slot-definition* является либо символом, который обозначает слот либо списком,

```
(slot-name-symbol . slot-options)
```

где имя слота *slot-name-symbol* это символ, а опции слота *slot-options* это список с четным числом элементов. Элементы с четными номерами слотов *slot-options*(считая от нуля) являются ключевыми словами; элементы с нечетными номерами являются соотвествующими значениями для этих ключевых слов.

Каждая опция класса *class-option* это ключевое слово и соотвествующее значение.

В качестве примера, давайте определим тип для представления комплексного числа как двух вещественных чисел.[2] Это можно сделать с помощью следующего определения класса:

```
(define-class <my-complex> (<number>)
   r i)
```

Это определение связывает переменную `<my-complex>` с новым классо экземпляры которого будут содержать два слота. Эти слоты называются `r` и `i` и будут содержать действительную и мнимую части комплексного числа. Обратите внимание, что этот класс наследует от класса `<number>`, который является предопределенным классом.[3]

Параметры слотов описаны в следующем разделе. Возможны следующие опции класса:

`#:metaclass` *metaclass* [class option]

Опция класса `#:metaclass` задает метакласс класса, который определяется. *metaclass* должен быть классом, который наследует от `<class>`. Для использоания метаклассов, см Раздел 8.11.1 [Metaobjects and the Metaobject Protocol], страница 813, и Раздел 8.11.2 [Metaclasses], страница 814.

Если опция `#:metaclass` отсутствует, GOOPS повторно использует или создает метакласс для нового класса, вызвав `ensure-metaclass` (см. Раздел 8.11.5 [ensure-metaclass], страница 817).

`#:name` *name* [class option]

Опция класса `#:name` указывает имя нового класса. Это имя используется для идентификации класса, когда печатаются связанные объекты - сам класс, его экземпляры и его подклассы.

Если параметр `#:name` отсутствует, GOOPS использует первый аргумент для `define-class` как имя класса.

---

[2] Конечно Guile уже имеет определение комплексных чисел. А `<complex>` на самом деле является предопределенным классом в GOOPS; но наше определение полезно здесь в качестве примера.

[3] `<number>` это прямой суперкласс предопределенного класса `<complex>`; `<complex>` является суперклассом для `<real>`, и `<real>` является суперклассом для `<integer>`.

8.3 Создание экземпляров и доступ к слотам

Экземпляр (или Объект) определенного класса может быть создан с помощью `make`. `make` принимает один обязательный параметр, который является классом создаваемого экземпляра, и список необязательных аргументов, которые будут использоваться для инициализации слотов нового экземпляра. Например, следующая форма:

```
(define c (make <my-complex>))
```

создает новый объект `<my-complex>` и связывает его с переменно Scheme `c`.

`make` [generic]
`make` (*class* <*class*>) *initarg* . . . [method]
Создает и инициализирует новый экземпляр класса *class*, используя аргументы *initarg* . . . .

Теоретически, *initarg* . . . может иметь любую структуру, которая понимается методами get, когда вызывается обобщенная функция `initialize` для вновь созданного объекта.

На практике, специализированные методы `initialize` обычно вызывают `(next-method)`, и поэтому в конечно итоге применяются стандартные методы инициализации GOOPS `initialize`. Эти методы ожидают. что *initargs* будет списком с четным числом аргументов, где четные элементы (начиная с нуля) это ключевые слова, а элементы с нечетным номером - соответствующие значения.

Процесс инициализации GOOPS автоматически обрабатывает аргументы ключевые слова для слотов, определение которых включает опцию `#:init-keyword` (см. Раздел 8.4 [init-keyword], страница 790). Другие пары значений ключевого слова могут обрабатываться только методом инициализации `initialize` который специализируется на классе нового экземпляра. Любые пары необработанных ключевых слов игнорируются.

`make-instance` [generic]
`make-instance` (*class* <*class*>) *initarg* . . . [method]
`make-instance` is an alias for `make`.

Доступ к слотам нового комплексного числа можно получить используя `slot-ref` и `slot-set!`. `slot-set!` устанавливает значение слота объекта и `slot-ref` извлекает его.

```
(slot-set! c 'r 10)
(slot-set! c 'i 3)
(slot-ref c 'r) ⇒ 10
(slot-ref c 'i) ⇒ 3
```

Модуль `(oop goops describe)` предоставляет функцию `describe`, которая полезна для просмотра всех слотов объекта; она печатает слоты и их значения на стандартный вывод.

```
(describe c)
⊣
#<<my-complex> 401d8638> is an instance of class <my-complex>
Slots are:
```

```
r = 10
i = 3
```

8.4 Опции Слотов(Slot Options)

При определении слота (в форме a `(define-class ...)`), могут быть указаны различные параметры в дополнение к названию слота. Каждый параметр задается ключевым словом. Список возможных ключевых слов следующий:

`#:init-value` *init-value* [slot option]
`#:init-form` *init-form* [slot option]
`#:init-thunk` *init-thunk* [slot option]
`#:init-keyword` *init-keyword* [slot option]

Эти параметры предоставляют различные способы указать, как инициализировать начальное значение слота во время создания нового экземпляра.

init-value задает фиксированное начальное значение слота (совместно используемое во всех новых экземплярах класса).

init-thunk указывает преобразователь(thunk) который предоставляет значение по умолчанию для слота. Преобразователь вызывается когда создается новый экземпляр класса и должен возвращать новое начальное значение слота.

init-form указывает форму, которая при вызове возвращает начальное значение для слота. Форма выполняется каждый раз, когда создается экземпляр класса, в лексической среде содержащей выражение `define-class`.

init-keyword указывает ключевое слово, которое может использоваться для передачи начального значения слота через вызов `make`, которая создает новый экземпляр класса.

Обратите внимание: поскольку значение `init-value` является общим для всех экземпляров класса, вы должны использовать его тогда, когда начальное значение является неизменным, таким как константа. Если вы хотите инициализировать слот новым, независимо изменяемым значением, вы должны вместо этого использовать `init-thunk` или `init-form`. Рассмотрим пример.

```
(define-class <chbouib> ()
  (hashtab #:init-value (make-hash-table)))
```

Здесь создается только одна хеш-таблица, и все экземпляры класса `<chbouib>` имеют свой слот `hashtab`. Чтобы каждый экземпляр класса `<chbouib>` ссылался на новую хеш-таблицу, вы должны написать:

```
(define-class <chbouib> ()
  (hashtab #:init-thunk make-hash-table))
```

или:

```
(define-class <chbouib> ()
  (hashtab #:init-form (make-hash-table)))
```

Если для одного из слотов указано больее одного из этих параметров, порядок приоритета инициализации устанавливается вначале для:

- `#:init-keyword`, if *init-keyword* is present in the options passed to `make`
- `#:init-thunk`, `#:init-form` or `#:init-value`.

Если определение слота содержит более одного параметра инициализации того же приоритета, более поздние из них игнорируются. Если слот не инициализирован вообще, его значение не устанавливается.

В общем случае слоты, которые разделяются между несколькими экземплярами, инициализируются только в момент создания нового экземпляраа, если значение слота еще не было связано. Однако, если при создание нового экземпляра указывается ключевое слово init и значение для общего слота, слот инициализируется не зависимо от его предыдущего значения.

Обратите внимание, однако, что мощь протокола метаобъектов GOOPS означает, что все написанные здесь процедуры, могут быть настроены или перенастроены для определенных классов! Слот инициализации, описанный здесь, выполняется наименее специализированным методом обобщенной функции `initialize`, чья сигнатура такова:

```
(define-method (initialize (object <object>) initargs) ...)
```

Инициализация экземпляров любого данного класса может быть настроена путем определения метода инициализации `initialize`, который является специализированным для этого класса, и автор специализированного метода может решить вызвать codenext-method - менее специализированный метод инициализации `initialize` - и в любой точке специализированного кода это правила могут быть переопределены и перестают работать. В общем, поэтому механизмы инициализации, описанные здесь, могут быть изменены или переопределены более специализированным кодом или вообще могут не поддерживаться для определенных классов.

`#:getter` *getter* [slot option]
`#:setter` *setter* [slot option]
`#:accessor` *accessor* [slot option]

Доступный объект *obj* имеющий слоты с именами `foo` и `bar`, можно всегда читать и писать, вызывая методы `slot-ref` и `slot-set!` с соответствующим именем слота; например:

```
(slot-ref obj 'foo)
(slot-set! obj 'bar 25)
```

Опции `#:getter`, `#:setter` и `#:accessor`, если они присутствуют, сообщают GOOPS о создании обобщенной функции и определении методов, которые можно использовать для получения и установки значения слота. *getter* указывает обобщенную функцию, к которой GOOPS добавит метод получения значения слота. *setter* указывает обобщенную функцию, к которой GOOPS добавит метод установки значения слота. *accessor* указывает обобщенную функцию accessor которой GOOPS добавит методы для получения и установки значения слота.

Поэтому, если класс включает определение слота, например:

```
(c #:getter get-count #:setter set-count #:accessor count)
```

GOOPS определяет методы обобщенной функций, так что на значение слота можно ссылаться используя либо getter либо accessor -

```
(let ((current-count (get-count obj))) ...)
(let ((current-count (count obj))) ...)
```

- и устанавливать с помощью либо setter или accessora

```
(set-count obj (+ 1 current-count))
(set! (count obj) (+ 1 current-count))
```

Обратите внимание, что

- с accesor'ом значение слота устанавливается с использованием обобщенного синтаксиса `set!`
- на практике для слота неиспользуют все три из этих опций: только для чтения, только для записи и для чтения-записи `#:getter`, `#:setter` и `#:accessor` соответственно.

Связывание указанных имен выполняется в среде определения класса `define-class`. Если имена уже связаны (в этой среде) значениями которые не могут быть обновлены до обобщеннх функций, эти значения будут перезаписаны, если выполняется выражение the `define-class`. Более подробно, См. Раздел 8.11.9 [ensure-generic], страница 824.

`#:allocation` *allocation* [slot option]

Опция `#:allocation` указывает GOOPS, как распределить памиять, для слота. Возможные значения для *allocation*:

- `#:instance`

  Указывает, что GOOPS должен создавать отдельное хранилище для этого слота в каждом новом экземпляре содержащего его класса (и его подклассов). Это значение устанавливается по умолчанию.

- `#:class`

  Указывает, что GOOPS должен создвать хранилище для слота, которое будет использоваться всеми экземплярами содержащего его класса (и его подклассов). Другими словами, слот в классе *C* с распределением типа `#:class` разделяется всеми экземплярами *instance*s для которых выполнено `(is-a?` *`instance c`*`)`. Это позволяет определенть некую глобальную переменную, к которой можно получить доступ только из прямых наследников класса, который определяет данный слот.

- `#:each-subclass`

  Указывает, что GOOPS должен создать хранилище для этого слота, который будет использоваться всеми непосредственными *direct* экземплярами этого класса и что всякий раз, когда определяется подкласс данного класса, GOOPS должен создать новое хранилище для слота, который будет разделяться всеми непосредственными экземплярами данного подкласса. Другими словами, слот с `#:each-subclass` разделяется всеми экземплярами одного класса `class-of`.

- `#:virtual`

  Указывает, что GOOPS не должен выделять память для этого слота. Определение слота также должны включать в себя опции `#:slot-ref` и `#:slot-set!` для получения значения или его установки для данного слота. Смотри пример ниже.

Параметры рапределения слотов обрабатываются при определении нового класса с помощью обобщенной функции `compute-get-n-set`, которая определяется в метаклассе класса. Следовательно, новые типы распределения слотов могут быть реализованы путем определения нового метакласса и метода для `compute-get-n-set` который будет специализирован для нового метакласса. Пример того, как это делается, см. Раздел 8.11.6 [Customizing Class Definition], страница 820.

`#:slot-ref` *getter* [slot option]
`#:slot-set!` *setter* [slot option]

Опции `#:slot-ref` и `#:slot-set!` должны быть указаны, если распределение слотов является `#:virtual`, и игнорируются иначе.

getter должен быть замыканием, принимающим один параметр *instance*, которое возвращает текущее значение слота. *setter* должен быть замыканием получающим два параметра - *instance* и *new-val* - которое устанавливает новое значение слота в *new-val*.

8.5 Илюстрирование Описание Слота

Чтобы проилюстрировать описание слота, мы можем переопределить класс `<my-complex>` рассмотренный ранее. Определение может быть:

```
(define-class <my-complex> (<number>)
   (r #:init-value 0 #:getter get-r #:setter set-r! #:init-keyword #:r)
   (i #:init-value 0 #:getter get-i #:setter set-i! #:init-keyword #:i))
```

При этом определении, слоты `r` и `i` по умолчанию устанавливаются в 0 и могут быть инициализированны другими значениями путем вызова `make` с ключевыми словами `#:r` и `#:i`. Также обобщенные функции `get-r`, `set-r!`, `get-i` и `set-i!` автоматически определяются для чтения и записи слотов.

```
(define c1 (make <my-complex> #:r 1 #:i 2))
(get-r c1) ⇒ 1
(set-r! c1 12)
(get-r c1) ⇒ 12
(define c2 (make <my-complex> #:r 2))
(get-r c2) ⇒ 2
(get-i c2) ⇒ 0
```

Аксессоры могут читать и записывать слот. Итак, другое определение класса `<my-complex>`, использующее опцию `#:accessor`, может быть:

```
(define-class <my-complex> (<number>)
   (r #:init-value 0 #:accessor real-part #:init-keyword #:r)
   (i #:init-value 0 #:accessor imag-part #:init-keyword #:i))
```

с помощью этого определения слот `r` можно прочитать с помощью:

```
(real-part c)
```

и установить с помощью:

```
(set! (real-part c) new-value)
```

Предположим теперь, что мы хотим манипулировать комплексными числами как прямоугольными, так и полярными координатами. Одним из решений может быть определение комплексных чисел, которое использует одно конкретное представление и некоторые функции преобразования для перехода из одного представления в другое. Лучшее решение - использовать виртуальные слоты, например:

```
(define-class <my-complex> (<number>)
   ;; True slots use rectangular coordinates
   (r #:init-value 0 #:accessor real-part #:init-keyword #:r)
   (i #:init-value 0 #:accessor imag-part #:init-keyword #:i)
   ;; Virtual slots access do the conversion
   (m #:accessor magnitude #:init-keyword #:magn
      #:allocation #:virtual
      #:slot-ref (lambda (o)
                   (let ((r (slot-ref o 'r)) (i (slot-ref o 'i)))
                     (sqrt (+ (* r r) (* i i)))))
      #:slot-set! (lambda (o m)
                    (let ((a (slot-ref o 'a)))
                      (slot-set! o 'r (* m (cos a)))
                      (slot-set! o 'i (* m (sin a))))))
   (a #:accessor angle #:init-keyword #:angle
      #:allocation #:virtual
      #:slot-ref (lambda (o)
                   (atan (slot-ref o 'i) (slot-ref o 'r)))
      #:slot-set! (lambda(o a)
                    (let ((m (slot-ref o 'm)))
                      (slot-set! o 'r (* m (cos a)))
                      (slot-set! o 'i (* m (sin a)))))))
```

В этом определении класса, слоты размах(magnitude) `m` и угол(angle) `a` виртуальные, и вчисляются когда при ссылке, из значений нормальных слотов (т.е. `#:allocation #:instance`) `r` и `i`, путем вызова функций, определенных в соответствующем параметре `#:slot-ref`. Соответственно, запись `m` или `a` приводит к вызову функций определенных в опции `#:slot-set!`. Таким образом следующее выражение

```
(slot-set! c 'a 3)
```

позволяет установить угол комплексного числа `c`.

```
(define c (make <my-complex> #:r 12 #:i 20))
(real-part c) ⇒ 12
(angle c) ⇒ 1.03037682652431
(slot-set! c 'i 10)
(set! (real-part c) 1)
(describe c)
⊣
#<<my-complex> 401e9b58> is an instance of class <my-complex>
Slots are:
     r = 1
```

```
i = 10
m = 10.0498756211209
a = 1.47112767430373
```

Поскольку ключевые слова инициализации были определены для четырех слотов, мы можем теперь определить стандартные примитивы Scheme `make-rectangular` и `make-polar`.

```
(define make-rectangular
   (lambda (x y) (make <my-complex> #:r x #:i y)))

(define make-polar
   (lambda (x y) (make <my-complex> #:magn x #:angle y)))
```

8.6 Методы и Обобщенные Функции

Метод GOOPS похож на процедуру Scheme, за исключением того, что он специализирован для конкретного набора классов аргументов и будет использоваться только тогда, когда фактические аргументы в вызове совпадают с классами в определении метода.

```
(define-method (+ (x <string>) (y <string>))
  (string-append x y))

(+ "abc" "de") ⇒ "abcde"
```

Метод не формально связан с каким-либо одним классом (как и во многих других объектно-ориентированных языках), поскольку метод может быть специализирован для комбинации нескольких классов. Если вы изучали ООП на примере не Лисп языков, вы можете помнить обсуждения, такие как метод растягивания(stretch) графического изображения вокруг поверхности(surface) должен быть методом класса изображения(image), с поверхностью(surface) в качестве параметра, или методом класса поверхность(surface), с изображением (image) в качестве параметра. В GOOPS вы просто напишите:

```
(define-method (stretch (im <image>) (sf <surface>))
  ...)
```

и вопрос о том, к какому классу этот метод привязан, не нуждаеся в ответе.

Одновременно может быть несколько методов с тем же именем, но разным набором указанных аргументов классов; например:

```
(define-method (+ (x <string>) (y <string)) ...)
(define-method (+ (x <matrix>) (y <matrix>)) ...)
(define-method (+ (f <fish>) (b <bicycle>)) ...)
(define-method (+ (a <foo>) (b <bar>) (c <baz>)) ...)
```

Обобщенная функция является контейнером для набора таких методов, которые программа намерена использовать.

Если вы посмотрите на исходный код программы и увидите где то в ней `(+ x y)`, концептуально здесь происходит то, что программа в этот момент вызывает обобщенную функцию (в этом случае, обобщенная функция связана с идентификатором `+`). Когда это происходит, Guile разрбирается какой из методов обобщенной функции

является наиболее подходящим для аргументов с которыми вызвана эта функция; после чего вычисляет этот метод с аргументами как формальными параметрами метода. Это происходит каждый раз когда выполняется вызов обобщенной функции — это не предполагает, что вызов данного исходного кода будет вызывать один и тот же метод каждый раз.

Определение идентификатора как обобщенной функции выполняется с помощью макроса `define-generic`. Определение нового метода выполняется с помощью макроса `define-method`. Обратите внимание, что определение метода `define-method` автоматически выполняет `define-generic`, если соответствующий идентификатор еще не является обобщенной функцией, поэтому часто не требуется явно выраженного вызвова `define-generic`.

`define-generic` *symbol* [syntax]

Создает обобщенную функцию с именем *symbol* и связывает ее с переменной *symbol*. Если ранее *symbol* был связан с процедурой Scheme procedure (или процедурой с устновщиком), старая процедура (и установщик) включаются в новую обобщенную функцию как процедура по умолчанию(и установщик). Любое другое предыдущее значение, включая существующую обобщенную функцию, отбрасывается и заменяется новой, пустой обобщенной функцией.

`define-method` (*generic parameter ...*) *body ...* [syntax]

Определяет метод для обобщенной функции или обобщенного аксессора *generic* параметрами *parameter* и телом *body ...*.

generic — это Обобщенная функция. Если *generic* это переменная, которая еще не связана с объектом обобщенной функции, расширение `define-method` будет включать вызов `define-generic`. Если *generic* это (`setter` *generic-with-setter*), где *generic-with-setter* это переменная еще не связанная с объктом generic-with-setter, расширение будет включать вызов `define-accessor`.

Каждый *parameter* должен быть либо символом, либо двухэлементным списком (*symbol class*). Символы относятся к переменным в теле форм, которые привязаны к параметрам предоставляемые вызывающим, при вызове этого метода. *class*, если он есть, указывают возможные комбинации параметров, к которым может применяться этот метод

body ... является телом определения метода.

`define-method` выражения немного напоминают форму определения процедур Scheme

```
(define (name formals ...) . body)
```

Важным отличием является то, что каждый формальный параметр, помимо возможного аргумента “rest”, может быть ограничен именем класса: *formal* становиться (*formal class*). Значение этого ограничения является то, что определяемый метод будет применим только в конкретном вызове обобщенной функции, если соответствующий аргумент является экземпляром классса *class*(или одного из его подклассов). Если более чем один из формальных параметров ограничен подобным образом, то метод будет применим только в том случае, если каждый из соответствующих аргументов является экземпляром соответствующим указанному классу.

Обратите внимание, что не ограниченные формальные параметры действуют так, как если бы они были ограничены классом `<top>`, который GOOPS использует для обозначения суперкласса всех допустимых типов Scheme, включая примитивные типы и GOOPS классы.

Например, если метод обобщенной функции определен с параметрами(*parameter*) `(s1 <square>)` и `(n <number>)`, этот метод применим только к вызовам его обобщенной функции которые имеют два параметра, где первый параметр является экземпляром класса `<square>` и второй параметр является числом.

8.6.1 Аксессоры(методы доступа к значениям)

Аксессор — это обобщенная функция, которая может использоваться с обобщенным вызовом `set!` синтаксис (см. Раздел 6.9.8 [Procedures with Setters], страница 274). Guile будет обрабатывать вызов как

```
(set! (accessor args...) value)
```

вызывая наиболее специализированный метод доступа(аксессор/`accessor`) соответствующий классам `args` и `value`. `define-accessor` используется для привязки идентификатора к аксессору.

`define-accessor` *symbol* [syntax]

Создает аксессор с именем *symbol* и связывает его с переменной *symbol*. Если *symbol* ранее был связан с процедурой Scheme (илиг procedure-with-setter), старая процедура (и установщик) включается в новый аксессор в качестве процедуры по умолчанию (и установщик). Любое другое предыдущее значение, включая существующую обобщенную функцию или акцессор заменяются новым, пустым аксессором.

8.6.2 Расширение Примитивов

Многие из примитивных процедур Guile можно расширить, предоставив им обобщенную функцию которая работает в сочетании с их обычной Си-кодированной реализацией. Когда примитив расширяется таким образом, он ведет себя как обобщенная функция с Си кодированным методом по умолчанию.

Это расширение происходит автоматически, если метод определен(вызовом `define-method`) для переменной, текущее значение которой является примитивным. Но его также можно выполнить принудительно вызвав `enable-primitive-generic!`.

`enable-primitive-generic!` *primitive* [primitive procedure]

Принудительное создание определения обобщенной функции для *primitive*.

Как только было создано определение обобщенной функции для примитива, оно может быть востановлено используя `primitive-generic-generic`.

`primitive-generic-generic` *primitive* [primitive procedure]

Возвращает обобщенной функции определение примитива *primitive*.

`primitive-generic-generic` генерирует ошибку, если *primitive* не является примитивом с обобщенным расширением.

8.6.3 Объединение Обобщенных функций

Обобщенные функции и аксессоры GOOPS часто имеют короткие, обобщенныые имена. Например, сли векторный пакет предоставляет аксессор для координаты X вектора, этот аксессор может быть просто назван `x`. Его не нужно называть, например так, `vector:x`, потому что GOOPS будет работатт, когда видит код, подобный (`x` *`obj`*), что должен быть ввызван вектор-специфичный метод `x`, если *obj* это вектор.

Тем не менее возникает вопрос о том, что происходит когда разные пакеты определяют обобщенную функцию с тем же именем. Предположим, мы работаем с графическим пакетом, который должен использовать два независимых векторных пакета для 2D и 3D векторов соответственно. Если оба пакета экспортируют `x`, что делает код при использовании этих пакетов?

Раздел 6.20.3 [duplicate binding handlers], страница 434, объясняет, как это разрешается для противоречивых привязок в общем. Для обобщенных, существует специальный обработчик дубликатов, `merge-generics`, который предписывает модульной системе объединить обобщенные функции с одинаковыми именами. Вот пример:

```
(define-module (math 2D-vectors)
  #:use-module (oop goops)
  #:export (x y ...))

(define-module (math 3D-vectors)
  #:use-module (oop goops)
  #:export (x y z ...))

(define-module (my-module)
  #:use-module (oop goops)
  #:use-module (math 2D-vectors)
  #:use-module (math 3D-vectors)
  #:duplicates (merge-generics))
```

Обобщенная функция `x` в модуле `(my-module)` теперь будет включать все методы `x` из обоих импортированных модулей.

Чтобы быть точным, теперь будут три различные обобщенные функции с именем `x`: `x` в `(math 2D-vectors)`, `x` в `(math 3D-vectors)`, и `x` в `(my-module)`; и эти функции разделяют их методы интересным и динамичным способом.

Чтобы объяснить, давайте назовем импортированные обобщенные функции (в `(math 2D-vectors)` и `(math 3D-vectors)`) предками(*ancestors*), и объединенную обобщенную функцию (в `(my-module)`), потомком(*descendant*). Общее правило заключается в том, что для любой обобщенной функции G, применимы методы выбранные из объединения методов функций потомков G, сами методы G и методы функций пердков G.

Таким образом, функции предков эффективно делят методы с их потомками и наоборот. В приведенном выше примере `x` в `(math 2D-vectors)` будет совместно использовать методы `x` в `(my-module)` и наоборот.[4] Обмен динамический, поэтому до-

[4] Но обратите внимание, что `x` в `(math 2D-vectors)` не разделяет методы с `x` в `(math 3D-vectors)`, поэтому модульность сохраняется.

бавление другого нового метода к потомству подразумевает добавление его к предкам потомка.

8.6.4 Next-method

Когда вы вызываете обобщенную функцию с определенным набором аргументов, GOOPS создает список всех методов, которые применимы к этим аргументам и упорядочивает его, тем, насколько близко определение метода соответствует фактическому типу аргументов. Затем он вызывает метод с верху этого списка. Если код выбранного метода хочет вызвать следующий метод в этом списке, он может это сделать используя `next-method`.

```
(define-method (Test (a <integer>)) (cons 'integer (next-method)))
(define-method (Test (a <number>))  (cons 'number  (next-method)))
(define-method (Test a)             (list 'top))
```

С этими определениями,

```
(Test 1)   ⇒ (integer number top)
(Test 1.0) ⇒ (number top)
(Test #t)  ⇒ (top)
```

`next-method` всегда вызывается просто `(next-method)`. Аргументы для вызова следующего метода всегда неявные и сегда те же, что и для вызова исходного метода.

Если вы хотите вызвать метод с тем же именем, но с другим набором аргументов(например, как вы можете с перегруженными методами Си++), не используйте `next-method`, но в место этого просто напишите новый вызов, как обычно:

```
(define-method (Test (a <number>) min max)
  (if (and (>= a min) (<= a max))
      (display "Number is in range\n"))
  (Test a))

(Test 2 1 10)
⊣
Number is in range
⇒
(integer number top)
```

(В этом случае вы должны быть осторожны, чтобы вызов `Test` не привел к бесконечной рекурсии, но соображение такое же, как и в любом коде Scheme.)

8.6.5 Примеры обобщенных функций и Методов

Рассмотрим следующие определения:

```
(define-generic G)
(define-method (G (a <integer>) b) 'integer)
(define-method (G (a <real>) b) 'real)
(define-method (G a b) 'top)
```

Вызов `define-generic` определяет *G* как обобщенную функцию. Три следующие строки определяют методы для *G*. Каждый метод использует последовательность специализированных параметров (*parameter specializers*), которые определяют, когда

данный метод применим. Специализатор позволяет указать класс применяемого параметра, которому он должен принадлежать(напрямую или опосредованно). Если специализатор не указан, система по умолчанию присваивает значение <top>. Таким образом, первое определение метода эквивалентно:

```
(define-method (G (a <integer>) (b <top>)) 'integer)
```

Теперь давайте рассмотрим некоторые возможные вызовы обобщенной функции G:

```
(G 2 3)    ⇒ integer
(G 2 #t)   ⇒ integer
(G 1.2 'a) ⇒ real
(G #t #f)  ⇒ top
(G 1 2 3)  ⇒ error (since no method exists for 3 parameters)
```

Приведенные выше методы используют только один специализированный параметр в списке параметров. Но в целом, любые или все параметры метода могут быть специализированы. Предположим, что мы определим теперь:

```
(define-method (G (a <integer>) (b <number>))  'integer-number)
(define-method (G (a <integer>) (b <real>))    'integer-real)
(define-method (G (a <integer>) (b <integer>)) 'integer-integer)
(define-method (G a (b <number>))              'top-number)
```

С этими определениями:

```
(G 1 2)   ⇒ integer-integer
(G 1 1.0) ⇒ integer-real
(G 1 #t)  ⇒ integer
(G 'a 1)  ⇒ top-number
```

В качестве дальнейшего примера мы продолжим определять операции над классом <my-complex>. Предположим, что мы хотим использовать его для полной реализации комплексных чисел. Например, определение для сложения двух комплексных чисел может быть

```
(define-method (new-+ (a <my-complex>) (b <my-complex>))
  (make-rectangular (+ (real-part a) (real-part b))
                    (+ (imag-part a) (imag-part b))))
```

Чтобы убедиться, что метод + используемый в методе new-+ является стандартным суммированием мы можем сделать:

```
(define-generic new-+)

(let ((+ +))
  (define-method (new-+ (a <my-complex>) (b <my-complex>))
    (make-rectangular (+ (real-part a) (real-part b))
                      (+ (imag-part a) (imag-part b)))))
```

define-generic обеспечивает здесь, что new-+ будет определяться в глобальном окружении. Как только это будет сделано, мы можем добавить методы к обобщенной функции new-+, которые делают замыкание на символ +. Полная запись метода new-+ показана на Figure 8.1.

```
(define-generic new-+)

(let ((+ +))

  (define-method (new-+ (a <real>) (b <real>)) (+ a b))

  (define-method (new-+ (a <real>) (b <my-complex>))
    (make-rectangular (+ a (real-part b)) (imag-part b)))

  (define-method (new-+ (a <my-complex>) (b <real>))
    (make-rectangular (+ (real-part a) b) (imag-part a)))

  (define-method (new-+ (a <my-complex>) (b <my-complex>))
    (make-rectangular (+ (real-part a) (real-part b))
                      (+ (imag-part a) (imag-part b))))

  (define-method (new-+ (a <number>))  a)

  (define-method (new-+) 0)

  (define-method (new-+ . args)
    (new-+ (car args)
      (apply new-+ (cdr args)))))

(set! + new-+)
```

Figure 8.1: Расширение + для обработки комплексных чисел

Мы используем тот факр, что обобщенная функция не обязана иметь фиксированное количество параметров. Четыре первых метода реализуют двухэлементное суммирование. Пятый метод говорит что добавление одного элемента это сам элемент. Шестой метод говорит, что использование добавления без параметра всегда возвращает 0 (это верно и для примитива +). Последний метод принимает произвольное количество параметров parameters[5]. Этот метод действует как `reduce`: он вызывает двуэлементное суммирование для каждого элемента *car* из списка и результата его последнего выполнения. В завершении, `set!` позволяет переопределить символ + нашему расширенному дополнению.

Чтобы завершить нашу реализацию комплексных чисел, мы могли бы переопределить стандартные предикаты Scheme следующим образом:

```
(define-method (complex? c <my-complex>) #t)
(define-method (complex? c)           #f)

(define-method (number? n <number>) #t)
(define-method (number? n)          #f)
```

[5] The parameter list for a `define-method` follows the conventions used for Scheme procedures. In particular it can use the dot notation or a symbol to denote an arbitrary number of parameters

...

Стандартные примитивы, в которых задействованы комплексные числа, также могут быть переопределены в той же манере.

8.6.6 Обработка Ошибок Обращения

Если вызывается обобщенная функция с комбинацией параметров, для которых нет применимого метода, GOOPS вызывает ошибку.

`no-method` [generic]
`no-method` (*gf* <*generic*>) *args* [method]

Когда приложение вызывает обобщенную функцию, и никакие методы не были определены для этой обобщенной функции, GOOPS вызывает универсальную обобщенную функцию `no-method`. По умолчанию этот метод вызывает `goops-error` с сообщением об ошибке.

`no-applicable-method` [generic]
`no-applicable-method` (*gf* <*generic*>) *args* [method]

Когда приложение применяет обобщенную функцию к набору аргументов и нет метода который был бы определен для этих типов аргументов, GOOPS вызывает обобщенную функцию `no-applicable-method`. Метод по умолчанию вызывает `goops-error` с сообщением об ошибке.

`no-next-method` [generic]
`no-next-method` (*gf* <*generic*>) *args* [method]

Когда метод обобщенной функции вызывает `(next-method)` для вызова следующего менее специаизированного метода для этой обобщенной функции, и нет менее специализированного метода определенного для данной обобщенной функции для данного набора аргументов, GOOPS вызывает обобщенную функцию `no-next-method`. Метод по умолчанию вызывает `goops-error` с соответствующим сообщением.

8.7 Наследование

Вот несколько определений классов, которые помогут проилюстрировать наследование

```
(define-class A () a)
(define-class B () b)
(define-class C () c)
(define-class D (A B) d a)
(define-class E (A C) e c)
(define-class F (D E) f)
```

`A`, `B`, `C` имеют нулевой список суперклассов. В этом случае, система заменит пустой список на список содержащий только класс `<object>`, это корень всех классов определяемых вызовом `define-class`. `D`, `E`, `F` используют множественное наследование: каждый класс наследуется от двух ранее определенных классов. Т.е определения классов определяют иерархию, которая показана на Figure 8.2. На этом рисунке так же показан класс `<top>`; этот класс является суперклассом всех объектов Scheme. В частности, `<top>` является суперклассом для всех стандартных типов Scheme.

Figure 8.2: A class hierarchy.

Когда класс имеет суперклассы, его набор слотов вычисляется путем объединения его собственных слотов и всех его суперклассов. Таким образом, каждый экземпляр класса D будет иметь три слота, `a`, `b` и `d`). Слоты класса могут быть просмотрены с помощью примитива `class-slots`. Для классов,

```
(class-slots A) ⇒ ((a))
(class-slots E) ⇒ ((a) (e) (c))
(class-slots F) ⇒ ((e) (c) (b) (d) (a) (f))
```

Порядо вывода слотов не определен.

8.7.1 Список Старшинства(предшествования) Классов (Class Precedence List)

Что происходит, когда класс наследует от двух или более суперклассов, имеющих слот с одним и темже именем, но с несовместимыми определениями — например, разными значениями инициализации или размещения слота? Нам нужно правило для определения того, какое определение слота получит производный класс, и это правило обеспечивается списком старшинства классов. *Class Precedence List*.[6]

Другая проблема возникает при вызове обобщенной функции, и есть более одного метода, которые можно применить к аргументам вызова. Здесь нам нужен способ упорядочения применения методов, так что бы Guile знал, какой метод использовать первым, чтобы использовать следующий метод, если этот метод вызывает `next-method`, и т.д. Одним из компонентов определения этого порядка является определение для каждого заданного аргумента вызова, котоыре из специализированных классов, из каждого определения применимого метода являющегося наиболее специфичным для этих аргументов, и здесь снова нам помогает список старшинства классов.

Если наследование было ограничено, так что каждый класс мог иметь только один суперкласс — также известное как одиночное *single* наследование — упорядочение классов было бы легким. Правило было бы просто, что подкласс считается более конкретным, чем его суперкласс.

При множественном наследовании упорядочение менее очевидно, и мы вынуждены определять правило для определения приоритета. Предопложим, что мы имеем:

```
(define-class X ()
   (x #:init-value 1))

(define-class Y ()
   (x #:init-value 2))

(define-class Z (X Y)
   (...))
```

Очевидно, что `Z` класс является более конкретным(Старшим), чем `X` или `Y`, для экземпляров `Z`. Но какой из классов более конкретен(Старш) `X` или `Y` — и, следовательно,

[6] Данная секция является адаптацией материала Деффа Далтона (J.Dalton@ed.ac.uk) *Brief introduction to CLOS*

для определений выше, какое значение `#:init-value` вступит в силу при создании экземпляра класса `Z`? Правило GOOPS заключается в том, что суперклассы перечисленные ранее, более конкретны(Старши), чем перечисленные позднее. Следовательно `X` более конкретный(Старш) чем `Y`, и значение `#:init-value` для слота `x` в экземпляре `Z` будет равно 1.

Следовательно, существует линейный порядок для класса и всех его суперклассов, от наименее конкретных к более конкретным, и этот порядок называется Списком Старшинства(Предшествования) Классов(Class Precedence List) класса.

На самом деле вышеприведенных правил не достаточно, чтобы всегда определять уникальный порядок, но они дают представление о том, как все работает. Например, для класса `F` показанного на Figure 8.2, список старшинства классов такой

```
(f d e a c b <object> <top>)
```

В тех случаях, когда существует какая-либо двусмысленность(например, как эта), для программиста плохой идеей будет, полагаться именно на порядок. Если порядок для некоторых суперклассов важен , это может быть выражено непосредственно в определении классов.

Список Старшинства Классов можно поличть, вызвав `class-precedence-list`. Эта функция возвращает упорядоченный список, первым элементом которого является наиболее Старший(конкретный) класс. Например:

```
(class-precedence-list B) ⇒ (#<<class> B 401b97c8>
                                        #<<class> <object> 401e4a10>
                                        #<<class> <top> 4026a9d8>)
```

или для более быстрого чтения:

```
(map class-name (class-precedence-list B)) ⇒ (B <object> <top>)
```

8.7.2 Упорядочивание Методов

Теперь, с идеей Списка Старшинста Классов, мы можем точно указать как можно упорядочить методы, когда применимы более чем один из методов обобщенной функции к аргументам вызова.

Правила таковы:

- применяемые методы упорядочиваются по порядку конкретизации, а наиболее конкретный метод используется вначале, а затем следующий, если этот метод вызывает `next-method`, и т.д.
- метод M1 более конкретен, чем другой метод M2, если первый конкретизирующий класс, который отличается, между определиниями M1 и M2, более конкретен, в определении M1, для соответствующих фактических аргументов вызова, чем конкретизирующий класс в определении M2
- Класс C1 более конкретен. чем другой класс C2, для объекта фактического класса C, если C1 предшествует(Старше) C2 в списке приоритетов класса C.

8.8 Интроспекция(Самонаблюдение у Классов-Introspection)

*Introspection*Интроспекция, или *reflection*Рефлексия, означает возможность динамического получения информации об объектах GOOPS. Это пожалуй лучше

всего илюстрируется рассмотрением объектно-ориентированнго языка который не обеспечивает никакой интроспекции, а именно Си++.

Ничто в Си++ не позволяет программе получать ответы на следующие типы вопросов:

- Что представляют собой данные этого объекта или класса?
- Какие классы наследует этот класс?
- Этот метод метод является виртуальным или не виртуальным?
- Если я вызываю `Employee::adjustHoliday()`, то какой класс содержит метод `adjustHoliday()` который будет применяться?

В Си++ ответы на эти вопросы могут быть даны только при просмотре исходного кода, если у вас есть к нему доступ. С другой стороны, GOOPS включает процедуры, позволяющие отвечать на эти вопросы - или их эквиваленты GOOPS - динамическе, т.е во время работы.

8.8.1 Классы

Класс GOOPS сам является экзепляром класса `<class>`, или подкласса `<class>`. Определение класса `<class>` имеет слоты, которые используются для описания свойств класса, включая следующие.

`class-name` *class* [primitive procedure]
Возвращает имя класса *class*. Им является значение слота `name` в *class*.

`class-direct-supers` *class* [primitive procedure]
Возвращает список, содержащий прямые суперклассы *class*. Это значение слота `direct-supers` переменной *class*.

`class-direct-slots` *class* [primitive procedure]
Взвращает список, содержащий определения словтов непосредственно заданных в *class*. Это значение слота `direct-slots` переменной *class*.

`class-direct-subclasses` *class* [primitive procedure]
возвращает список содержащий прямые подклассы класса *class*. Это значение слота `direct-subclasses` переменной *class*

`class-direct-methods` *class* [primitive procedure]
возвращает список всех методов обобщенных функций, которые используют класс *class* как формальный параметр конкретизации. Это значение слота непосредственно определенных методов класса `direct-methods`.

`class-precedence-list` *class* [primitive procedure]
Возвращает список старшинства классов для класса *class* (см. Раздел 8.7.1 [Class Precedence List], страница 803). Это значение слота `cpl` для *class*.

`class-slots` *class* [primitive procedure]
Возвращает список, содержащий определения слотов для всех слотов класса, включая любые слоты, котрые наследуются от суперклассов. Это значение слота `slots` в *class*.

`class-subclasses` *class* [procedure]
Возвращает список всех подклассов класса *class*.

`class-methods` *class* [procedure]
Возвращает список всех методов которые использует *class* или подклассы *class*, как один из его формальных параметров конкретизирующие параметры.

8.8.2 Экземпляры(Instances)

`class-of` *value* [primitive procedure]
Возвращает класс GOOPS для произвольного значения Scheme *value*.

`instance?` *object* [primitive procedure]
Возвращает `#t` если *object* является экземпляром какого либо класса GOOPS, иначе `#f`.

`is-a?` *object class* [procedure]
Взвращает `#t` если *object* является экземпляром класса *class* или одного из его подклассов.

Вы можете использовать предикат `is-a?`, чтобы узнать, принадлежит ли какое либо заданное значение данному классу, или `class-of` чтобы узнать класс заданного значения. Обратите внимание, что при загрузке GOOPS (код использующий модуль `(oop goops)`) встроенные классы, такие как `<string>`, `<list>` и `<number>` автоматически настраиваются, соотвественно всем типам Guile Scheme.

```
(is-a? 2.3 <number>) ⇒ #t
(is-a? 2.3 <real>) ⇒ #t
(is-a? 2.3 <string>) ⇒ #f
(is-a? '("a" "b") <string>) ⇒ #f
(is-a? '("a" "b") <list>) ⇒ #t
(is-a? (car '("a" "b")) <string>) ⇒ #t
(is-a? <string> <class>) ⇒ #t
(is-a? <class> <string>) ⇒ #f

(class-of 2.3) ⇒ #<<class> <real> 908c708>
(class-of #(1 2 3)) ⇒ #<<class> <vector> 908cd20>
(class-of <string>) ⇒ #<<class> <class> 8bd3e10>
(class-of <class>) ⇒ #<<class> <class> 8bd3e10>
```

8.8.3 Слоты(Slots)

`class-slot-definition` *class slot-name* [procedure]
Возвращает определение слота для слота с именем *slot-name* в классе *class*. *slot-name* должно быть символом.

`slot-definition-name` *slot-def* [procedure]
Извлекает и возвращает имя слота из *slot-def*.

`slot-definition-options` *slot-def* [procedure]
Извлекает и возвращает параметры слота из *slot-def*.

`slot-definition-allocation` *slot-def* [procedure]
Извлекает и возвращает параметры размещения из *slot-def*. Это значение ключевого слова `#:allocation` (см. Раздел 8.4 [allocation], страница 790), или `#:instance` если ключевое слово `#:allocation` отсутствует.

`slot-definition-getter` *slot-def* [procedure]
Извлекает и возвращает параметр getter из *slot-def*. Это значение ключевого слова `#:getter` (см. Раздел 8.4 [getter], страница 790), или `#f` если ключевое слово `#:getter` не задано.

`slot-definition-setter` *slot-def* [procedure]
Извлекает и возвращает параметр слота setter из *slot-def*. Это значение ключевого слова `#:setter` (см. Раздел 8.4 [setter], страница 790), или `#f` если ключевое слово `#:setter` отсутствует.

`slot-definition-accessor` *slot-def* [procedure]
Извлекает и возвращает параметр слота accessor из *slot-def*. Это значение ключевого слова `#:accessor` (см. Раздел 8.4 [accessor], страница 790), или `#f` если ключевое слово `#:accessor` отсутствует.

`slot-definition-init-value` *slot-def* [procedure]
Извлекает и возвращает параметр слота начальное значение(init-value) из *slot-def*. Это значение ключевого слова `#:init-value` (см. Раздел 8.4 [init-value], страница 790), или значение unbound если ключевое слово `#:init-value` отсутствует.

`slot-definition-init-form` *slot-def* [procedure]
Извлекает и возвращает опцию инициализации слота init-form из *slot-def*. Это значение ключевого слова `#:init-form` (см. Раздел 8.4 [init-form], страница 790), или unbound если значение ключевого слова `#:init-form` отсутствует.

`slot-definition-init-thunk` *slot-def* [procedure]
Извлекает и возвращает опцию слота init-thunk из *slot-def*. Это значение ключевого слова `#:init-thunk` (см. Раздел 8.4 [init-thunk], страница 790), или `#f` если ключевое слово `#:init-thunk` отстутствует.

`slot-definition-init-keyword` *slot-def* [procedure]
Извлекает и возвращает параметр слота init-keyword из *slot-def*. Это значение ключевого слова `#:init-keyword` (см. Раздел 8.4 [init-keyword], страница 790), или `#f` если ключевое слово `#:init-keyword` отсутствует.

`slot-init-function` *class slot-name* [procedure]
Возвращает функцию инициализации для слота с именем *slot-name* в классе *class*. *slot-name* должно быть символом.

Возвращенная функция инициализации включает эффекты стандартных опций слота `#:init-thunk`, `#:init-form` и `#:init-value`. Эти инициализации могут быть переопределены опцией слота `#:init-keyword` или специальным методом `initialize`, таким образом, в общем случае, возвращаемая функцией `slot-init-function` может быть недостаточна. Более подробное обсуждение, см. Раздел 8.4 [init-value], страница 790.

8.8.4 обобщенные Функции(Generic Functions)

обобщенная функция является экземпляром класса `<generic>`, или подкласса `<generic>`. Определение класса `<generic>` имеет слоты, которые используются для описания свойств обобщенной функции.

`generic-function-name` *gf* [primitive procedure]
Возвращает имя обобщенной функции *gf*.

`generic-function-methods` *gf* [primitive procedure]
Возвращает список методов обобщенной функции *gf*. Этот список является значением слота `methods` объекта *gf*.

Аналогичным образом, метод является экземпляром класса `<method>`, или подкласса `<method>`; и определение класса `<method>` имеет слоты, которые используются для описания свойств метода.

`method-generic-function` *method* [primitive procedure]
Возвращает обобщенную функцию, к которой принадлежит метод(*method*). Это значение слота *method* обобщенной функции `generic-function`.

`method-specializers` *method* [primitive procedure]
Возвращает список формальных параметров конкретизирующих метод *method*. Это значение слота `specializers` в методе(*method*).

`method-procedure` *method* [primitive procedure]
Возвращает процедуру, реализующую *method*. Это значение слота `procedure` в методе(*method*).

`method-source` [generic]
`method-source` (*m* <*method*>) [method]
Возвращает выражение для печати показывающее определение метода *m*.

```
(define-generic cube)

(define-method (cube (n <number>))
  (* n n n))

(map method-source (generic-function-methods cube))
⇒
((method ((n <number>)) (* n n n)))
```

8.8.5 Доступ к Слотам(Accessing Slots)

Любой слот, независимо от его размещения, может быть проверен, получен или установлен с использованием следующих четырех процедур.

`slot-exists?` *obj slot-name* [primitive procedure]
Возвращает `#t` если объект *obj* имеет слот с именем *slot-name*, иначе `#f`.

`slot-bound?` *obj slot-name* [primitive procedure]
Возвращает `#t` если слот с именем *slot-name* в объекте *obj* имеет значение, иначе `#f`.

`slot-bound?` вызывает обобщенную функцию `slot-missing` если объект *obj* не имеет слота именуемого *slot-name* (см. Раздел 8.8.5 [Accessing Slots], страница 808).

`slot-ref` *obj slot-name* [primitive procedure]
Возвращает значение слота с именем *slot-name* в объекте *obj*.

`slot-ref` вызывает обобщенную функцию `slot-missing` если объект *obj* не имеет слота называемого *slot-name* (см. Раздел 8.8.5 [Accessing Slots], страница 808).

`slot-ref` вызывает обобщенную функцию `slot-unbound` если именуемый слот объекта *obj* не имеет значения (см. Раздел 8.8.5 [Accessing Slots], страница 808).

`slot-set!` *obj slot-name value* [primitive procedure]
Устанавливает значение слота с именем *slot-name* в объекте *obj* значением *value*.

`slot-set!` вызывает обобщенную функцию `slot-missing` если объект *obj* не имеет слота именуемого *slot-name* (см. Раздел 8.8.5 [Accessing Slots], страница 808).

GOOPS хранит информацию о слотах в классах. Внутренне все эти процедуры работают проматривая определение слота для слота с именем *slot-name* в классе `(class-of` *`obj`*`)`, а затем используют определенные для слотов функции доступа "getter" и "setter" чтобы получить или установить значение этих слотов.

Следующие четыре процедуры отличаются от предыдущих, поскольку они принимают класс как явный аргумент, а не предполагают `(class-of` *`obj`*`)`. Поэтому они позволяют применить замыкания "getter" и "setter" определенные слотами в одном классе к экземпляру другого класса.

`slot-exists-using-class?` *class obj slot-name* [primitive procedure]
Возвращает `#t` если класс *class* имеет определение слота с именем *slot-name*, иначе `#f`.

`slot-bound-using-class?` *class obj slot-name* [primitive procedure]
Возвращет `#t` если применение функции `slot-ref-using-class` к тем же аргументам вызовет обобщенную функцию `slot-unbound`, иначе `#f`.

`slot-bound-using-class?` вызывает обобщенную функцию `slot-missing` если класс *class* не имеет определения слота для слота с именем *slot-name* (см. Раздел 8.8.5 [Accessing Slots], страница 808).

`slot-ref-using-class` *class obj slot-name* [primitive procedure]
Применяет замыкание "getter" для слота с именем *slot-name* класса *class* к объекту *obj*, и возвращает результат.

`slot-ref-using-class` вызывает обобщенную функцию `slot-missing` если класс *class* не имеет определения определения слота с именем *slot-name* (см. Раздел 8.8.5 [Accessing Slots], страница 808).

`slot-ref-using-class` вызывает обобщенную функцию `slot-unbound` если применение замыкания "getter" к объекту *obj* возвращает значение unbound (см. Раздел 8.8.5 [Accessing Slots], страница 808).

`slot-set-using-class!` *class obj slot-name value* [primitive procedure]
Применяет замыкание "setter" для слота с именем *slot-name* в классе *class* объекта *obj* и новым значением *value*.

`slot-set-using-class!` вызывает обобщенную функцию `slot-missing` если класс *class* не имеет определения слота с именем *slot-name* (см. Раздел 8.8.5 [Accessing Slots], страница 808).

Слоты, чье размещение определено для класса, а не для каждого экземпляра, можно ссылаться и устанавливать без необходимости указывать какой либо конкретный экземпляр.

`class-slot-ref` *class slot-name* [procedure]
Возвращает значение слота с именем *slot-name* в классе *class*. Именованный слот должен иметь распределение типа `#:class` или `#:each-subclass` (см. Раздел 8.4 [allocation], страница 790).

Если нет такого слота с кодом распределения `#:class` или `#:each-subclass`, `class-slot-ref` вызывает обобщенную функцию `slot-missing` с аргументами *class* и *slot-name*. Иначе, если значение слота не связано, `class-slot-ref` вызывает обобщенную функцию `slot-unbound`, с теми же самыми аргументами.

`class-slot-set!` *class slot-name value* [procedure]
Задает значение слота с именем *slot-name* в классе *class* значением *value*. Именованный слот должен иметь распределение типа `#:class` или `#:each-subclass` (см. Раздел 8.4 [allocation], страница 790).

Если он не имеет тип размещения `#:class` или `#:each-subclass`, `class-slot-ref` вызывает обобщенную функцию `slot-missing` с аргументами *class* или *slot-name*.

Когда вызов `slot-ref` или `slot-set!` указывает несуществующие имя слота или пытается получить ссылку на слот, значение которого не установлено, GOOPS вызывает одну из следующих обобщенных функций.

`slot-missing` [generic]
`slot-missing` (*class* <*class*>) *slot-name* [method]
`slot-missing` (*class* <*class*>) (*object* <*object*>) *slot-name* [method]
`slot-missing` (*class* <*class*>) (*object* <*object*>) *slot-name value* [method]
Когда приложение пытается ссылаться или устанавливать слот класса или экземпляра по имени и с недопустимым именем слота для данного класса *class* или объекта *object*, GOOPS вызывает обобщенную функцию `slot-missing`.

По умолчанию все методы вызывают `goops-error` с соответствующим сообщением.

`slot-unbound` [generic]
`slot-unbound` (*object* <*object*>) [method]
`slot-unbound` (*class* <*class*>) *slot-name* [method]
`slot-unbound` (*class* <*class*>) (*object* <*object*>) *slot-name* [method]
Когда приложение пытается ссылатья на слот класса или экземпляра, а слот не имеет связанного значения, GOOPS вызывает обобщенную функцию `slot-unbound`.

По умолчанию все методы вызывают `goops-error` с соответствующим сообщением.

8.9 Обработка Ошибок

Процедура `goops-error` вызывается методами по умолчанию для генерации соответствующей ошибки следующими обобщенными функциями:

- `slot-missing` (см. Раздел 8.8.5 [slot-missing], страница 808)
- `slot-unbound` (см. Раздел 8.8.5 [slot-unbound], страница 808)
- `no-method` (см. Раздел 8.6.6 [no-method], страница 802)
- `no-applicable-method` (см. Раздел 8.6.6 [no-applicable-method], страница 802)
- `no-next-method` (см. Раздел 8.6.6 [no-next-method], страница 802)

Если вы настраиваете эти функции для определенных классов или метаклассов, вы все равно можете захотеть использовать `goops-error`, чтобы сигнализировать о любых обнаруженных ошибках.

`goops-error` *format-string arg ...* [procedure]
: Выбрасывает ошибку с ключем `goops-error` и сообщением об ошибке, построенному из *format-string* и аргументов *arg ...*. Форматирование сообщений об ошибках выполняется с помощью `scm-error`.

8.10 Некоторые функции работы с Объектами GOOPS

Здесь мы рассмотрим некоторые моменты, касающиеся объектов GOOPS, которые не достаточно существенны, что бы им посвятить отдельные разделы.

Равенство Объектов

Когда GOOPS загружается, `eqv?`, `equal?` и `=` становятся обобщенными функциями, и вы можете определять методы для них, специалзированные для ваших классов, чтобы контролировать то, как работают различные виды равенств для ваших классов.

Например, процедура `assoc`, для поиска записи в ассоциированном списке alist, настроена на использование `equal?` для определения когда начало списка alist(car) совпадет с ключевым параметром, с которым вызван `assoc`. Следовательно, если вы определили новый класс, и хотите испольовать экземпляры этого класса в качестве ключей alist, вы можете определить метод `equal?`, для вашего класса, чтобы точно контролировать соответствие при работе `assoc`.

Клонирование Объектов

`shallow-clone` [generic]
`shallow-clone` (*self* <*object*>) [method]
: Возвращает "поверхностную" копию объекта(*self*). По умолчанию метод создает поверхностную копию размещающую новый экземплфр и копируюя значения слотов из себя в новый экземпляр. Каждое значение слота копируется как немедленное значение или как ссылка.

`deep-clone` [generic]
`deep-clone` (*self* <*object*>) [method]

Возвращает “глубокую” копию объекта(*self*). По умолчанию метод делающий глубокую копию выделяет место для нового экземпляра и копирует или клонирует значение слотов от себе в новый экземпляр. Если значение слота является экземпляром (удовлетворяет `instance?`), оно клонируется вызовом `deep-clone` с этим значением. Другие слоты копируются либо как непосредственные значения, либо по ссылке.

Write and Display

`write` *object port* [primitive generic]
`display` *object port* [primitive generic]

Когда GOOPS загружается, `write` и `display` становятся обобщенными функциями со специальными методами для печати

- objects - экземпляр класса `<object>`
- foreign objects - экземпляр класса `<foreign-object>`
- classes - экземпляр класса `<class>`
- generic functions - экземпляр класса `<generic>`
- methods - экземпляр класса `<method>`.

`write` и `display` печатают не-GOOPS значения, также как примитивные Guile функции `write` и `display`.

В дополнение к упомянутым случаям вы можете конечно определить методы `write` и `display` для ваших классов, чтобы настроить, как будут печататься экземпляры этих классов.

8.11 Метаобъектный Протокол

До сего момента, мы говорили только о GOOPS, без сопоставления его с идеей протокола метаобъектов. Есть еще несколько тем, которые могут обсуждаться отдельно, первая - это переопределение класса, и вторая изменение класса с существующими экземплярами, но на практике разработчики использующие их, должны быть достаточно продвинутыми, чтобы понять также и протокол метаобъектов, и, вероятно, точно будут использовать этот протокол для настройки того, что должно происходить во время этих событий.

Итак давайте начнем. GOOPS основан на “метаобъектном протоколе” (также известный как “МОП”) производном из используемых в CLOS (the Common Lisp Object System), tiny-clos (небольшая реализация Scheme подмножества функций CLOS) и STKlos.

МОП лежит в основе многих возможных настроек GOOPS — таких как определение метода `initialize` для настройки инициализации экземпляров определенного класса приложений и понимание МОП делает гораздо более легким объяснение таких настроек в правильном порядке. И на более глубоком уровне понимание МОП является ключевой частью понимания GOOPS, и также полного использования мощи GOOPS, настройки поведения самого GOOPS.

8.11.1 Метаобъекты и Метаобъектный Протокол(МОП)

Основными строительными блоками GOOPS являются классы, определения слотов, экзепляров, обобщенных функций и методов. Класс - это группировка отношений наследования и определения слотов. Экземпляр объекта со слотами, которые размещаются следуя правилам, предусмотренными суперклассами этого класса и определениями слотов. Обобщенная функция представляет собой набор методов и правил определения какой из этих методов применяется при вызове обобщенной функции. Метод представляет собой процедуру и набор спецификаторов определяющих тип аргументов, к которым применима процедура.

Из этих объектов, GOOPS представляет классы, обобщенные функции и методы как "метаобъекты". Другими словами, значения в программе GOOPS описывающие классы, обобщенные функции и методы, сами являются экземплярами (или "объектами") специальных классов GOOPS которые инкапсулируют поведение соответственно, классов, обобщенных функций и методов.

(Другими двумя объектами являются определение слотов и экземпляры. Определения слотов не являются строго экземплярами, но каждое определение слота связано с классом GOOPS, который определяет поведение слота в отношении доступности и защиты от сборки мусора. Экземпляры, конечно, объекты в обычном смысле, и нет никакой пользы от мысли о них как о метаобъектах.)

"Метаобъектный протокол" (или "МОП") это спецификация обобщенных функций, которые определяют поведение этих метаобъектов и обстоятельств при которых эти обобщенные функции вызываются.

В качесте конкретного примера того, что это означает, рассмотрим как GOOPS вычисляет набор слотов для класса, который определяется с использованием `define-class`. Желательным набором слотов является объединение прямых слотов и слотов всех его суперклассов. Но `define-class` сам не выполняет этот расчет. Вмето этого существует метод `initialize` обобщенной функции который специализирован для экземпляров типа `<class>`, и именно этот метод выполняет расчет слотов.

`initialize` это обобщенная функция, которую GOOPS вызывает всякий раз, когда создается новый экземпляр, сразу после выделения памяти для нового экземпляра, чтобы инициализировать слоты нового экземпляра. Последовательность шагов следующая.

- `define-class` использует `make` для создания нового экземпляра класса `<class>`, передающегося как аргумент инициализации суперкласса, определения слотов и параметров класса, которые были указаны в форме `define-class`.
- `make` выделяет память для нового экземпляра и вызывает обобщенную функцию `initialize` для инициализации слотов нового экземляра.
- Обобщенная функция `initialize` применяет метод, который специализирован для экземпляров типа `<class>`, и этот метод выполняет расчет слотов.

Другими словами, вместо того чтобы быть жестко закодированным в `define-class`, поведение определения класса по умолчанию инкапсулируется методами обобщенной функции, которые специализируются для класса `<class>`.

Можно создать новый класс, который наследует от `<class>`, который называется "метаклассом", и написать новый метод `initialize`, который специализирован для

экземпляров нового метакласса. Затем, если форма `define-class` включает опцию класса `#:metaclass`, чье значение это новый метакласс, класс который будет определен формой `define-class` будет экземпляром нового метакласса, а не стандартного `<class>`, и будет определен в соответствии с новым методом инициализации `initialize`. Таким образом, расчет слотов по умолчанию, а также любой другой аспект отношения нового класса с его суперклассами может быть изменен или переопределен.

Аналогичным образом, поведение обобщенных функций может быть изменено или переопределено созданием нового класса, который наследуется от стандартного класса обобщенных функций `<generic>`, написанием соответствующих методов, которые конкретизируются в новом классе, и создают новые обобщенные функции, являющиеся экземплярами нового класса обобщенных функций.

То же самое верно и для метаобъектов методов. И тот же самый базовый механизм применяется автором класса для написания метода `initialize` который конкретизирован для применения их класса, чтобы инициализировать экземпляры этого класса.

Такова мощь МОП. Обратите внимание, что `initialize` это просто одна из большого числа обобщенных функций, которые могут быть настроены для изменения поведения объектов приложения и классов и самого GOOPS. Каждый следующий раздел охватывает конкретную область функциональности GOOPS, и описывает обобщенные функции, которые имеют отношение к настройке этой области.

8.11.2 Метаклассы

Метакласс(*metaclass*) это класс, объекты которого представляют классы GOOPS. Или более лаконично, метакласс это класс классов.

Большинство классов GOOPS имеют в качестве метакласса класс `<class>` и по умолчанию, любой новый класс создаваемый с использованием `define-class` имеет в качестве метакласса класс `<class>`.

Но что это значит? Чтобы узнать, давайте подробнее рассмотрим, что происходит когда создается новый класс с помощью `define-class`:

```
(define-class <my-class> (<object>) . slots)
```

Guile расширяет это до чего-то вроде:

```
(define <my-class> (class (<object>) . slots))
```

который в свою очередь расширяется до:

```
(define <my-class>
  (make <class> #:dsupers (list <object>) #:slots slots))
```

Как видно из этого расширения, результирующее значение `<my-class>` является экземпляром класса `<class>` со значениями слотов определяемыми суперклассами и определением слотов для класса `<my-class>`. (`#:dsupers` и `#:slots` являются ключевыми словами инициализации для слотов `dsupers` и `dslots` в классе `<class>`.)

Предположим теперь, что вы хотите определить новый класс с метаклассом, отличным от стандартного `<class>`. Это делается путем написания:

```
(define-class <my-class2> (<object>)
   slot ...
   #:metaclass <my-metaclass>)
```

и Guile расширяет это(*this*) определение до чегото вроде:

```
(define <my-class2>
  (make <my-metaclass> #:dsupers (list <object>) #:slots slots))
```

В этом случае значение `<my-class2>` является экземпляром более специализированного класса `<my-metaclass>`. Обратите внимание, что сам `<my-metaclass>` должен быть предварительно определен как подкласс класса `<class>`. Для более полного обсуждения того, когда и как полезно определять новые метаклассы, см. Раздел 8.11.3 [MOP Specification], страница 815.

Теперь давайте создадим экземпляр `<my-class2>`:

```
(define my-object (make <my-class2> ...))
```

Все последующие утверждения являются правильными выражениями отношений между `my-object`, `<my-class2>`, `<my-metaclass>` и `<class>`.

- `my-object` это экземпляр класса `<my-class2>`.
- `<my-class2>` это экземпляр класса `<my-metaclass>`.
- `<my-metaclass>` это экземпляр класса `<class>`.
- Класс объекта `my-object` это `<my-class2>`.
- Класс объекта `<my-class2>` это `<my-metaclass>`.
- Класс объекта `<my-metaclass>` это `<class>`.

8.11.3 Спецификация МОП

Цель спецификации МОП в этой главе — указать все настраиваемые обобщенные функции вызовы которых могут выполняться стандартным синтаксисом GOOPS, процедурами и методами, и объяснить протокол для настройки таких вызовов.

Вызовы обобщенных функций настраиваются, если типы аргументов, к которым она применяется не полностью определяются лексическим контекстом, в котором этот вызов появляется. Например, вызов инициализации (`initialize` *`instance initargs`*) по умолчанию метод `make-instance` настраивается, поскольку тип аргумента *`instance`* определяемый классом, передается `make-instance`.

(Тогда как — даем контр-пример — (`make` `<generic>` `#:name ',name`) вызов в `define-generic` не настраивается, потому что все его аргументы лексически определенные типы.)

При использовании этого правила для определения того, настраивается ли данный вызов обобщенной функции, мы игнорируем аргументы, которые, как ожидается, будут обрабатываться в определениях методов как один списковый аргумент “rest”.

Для каждого настраиваемого вызова обобщенной функции, протокол вызова (*invocation protocol*) разъясняется указанием:

- что, Концептуально, для чего предназначен применяемый метод
- какие предположения, если таковые имеются, вызывающий делает о побочных эффктах применяемого метода.
- что вызывающий ожидает получить в качестве возвращаемого значения применяемого метода.

8.11.4 Протокол Создания Экземпляров

`make <class> . initargs` (method)

- `allocate-instance class initargs` (generic)

  Применяемый метод `allocate-instance` должен выделять хранилище для нового экземпляра класса *class* и вернуть не инициализированный экзепляр.

- `initialize instance initargs` (generic)

  instance это не инициализированный экземпляр, возвращаемый `allocate-instance`. Применяемый метод должен инициализировать новый экземпляр в любом смысле, подходящем для его класса. Возвращаемое методом значение игнорируется.

`make` сама является обобщенной функцией. Следовательно сам вызов `make` может быть настроен в зависимости от метакласса нового экземпляра, который является более специализированным чем класс по умолчанию `<class>`, определением метода `make` специлизированного для этого метакласса.

Обычно, однако, применяется метод для классов с метаклассом `<class>`. Этот метод вызывает две обобщенные функции:

- (allocate-instance *class* . *initargs*)
- (initialize *instance* . *initargs*)

`allocate-instance` распределяет память для нового экземпляра и возвращает новый экземпляр, не инициализированным. Вы можете настроить `allocate-instance`, например, если вы хотите предоставить GOOPS обертку вокруг некоторой другой объектной системы программирования.

Для этого вы создадите специализированный метакласс, который будет дейстовать как метакласс для всех классов и экземпляров из другой системы. Затем определяйте метод `allocate-instance`, специализированный для этого метакласса, который вызывает примитив Guile Си функцию (или код FFI), который в свою очередь, выделит паамять для нового экземпляра с использованием интерфейса другой объектной системы.

В этом случае для полной системы вам также потребуется настроить несколько других обобщенных функций, таких как `make` и `initialize`, так что GOOPS знает как создавать классы из другой системы, получать доступ к слотам объектов, и т.д.

`initialize` инициализирует экземпляр, возвращаемый `allocate-instance`. Стадартный метод GOOPS выполняет инициализацию, соответствующую классу экземпляра.

- На наименее специализированном уровне выполняется метод для экземпляров типа `<object>` внутренняя инициализация экземпляра GOOPS, и инициализирует слоты экземпляра в соответствии с определениями слотов и любых ключевых слов инициализации слота, которые появляются в *initargs*.
- Метод для экземпляров типа `<class>` вызывает `(next-method)`, затем выполняет инициализацию класса, описанную в разделе Раздел 8.11.5 [Class Definition Protocol], страница 817.
- и т.д. для обобщенных функций, методов, операторов классов . . .

Аналогичным образом вы можете настроить инициализацию экземпляров любых определенных в приложении классов определяя метод `initialize` специализированный для этого класса.

Представьте себе класс, чьи экземпляры должны быть инициализированы во время создания экземпляра путем запросов к базе данных. Хотя это может быть возможно, это сочетание ключевых слов `#:init-thunk` и замыкания в определениях слота, возможно более аккуратно написать метод `initialize` для класса, который опрашивает базу данных один раз и инициализирует все зависимые значения слотов в соответствии с результатами запроса.

8.11.5 Протокол Определения Класса

Ниже приведена сводная диаграмма синтаксиса, процедур и обобщенных функций, которые могут учавствовать в определении класса.

`define-class` (syntax)

- `class` (syntax)
  - `make-class` (procedure)
    - `ensure-metaclass` (procedure)
    - `make` *`metaclass`* `...` (generic)
      - `allocate-instance` (generic)
      - `initialize` (generic)
        - `compute-cpl` (generic)
          - `compute-std-cpl` (procedure)
        - `compute-slots` (generic)
        - `compute-get-n-set` (generic)
        - `compute-getter-method` (generic)
        - `compute-setter-method` (generic)
- `class-redefinition` (generic)
  - `remove-class-accessors` (generic)
  - `update-direct-method!` (generic)
  - `update-direct-subclass!` (generic)

Каждый вышеописанный шаг обозначенный как “обобщенный(generic)”, может быть настроен и детально покажем ниже как это сделать “правильно(correct)” поскольку он описывает какой метод по умолчанию используется обобщенной функцией. Например, если вы пишете метод `initialize`, для некоторого метакласса, который не вызывает `next-method` и не вызывает `compute-cpl`, тогда `compute-cpl` не будет вызван когда класс определяется с этим метаклассом.

Форма `(define-class ...)` (см. Раздел 8.2 [Class Definition], страница 787) расширяется до выражения, которое:

- проверяет, что оно вычисляется только на верхнем уровне
- определяет любые аксессоры, которые подразумеваются определениями слотов *slot-definition*

- использует `class` для создания нового класса
- проверяет предыдущее определение для класса с именем *name* и, если оно найдено, обрабатывает переопределение путем вызова `class-redefinition` (см. Раздел 8.12 [Redefining a Class], страница 825).

`class` *name* (*super ...*) *slot-definition ... class-option ...* [syntax]

Возвращает вновь созданный класс, который наследуется от *super*, с непосредственно определенными слотами, определенными *slot-definition* и *class-option*. Формат *slot-definition* и *class-option*, см. Раздел 8.2 [define-class], страница 787.

`class` расширяется до выражения, которое

- обрабатывает параметры определения класса и слота, чтобы проверить правильно ли они сформированы, чтобы конвертировать параметр `#:init-form` в параметр `#:init-thunk`, поставляющий параметр среды по умолчанию, (текущая среда верхнего уровня) и вычисляет все биты которые необходимо вычислить.
- вызывает `make-class` для создания класса с обработанными и вычисленными параметрами.

`make-class` *supers slots class-option ...* [procedure]

Возвращает вновь созданный класс, который наследуется от *supers*, с непосредственно определенными слотами *slots* и *class-option*. Формат определения слотов *slots* и *class-option*s, см Раздел 8.2 [define-class], страница 787, за исключением того, что для `make-class`, *slots* представляет собой отдельный список определения слотов.

`make-class`

- добавляет `<object>` в список *supers* если *supers* пустой или если ни один из классов в *supers* не имеет `<object>` в своем списке старшинства классов.
- по умолчанию заданы параметры `#:environment`, `#:name` и `#:metaclass`, если они не являются определенными в *options*, в текущей среде верхнего уровня несвязанное значени и `(ensure-metaclass supers)` соответственно
- проверяет наличие повторяющихся классов в *supers* и дублиурующиеся имена слотов в *slots*, и сигнализирует об ошибке если имеются дубликаты.
- вызывает `make`, передает метакласс как первый параметр и все остальные параметры как ключевые слова со значениями.

`ensure-metaclass` *supers env* [procedure]

Возвращает метакласс, подходящий для класса, который наследуется от списка классов в *supers*. Возвращенный метакласс это объединение наследуемых метаклассов из классов в *supers*.

В простейшем случае, все the *supers* являются простыми классами с метаклассом `<class>` и возвращаемый метакласс это просто класс `<class>`.

Для более сложного примера предположим, что *supers* содержит один класс с метаклассом `<operator-class>` и один класс с метаклассом `<foreign-object-class>`. Тогда возвращенный метакласс будет классом, который наследуется от обоих метаклассов от `<operator-class>` и от `<foreign-object-class>`.

Если *supers* это пустой список, `ensure-metaclass` возвращает метакласс GOOPS по умолчанию, т.е. метакласс `<class>`.

GOOPS хранит список метаклассов, созданных `ensure-metaclass`, так что каждый требуемый тип метакласса создается только один раз.

Параметр `env` игнорируется.

`make` *metaclass initarg ...* [generic]
metaclass это метакласс класса, который определяется, либо берется из опции класса `#:metaclass`, либо вычисляется с помощью `ensure-metaclass`. Применяемый метод должен создать и вернуть полностью инициализированный метаобъект класса для определения нового класса.

Вызов (`make` ***metaclass initarg ...***) является частным случаем протокола создания экземпляра, рассмотренный в предыдущем разделе. Он создаст метаобъект класса с метаклассом *metaclass*. По умолчанию этот метаобъект будет инициализирован методом `initialize` специализированный для экземпляров типа `<class>`.

Метод `initialize` для классов (signature (`initialize <class> initargs`)) вызывает следующие обобщенные функции.

- `compute-cpl` ***class*** (generic)

  Применяемый метод должен вычислить и вернуть список старшинства классов для класса *class* как список метаобъектов классов. Когда вызывается `compute-cpl` следующие слоты метаобъекта *class* должны быть инициализированы: `name`, `direct-supers`, `direct-slots`, `direct-subclasses` (empty), `direct-methods`. Значение возвращаемое `compute-cpl` будет храниться в слоте `cpl`.

- `compute-slots` ***class*** (generic)

  Применяемый метод должен вычислять и возвращать слоты (объединение прямых и наследуемых) для класса *class* как список определений слотов. Когда вызывается `compute-slots` все слоты метаобъекта *class* упомянутые для `compute-cpl` должны быть инициализированы, плюс следующие слоты: following: `cpl`, `redefined` (`#f`), `environment`. Значение возвращаемое `compute-slots` будет храниться в слоте `slots`.

- `compute-get-n-set` ***class slot-def*** (generic)

  `initialize` вызывает `compute-get-n-set` для каждого слота вычисленного `compute-slots`. Применяемый метод должен вычислить и вернуть пару замыканий, которые соответственно, получают и устанавливают значение указанного слота. get замыкание должно иметь арность 1 и ожидать единственный аргумент, который является экземпляром слота, значение которого должно быть извлечено. set замыкание должно иметь арность 2 и принимать два аргумента, где первым аргументом является экземпляр слота значение которого должно быть установлено, а второй аргумент - новое значение для этого слота. Замыкания должны быть возвращены в виде списка из двух элементов: (`list` ***get set***).

  Замыкания возвращаемые `compute-get-n-set` сохраняются как часть значения слота `getters-n-setters` метаобъекта *class*. В частности, значение этого слота

представляет собой список с тем же количеством элементов, что и слоты в классе, и каждый элемент выглядит так:

(*slot-name-symbol init-function . index*)

или так:

(*slot-name-symbol init-function get set*)

Где замыкания get и set заменяются индексом *index*, слот является экземпляром слота и индекса *index* - индекс слота в базовой структуре: GOOPS знает как получить и установить значение такого слота, и следовательно, не требуется специально конструировать get и set замыкания для этого случая. В противном случае замыкания *get* и *set* возвращаемые `compute-get-n-set`.

Структура значения слотов `getters-n-setters` важна при понимании следующих настраиваемых обобщенных функций которые вызывает `initialize` . . .

- `compute-getter-method` ***class gns*** (generic)

  `initialize` вызывает `compute-getter-method` для каждого из слотов класса(определенных `compute-slots`), который включает в себя опцию слота `#:getter` или `#:accessor`. *gns* это элемент метаобъекта класса *class* слот `getters-n-setters` который определяется как слот с запросами ссылки и установки, как описано выше, в `compute-get-n-set`. Применяемый метод должен создавать и возвращать метод, который специализирован для экземпляров типа *class* и использовать get замыкание для получения значения слота. `initialize` использует `add-method!` для добавления возвращаемого метода к обобщенной функции именуемой определением опции слота `#:getter` или `#:accessor`.

- `compute-setter-method` ***class gns*** (generic)

  `compute-setter-method` вызывается с теми же аргументами, что и `compute-getter-method`, для каждого слота класса, который включает в себе опцию `#:setter` или `#:accessor`. Применяемый метод должен создать и вернуть метод, который специализирован для экземпляра типа *class* и использовать замыкание set для установки значения слота. `initialize` использует `add-method!` для добавления возвращаемого метода к обобщенной функции именуемой определением опции слота `#:setter` или `#:accessor`.

8.11.6 Настройка Определения Класса

Если метакласс нового класса является чем-то более специализированным, чем стандартный класс `<class>`, то тип *class* в вышеперечисленных вызовах более специализирован чем `<class>`, и следовательно, он может определить методы обобщенных функций, специализированные для метакласса нового класса, которые могут изменить или отменять поведение по умолчанию для `initialize`, `compute-cpl` или `compute-get-n-set`.

`compute-cpl` вычисляет список старшинства классов (“CPL”) для нового класса (см. Раздел 8.7.1 [Class Precedence List], страница 803), и возвращает его как список объектов класса. CPL важен, поскольку он определеят упорядочение суперклассов, которые используются, когда обобщенная функция вызывается для экземпляра класса, чтобы решить, какой из доступных методов обобщенной функции является наиболее конкретным(подходящим). Следовательно, `compute-cpl` можно настроить в

порядке изменения для изменения алгоритма упорядочения CPL для всех классов со специальным метаклассом.

Алгоритм CPL по умолчанию инкапсулируется процедурой `compute-std-cpl`, которая вызывается методом по умолчанию `compute-cpl`.

`compute-std-cpl` *class* [procedure]

Вычисляет и возвращает список старшинства классов для класса *class* в соответствии с алгоритмом описанном в Раздел 8.7.1 [Class Precedence List], страница 803.

`compute-slots` вычисляет и возвращает список всех определений слотов для нового класса. По умолчанию, этот список включает в себя непосредственно определенные в классе слоты из формы определения `define-class`, плюс определения которые определения слотов, которые унаследованы от суперклассов. По умолчанию метод `compute-slots` использует CPL вычисленный `compute-cpl` для вычисления этого объединения слотов, с правилом, что слоты унаследованные от суперклассов, заменяются непосредственными слотами с тем же именем. Одной из возможных причин для настройки `compute-slots` будет реализация альтернативной стратегии разрешения конфликтов имен слотов.

`compute-get-n-set` вычисляет низкоуровневые замыкания, которые будут использоваться для получения и установки значений определенных слотов, и возвращет их в списке с двумя элементами.

Возвращемые замыкания зависят от того как распределяется хранилище для данного слота. Стандартный метод `compute-get-n-set` специализированный для классов типа `<class>`, обрабатывает стандартные значения GOOPS для опций слота `#:allocation` (см. Раздел 8.4 [allocation], страница 790). Определяя новый метод `compute-get-n-set` для более специализированного метакласса, возможна поддержка нового типа распределения слотов.

Предположим, вы хотите создать большое количество экзепляров некоторого класса со слотом, который должен быть разделен между некоторыми, но не всеми экземплярами этого класса, скажем каждый 10 экземпляров должны испльзовать одно и тоже хранилище. В следующем примере показано, как реализовать и использовать для этого новый тип расределения слотов.

```
(define-class <batched-allocation-metaclass> (<class>))

(let ((batch-allocation-count 0)
      (batch-get-n-set #f))
  (define-method (compute-get-n-set
                  (class <batched-allocation-metaclass>) s)
    (case (slot-definition-allocation s)
      ((#:batched)
       ;; If we've already used the same slot storage for 10 instances,
       ;; reset variables.
       (if (= batch-allocation-count 10)
           (begin
             (set! batch-allocation-count 0)
```

```
            (set! batch-get-n-set #f)))
          ;; If we don't have a current pair of get and set closures,
          ;; create one.  make-closure-variable returns a pair of closures
          ;; around a single Scheme variable - see goops.scm for details.
          (or batch-get-n-set
              (set! batch-get-n-set (make-closure-variable)))
          ;; Increment the batch allocation count.
          (set! batch-allocation-count (+ batch-allocation-count 1))
          batch-get-n-set)

        ;; Call next-method to handle standard allocation types.
        (else (next-method)))))

(define-class <class-using-batched-slot> ()
  ...
  (c #:allocation #:batched)
  ...
  #:metaclass <batched-allocation-metaclass>)
```

Использование методов `compute-getter-method` и `compute-setter-method` описано в Раздел 8.11.5 [Class Definition Protocol], страница 817.

`compute-cpl` и `compute-get-n-set` вызываются стандартным методом `initialize` для классов, метаклассом которых является класс `<class>`. Но сам `initialize` можно также модифицировать, определяя метод `initialize` специализированный для нового класса метакласса. Такой метод может полностью переопределить стандартное поведение, не вызывая методов `(next-method)` вообще, но более типично он выполнял бы дополнительные шаги инициализации класса до или/и после вызова `(next-method)` для стандартного поведения.

8.11.7 Определение Методов

`define-method` (syntax)

- `add-method!` *`target method`* (generic)

`define-method` вызывает обобщенную функцию `add-method!` для обработки добавления нового метода к множеству возможных целей. GOOPS включает методы обработки *target* такие как:

- обобщенная функция (the most common case)
- процедура
- обобщенный примитив (см. Раздел 8.6.2 [Extending Primitives], страница 797)

Определив дальнейшие методы для for `add-method!`, вы теоретически можете обрабатывать добавление методов к другим типам целей.

8.11.8 Определение Методов Изнутри

`define-method`:

- проверяет форму первого параметра и применяет следующие шаги к аксессору установщика если он имеет форму `(setter ...)`

- интерполирует вызов `define-generic` или `define-accessor` если обобщенная функция не является уже определенной с указанным именем.
- вызывает `method` с параметром *parameter*s и телом *body*, чтобы создать экземпляр нового метода.
- вызывает `add-method!` чтобы добавить этот метод к соответствующей обобщенной функции.

`method` (*parameter* ...) *body* ... [syntax]
Создает метод, специализаторы которого определяются классами в параметрах *parameter* и чьё определение процедуры строиться из символов *parameter* и формы *body*.

Параметры *parameter* и *body* должны быть такими же, как и для `define-method` (см. Раздел 8.6 [define-method], страница 795).

`method`:

- извлекает формальные и специализированные классы из *parameter*, по умолчанию класс не указанных параметров `<top>`
- создает замыкание, используя формальные параметры и форму *body*
- вызывает `make` с метаклассом `<method>` и специализаторами и замыканием используя ключевые слова `#:specializers` и `#:procedure`.

`make-method` *specializers procedure* [procedure]
Создает метод используя *specializers* и *procedure*.

specializers должен быть списком классов, который задает комбинацию параметров к которым этот метод применим.

procedure должна быть замыканием, которое будет применяться к параметрам обобщенной функции когда этот метод вызывается.

`make-method` это простая обертка вокруг `make` с метаклассом `<method>`.

`add-method!` *target method* [generic]
Обобщенная функция для добавления метода *method* к *target*.

`add-method!` (*generic* <*generic*>) (*method* <*method*>) [method]
Добавляет метод *method* к обобщенной функции *generic*.

`add-method!` (*proc* <*procedure*>) (*method* <*method*>) [method]
Если *proc* это процедура с возможностью обобщения (см. Раздел 8.6.2 [generic-capability?], страница 797), обновляет ее до обобщенного примитива и добавляет метод *method* к его определению обобщенной функции.

`add-method!` (*pg* <*primitive-generic*>) (*method* <*method*>) [method]
Добавляет метод *method* к определению обобщенной функции *pg*.

Реализация: `(add-method! (primitive-generic-generic pg) method)`.

`add-method!` (*whatever* <*top*>) (*method* <*method*>) [method]
Выбрасывает ошибку укзывающую, что *whatever* не является допустимой обобщенной функцией.

8.11.9 Обобщенные Функции изнутри

`define-generic` вызов `ensure-generic` обновляет ранее существующее значение процедуры, или `make` с укзанием в качестве метакласса `<generic>` создает новую обобщенную функцию.

`define-accessor` вызов `ensure-accessor` обновляет ранее существовавшее значение процедуры, или `make-accessor` создает новый аксессор.

`ensure-generic` *old-definition* [*name*] [procedure]
Возвращает обобщенную функцию с именем *name*, если это возможно, используя или обновляя старое определение(*old-definition*). Если не указано *name* по умолчанию равно `#f`.

Если старое определение(*old-definition*) уже является обобщенной функцией, вызов возвращает unchanged.

Если старое определение(*old-definition*) это процедура Scheme или процедура с установщиком(procedure-with-setter), `ensure-generic` возвращает новую обобщенную функцию, которая использует старое определние(*old-definition*) для процедуры по умолчанию и для установщика.

В противном случае `ensure-generic` возвращает новую обобщенную процедуру без значения по умолчанию и без методов.

`make-generic` [*name*] [procedure]
Возвращает новую обобщенную функцию с именем `(car` *name*`)`. Если неуказано имя *name* по умолчанию `#f`.

`ensure-generic` вызывает `make` с метаклассом `<generic>` и `<generic-with-setter>`, в зависимости от предыдущего значения переменной, которую он пытается обновить.

`make-generic` это просто оболочка для `make` с метаклассом `<generic>`.

`ensure-accessor` *proc* [*name*] [procedure]
Возвращает аксессор с именем *name*, если возможно используя или обновляя *proc*. Если не задано имя *name* по умолчанию `#f`.

Если *proc* уже является аксессором, он возвращает unchanged.

Если *proc* это процедура Scheme, процедура с установщиком (procedure-with-setter) или обобщенная процедура, `ensure-accessor` возвращает аксессор который повторно использует элементы *proc*.

В противном случае `ensure-accessor` возвращает новый аксессор без умолчания и без методов.

`make-accessor` [*name*] [procedure]
Возвращает новый аксессор с именем `(car` *name*`)`. Если не указано имя *name* по умолчанию `#f`.

`ensure-accessor` вызывает `make` с метаклассом `<generic-with-setter>`, а также вызывает `ensure-generic`, `make-accessor` и (хвостовой рекурсией) `ensure-accessor`.

`make-accessor` вызывает `make` дважды, первый раз с метаклассом `<generic>` чтобы создать обобщенную функцию установщика, затем с метаклассом `<generic-with-setter>` для создания аксессора, передавая обобщенную функцию установщик как значение ключевого слова `#:setter`.

8.11.10 Вызовы Обобщенной Функции

Существует подробный и настраиваемый протокол, использующийся в процессе вызова обобщенной функции — т.е., в процессе определения того какие из методов обобщенной функции применимы к текущим аргументам и какие из них применять. Вот краткая диаграмма запутанного протокола применения обобщенной функции.

`apply-generic` (generic)

- `no-method` (generic)
- `compute-applicable-methods` (generic)
- `sort-applicable-methods` (generic)
  - `method-more-specific?` (generic)
- `apply-methods` (generic)
  - `apply-method` (generic)
  - `no-next-method` (generic)
- `no-applicable-method`

У нас нет пока полной документации. См. Код для полной информации (`oop/goops.scm`).

8.12 Переопределение класса(Redefining a Class)

Предположим, что класс `<my-class>` определяется с помощью `define-class` (см. Раздел 8.2 [define-class], страница 787), со слотами, которые имеют функции доступа, и что приложение создало несколько экземпляров `<my-class>` используя `make` (см. Раздел 8.3 [make], страница 789). Что происходит, если `<my-class>` переопределяется вызовом `define-class`?

8.12.1 Поведение по умолчанию при пероопределении класса(Default Class Redefinition Behaviour)

Ответ GOOPS по умолчанию на этот вопрос выглядит следующим образом:

- Все существующие прямые экземпляры `<my-class>` преобразуются в экземпляры нового класса. Это достигается за счет сохранения значений слотов, которые существуют как в старом, так и в новом определении класса и инициализируя значение новых слотов обычным способом. (см. Раздел 8.3 [make], страница 789).
- Все существующие подклассы `<my-class>` переопределяются, как будто выражения `define-class` которые их определили, были выполнены после переопределения `<my-class>`, и описанный здесь процесс переопределения класса применяется рекурсивно к переопределяемым подклассам.
- Как только все его экземпляры и подклассы обновлены, метообъект класса ранее связанный с переменной `<my-class>` больше не нужен и поэтому можно отправить его в мусор(он будет уничтожен сборщиком мусора).

Чтобы все было в порядке, GOOPS также необходимо немного поработать над методами, которые связаны с переопредленным классом.

- Слоты методов доступа для слотов из старого определения должны быть удалены из обобщенной функции. Они будут заменены методами доступа для слотов нового определения класса.
- Любой метод обобщенной функции, который использует старый мета-объект `<my-class>` как один из его формальных параметров должен быть обновлен, чтобы ссылаться на новый метаобъект `<my-class>`. (Всякий раз, когда определяется новый метод обобщенной функции, `define-method` добавляет метод к списку, хранящемуся в метаобъекте класса для каждого класса, используемого в качестве указанного формального параметра., поэтому легко найти все методы, которые необходимо обновить, когда переопределяется класс.)

Если эта стратегия переопределения класса поражает вас как довольно противоречащая интуиции, имейте в виду что она получена из аналогичного поведения в других объектных системах, таких как CLOS, и что опыт в этих системах показал, что она очень полезна на практике.

Так же имейте в виду, что как и большинство действий по умолчанию GOOPS, это поведение можно настроить . . .

8.12.2 Настройка переопределения класса(Customizing Class Redefinition)

Когда `define-class` замечает, что класс переопределяется, он создает новый метаобъект класса, как обычно, затем вызывает обобщенную функцию `class-redefinition` со старым и новым классом в качестве аргументов. Поэтому, если у старых или новых классов есть метаклассы отличные от класса по умолчанию `<class>`, поведение переопределения класса можно настроить определив метод `class-redefinition`, который специализирован для соответствующих метаклассов.

`class-redefinition` [generic]
: Handle the class redefinition from *old-class* to *new-class*, and return the new class metaobject that should be bound to the variable specified by `define-class`'s first argument.

`class-redefinition` (*old-class* `<`*class*`>`) (*new-class* `<`*class*`>`) [method]
: Implements GOOPS' default class redefinition behaviour, as described in Раздел 8.12.1 [Default Class Redefinition Behaviour], страница 825. Returns the metaobject for the new class definition.

Метод по умолчанию `class-redefinition`, для классов с метаклассом по умолчанию `<class>`, вызывает следующие обобщенные функции, которые, конечно могут быть индивидуально настроены.

`remove-class-accessors!` *old* [generic]
: The default `remove-class-accessors!` method removes the accessor methods of the old class from all classes which they specialize.

`update-direct-method!` *method old new* [generic]
The default `update-direct-method!` method substitutes the new class for the old in all methods specialized to the old class.

`update-direct-subclass!` *subclass old new* [generic]
The default `update-direct-subclass!` method invokes `class-redefinition` recursively to handle the redefinition of subclasses.

Альтернативная стратегия переопределения класса может заключатся в том, чтобы объявить все существующие экземпляры экземплярами старого класса, но признавая что старый класс теперь "безымянный(nameless)", поскольку его имя было получено новым определением. В этойо стратегии любые существующие подклассы могли бы также остаться такими, как они есть, при том понимании, чо они наследутся от безымянного класса.

Эта стратегия легко реализуется в GOOPS, определением нового метакласса, который будет использоваться для всех классов, к которым должна применяться эта стратегия, а затем определяется метод переопределения класса `class-redefinition` который специализирован для этого метакласса:

```
(define-class <can-be-nameless> (<class>))

(define-method (class-redefinition (old <can-be-nameless>)
                                   (new <class>))
  new)
```

Когда настройка может быть такой же простой, разве вы не рады, что GOOPS реализует гораздо более сложную стратегию по умолчанию!?

8.13 Изменение Класса и Экземпляра

Когда класс переопределяется, любой существующий экземпляр переопределенного класса будет изменен под определение новое определение класса до того момента, когда на любой из слотов экземпляра будут ссылаться или пытаться его установить. GOOPS изменяет каждый экземпляр, вызывая обобщенную функцию `change-class`.

В более общем плане вы можете изменить класс существующего экземпляра в любое время, вызвав обобщенную функцию `change-class` с двумя аргументами: экземпляр(instance) и новый класс(new class).

По умолчанию метод `change-class` рашает, как реализовать изменение класса глядя на определения слотов для существующего экземпляра класса и нового класса. Если новый класс имеет слоты с тем же именем, что и слоты в существующем классе, значения для этих слотов сохранаются. Слоты которые присутствуют только в существующем классе, отбрасываются. Слоты которые присутствуют только в новом классе, инициализируются с использованием соответствующего определения функции инициализации слота. (см. Раздел 8.8.1 [slot-init-function], страница 805).

`change-class` *instance new-class* [generic]

`change-class` (*obj* <*object*>) (*new* <*class*>) [method]
Изменить экземпляр *obj* что бы сделать его экземпляром класса *new*.

Значение каждого из слотов *obj* сохраняется только в том случае, если аналогичный слот существует в новом *new* классе; любые другие значения слотов отбрасываются.

Стоты нового класса*new*, которым нет соответствия ни однго из ранее существовавших слотов в *obj*, инициализируются в соответствии с функциями инициализации определенных для них в определении слотов класса *new*.

По умолчанию метод `change-class` также вызывает другую обобщенную функцию, `update-instance-for-different-class`, в конце своей работы, пред возвратом. Примененный метод `update-instance-for-different-class` может в дальнейшем корректировать новые экземпляры *new-instance* что необходимо для завершения изменения класса или изменять класс. Возвращаемое значение применяемого метода игнорируется.

`update-instance-for-different-class` *old-instance new-instance* [generic]
обобщенная функция. которая может быть настроена для того, чтобы поместить завершающие штрихи в экземпляры чей класс был изменен. По умолчанию метод `update-instance-for-different-class` ничего не делает.

Индивидуальное изменение поведения класса может быть реализовано путем определения метода `change-class`, который специализируется либо по экземпляру класса, подлежащему модификации, либо по метаклассу нового класса.

9 Guile Implementation

В какой-то момент, после того, как вы начали программировать на Scheme, на другом уровне Scheme знание реализации: того как Scheme реализована, оказывается необходимым что бы стать экспертом-хакером. Как отмечает Peter Norvig в своей ретроспективе PAIP[1], "Эксперт-программист Lisp в конечном итоге развивает хорошую 'модель эффективности'."

Благодаря этому Норвиг отмечает, что со временем, хакер Lisp в конечном итоге развивает понимание чего стоит его код с точки зрения пространства и времени.

Эта глава описывает Guile как реализацию Scheme: ее историю, ее представление и оценивает ее данные и ее компилятор. Эти знания помогут вам сделать шаг от того чтобы просто быть знакомым со Scheme к уровню настоящего хакера.

9.1 Краткая история Guile

Guile это артефакт исторических процессов, как кода, так и сообщества хакеров. Эту историю иногда полезно знать при изучении исходного кода, знать о прошлых решениях и о будущих направлениях.

Конечно, настоящая история Guile пишется хакерами, а не писателями поэтому мы завершаем раздел с описанием текущего статуса и будущих направлений.

9.1.1 Тезисы Emacs

История Guile - это рассказа о том, как перенести опыт Emacs в развитие программ в системе GNU.

Emacs, когда он впервыпе был создан в форме GNU в 1984, стал новым решением проблемы "как сделать программу". Тезис Emacs заключается в том, что очень приятно создавать сложные программы на основе ортогонального ядра, написанного на низкоуровневом языке, вместе с мощным, высокоуровневым языком расширения.

Язык Расширения способствует расширению программ, программ которые легко адаптируются к различным пользователям и временным изменениям. Доказательство этого можно увидеть в текущем Emacs и продолжении его существования, охватывающем более четверти века.

Помимо предоставления модификации программы другими, язык расширения хороши также для усиления. Программы, созданные в "Emacs стиле" приятны и легки для их авторов, что бы понять, какие функции им нужны.

После того как опыт Emacs был оценен более широко, ряд хакеров начали рассмотрение вопроса, как распространить этот опыт на остальную часть системы GNU. Было ясно, что самый простой способ Emacsify(емаксировать) программы - это внедрить в них реализацию общего языка.

---

[1] PAIP это абревиатура для *Paradigms of Artificial Intelligence Programming*, старый, но полезный текст на Lisp. Ретроспектива Norvig суммирует уроки PAIP, и может быть найдена на `http://norvig.com/Lisp-retro.html`.

9.1.2 Первые Дни

Tom Lord был первым, кто полностью сконцентрировал свои усилия на внедрение языка времени выполнения, который он назвал “GEL”, языком расширения GNU.

GEL был продуктом преобразования SCM, реализации Scheme Aubrey Jaffer, в нечто более подходящее для внедрения в качестве библиотеки. (Сам SCM базировался на реализации выполненной George Carrette, SIOD.)

Lord удалось убедить Richard Stallman перевести GEL в статус официального языка расширений для проекта GNU. Это было естественным подходом, учитывая что Scheme был более чистым, более современным Lisp чем Emacs Lisp. Часть аргументов заключалась в том, что в конечном итоге, когда GEL станет более мощным, он могбы научиться выполнять программы на других языках, особенно Emacs Lisp.

Из-за конфликта имен с другим языком програмирования, Jim Blandy предложил новое название для GEL: “Guile”. Помимо рекурсивного акронима, “Guile” хитрое следование имен его предков, “Planner”, “Conniver”, и “Schemer”. (Последний был усечен до “Scheme” из-за ограничений длины имени файлов в 6-символов на старой операционной системе.) Наконец, “Guile” созвучен “guy-ell”, или “Guy L. Steele”, который вместе с Gerald Sussman, первоначально создали схему.

Примерно в тоже время, когда Guile (GEL) готовился к публичному выпуску, другой язык расширения набирал популярность,Tcl. Многие разработчики нашли преимущества в Tcl из-за его shell-подобного синтаксиса и его хорошо развитой библиотеки графических виджетов, Tk. Кроме того, в то время был большой маркетинговый толчок, представлявший Tcl как “универсальный язык расширений”.

Richard Stallman, как главный автор GNU Emacs, имел особое виденье какие языки расширения должны быть, и Tcl не казался ему столь же способным как Emacs Lisp. Он отправил критику в группу новостей comp.lang.tcl, вызвав одну из легендарных интернет дискуссий. В части эти рассуждения окрестили как “Tcl Войны”, он объявил о намерении фонда СПО продвигать Guile как язык расширений для проекта GNU.

Это распространенное заблуждение, что Guile был создан как реакция на Tcl. Хотя это правда, что публичное объявление Guile произошло одновременно с “Tcl войнами”, Guile был создан из условий которые существовали вне полемики. Действительно, требование необходимости наличия мощного языка, что бы преодолеть разрыв между расширением существующих приложений и динамической средой программирования, по прежнему остается с нами и сегодня.

9.1.3 A Scheme of Many Maintainers

Обследовав поле, кажеться, что число реализаций Scheme соответствует числу их сопровождаеющих maintainers в отношении N-to-1. То есть, те люди которые реализуют Schemes, могут делать это несколько раз, но время жизни данной Scheme связано с сопровождением одного человека.

В этом отношении Guile не типично.

Tom Lord поддерживал Guile в течении первых полутора лет или около того, что соответствует концу 1994 года до середины 1996. Выпуски, сделанные за это время, составляют дугу от SCM к Guile как автономной программе, переиспользуемой, внедряемой библиотеки, но проходящей через взрыв характеристик: встроенные Tcl и Tk,

инструментальная цепочка для дизассемблирования Java, добавление Си подобного синтаксиса, создания модульной системы и запуска богатого интерфейса POSIX.

Только некоторые из этих функций остались в Guile. Постоянное напряжение между предоставлением небольшого вложенного языка, и того, который имеет все функции(например графический инструментарий), что может понадобиться современному Emacs. В конце концов, по мере того как Guile формировалась, команда разработчиков решила сосредоточится на глубине, документации и ортогональности, чем на широте. С тех пор это было в центре внимания Guile, хотя существует широкий диапазон библиотек для Guile.

Джим Бленди председательствовал на этот период стабилизации, за три года, до конца 1999 года, когда он тоже перешел к другим проектам. С тех пор у Guile была группа сопровождающих. Первой группой были Мацей Стаховак, Микаэль Джульфельд и Мариус Фоллмер, с Воллмером, остающимся надолго. К концу 2007 года Фолмер в основном перешел к другим вещам, поэтому Нейл Джеррам и Людовик Курт взяли на себя основную ответственность за сопровождение. В конце 2009 года к Джерри и Курту присоединился Энди Винго.

Конечно, значительная часть фактической работы с Guile исходит от других участников которых слишком много, чтобы упомянуть, но без которых мир был бы беднее.

9.1.4 Временная шкала выпусков Guile

guile-i — 4 February 1995
: SCM, превратился в библиотеку.

guile-ii — 6 April 1995
: Была добавлена низкоуровневая модульная система. Добавлена поддержка Tcl/Tk, позволяющая расширение Scheme через Tcl и наоборот. Поддержка POSIX была улучешна, и была предпринята эксперементальная попытка интеграции с Java.

guile-iii — 18 August 1995
: Си-подобный синткасис был улучшен, но в основном этот выпуск обозначил начало разбиения Guile на части.

1.0 — 5 January 1997
: `#f` теперь отличается от `'()`. Добавлена многопоточность на уровне пользователя. Отладка исходного кода стала более удобной, было начато написание руководства для программистов и пользователей. Модульная система получила интерфейс высокого уровня, который до сих пор в более менее неизменной форме.

1.1 — 16 May 1997
1.2 — 24 June 1997
: Поддержка Tcl/Tk и ctax была разделена по отдельным пакетам и с тех пор остается там. Guile стал более совместимым с SCSH, и стал более полезным как язык сценариев UNIX. Теперь Libguile может быть построена как общая библиотека и сторонние расширения, написанные на Си, могут загружаться через динамическую линковку.

1.3.0 — 19 October 1998
: Редактирование командной строки стало намного приятнее благодаря использованию библиотеки readline. Первоначальная поддержка интернационализацииThe с помощью многобайтовых строк была удалена; Прошло 10 лет, прежде чем надлежащая интернационализация вернулась. Начата поддержка Emacs Lisp, порты получили лучшую поддержку для файловых дескрипторов, и добавлены fluids.

1.3.2 — 20 August 1999
1.3.4 — 25 September 1999
1.4 — 21 June 2000
: Добавлен длинный список лисповых реализаций: hooks, Common Lisp's `format`, необязательные и ключевые слова в аргументах процедур, `getopt-long`, сортировка, случайные числа, и много других изменений и улучшений. Guile также получил интерактивный дебагер, интерактивную справку и трассировщик.

1.6 — 6 September 2002
: Guile получила поддержку стандарта R5RS, и добавила ряд модулей SRFI. Модульная ситема была расширена программной продержкой для выбора и переименования индентификаторов. Объектная система GOOPS была объединена с ядром Guile.

1.8 — 20 February 2006
: Guile переключилась на использование библиотеки GMP в расчетах арифметики произвольной точности и добавлена поддержка точных рациональных чисел. Встраивание пользовательского пространства Guile было удалено в пользу упреждающих потоков POSIX, обеспечивающих истинную много процессорность. Добавлена поддержка Gettext, и Guile Си интерфейс API был очищен и ортогонализирован.

2.0 — 16 February 2010
: Виртуальная машина была добавлена в Guile, вместе с соотвествующим компилятором и набором утилит. Впоследствии поддержка интернационализации была вновь реализована в терминах unicode, locales, и libunistring. Запуск экземпляров Guile стал управляемым и отлаживаемый из Emacs через Geiser. Guile добавил функции обнаруженные в других Schemes: SRFI-18 потоки, module-hygienic macros, профайлер, трассировщик, отладчик, интеграция SSAX XML, байтовые вектора, и динамический FFI, delimited continuations, версии модулей, и частичная поддержка R6RS.

2.2 — mid-2014
: Виртуальная машина, введенная в версии 2.0 была полностью переписана вместе с большей частью копилятора и набора утилит. Это ускорило многие программы Guile, а также сократило время запуска и использование памяти. Инструментарий анализатора PEG был добавлен, что упростило работу с другими языковыми интерфейсами.

9.1.5 Состояние, или: Необходима Ваша Помощь.

Guile добилась многого, но многое еще предстоит сделать.

Есть еще старая проблема приведения существующих приложений в более подобные Emacs опыту. В этом отношении у Guile были некоторые успехи, но все же, большинство приложений в системе GNU находятся без интеграции с Guile.

Внедрение Guile в эти приложения требует инвестиций “хакерской энергии” для подключение Guile к программе, которая рассчитывается только тогда, когда она уже достаточно хороша, чтобы внедрить в нее новые виды поведения. Это было бы отличным способом для новых хакеров: приложение которое вы испльзуете, и которое вы хорошо знаете, придумать что оно еще не может сделать и выяснить способ интеграции с Guile и реализовать эту задачу в Guile.

Со временем, возможно, эта экспозиция может перевернуть сама себя, в результате чего программы смогут работать под Guile, что в конечном итоге приведет к Емаксификации(Emacsification) всей системы GNU. В самом деле, это причина для обозначения многих модулей Guile, которые живут в пространстве имен `ice-9`, как реверанс к вымышленной субстанции Льда из новел Kurt Vonnegut, "Колыбель кошки", способной действовать в качестве затравочного кристала для кристализации массы программного обеспечения.

Неявной для всего этого обсуждения является идея о том, что динамические языки как-то лучше чем такие языки как Си. Хотя такие языки как Си, имеют свое место. Guile отвечает на этот вопрос - да, Scheme более выразительна чем Си, и на ней более интересно писать. Эта реализация несет в себе обязательство писать как можно больше кода на Scheme, не на других языках.

В наши дни можно писать расширяемые приложения почти полностью из языков высокого уровня, через байт-код и собственную компиляцию, обеспчивающих прирост скорости выполнения в базовых аппратных и внешних интерфейсах вызовов высокоуровневых языков. Сиседмы Smalltalk похожи на это, как и обычные системы на основе Common Lisp. Хотя еще существует ряд чистокровных приложений там, где пользователям по-прежнему необходимо опускаться до Си для выполнения некоторых задач: сопряжения системных библиотек, у которых нет готовых интерфейсов к Guile, и для некоторых задач, требующих высокого быстродействия. Нативная компиляция времени выполнения запланирована для Guile 3.0, должна помочь в этом.

Тем, не менее при использовании приложения все-в-Guile, иногда вы хотите предоставить возможность для пользователей, чтобы расширить вашу программу языком, синтаксисом который ближе к Си или Python. Еще одна интересная идея, которую стоит рассмотреть, заключается в компиляци, например Python в Guile. Это не надуманная идея: см. например IronPython или JRuby.

И тогда есть Emacs. Поддержка Guile Emacs Lisp достигла отличного уровня правильности, надежности и скорости. Однако попрежнему предстоит сделать работу, чтобы закончить интеграцию в Emacs. Это даст много интересного Emacs: родные(для архитектуры процессора) потоки, реальную объектную систему, более сложные типы данных, более чистый синтаксис и доступ ко всем Guile расширениям.

Наконец, существует еще одна ось кристализации, ось между различными реализациями Scheme. Guile еще не поддерживает последний стандарт Scheme, R7RS, и должен еще это реализовать. Как и все стандарты, R7RS является несовершенным, но его поддержка позволит большему коду работать на Guile без изменений и позво-

лит хакерам Guile создавать код, совместимый с другими версиями scheme. Помощь в этом отношении будет высоко оценена.

9.2 Представление данных

Scheme - это язык со скрытой типизацией; это означает, что система не может определить тип данных во время компиляции. Типы данных становяться известными во время выполнения программы. Переменные не имеют фиксированных типов; Переменная может содержать пару(pair) в одном месте программы, целое число в следующем и тысячаэлемнтный вектор позже. Вместо переменных, значения имеют фиксированный тип.

Для реализации стандартных функций Scheme, таких как `pair?` и `string?` и обеспечения сборки мусора, представление каждого значения должно содержать достаточно информации для точного определения его типа во время выполнения. Часто системы Scheme также используют эту информацию, чтобы определить, пыталась ли программа применить операцию к неуместному типу значения(например применить `car` к строке(string)).

Поскольку переменные, пары и векторы могут содержать значения любого типа, реализация Scheme использует единое представление для значений — один тип, достаточно большой, чтобы содержать либо полное значение либо указатель на полное значение, а также необходимую информацию о типе.

В следующих разделах будет представлена простая система типов, а затем сделаны некоторые уточнения для устранения основных ее недостатков. Затем мы завершаем обсуждение, конкретным выбором, который выбран для Guile в отношении сборки мусора и представления данных.

9.2.1 Простейшее представление

Простейший способ представления значений Scheme в Си представить каждое значение как указатель на структуру, содержащую индикатор типа, за которым следует объединение, несущее реальное значение. Предполагая, что `SCM` это имя нашего универсального типа, мы можем написать:

```
enum type { integer, pair, string, vector, ... };

typedef struct value *SCM;

struct value {
  enum type type;
  union {
    int integer;
    struct { SCM car, cdr; } pair;
    struct { int length; char *elts; } string;
    struct { int length; SCM  *elts; } vector;
    ...
  } value;
};
```

с точками заменяющими код для остальных типо Scheme.

Это представление достаточно для реализации всей семантики Scheme. Если *x* является значением `SCM`:

- Для проверки что *x* является целым(integer), мы можем написать *x*`->type == integer`.
- Чтобы найти его значение, мы можем написать *x*`->value.integer`.
- Для проверки что *x* это вектор(vector), мы можем наисать *x*`->type == vector`.
- Если мы знаем что *x* это вектор, мы можем написать *x*`->value.vector.elts[0]` чтобы сослаться на первый элемент.
- Если мы знаем что *x* это пара, мы можем написать *x*`->value.pair.car` чтобы извлечь начало пары(car).

9.2.2 Быстрые Целые числа

К сожалению, вышеупомянутое представление имеет серьезный недостаток. Чтобы вернуть целое(integer), выражение должно выделить память под структуру значения `struct value`, инициализировать его, чтобы представить это целое число и вернуть указатель на него. Кроме того, для получения значения целого числа требуется ссылка на память, которая намного медленне, чем ссылка на регистры для большинства процессоров. Поскольку целые числа чрезвычайно распространены, это представление является слишком дорогостоящим, как по времени, так и по памяти. Целые должны быть очень дешевыми, для создания и манипулирования ими.

Одним из возможных решений кроется в наблюдении, что на многих архитектурах, выделеное в куче(heap) место для данных (т.е то, что вы получаете при вызове `malloc`) должно быть выровнено по воьмибайтовой границе. (Независимо от того, действительно ли машина требует этого, мы можем написать собственный распределитель (allocator) для объектов структур значений`struct value` который будет гарантировать что это так) В этом случае нижние три бита адреса структуры, как известно, равны нулю.

Это дает нам место, которое поможет обеспечить улучшенное представление целых чисел. Мы создадим последовательность правил:

- Если нижние три бита значения `SCM` равны нулю, SCM значение является указателем на структуру значения `struct value`, и все работает как и раньше.
- В противном случае, значение `SCM` представляет собой целое число, значение которого отображается в его верхних битах.

Вот код Си, реализующий это соглашение:

```
enum type { pair, string, vector, ... };

typedef struct value *SCM;

struct value {
  enum type type;
  union {
    struct { SCM car, cdr; } pair;
    struct { int length; char *elts; } string;
    struct { int length; SCM  *elts; } vector;
```

```
    ...
  } value;
};

#define POINTER_P(x) (((int) (x) & 7) == 0)
#define INTEGER_P(x) (! POINTER_P (x))

#define GET_INTEGER(x)  ((int) (x) >> 3)
#define MAKE_INTEGER(x) ((SCM) (((x) << 3) | 1))
```

Обратите внимание, что целое(`integer`) больше не отображается как элемент типа перечисления`enum type`, и объединение теряет целочисленный член. Вместо этого мы используем макросы `POINTER_P` и `INTEGER_P` для создания грубого разделения значений на целые и не целые числа, а также тестирования типов прежде использования.

Здесь мы ответим на поставленные выше вопросы(опять же предположим что *x* это значение `SCM`):

- Чтобы проверить, является ли *x* целым числом, мы можем написать `INTEGER_P (x)`.
- Чтобы найти его значение, мы можем написать `GET_INTEGER (x)`.
- Чтобы проверить, является ли *x* вектором, мы можем написать:

  ```
  POINTER_P (x) && x->type == vector
  ```

  Учитывая новое представление, мы должны убедиться, что *x* действительно является указателем перед разыменовыванием, чтобы определить его полный тип.
- Если мы знаем что *x* это вектор, мы можем записать `x->value.vector.elts[0]`, чтобы ссылаться на его первый элемент, как и раньше
- Если мы знаем что *x* это пара, мы можем записать `x->value.pair.car` чтобы извлечь первый элемент пары(car), также как и раньше.

Это представление позволяет более эффективно работать с целыми числами, чем первое. Например если известно что *x* и *y* целые числа, мы можем вычислить их сумму следующим образом:

```
MAKE_INTEGER (GET_INTEGER (x) + GET_INTEGER (y))
```

Теперь, целочисленная математика не требует выделения памяти или ссылок. Реальные системы Scheme реализуют сложение и другие операции, используя еще более эффективный алгоритм, но это эссе не о bit-twiddling. (Подсказка: как вы решаете, когда переполниться большое число? Как вы это сделаете на ассемблере?)

9.2.3 Дешевые Пары

Однако, есть еще одна проблема, с которой можно столкнуться. Значительная часть кучи(heap) Scheme содержат пары, их большее чем других типов объектов; Как то Jonathan Rees обратил внимание на то, что пары занимаеют 45% кучи в его реализации Scheme (Scheme 48). Однако наше представление, выше, тратит три слова `SCM` на пару - одно для типа и два для car и cdr. Есть ли способ представить пары, используя только два слова?

Давайте уточним соглашение, которое мы установили ранее для представления целых integer. Допустим, что: that:

- Если нижние три бита значения `SCM` равны `#b000`, то это указатель, как и раньше.
- Если нижние три бита равны `#b001`, то верхние биты являются целым числом. Это немного более строгое ограничение чем раньше.
- Если нижние три бита равны `#b010`, тогда значение с нижними тремя битами отбрасывается, а верхние являются адресом пары.

Здесь представлен новый Си код:

```
enum type { string, vector, ... };

typedef struct value *SCM;

struct value {
  enum type type;
  union {
    struct { int length; char *elts; } string;
    struct { int length; SCM  *elts; } vector;
    ...
  } value;
};

struct pair {
  SCM car, cdr;
};

#define POINTER_P(x) (((int) (x) & 7) == 0)

#define INTEGER_P(x)  (((int) (x) & 7) == 1)
#define GET_INTEGER(x)  ((int) (x) >> 3)
#define MAKE_INTEGER(x) ((SCM) (((x) << 3) | 1))

#define PAIR_P(x) (((int) (x) & 7) == 2)
#define GET_PAIR(x) ((struct pair *) ((int) (x) & ~7))
```

Обратите внимание, что перечисление(`enum type`) и структура значения (`struct value`) теперь содержат только положения для векторов и строк; как целые числа, так и пары стали особыми случаями. Выше приведеный код также предполагает что `int` достаточно велик, чтобы содержать указатель, что обычно не так.

Ниже приведен список наших примеров:

- Чтобы проверить, является ли *x* целым числом, мы можем написать `INTEGER_P (x)`; как и раньше.
- Чтобы получить значение, мы можем написать `GET_INTEGER (x)`, как и раньше.
- Чтобы проверить является ли *x* вектором, мы можем записать:

  ```
  POINTER_P (x) && x->type == vector
  ```

  Мы все равно должны убедиться, что *x* является указателем перед разыменовыванием `struct value` в соответстви с типом.
- Если мы знаем, что тип *x* это вектор, мы можем записать `x->value.vector.elts[0]`, чтобы ссылаться на на его первый элемент, как и раньше.

- Мы можем написать `PAIR_P (x)` чтобы определить является ли *x* парой, а затем записать `GET_PAIR (x)->car` чтобы получить ссылку на ее первый элемент car.

Это изменение представления уменьшает размер нашей кучи на 15%. Это также делает его более быстрым при решении является ли значение парой или нет, поскольку ссылки на память для этого не нужны; достаточно проверить нижние два бита значения `SCM`. Это может быть значительным при просмотре списков, являющемся основной деятельностью системы Scheme.

Опять же, большинство реальных систем Scheme используют немного другую реализацию; например, если GET_PAIR вычитает младшие биты `x`, а не маскирует их, оптимизатор может сочетать это вычитание с добавлением смещения члена структуры, на который мы ссылаемся, делая измененный указатель столь же быстрым в использовании, как и немодифицированный указатель.

9.2.4 Консервативная сборка мусора

Помимо скрытой типизации, основным источником ограничений на реализацию в Scheme представлений данных накладывает сбощик мусора. Сборщик должен иметь возможность, чтобы определить какой объект живой в куче, а какой не живой и следовательно, должен быть собран.

Существует много способов реализовать его. Сборщик мусора Guile построен на библиотеке консервативного сборщика мусора Boehm-Demers-Weiser (BDW-GC). BDW-GC "просто работает", по большей части. Но поскольку интересно знать, как он работает, мы включаем здесь описание на высоком уровне, что делает BDW-GC.

Сбор мусор имеет две логические фазы% фазу *пометки(mark)*, в которой устанавливается набор живых объектов, и фазу *чистки(sweep)*, в которой объекты не прошедшие пометку убираются. Корректное функционирование сборщика зависит от возможности пройти весь набор живых объектов.

В фазе пометки коллекционер сканирует глобальные переменные системы и локальные переменные в стеке, чтобы определить какие объекты сразу(непосредственно) доступны Си-коду. Тогда он сканирует эти объпкты, чтобы айти объекты, на которые они указывают и так далее. Сборщик логически устанавливает *бит метки(mark bit)* для каждого найденного объекта, поэтому каждый объект проходиться только один раз.

Когда сборщик мусора не может найти никаких немаркированных объектов, на которые указывают отмеченные объекты, предполагается, что любые объекты, которые все еще не отмечены, никогда не будут использоваться программой (т.к. нет разметки из любой глобальной или локальной переменной, которая достигает их) и освобождает их.

В приведенных выше пунктах мы не указали, как сборщик мусора находит глобальные и локальные перменные; как обычно существует много разных подходов. Часто программист должен содержать список указателей на все глобальные переменные, относящиеся к куче, и другой список (корректирующийся при входе и выходе из каждой функции) локальных переменных, для работы сборщика.

Список глобальных переменных обычно не так сложно поддерживать, поскольку глобальные переменные относительно редки. Тем не менее, явно поддерживаемый список локальных переменных(по личному опыту) - это кошмар для поддержания.

Таким образом, BDW-GC использует технику так называемой *консервативной сборки мусора*, чтобы сделать список локальных переменных не нужным.

Трюк консервативной сборки мусора заключается в том, чтобы рассматривать стек как обычный диапазон памяти, и предполагать, что каждое слово в стеке является указателем на кучу. Таким образом, сборщик отмечает все объекты, адреса которых отображаются в любом месте стека, не зная наверняка как это слово правильно толковать.

Помимо стека, BDW-GC также сканирует разделы статических данных. Это означает что глобальные переменные также сканируются при поиске живых объектов Scheme.

Очевидно, что такая система будет иногда сохранять объекты, которые на самом деле являются мусором, и должны быть освобождены. На практике это не является проблемой. Альтернатива, с явно поддерживаемым списком адресов локальных переменных, значительно менее надежна, из-за ошибок программиста. Заинтересованные читатели должны посмотреть веб страницу BDW-GC по адресу `http://www.hboehm.info/gc/`, чтобы получить больше информации.

9.2.5 Тип SCM в Guile

Guile разделяет объекты Scheme на два вида: те которые полностью соответствуют `SCM`, и те которые требуют храненения в куче.

Первый класс называется *непосредственным(immediates)*. Класс непосредственных включает в себя малые целые числа, символы, знаковые символы, логические значения, пустой список, таинственный объект конца файла и некоторые другие.

Оставшиеся типы называются, что неудивительно, *опосредованными (non-immediates)*. Они включают в себя пары, процедуры, строки, вектор и все другие типы данных в Guile. Для опосредованных, слово `SCM` содержит указатель на данные в куче, с дополнительной информацией об объекте в вопросе что храниться в этих данных.

В этом разделе описывается, как тип `SCM` фактически представлен и исползуется на уровне Си. Заинтересованные читатели должны посмотреть файл `libguile/tags.h` для представления того как Guile хранит инофрмацию о типах.

Фактически, для представления объектов в Guile существуют два базовых типа данных Си: `SCM` и `scm_t_bits`.

9.2.5.1 Связь между `SCM` и `scm_t_bits`

Переменная типа `SCM` гарантированно содержит действительный объект Scheme. С другой стороны, переменная типа `scm_t_bits`, может содержать представление значения `SCM` как интегрального значения Си типа, но также может содержать любое значение Си, даже если оно не соответствует действительному объекту Scheme.

Для переменной x типа `SCM`, информация о типе объекта Scheme храниться в форме, которая не может быть использована непосредственно. Чтобы иметь возможность рабоать с типом кодирующим значение scheme, переменная `SCM` должна быть преобразована в соответствующее представление, такое как `scm_t_bits` переменной y, используя макрос `SCM_UNPACK`. Как только это будет сделано, тип объекта scheme x

может быть получен из содержимого битов значения `scm_t_bits` *y*, способом проиллюстрированным ранее в этой главе (см. Раздел 9.2.3 [Cheaper Pairs], страница 836). И наоборот, действительное битовое кодирование значения переменной `scm_t_bits` может быть преобразовано в соответствующее значение `SCM` используя макрос `SCM_PACK`.

9.2.5.2 Непосредственные Объекты

Объект Scheme может быть либо непосредственным, т.е.: содержать всю необходимую информацию в себе, или он может содержать ссылку на *ячейку(cell)* с дополнительной инфомрацией в куче. Хотя в общем случае для кода пользователя не должно быть никакого значения, является ли объект непосредственным или нет, в собственном коде Guile различие иногда имеет значение. Работа обеспечивается следующими предоставляемыми низкоуровневыми макросами:

`int SCM_IMP` (*SCM x*) [Macro]

Объект Scheme является непосредственным, если для него выполняется предикат `SCM_IMP`, в противном случае он содержит закодированную ссылку на ячейку в куче. Результат предиката предоставляется Си как булево значение. Коду пользователя и коду, который расширяет Guile, обычно не требуется использовать этот макрос.

Резюмируем:

- Если задан Scheme объект *x* неизвестного типа, сначала проверьте с помощью `SCM_IMP (x)` не непосредственный ли это объект.
- Если это так, всю информацию о типе и значении можно определить из значения `scm_t_bits`, которое предоставляется макросом `SCM_UNPACK(x)`.

В Scheme имеется ряд специальных значений, большинство из которых задокументировано в других местах этого руководства. Это не совсем подходящее место для их размещения, но пока вот список Си имен, присвоенных некоторым из этих значений:

`SCM SCM_EOL` [Macro]

Объект пустого списка Scheme или объект “End Of List”, обычно записываемый в Scheme как `'()`.

`SCM SCM_EOF_VAL` [Macro]

Значение - конец файла(end-of-file) в Scheme. Оне не имеет стандартного представления, по очевидным причинам.

`SCM SCM_UNSPECIFIED` [Macro]

Значение, возвращаемое некоторыми(но не всеми) выражениями, указанными в стандарте Scheme возвращающими “неопределенное(unspecified)” значение.

Это своего рода своего рода буквальный способ использования стандарта, цикл read-eval-print не печатает ничего, когда выражение возвращает это значение. Поэтому неплохо вернуть это значение, когда вы не можете придумать ничего полезного.

`SCM SCM_UNDEFINED` [Macro]

"Неопределенное" значение. Его наиболее важным свойством является то, что он не равен действительным значениям Scheme. Это связано с различными внутренними потребностями используемого Си кода, взаимодействующего с Guile.

Например, когда вы пишите функцию Си, вызываемую из Scheme и которая принимает необязательные аргументы, интерпретатор передает `SCM_UNDEFINED` для любых аргументов которые вы не получили.

Мы также используем это для обозначения несвязанных переменных.

`int SCM_UNBNDP` (*SCM x*) [Macro]

Возвращает true, если *x* равна `SCM_UNDEFINED`. Обратите внимание, что это не проверка является ли *x* - `SCM_UNBOUND`(является ли x связанным). История не простит такое название нам.

9.2.5.3 Опосредованные(Non-immediate) Объекты

Объект Scheme типа `SCM`, для которого не выполнен предикат `SCM_IMP`, содержит закодированную ссылку на ячеку из кучи. Эта ссылка может быть декодирована в Си указатель на ячеку кучи, использованием макроса `SCM2PTR`. Кодирование указателя на ячейку кучи в значение `SCM` выполняется с использоанием макроса `PTR2SCM`.

`scm_t_cell * SCM2PTR` (*SCM x*) [Macro]

Извлечь и вернуть указатель ячеки кучи из опосредствованого объекта `SCM` *x*.

`SCM PTR2SCM` (*scm_t_cell * x*) [Macro]

Возвращает значение `SCM`, которое кодирует ссылку на указатель *x* кучи.

Обратите внимание, что также возможно преобразовать опосредствованное значение `SCM` используя `SCM_UNPACK` в переменную `scm_t_bits`. Однако, результат `SCM_UNPACK` не может быть использован как указатель на `scm_t_cell`: только `SCM2PTR` гарантированно преобразует объект `SCM` в действительный указатель на ячейку кучи. Кроме того, не разрешается применять `PTR2SCM` ко всему, что не является действительным указателем на ячейки кучи.

Резюме:

- Используйте `SCM2PTR` только для значений `SCM`, для которых `SCM_IMP` является ложным!
- Не используйте `(scm_t_cell *) SCM_UNPACK (x)`! Вместо этог используйте SCM2PTR (*x*)!
- Не ипользуйте `PTR2SCM` для чего либо, кроме как указателя на ячейку из кучи!

9.2.5.4 Выделение Ячеек

Guile предоставляет как обычные ячейки с двумя слотами, так и двойные ячейки с четырьмя слотами. Следующие функции являются наибольее примитивным способом выделения ячеек.

Если вызывающий намерен использовать его в качестве заголовка для какого-либо другого типа, он должен передать соответствующее магическое значение в *word_0*, чтобы пометить его как член этого типа и передать все значение как *word_1*, и т.д.,

которое необходимо данному типу. Обычно вам не нужны эти функции, если вы не внедряете новый тип данных лезете глубоко в код `<libguile/tags.h>`.

Если вы просто хотите выделить пару, используйте `scm_cons`.

`SCM scm_cell` (*scm_t_bits word_0, scm_t_bits word_1*) [Function]
Выделяет новую ячейку, инициализирует два слота значениями *word_0* и *word_1*, и возвращает ее.

Обратите внимание, что слова *word_0* и *word_1* имеют тип `scm_t_bits`. Если вы хотите передать объект `SCM` вам необдимо использовать `SCM_UNPACK`.

`SCM scm_double_cell` (*scm_t_bits word_0, scm_t_bits word_1, scm_t_bits word_2, scm_t_bits word_3*) [Function]
Подобно `scm_cell`, но выделяет двойную ячейку с четыремя слотами.

9.2.5.5 Ячейка Кучи Информация о типе

Ячейки кучи содержат несколько записей, каждая из которых является либо объектом scheme типа `SCM` или наобработанным значением Си типа `scm_t_bits`. Какая из записей ячейки содержит объекты Scheme и какая содержит необработанные значения Си, определяется первой записью ячейки, которая содержит ячейку типа информации.

`scm_t_bits SCM_CELL_TYPE` (*SCM x*) [Macro]
Для опосредованного объекта Scheme *x*, достает содержимое первой записи ячейки кучи, на которую ссылается *x*. Это значение содержит информацию о типе ячейки.

`void SCM_SET_CELL_TYPE` (*SCM x, scm_t_bits t*) [Macro]
Для опосредованного объекта Scheme *x*, записывает значение *t* в первую запись ячеейки кучи на которую ссылается *x*. Значение *t* должно содержать действительный тип ячекйки.

9.2.5.6 Доступ к содержимому Ячеек

Для опосредованного объекта Scheme *x*, тип объекта можно определить прочитав ячейку типа используя макрос `SCM_CELL_TYPE`. Для каждого типа ячкейки известно какие записи содержат ячейки объектов Scheme и и какие записи содержат ячейки необработанных Си данных. Для доступа к различным записям ячеек соответственно, предоставляются следующие макросы.

`scm_t_bits SCM_CELL_WORD` (*SCM x, unsigned int n*) [Macro]
Достать запись ячейки *n*, на которую ссылается опосредованный объект Scheme *x* как необработанные данные. Это незаконно, для доступа к ячейкам, в которых объекты Scheme используют эти макросы. Для удобства предусмотрены следующие макросы:

- SCM_CELL_WORD_0 (*x*) ⇒ SCM_CELL_WORD (*x*, 0)
- SCM_CELL_WORD_1 (*x*) ⇒ SCM_CELL_WORD (*x*, 1)
- ...
- SCM_CELL_WORD_*n* (*x*) ⇒ SCM_CELL_WORD (*x*, *n*)

`SCM SCM_CELL_OBJECT` (*SCM* ***x***, *unsigned int* ***n***) [Macro]

Достать запись *n* ячейки из кучи, на которую ссылается опосредованный объект Scheme *x* как объект Scheme. Это незаконно, для доступа к записям ячеек которые не хранят объекты Scheme с помощью использования данных макросов. для удобства, предусмотрены следующие макросы.

- SCM_CELL_OBJECT_0 (*x*) ⇒ SCM_CELL_OBJECT (*x*, 0)
- SCM_CELL_OBJECT_1 (*x*) ⇒ SCM_CELL_OBJECT (*x*, 1)
- ...
- SCM_CELL_OBJECT_*n* (*x*) ⇒ SCM_CELL_OBJECT (*x*, *n*)

`void SCM_SET_CELL_WORD` (*SCM* ***x***, *unsigned int* ***n***, *scm_t_bits* ***w***) [Macro]

Записать необработанное значение Си *w* в запись с номером *n* ячейки кучи опосредованного значения Scheme *x*. Значения, которые записываются в ячейки таким образом, могут считываться из ячеек с использованием макросов `SCM_CELL_WORD` или, в случае, если это нулевая запись ячейки макросом `SCM_CELL_TYPE`. Для частного случая нулевой записи ячейки надо убедиться что *w* содержит информацию о типе ячейки, которая не описывает объект Scheme. Для удобства, предусмотрены следующие макросы.

- SCM_SET_CELL_WORD_0 (*x*, *w*) ⇒ SCM_SET_CELL_WORD (*x*, 0, *w*)
- SCM_SET_CELL_WORD_1 (*x*, *w*) ⇒ SCM_SET_CELL_WORD (*x*, 1, *w*)
- ...
- SCM_SET_CELL_WORD_*n* (*x*, *w*) ⇒ SCM_SET_CELL_WORD (*x*, *n*, *w*)

`void SCM_SET_CELL_OBJECT` (*SCM* ***x***, *unsigned int* ***n***, *SCM* ***o***) [Macro]

Записать объект Scheme *o* в запись с номером *n* ячейки кучи опосредованного значения Scheme *x*. Значения, которые записываются в ячейки таким образом, могут считываться из ячеек с использованием макросов `SCM_CELL_OBJECT` или, в случае, если это нулевая запись ячейки макросом `SCM_CELL_TYPE`. Для частного случая нулевой запись объекта Scheme в эту ячейку разрешена только в том случае, если ячейка формирует пару. Для удобства, предусмотрены следующие макросы.

- SCM_SET_CELL_OBJECT_0 (*x*, *o*) ⇒ SCM_SET_CELL_OBJECT (*x*, 0, *o*)
- SCM_SET_CELL_OBJECT_1 (*x*, *o*) ⇒ SCM_SET_CELL_OBJECT (*x*, 1, *o*)
- ...
- SCM_SET_CELL_OBJECT_*n* (*x*, *o*) ⇒ SCM_SET_CELL_OBJECT (*x*, *n*, *o*)

Резюме:

- Для опосредованного объекта Scheme *x* неизвестного типа, получить тип можно используя макрос `SCM_CELL_TYPE (x)`.
- Как только информация о типе ячейки будет доступна, используйте только соответствующие методы доступа для чтения и записи данных в разные записи ячейки.

9.3 Виртуальная машина для Guile

У Guile есть и интерпретатор и компилятор. Для пользователя разница является прозрачной — интерпретируемые и скомпилированные процедуры могут вызывать друг друга, как им нравиться.

Разница заключается в том, что компилятор создает и интерпретирует байт-код для пользовательской виртуальной машины, вместо интерпретации S-выражений напрямую. Загрузка и выполнение скомпилированного кода, быстрее, чем загрузка и выполнение исходного кода.

Виртуальная машина, которая интерпретирует байт-код, является частью самого Guile. Этот раздел описывает сущность виртуальой машины Guile.

9.3.1 Почему Виртуальная Машина(ВМ)?

Долгое время в Guile был только интерпретатор. Интерпретатор Guile обрабатывал на прямую S-выражения представляющие исходный код на Scheme.

Но даже в то время когда интерпретатор был очень оптимизирован и настроен вручную, он все еще выполнял множество ненужных вычислений. Например, применение функции к аргументам бесполезно повторяет аргументы в списке. При вычисление выражения всегда приходилось выяснять, что такое начало(car) выражения — это процедура, форма или что нибудь еще. Все значения должны быт размещены в куче. И т.д.

Решение этой проблемы состояло в том, чтобы скомпилировать язык более высокого уровня Scheme, в язык более низкого уровня, для которого все проверки и размещения уже выполнены. Код заменяется на минимальный минимумЖ необходимый для “выполнения задания”.

Тогда возникает вопрос, какой язык низкого уровня выбрать? Есть много вариантов. Мы можем напрямую скомпилировать программу в исполняемый код процессора. Но это создает проблемы с переносимостью для Guile, поскольку это очень кросс-платформенный проект.

Таким образом, мы хотим получить прирост производительности, который дает компилияция, но мы также хотим поддерживать преимущества переносимости выполняемого кода. Очевидное решение состоит в том, чтобы создать виртуальную машину, которая будет присутствовать во всех инсталяциях Guile.

Самый простой (и самый интересный) способ зависеть от виртуальной машины — это реализовать виртуальную машину внутри самого Guile. Guile содержит интерпретатор байт-кода (написанного на Си) и компилятор Scheme в байт-код (написанный на Scheme). Этот способ построения виртуальной машины обеспечивает необходимые Scheme конструкции (хвостовые вызовы(tail calls), множественные переменные (multiple values), (продолжения) `call/cc`) и может обеспечить оптимизацию встроенных инструкций Guile (`cons`, `struct-ref`, и т.д.).

Вот что делает Guile. В остальной части этого раздела описывается ВМ которая реализована в Guile и скомпилированные процедуры которые выполняются в ней.

Однако, прежде чем двигаться дальше, мы должны отметить, что, хотя мы говорили об интерпретаторе в прошедшем времени, у Guile все еще есть интерпретатор. Разница в том, что раньше это был главный исполнитель, и поэтому был реализован

в сильно оптимизированном Си коде; теперь он фактически реализован на Scheme, и скомпилирован до байт-кода ВМ, как и любая другая программа на Guile.(Там есть все еще и интерпретатор на Си, используемый для загрузки компилятора, но он обычно не используется во время выполнения.)

Потенциал реализованного интерпретатора на Scheme состоит в том, что мы сохраняем хвостовые вызовы и обработку множественных значений между интерпретируемым и скомпилированным кодом. Недостатком является то, что интерпретатор в Guile 2.2 все еще в два раза медленнее, чем интерпретатор в 1.8. Поскольку пользователи Scheme в основном вполняют скомпилированный код, скорость скомпилированного кода выше, что компенсирует потерю скорости интерпретатором. В любом случае, если у нас есть встроенная компиляция кода Scheme, мы ожидаем, что самообслуживаемый интерпретатор, легко обыграет старую настроенную в ручную Си реализацию.

Также обратите внимание, что это решение реализовать компилятор байт-кода не исключает компиляцию в исполняемый код базовой системы. Мы можем скомпилировать из байт-кода в исполняемый код во время выполнения, или даже сделать предварительную компиляцию. Дополнительные возможности обсуждаются в Раздел 9.4.7 [Extending the Compiler], страница 891.

9.3.2 Концепция Виртуальной Машины(ВМ)

Скомпилированный код запускается виртуальной машиной(ВМ). Каждый поток имеет собственную виртуальную машину. Виртуальная машина выполняет последовательность инструкций в процедуре.

Каждая инструкция ВМ начинается с указания, что это за операция, а затем следуют закодированные ее исходные и целевые операнды. Каждая процедура объявляет, какое она имеет количество локальных переменных, включая аргументы функции. Эти локальные переменные образуют доступные операнды в процедуре, и доступ к ним осуществляется по индексу.

Локальные переменные для процедуры храняться в стеке. Вызов процедуры обычно увеличивает стек, и возврат из процедуры сокращает его. Память стека является ограниченной по доступу виртуальной машиной, которой она принадлежит.

В дополнение к своим стекам виртуальные машины также имеют доступ к глобальной памяти (модули(modules), глобальные связанные перменные (global bindings), и т.д), которые разделяются между другими частями Guile, включая другие ВМ.

Регистры имеющиеся у ВМ следующие:

- ip - Указатель инструкции(Instruction pointer)
- sp - Указатель стеа(Stack pointer)
- fp - Указатель кадра(Frame pointer)

В других архитектурах указатель команд иногда называют “программным счетчиком(program counter)” (pc). Этот набор регистров довольно типичен для виртуальных машин; их точные значения в контексте виртуальной машины Guile описаны в следующем разделе.

9.3.3 Распределение Стека

Стек виртуальной машины Guile состоит из кадров(*frames*). Каждому кадру соответствует применение одной скомпилированной процедуры, и содержит пространство для хранения аргументов, локальных переменных и некоторой другой учетной информации (например, что делать после завершения кадра).

Хотя компилятор может делать все, что захочет, до тех пор пока семантика вычислений сохраняется, на практике каждый раз, когда вы вызываете функцию, создается новый кадр. (Заметным исключением, конечно, является случай хвостового вызова, см. Раздел 3.3.2 [Tail Calls], страница 25.)

Структура верхнего стекового кадра выглядит следующим образом:

```
| ...              |
+==================+ <- fp + 2 = SCM_FRAME_PREVIOUS_SP (fp)
| Dynamic link     |
+------------------+
| Return address   |
+==================+ <- fp
| Local 0          |
+------------------+
| Local 1          |
+------------------+
| ...              |
+------------------+
| Local N-1        |
\------------------/ <- sp
```

В приведенном выше рисунке стек растет вниз. В начале вызова функции, применяемая процедура находиться в Local 0, за которым следуют аргументы из Local 1. После проверки процедурой, что ей передаются совместимый набор аргументов, процедура выделяет некоторое дополнительное пространство в кадре для хранения локальных для функции переменных.

Обратите внимание, что после того, как значение в слоте локальной переменной больше не треубется, Guile может повторно использовать этот слот. Это относиться и к слотам, которые первоначально использовались для аргументов и вызываемого(?). По этой причине, обратная трасса в Guile не всегда может показать все аргументы: это может быть вызвано тем, что слот, соответствующий этому аргументу, был повторно использован какой-либо другой переменной.

Обратный адрес(return address) это `ip`, действоваший до применения этой программы. Когда мы вернемся из этого активного кадра, мы вернемся к этому `ip`. Аналогичным образом, динамическая ссылка(*dynamic link*) представляет собой смещение `fp` которое действовало до применения этой программы, относительно текущего `fp`.

Чтобы подготовиться к не-хвостовому приложению, ВМ Guile выполняет код который меняет размещает применяемую функцию и ее аргументы в соответствующие слоты стека с двумя свободными слотами под ними. Затем вызов инициализирует эти два свободных слота текущими `ip` и `fp`, и изменяет `ip` на точку входа в функцию, и `fp` на точку кадра нового вызова.

Таким образом, динамическая ссылка(dynamic link) связывает текущий кадр стека с предыдущим. Вычисление трассировки стека включает в себя перемещение по этим кадрам.

Каждый локальный стек Guile имеет ширину 64 бита, даже на 32-битных архитектурах. Это позволяет Guile сохранять единообразную обработку локального стека, позволяя при необходимости использовать арифметику 64-битных целых и с плавающей точкой чисел. См. См. Раздел 9.3.7 [Instruction Set], страница 851, для получения дополнительной информации об распакованной арифметике.

В качестве детали реализации, мы фактически сохраняем динамическую ссылку как смещение, а не абсолютное значение, потому что стек может перемещаться во время выполнения по мере его расширения или во время вызова продолжений(continuation). Если бы это было абсолютное значение, нам пришлось бы ходить по кадрам, и пересчитывать указатели кадров.

9.3.4 Переменные и Виртуальная Машина

В качестве примера рассмотрим следующий код Scheme:

```
(define (foo a)
  (lambda (b) (list foo a b)))
```

Внутри лямбда выражения, `foo` — это переменная верхнего уровня, `a` — это лексически захваченная переменная, и `b` это локальная перменная.

Другой путь, ссылки на `a` и `b` состоит в том, чтобы сказать, что `a` является "свободной" переменной, поскольку не определена в пределах лямбды, а `b` это "сязанная(bound)" переменная. Это термины, исползуемые в лямбда исчислении (*lambda calculus*), математическом обозначении для описания функций. Лямбда исчисление полезно потому что это язык, на котором можно точно рассуждать о функциях и переменных. Оно особенно хорошо при описании сферы отношений, и именно по этой причине мы упоминаем об этом здесь.

Guile выделяет все переменные в стеке. Когда лексически замкнутая процедура со свободными переменными — замыкание(*closure*)— создается, она копирует эти переменные в вектор своей свободной переменной. Ссылки на свободные переменные затем перенаправляются через вектор свободной переменной.

Если переменная когда-либо установлена `set!`, она должна быть выделена в куче(heap-allocated) вместо выделения в стеке, так что разные замыкания, которые захватывают одну и туже переменную, могут видеть одно и тоже значение. Кроме того, это позволяеет продолжениям(continuations) захватывать ссылки на переменную, а не на ее текущее значение. По этим причинам, `set!` переменные выделяемые в "рамках на рисунке стека(boxes)"—фактически, в ячейках переменных кучи. Дополнительную информацию см. См. Раздел 6.20.7 [Variables], страница 440, Ссылка на переменные в `set!` являются косвенными указывающими через boxes на ячекий в куче.

Таким образом, противоинтуитивное, казалось бы лучшее размещение "ближе к телу", а именно `set!`, на самом деле способствует дополнительному выделению памяти и косвенной адресации ячейки. Иногда оптимизатор Guile может удалить это выделение, но не всегда.

Возвращаясь к нашему примеру, `b` может быть выделен в стеке, поскольку он никогда не изменяется.

`a` также может быть выделено в стеке, поскольку она также никогда не изменяется. В лямбда замыкании, его значение будет скопировано в (и указано откуда) из вектора свободных переменных.

`foo` — это переменная верхнего уровня, потому что `foo` не лексически связан в этом примере.

9.3.5 Скомпилированные Процедуры это программы Виртуальной Машины

По умолчанию, когда вы вводите выражения в Guile REPL, они сначала компилируются в байт-код. Затем этот байт-код выполняется для создания значения. Если вычисляемое выражение является процедурой, результатом этого процесса будет скомпилированная процедура.

Скомпилированная процедура представляет собой составной объект, состоящий из его байт-кода и ссылки на любые лексически захваченные переменные. Кроме того, когда процедура компилируется, она ассоциируется с метаданными записанными как таблица, например, сопоставление кода и номеров строк, или docstring. Вы можете посмотреть эти части с помощью функций доступа из модуля `(system vm program)`. См. Раздел 6.9.3 [Compiled Procedures], страница 264, для полной ссылки на API.

Процедура может ссылаться на данные, которые были статически выделелены(распределены) когда процедура была скомпилирована. Например, пара непосредственных(immediate) объектов (см. Раздел 9.2.5.2 [Immediate objects], страница 840) может быть выделена непосредственно в сегменте памяти, который содержит скомпилированный байт-код и доступна непосредственно этому байт-коду.

Другое использование для статически распределенных данных - служить кешем для байт-кода. Запросы к переменным верхнего уровеня обрабатываются таким образом. Если инструкция `toplevel-box` обнаруживает, что она не имеет кэшированной переменной для ссылки верхнего уровня, она обращается к другим статическим данным для разрешения ссылки и заполняет ей слот кеша. После этого весь доступ к переменной происходит через кэш-ячейку. Значение переменной может измениться в будущем, но сама переменная изменена не будет.

Мы можем видеть, как эти понятия объединяются путем разбора функции `foo`, которую мы определили раньше. Чтоыбы посмотреть что происходит:

```
scheme@(guile-user)> (define (foo a) (lambda (b) (list foo a b)))
scheme@(guile-user)> ,x foo
Disassembly of #<procedure foo (a)> at #xea4ce4:

   0    (assert-nargs-ee/locals 2 0)    ;; 2 slots (1 arg)    at (unknown file):1:0
   1    (make-closure 1 7 1)            ;; anonymous procedure at #xea4d04 (1 free var)
   4    (free-set! 1 0 0)               ;; free var 0
   6    (mov 0 1)
   7    (return-values 2)               ;; 1 value

----------------------------------------
Disassembly of anonymous procedure at #xea4d04:

   0    (assert-nargs-ee/locals 2 2)    ;; 4 slots (1 arg)    at (unknown file):1:16
   1    (toplevel-box 1 74 58 68 #t)    ;; `foo'
   6    (box-ref 1 1)
   7    (make-short-immediate 0 772)    ;; ()                 at (unknown file):1:28
```

```
 8    (cons 2 2 0)
 9    (free-ref 3 3 0)               ;; free var 0
11    (cons 3 3 2)
12    (cons 2 1 3)
13    (return-values 2)              ;; 1 value
```

Сначала идет небольшая прелюдия, где `foo` проверяет, что она была вызвана только с 1 аргументом. Затем на `ip` 1, мы выделяем новое замыкание и сохраняем его в слоте 1 относительно `sp`.

Во время выполнения, локальные переменные в Guile обычно адресуются относительно указателя стека, что приводит к приятному эффективному доступу `sp[`*n*`]`. Однако это может затруднить дизасемблирование, потому что `sp` может меняться во время работы функции и потому что входящие аргумекнты относятся к `fp`, а не `sp`.

Чтобы узнать, какой ссылке на слот относительно `fp` соответствует ссылка относитьно `sp`, просматривайте дизассемблированный код вверх, пока вы не дойдете до анотации "*n* slots"; в нашем случае 2, что указывает на то, что кадру выделено место из 2 слотов. Таким образом слот адресуемый относительно `sp` с индексом 1 соответствует слоту адресуему относительно `fp` с индексом 0, который первоначально содержал значение вызванного замыкания. Это означает. что Guile значение этого замыкания не требуется и поэтому слот 0 был освобожден для повторного использования, в данном случае для результата создания нового замыкания.

Замыкание это код с данными. `6` в вызове создания замыкания `(make-closure 1 6 1)` является относительным смещением от точки указателя кода(ip) для замыкания, и конечная `1` указывает, что замыкание располагает местом для 1 свободной переменной. `ip` 4 инициализирует свободную переменную 0 в новом замыкании с помощью значения из слота расположенного относительно `sp`-со смещением 0, что соответствует слоту относительно `fp` со смещением 1, это первый аргумент функции `foo`: `a`. Наконец мы возвращаем замыкание.

Вторая строфа разбирает код для замыкания. После прелюдии, мы загружаем переменную для переменной верхнего уровня `foo` в слот 1. Этот поиск происходит лениво, в первое время переменная, на которое фактически ссылается переменная, и местоположение поиска кешируются так, что будущие ссылки на них очень дешевы. См. Раздел 9.3.7.2 [Top-Level Environment Instructions], страница 854, для более подробной информации. `box-ref` разыменовывает ячейку переменной, заменяя содержимое слота 1.

Ниже следует последовательность созданий пар(conses) для построения результирующего списка. `Ip` 7 создает хвост списка(непосредственный пустой список). `Ip` 8 conses создает пару из значения в слоте 2, соответствующее первому аргументу замыкания: `b` и хвоста. `Ip` 9 загружает свободную переменную 0 слота 3– вызыванной процедуры(a), в `fp`-относительно слота 0 – будет слот 3, `ip` 11 conses создает следующую пару списка. Наконец мы cons создаем пару, из значения в слоте 1, содержащую `foo` верхнего уровня(и хвоста), переднего плана списка, и затем возвращаем ее.

9.3.6 Формат Объектного Файла

Чтобы скомпилировать файл на диск, нам нужен формат для записи скомпилированного кода на диск и позже загрузки его в Guile. Хороший формат объектного файла *object file format*имеет ряд характеристик:

- Прежде всего, загрузка скомпилированного файла должна быть очень дешева.
- Должна быть возможность статически выделять константы в файле. Например, литерал байтовый вектор(bytevector) должен быть из исходного кода размещен непосредственно в объектный файл.
- Скомпилированный файл должен максимум кода и данных разделять между различными процессами.
- Скомпилированный файл должен содержать инофрмацию об отладке. такую как номера строк, но эта информация должна быть отделена от самого кода. должно быть возможность отлаживать информацию, если пространство заполнено.

Эти характеристики не специфичны для Scheme. Действительно, основные языки, такие как Си и Си++ решали эту проблему много раз в прошлом. Guile строит свою работу на принятии ELF, формата объектных файлов GNU и других Юникс подобных систем, как своего формата объектных файлов. Хотя Guile исползует ELF на всех платформах, мы не используем поддержку платформы для ELF. Guile реализует собственный компоновщик и загрузчик. Преимущество использования ELF не в обмене кодом, но в обмене идеями. ELF это просто хорошо продуманный формат объектного файла.

Файл ELF содержит две мета-таблицы, описывающих его содержимое. Первая мета-таблица для загрузчика, и называется таблицей программ (*program table*), а иногда таблицей сегментов(*segment table*). Таблица программ делит файл на большие куски, которые по разному обрабатываются загрузчиком. В основном разница между этими сгментами (*segments*) в их разрешениях(правах).

Обычно все сегменты файла ELF помечены как доступные только для чтения. за исключением той части которая представляет собой модифицируемые статические даные, которые необходио загрузить во время инициализации. Загрузка ELF файла так же проста, как функция отображение файла в оперативную память(mmapping) с флагом доступа только для чтения, затем используя таблицу сегментов, отмечают небольшой субрегион, как доступный для записи. Этот доступный для записи раздел обычно добавляют к корневому набору сборщика мусора.

Один сегмент ELF обозначается как динамический("dynamic"), что означает что он имеет данные, представляющие интерез для загрузчика. Guile использует этот сегмент для записи версии Guile, соответствующей этому файлу. Там также записывается в динамическом сегменте адрес точки инициализационного сегмента (thunk) который запускается для выполнения любой необходиой инициализации выполняемой во время связывания(link-time). (Это похоже на динамическое перемещение для обычных ELF разделяемых объкетов, за исключением того, что мы компилируем перестановки как процедуры, вместо того чтобы загрузчик интерпретировал таблицу перемещенй) Наконец, динамический сегмент обозначает местоположение входного сегмента("entry thunk") объектного файла. этот сегмент возвращается вызвавшему процедуру `load-thunk-from-memory` или `load-thunk-from-file`. При вызове, он выполнит "тело" скомпилированных выражений.

Другая мета-таблица в ELF файле это таблица разделов(*section table*). В то время как таблица программ делит файл ELF на большие куски для загрузчика, в таблице разделов указаны небольшие разделы для использования интроспективными инструментами, такими как отладчики или тому подоное. Один сегмент(запись в таблице

программ) обычно содержит много разделов. Также могут быть разделы вне любого сегмента.

Типичные разделы в файле Guile `.go` включают:

`.rtl-text`
: Байт-код(Bytecode).

`.data`
: Данные, которые требуют инициализации или которые могут быть изменены во время выполнения.

`.rodata`
: Статически распределенные данные, которые не требуют инициализации во время выполнения, и поэтому могут быть разделены между процессами.

`.dynamic`
: Динамический раздел, рассмотреный выше.

`.symtab`
`.strtab`
: Таблица отображения адресов в `.rtl-text` на имена процедур. `.strtab` используется `.symtab`.

`.guile.procprops`
`.guile.arities`
`.guile.arities.strtab`
`.guile.docstrs`
`.guile.docstrs.strtab`
: сторонние таблицы свойств процедур, arities и docstrings.

`.guile.docstrs.strtab`
: сторонние таблицы отображений кадров, описывающие набор живых слотов для любой точки возврата в тексте программы, и описывающих не являются ли эти слоты - указателями. Используется сборщиком мусора.

`.debug_info`
`.debug_abbrev`
`.debug_str`
`.debug_loc`
`.debug_line`
: Отладочная информация в формате DWARF. См. спецификацию DWARF, для большей информации.

`.shstrtab`
: Таблица сторок названий разделов.

Для получения дополнительной информации, см the elf(5) man page. См. the DWARF specification (`http://dwarfstd.org/`) для получения дополнительной информации о формате отладки DWARF. Или если вы авантюрный исследователь, попробуйте запустить `readelf` или `objdump` на скомпилированных файлах `.go`. Это хорошее время!

9.3.7 Набор Инструкций

В настоящее время на виртуальной машине Guile существует около 175. Эти инструкции представляют собой атомы единицы выполнения программы. В идеале они выполняют одну задачу безусловной ветви, затем отправляют на следующую инструкцию в потоке.

Инструкции сами по себе состоят из 1 или более 32 разрядных блоков. Младшие 8 бит первого слова указывают код операции, а остальные инструкции описывают операнды. Есть несколько различных способов закодировать операнды.

`sn` Беззнаковое целое *n*-бит, указывает относительно `sp` смещение локальной переменной.

`fn` Беззнаковое целое *n*-бит, указывает относительно `fp` смещение локальной переменной. Используется, когда продолжение(continuation) принимает переменное количество переменных, чтобы переставить полученные значения в известные местоположения в кадре.

`cn` Беззнаковое целое *n*-бит, указывет постоянное значение.

`l24` Смещение от текущего `ip`, в 32-битных единицах, является 24-битным значением со знаком. Указывает адрес байт-кода, для относительного перехода.

`i16`
`i32` Непосредственное значение Scheme (см. Раздел 9.2.5.2 [Immediate objects], страница 840), закодированное непосредственно в 16 или 32 битах.

`a32`
`b32` Непосредственные значения Scheme, закодированнве как пара 32-битных слов. Значения `a32` и `b32` всегда идут вместе для одного и тогоже кода операции, и указывают на старшие и младшие биты, соответственно. Обычно используются только в 64-битных системах.

`n32` Статически выделенная опосредованное значение. Адрес опосредованного значения закодированный как знаковое целое 32-битное число, и указывающее относительное смещение в 32-битных единицах. Думайте о нем как о `SCM x = ip + offset`.

`r32` Косвенное(не прямое) значение scheme, такое как `n32` но косвенное. Думайте о нем как о: `SCM *x = ip + offset`.

`l32`

`lo32` Адресс относительно ip, представляющий 32-битное знаковое целое. Может указывать на адрес байт-кода, как в `make-closure`, или на опосредованный адрес, как в `static-patch!`.

`l32` и `lo32` одинаковы с точки зрения виртуальной машины. Разница в том, что ассемблер может захотеть разрешить `lo32` как адрес указывающий на метку и некоторое количество слов смещения от этой метки, например когда исправляется поле для статически размещенного объекта.

`b1` Логическое значени: 1 для истины, иначе 0.

`xn` Игнорируемая последовательность из *n* бит.

Инструкция указывается путем указания ее имени, а затем описания ее операндов. Операнды упакованы в 32-битные слова, причем более ранние операнды занимают младшие биты.

Например, рассмотрим следующую спецификацию инструкции:

`free-set!` *s12:dst s12:src x8:_ c24:idx* [Instruction]
Установить свободную переменную *idx* из замыкания *dst* в *src*.

Первое слово в инструкции начнется с 8-битного значения, соответствующего коду операции *free-set!* в младших битах, за которым последуют *dst* и *src* как 12-битные значения. Второе слово начинается с 8 неиспользуемых битов, за которыми следует индех как 24-битное непосредственное значение.

Иногда компилятор может понять, что он компилирует специальный случай, который может быть запущен более эффективно. Так например, в то время как Guile предлагает общую инструкцию test-and-branch, он также предлагает конкретные инструкции для особых случаев, так что в следующих случаях все test-and-branch имеют свои инструкции:

```
(if pred then else)
(if (not pred) then else)
(if (null? l) then else)
(if (not (null? l)) then else)
```

Кроме того, некотоыре примитивы Scheme имеют свои встроенные реализации. Например, в предыдущем разделе мы видели `cons`.

Наконец, инструкции с операндами, которые кодируют ссылки на стек, интерпретируются от этих значений стека вверх до самой инсрукции. Большинство инструкций ожидают что их операнды помечены значениями SCM (представление `scm`), но некоторые инструкции ожидают распакованных целых (представление `u64` и `s64`) или числе с плавающей точкой (представление *f64*). Инструкции имеют статические типы: они должны получать свои операнды в формате, который они ожидают. Это зависит о компилятора, в данном случае. Если не указано иное, все операнды и результаты помещаются (упаковываются) в виде значений SCM.

9.3.7.1 Инстркукции Лексической Среды(Окружения)

Эти инструкции получают доступ и изменяют лексическую среду скомпилированной процедуры — ее свободных и связанных переменных. См. Раздел 9.3.3 [Stack Layout], страница 846, Для дополнительной информаций см. формат кадров стека.

`mov` *s12:dst s12:src* [Instruction]
`long-mov` *s24:dst x8:_ s24:src* [Instruction]
Скопировать значение из одного локального слота в другой.

Как обсуждалось ранее, аргументы процедуры и локальные переменные распределяеются в локальные слоты. Компилятор Guile пытается измежать перестановки переменных в различные слоты, что часто делает команды `mov` избыточными. Однако есть случаи, когда необходима перестановка, и в этих случаях `mov` это то, что нужно использовать.

`long-fmov` *f24:dst x8:_ f24:src* [Instruction]
Копирует значение из одного локального слота в другой, но адресация слотов относительно `fp` а не `sp`. Команда используется при перемещении значений после возврата множественного значения.

`make-closure` *s24:dst l32:offset x8:_ c24:nfree* [Instruction]
Создать новое замыкание и записать его в *dst*. Код для замыкания надо искать по смещению *offset* слов от текущего значения регистра `ip`. *offset* это 32-битное знаковое целое число. Пространство для *nfree* свободных переменных будет выделено.

Размер замыкания в настоящее время составляет два слова, плюс одно слово на каждую свободную переменную.

`free-ref` *s12:dst s12:src x8:_ c24:idx* [Instruction]
Загрузить свободную переменную *idx* из замыкания *src* в локальный слот *dst*.

`free-set!` *s12:dst s12:src x8:_ c24:idx* [Instruction]
Установить свободную перменную *idx* из замыкания *dst* в значение из *src*.

Эта инструкция обычно используется при инициализации свободных переменных замыкания, но не для изменения свободных переменных, поскольку назначенные переменные упакованы.

Напомним, что назначенные переменные обычно размещаются упакованными, так что продолжения и замыкания могут захватывать их идентификаторы, а не их значения которые они имеют в определенный момент времени. Переменные также используются при реализации высоко-уровневого связывания; см. следующий раздел для большей информации.

`box` *s12:dst s12:src* [Instruction]
Создать новую переменную хранящуюся(?) в *src*, поместить ее в *dst*.

`box-ref` *s12:dst s12:src* [Instruction]
Распаковать переменную из *src* в *dst*, проверяя что переменная является связанной.

`box-set!` *s12:dst s12:src* [Instruction]
Установить содержимое переменной *dst* в значение *set*.

9.3.7.2 Инструкции сред Верхнего Уровня

Эти инструкции получают значения в среде верхнего уровня: связывание которых было не лексическим в то время когда этот код был скомпилирован.

Место, в котором храниться связанное значение верхнего урвня, можно разыскать одирн раз и кэшировать для дальнейшего использования. Само связанное значение может меняться со временем, но ее местоположение останется постоянным.

`current-module` *s24:dst* [Instruction]
Сохранить текущий модуль в *dst*.

`resolve` *s24:dst b1:bound? x7:_ s24:sym* [Instruction]
Разрешить(найти) *sym* в текущем модуле и поместить полученную переменную в *dst*. Будет вызвана ошибка если переменная не найдена. Если *bound?* - истина(переменная связана), ошибка будет сигнализировать о том что переменная не связана.

`define!` *s12:dst s12:sym* [Instruction]

Найти связанную переменную для *sym* в текущем модуле, создать ее если необходимо, Хранить эту переменную в *dst*.

`toplevel-box` *s24:dst r32:var-offset r32:mod-offset n32:sym-offset b1:bound? x31:_* [Instruction]

Загружает значение. Значение будет извлекаться из памяти, используя значение *var-offset* в 32-битных словах от ip(текщей точки указателя программы) *var-offset* это значение со знаком. Здесь, `toplevel-box` как `static-ref`.

Затем, если загруженное значение является переменной, оно помещатеся в *dst* и выполнение продолжается.

Иначе, мы должны разрешить(найти) эту переменную. В этом случае мы загружаем модуль из *mod-offset*, также как мы грузим переменную. Обычно модуль получает установки, когда замыкание создается. *sym-offset* задает имя, как смещение к символу.

Мы используем модуль и символ для разрешения(поиска) переменной, помещая ее в *dst*, и кеширования разрешенной(найденой) переменной, чтобы в следующий раз мы попали в кеш. Если *bound?* это истина, а иначе выдается сообщение об ошибке если переменная не связана.

`module-box` *s24:dst r32:var-offset n32:mod-offset n32:sym-offset b1:bound? x31:_* [Instruction]

Подобно `toplevel-box`, за исключением *mod-offset* указывает на идентификатор модуля, вместо самого модуля. Идентификатор модуля - это имя модуля, как список, предваряемый логическим значением. Если это значение истинно, то переменная разрешена(найдена) относительно общедоступного интерфейса модуля, вместо частного интерфеса.

9.3.7.3 Инструкции вызова Процедур и Возврата

Как описано выше (см. Раздел 9.3.3 [Stack Layout], страница 846), соглашение о вызовах Guile такое, что аргументы передаются а значения возвращаются в стеке.

Для вызовов, как в хвостовом положении, так и нехвостовом., мы требуем, чтобы процедура и аргументы повторно перемещались на место перед инструкцией вызова. “На место” для хвостового вызова означает, что процедура должна быть в 0 слоте, относительно указателя кадра `fp`, и за ней должны следовать аргументы. Для не хвостового вызова, если процедура размещается в слоте *n* относительно `fp`, аргументы должны следовать за слотом *n*+1, и должно быть два свободных слота со смещением *n*-1 и *n*-2 для сохранения текущих `ip` и `fp`.

Возвращение значений аналогично. Многозначные значения повторно перемещаются вниз, чтобы начать со слота номер 1 относительно `fp` перед вызовом `return-values`. Мы начинаем со слота 1 вместо слота 0 что бы сделать квостовые вызовы (со значениями|значений?) `values` тривиальными.

В обоих вызовах и возрватах `sp` используется для указания вызываемомой процедуре числа аргументов или возвращаемых значений для вызвавшей процедры, соответственно. После получения возвращаемых значений, вызвавшая процедура обязана востанановить кадр *restore the frame* и сбросить `sp` в его прежнее значение.

`call` *f24:proc x8:_ c24:nlocals* [Instruction]
Вызо процедуры. *proc* является локальной соответствущей значению процедуре. Два значения ниже *proc* будут перезаписаны сохраненными данными кадра вызова. Новый кадр будет иметь пространство для *nlocals* локальных: одно для процедуры, а остальные для аргументов, которые уже должны быть уже размещены.

Кода вызов завершается, выполнение продолжается со следующей инструкции. В стеке может быть возвращено любое количество значений; точное их число может быть получено вычитанием адреса *proc* из установленого после возврата значения `sp`.

`call-label` *f24:proc x8:_ c24:nlocals l32:label* [Instruction]
Вызов процедуры в томже компиляционном блоке.

Эта инструкция похожа на вызов `call`, за исключением того, что вместо разыменовывания *proc* для поиска цели вызова, цель вызова как известно находиться на метке *label*, знаковое целое 32-битное смещение относительно текущего указателя программы `ip`. Поскольу *proc* на разыменовывается, это может быть какое-то другое представление замыкания.

`tail-call` *c24:nlocals* [Instruction]
Хвостовой вызов процедуры. Требует чтобы процедура и все аргументы уже были перемещены в позицию. Сбрасывает кадр на *nlocals*.

`tail-call-label` *c24:nlocals l32:label* [Instruction]
Хвостовой вызов известной процедуры. В качестве вызываемой `call` стоит метка `call-label`, `tail-call` is to `tail-call-label`.

`tail-call/shuffle` *f24:from* [Instruction]
Хвостовой вызов процедуры. Процедура уж должна быть уже размещена в 0 слоте. Остальная часть args берется из кадра, начиная с *from*, перемещаются вниз, чтобы начать со слота 0. Это является частью реализации встроенной `call-with-values`.

`receive` *f12:dst f12:proc x8:_ c24:nlocals* [Instruction]
Получает одно возвращаемое значение из вызова, чья процедура была *proc*, проверяя, что вызов фактически возвратил хотябы одно значение. После этого сбрасывает кадр на *nlocals* локальных.

`receive-values` *f24:proc b1:allow-extra? x7:_ c24:nvalues* [Instruction]
Получает возвращаемое множество значений из вызвова, чья процедура была *proc*. Если возвращеных значений меньше чем *nvalues*, сигнализирует об ошибке. Если не *allow-extra?* является истинным, требует чтобы число возвращаемых значений точно равнялось *nvalues*. После получения значений `receive-values`, значения могут быть скопированы с помощью `mov`, или использованы на месте.

`return-values` *c24:nlocals* [Instruction]
Возвращает несколько значений из кадра вызова. Этому коду операции соответствует применение значений `values` в хвостовой позиции. Как и в случае с

хвостовыми вызовами, мы ожидаем, что значения уже перемещаются вниз, начиная со слота 1. Если *nlocals* не ноль, is nonzero, кадр сбрасывается, что бы удерживать это количество локальных. Обратите внимание, что кадр сбрасывается до 1, local возвращает 0 значений.

`call/cc` *x24:_* [Instruction]
Захватывает текущее продолжение, и применяет(tail-apply) процедуру из локального слота 1 к ней. Эта инструкция является частью реализации `call/cc`, и не генерируется компилятором.

9.3.7.4 Инструкции начинающие функции

Вызов функции в Guile очень дешев: Виртуальная Машина просто контролирует процедуру. Сама процедура несет ответственность за проверку того, что ей передано приемлемое количество аргументов. Эта стратегия позволяет разбирать аргументы произвольной сложности анализирующими идиомами, без ущерба для общего случая.

Например, только вызывая процедуры с ключевыми аргументами процедура платит за разбор ключевых аргументов. (На момент написания, вызов процедур с ключевыми аргументами обычно в два-четыре раза дороже, чем вызов процедур с фиксированным набором аргументов.)

`assert-nargs-ee` *c24:expected* [Instruction]
`assert-nargs-ge` *c24:expected* [Instruction]
`assert-nargs-le` *c24:expected* [Instruction]
If the number of actual arguments is not `==`, `>=`, or `<=` *expected*, respectively, signal an error.

The number of arguments is determined by subtracting the stack pointer from the frame pointer (`fp - sp`). См. Раздел 9.3.3 [Stack Layout], страница 846, for more details on stack frames. Note that *expected* includes the procedure itself.

`br-if-nargs-ne` *c24:expected x8:_ l24:offset* [Instruction]
`br-if-nargs-lt` *c24:expected x8:_ l24:offset* [Instruction]
`br-if-nargs-gt` *c24:expected x8:_ l24:offset* [Instruction]
If the number of actual arguments is not equal, less than, or greater than *expected*, respectively, add *offset*, a signed 24-bit number, to the current instruction pointer. Note that *expected* includes the procedure itself.

These instructions are used to implement multiple arities, as in `case-lambda`. См. Раздел 6.9.5 [Case-lambda], страница 270, for more information.

`alloc-frame` *c24:nlocals* [Instruction]
Ensure that there is space on the stack for *nlocals* local variables, setting them all to `SCM_UNDEFINED`, except those values that are already on the stack.

`reset-frame` *c24:nlocals* [Instruction]
Like `alloc-frame`, but doesn't check that the stack is big enough, and doesn't initialize values to `SCM_UNDEFINED`. Used to reset the frame size to something less than the size that was previously set via alloc-frame.

`assert-nargs-ee/locals` *c12:expected c12:nlocals* [Instruction]
Equivalent to a sequence of `assert-nargs-ee` and `reserve-locals`. The number of locals reserved is *expected* + *nlocals*.

`br-if-npos-gt` *c24:nreq x8:_ c24:npos x8:_ l24:offset* [Instruction]
Find the first positional argument after *nreq*. If it is greater than *npos*, jump to *offset*.

This instruction is only emitted for functions with multiple clauses, and an earlier clause has keywords and no rest arguments. См. Раздел 6.9.5 [Case-lambda], страница 270, for more on how `case-lambda` chooses the clause to apply.

`bind-kwargs` *c24:nreq c8:flags c24:nreq-and-opt x8:_ c24:ntotal n32:kw-offset* [Instruction]
flags is a bitfield, whose lowest bit is *allow-other-keys*, second bit is *has-rest*, and whose following six bits are unused.

Find the last positional argument, and shuffle all the rest above *ntotal*. Initialize the intervening locals to `SCM_UNDEFINED`. Then load the constant at *kw-offset* words from the current *ip*, and use it and the *allow-other-keys* flag to bind keyword arguments. If *has-rest*, collect all shuffled arguments into a list, and store it in *nreq-and-opt*. Finally, clear the arguments that we shuffled up.

The parsing is driven by a keyword arguments association list, looked up using *kw-offset*. The alist is a list of pairs of the form `(`*kw* `.` *index*`)`, mapping keyword arguments to their local slot indices. Unless `allow-other-keys` is set, the parser will signal an error if an unknown key is found.

A macro-mega-instruction.

`bind-rest` *f24:dst* [Instruction]
Collect any arguments at or above *dst* into a list, and store that list at *dst*.

9.3.7.5 Инструкции Траплины

Хотя наиболее применимыми объектами в Guile являются процедуры, реализованные в байт-коде, это не все. Есть примитивы, продолжения и другие подобные процедурам объекты, которые имеют собственной соглашение о вызовах. Вместо добавления особых случаев в инстркцию вызова `call`, Guile обертывает эти другие применимые объекты в процедуры Трамплины Виртуальной Машины, затем предоставляя специальную поддержку этих объектов в байт коде.

Процедуры Трамплины обычно генерируются Guile во время выполнения, например, в ответ на вызов `scm_c_make_gsubr`. Таким образом, компилятор, вероятно, не должен генерировать код с этими инструкциями. Тем не менее, все еще интересно знать, как они работают, поэтому мы запишем эти trampoline инструкции здесь.

`subr-call` *x24:_* [Instruction]
Call a subr, passing all locals in this frame as arguments. Return from the calling frame.

`foreign-call` *c12:cif-idx c12:ptr-idx* [Instruction]
Call a foreign function. Fetch the *cif* and foreign pointer from *cif-idx* and *ptr-idx*, both free variables. Return from the calling frame. Arguments are taken from the stack.

`continuation-call` *c24:contregs* [Instruction]
Return to a continuation, nonlocally. The arguments to the continuation are taken from the stack. *contregs* is a free variable containing the reified continuation.

`compose-continuation` *c24:cont* [Instruction]
Compose a partial continution with the current continuation. The arguments to the continuation are taken from the stack. *cont* is a free variable containing the reified continuation.

`tail-apply` *x24:_* [Instruction]
Tail-apply the procedure in local slot 0 to the rest of the arguments. This instruction is part of the implementation of `apply`, and is not generated by the compiler.

`builtin-ref` *s12:dst c12:idx* [Instruction]
Load a builtin stub by index into *dst*.

`apply-non-program` *x24:_* [Instruction]
An instruction used only by a special trampoline that the VM uses to apply non-programs. Using that trampoline allows profilers and backtrace utilities to avoid seeing the instruction pointer from the calling frame.

9.3.7.6 Инструкции Ветвления

Все смещения в инструкциях ветвления это 24-битные числа со знаком, которые подсчитвают 32-битовые единицы. Это дает Guile эффективный 26-битный диапазон адреов для относительных переходов.

`br` *l24:offset* [Instruction]
Add *offset* to the current instruction pointer.

All the conditional branch instructions described below have an *invert* parameter, which if true reverses the test: `br-if-true` becomes `br-if-false`, and so on.

`br-if-true` *s24:test b1:invert x7:_ l24:offset* [Instruction]
If the value in *test* is true for the purposes of Scheme, add *offset* to the current instruction pointer.

`br-if-null` *s24:test b1:invert x7:_ l24:offset* [Instruction]
If the value in *test* is the end-of-list or Lisp nil, add *offset* to the current instruction pointer.

`br-if-nil` *s24:test b1:invert x7:_ l24:offset* [Instruction]
If the value in *test* is false to Lisp, add *offset* to the current instruction pointer.

`br-if-pair` *s24:test b1:invert x7:_ l24:offset* [Instruction]
If the value in *test* is a pair, add *offset* to the current instruction pointer.

`br-if-struct` *s24:test b1:invert x7:_ l24:offset* [Instruction]
If the value in *test* is a struct, add *offset* number to the current instruction pointer.

`br-if-char` *s24:test b1:invert x7:_ l24:offset* [Instruction]
If the value in *test* is a char, add *offset* to the current instruction pointer.

`br-if-tc7` *s24:test b1:invert u7:tc7 l24:offset* [Instruction]
If the value in *test* has the TC7 given in the second word, add *offset* to the current instruction pointer. TC7 codes are part of the way Guile represents non-immediate objects, and are deep wizardry. See `libguile/tags.h` for all the details.

`br-if-eq` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
`br-if-eqv` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
If the value in *a* is `eq?` or `eqv?` to the value in *b*, respectively, add *offset* to the current instruction pointer.

`br-if-=` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
`br-if-<` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
`br-if-<=` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
If the value in *a* is `=`, `<`, or `<=` to the value in *b*, respectively, add *offset* to the current instruction pointer.

`br-if-logtest` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
If the bitwise intersection of the integers in *a* and *b* is nonzero, add *offset* to the current instruction pointer.

9.3.7.7 Constant Instructions

Следующие инструкции загружают литеральные данные в программу. Есть два вида.

Первый набор инструкций загружает непосредственные значения. Эти инструкции кодируют непосредственные данные в поток команд.

`make-short-immediate` *s8:dst i16:low-bits* [Instruction]
Make an immediate whose low bits are *low-bits*, and whose top bits are 0.

`make-long-immediate` *s24:dst i32:low-bits* [Instruction]
Make an immediate whose low bits are *low-bits*, and whose top bits are 0.

`make-long-long-immediate` *s24:dst a32:high-bits b32:low-bits* [Instruction]
Make an immediate with *high-bits* and *low-bits*.

Опосредованные постоянные литералы адресованы прямо или косвенно. Например, Guile знает во время компиляции, как будет выглядеть размщение строки, и организует встраивание этого объекта непосредственно в скомпилированный образ. Ссылка на строку будет использовать вызов `make-non-immediate` для непосредственного обращения с указателем в скомпилированный блок как `SCM` значению.

`make-non-immediate` *s24:dst n32:offset* [Instruction]
Load a pointer to statically allocated memory into *dst*. The object's memory is will be found *offset* 32-bit words away from the current instruction pointer. Whether the object is mutable or immutable depends on where it was allocated by the compiler, and loaded by the loader.

Некоторые объект должны быть уникальрными во всей системе. Это относится к символам и ключевым словам. Для этих объектов, Guile организует инициализацию, когда скомпилированный блок загружается, сохраняя их в слоты на образе(загруженном). Ссылки идут косвенно, через этот слот. В этом случае используется `static-ref`.

`static-ref` *s24:dst r32:offset* [Instruction]
Load a *scm* value into *dst*. The *scm* value will be fetched from memory, *offset* 32-bit words away from the current instruction pointer. *offset* is a signed value.

Поля опосредованные возможно придется исправлять во время загрузки, поэтому мы не знаем заранее, по какому адресу они будут загружены. Это имеет место, наприимер для пары содержащей опосредованные данные в одном из полей. В этих ситуациях используются `static-ref` и `static-patch!`.

`static-set!` *s24:src lo32:offset* [Instruction]
Store a *scm* value into memory, *offset* 32-bit words away from the current instruction pointer. *offset* is a signed value.

`static-patch!` *x24:_ lo32:dst-offset l32:src-offset* [Instruction]
Patch a pointer at *dst-offset* to point to *src-offset*. Both offsets are signed 32-bit values, indicating a memory address as a number of 32-bit words away from the current instruction pointer.

Many kinds of literals can be loaded with the above instructions, once the compiler has prepared the statically allocated data. This is the case for vectors, strings, uniform vectors, pairs, and procedures with no free variables. Other kinds of data might need special initializers; those instructions follow.

`string->number` *s12:dst s12:src* [Instruction]
Parse a string in *src* to a number, and store in *dst*.

`string->symbol` *s12:dst s12:src* [Instruction]
Parse a string in *src* to a symbol, and store in *dst*.

`symbol->keyword` *s12:dst s12:src* [Instruction]
Make a keyword from the symbol in *src*, and store it in *dst*.

`load-typed-array` *s24:dst x8:_ s24:type x8:_ s24:shape n32:offset u32:len* [Instruction]
Load the contiguous typed array located at *offset* 32-bit words away from the instruction pointer, and store into *dst*. *len* is a byte length. *offset* is signed.

9.3.7.8 Инструкции Динамической Среды(Окружения)

Виртуальная машина Guile имеет низкоуровневую поддержку для `dynamic-wind`, динамического связывания и составных подсказок и прерываний.

`abort` *x24:_* [Instruction]
Abort to a prompt handler. The tag is expected in slot 1, and the rest of the values in the frame are returned to the prompt handler. This corresponds to a tail application of abort-to-prompt.

If no prompt can be found in the dynamic environment with the given tag, an error is signalled. Otherwise all arguments are passed to the prompt's handler, along with the captured continuation, if necessary.

If the prompt's handler can be proven to not reference the captured continuation, no continuation is allocated. This decision happens dynamically, at run-time; the general case is that the continuation may be captured, and thus resumed. A reinstated continuation will have its arguments pushed on the stack from slot 1, as if from a multiple-value return, and control resumes in the caller. Thus to the calling function, a call to `abort-to-prompt` looks like any other function call.

`prompt` *s24:tag b1:escape-only? x7:_ f24:proc-slot x8:_ l24:handler-offset* [Instruction]

Push a new prompt on the dynamic stack, with a tag from *tag* and a handler at *handler-offset* words from the current *ip*.

If an abort is made to this prompt, control will jump to the handler. The handler will expect a multiple-value return as if from a call with the procedure at *proc-slot*, with the reified partial continuation as the first argument, followed by the values returned to the handler. If control returns to the handler, the prompt is already popped off by the abort mechanism. (Guile's `prompt` implements Felleisen's *–F–* operator.)

If *escape-only?* is nonzero, the prompt will be marked as escape-only, which allows an abort to this prompt to avoid reifying the continuation.

См. Раздел 6.13.5 [Prompts], страница 323, for more information on prompts.

`wind` *s12:winder s12:unwinder* [Instruction]

Push wind and unwind procedures onto the dynamic stack. Note that neither are actually called; the compiler should emit calls to wind and unwind for the normal dynamic-wind control flow. Also note that the compiler should have inserted checks that they wind and unwind procs are thunks, if it could not prove that to be the case. См. Раздел 6.13.10 [Dynamic Wind], страница 339.

`unwind` *x24:_* [Instruction]

a normal exit from the dynamic extent of an expression. Pop the top entry off of the dynamic stack.

`push-fluid` *s12:fluid s12:value* [Instruction]

Dynamically bind *value* to *fluid* by creating a with-fluids object and pushing that object on the dynamic stack. См. Раздел 6.13.11 [Fluids and Dynamic States], страница 343.

`pop-fluid` *x24:_* [Instruction]

Leave the dynamic extent of a `with-fluid*` expression, restoring the fluid to its previous value. `push-fluid` should always be balanced with `pop-fluid`.

`fluid-ref` *s12:dst s12:src* [Instruction]

Reference the fluid in *src*, and place the value in *dst*.

`fluid-set!` *s12:fluid s12:val* [Instruction]

Set the value of the fluid in *dst* to the value in *src*.

`current-thread` *s24:dst* [Instruction]
Write the value of the current thread to *dst*.

`push-dynamic-state` *s24:state* [Instruction]
Save the current set of fluid bindings on the dynamic stack and instate the bindings from *state* instead. См. Раздел 6.13.11 [Fluids and Dynamic States], страница 343.

`pop-dynamic-state` *x24:_* [Instruction]
Restore a saved set of fluid bindings from the dynamic stack. `push-dynamic-state` should always be balanced with `pop-dynamic-state`.

9.3.7.9 Разные Инструкции

`halt` *x24:_* [Instruction]
Bring the VM to a halt, returning all the values from the stack. Used in the "boot continuation", which is used when entering the VM from C.

`push` *s24:src* [Instruction]
Bump the stack pointer by one word, and fill it with the value from slot *src*. The offset to *src* is calculated before the stack pointer is adjusted.

The `push` instruction is used when another instruction is unable to address an operand because the operand is encoded with fewer than 24 bits. In that case, Guile's assembler will transparently emit code that temporarily pushes any needed operands onto the stack, emits the original instruction to address those now-near variables, then shuffles the result (if any) back into place.

`pop` *s24:dst* [Instruction]
Pop the stack pointer, storing the value that was there in slot *dst*. The offset to *dst* is calculated after the stack pointer is adjusted.

`drop` *c24:count* [Instruction]
Pop the stack pointer by *count* words, discarding any values that were stored there.

`handle-interrupts` *x24:_* [Instruction]
Handle pending asynchronous interrupts (asyncs). См. Раздел 6.22.3 [Asyncs], страница 466. The compiler inserts `handle-interrupts` instructions before any call, return, or loop back-edge.

`return-from-interrupt` *x24:_* [Instruction]
A special instruction to return from a call and also pop off the stack frame from the call. Used when returning from asynchronous interrupts.

9.3.7.10 Встроенные Инструкции Scheme

Компилятор Scheme может распознать применение стандартных Scheme процедур. Он пытается, встроить эти небольшие операции, чтобы избежать накладных раходов на создание новых кадров стека. Это позволяет компилятору лучше оптимизировать код.

`make-vector` *s8:dst s8:length s8:init* [Instruction]
Make a vector and write it to *dst*. The vector will have space for *length* slots. They will be filled with the value in slot *init*.

`make-vector/immediate` *s8:dst s8:length c8:init* [Instruction]
Make a short vector of known size and write it to *dst*. The vector will have space for *length* slots, an immediate value. They will be filled with the value in slot *init*.

`vector-length` *s12:dst s12:src* [Instruction]
Store the length of the vector in *src* in *dst*, as an unboxed unsigned 64-bit integer.

`vector-ref` *s8:dst s8:src s8:idx* [Instruction]
Fetch the item at position *idx* in the vector in *src*, and store it in *dst*. The *idx* value should be an unboxed unsigned 64-bit integer.

`vector-ref/immediate` *s8:dst s8:src c8:idx* [Instruction]
Fill *dst* with the item *idx* elements into the vector at *src*. Useful for building data types using vectors.

`vector-set!` *s8:dst s8:idx s8:src* [Instruction]
Store *src* into the vector *dst* at index *idx*. The *idx* value should be an unboxed unsigned 64-bit integer.

`vector-set!/immediate` *s8:dst c8:idx s8:src* [Instruction]
Store *src* into the vector *dst* at index *idx*. Here *idx* is an immediate value.

`struct-vtable` *s12:dst s12:src* [Instruction]
Store the vtable of *src* into *dst*.

`allocate-struct` *s8:dst s8:vtable s8:nfields* [Instruction]
Allocate a new struct with *vtable*, and place it in *dst*. The struct will be constructed with space for *nfields* fields, which should correspond to the field count of the *vtable*. The *idx* value should be an unboxed unsigned 64-bit integer.

`struct-ref` *s8:dst s8:src s8:idx* [Instruction]
Fetch the item at slot *idx* in the struct in *src*, and store it in *dst*. The *idx* value should be an unboxed unsigned 64-bit integer.

`struct-set!` *s8:dst s8:idx s8:src* [Instruction]
Store *src* into the struct *dst* at slot *idx*. The *idx* value should be an unboxed unsigned 64-bit integer.

`allocate-struct/immediate` *s8:dst s8:vtable c8:nfields* [Instruction]
`struct-ref/immediate` *s8:dst s8:src c8:idx* [Instruction]
`struct-set!/immediate` *s8:dst c8:idx s8:src* [Instruction]
Variants of the struct instructions, but in which the *nfields* or *idx* fields are immediate values.

`class-of` *s12:dst s12:type* [Instruction]
Store the vtable of *src* into *dst*.

`make-array` *s24:dst x8:_ s24:type x8:_ s24:fill x8:_ s24:bounds* [Instruction]
Make a new array with *type*, *fill*, and *bounds*, storing it in *dst*.

`string-length` *s12:dst s12:src* [Instruction]
Store the length of the string in *src* in *dst*, as an unboxed unsigned 64-bit integer.

`string-ref` *s8:dst s8:src s8:idx* [Instruction]
Fetch the character at position *idx* in the string in *src*, and store it in *dst*. The *idx* value should be an unboxed unsigned 64-bit integer.

`string-set!` *s8:dst s8:idx s8:src* [Instruction]
Store the character *src* into the string *dst* at index *idx*. The *idx* value should be an unboxed unsigned 64-bit integer.

`cons` *s8:dst s8:car s8:cdr* [Instruction]
Cons *car* and *cdr*, and store the result in *dst*.

`car` *s12:dst s12:src* [Instruction]
Place the car of *src* in *dst*.

`cdr` *s12:dst s12:src* [Instruction]
Place the cdr of *src* in *dst*.

`set-car!` *s12:pair s12:car* [Instruction]
Set the car of *dst* to *src*.

`set-cdr!` *s12:pair s12:cdr* [Instruction]
Set the cdr of *dst* to *src*.

Note that `caddr` and friends compile to a series of `car` and `cdr` instructions.

`integer->char` *s12:dst s12:src* [Instruction]
Convert the `u64` value in *src* to a Scheme character, and place it in *dst*.

`char->integer` *s12:dst s12:src* [Instruction]
Convert the Scheme character in *src* to an integer, and place it in *dst* as an unboxed `u64` value.

9.3.7.11 Встроенные Атомарные Инструкции

См. Раздел 6.22.4 [Atomics], страница 468, для получения дополнительрной информации об атомарных операциях в Guile.

`make-atomic-box` *s12:dst s12:src* [Instruction]
Create a new atomic box initialized to *src*, and place it in *dst*.

`atomic-box-ref` *s12:dst s12:box* [Instruction]
Fetch the value of the atomic box at *box* into *dst*.

`atomic-box-set!` *s12:box s12:val* [Instruction]
Set the contents of the atomic box at *box* to *val*.

`atomic-box-swap!` *s12:dst s12:box x8:_ s24:val* [Instruction]
Replace the contents of the atomic box at *box* to *val* and store the previous value at *dst*.

`atomic-box-compare-and-swap!` *s12:dst s12:box x8:_ s24:expected x8:_ s24:desired* [Instruction]
If the value of the atomic box at *box* is the same as the SCM value at *expected* (in the sense of `eq?`), replace the contents of the box with the SCM value at *desired*. Otherwise does not update the box. Set *dst* to the previous value of the box in either case.

9.3.7.12 Инструкции Встроенной Математики

Встраивание математических операций имеет очевидное преимущесвтво при работе с фиксированными числами без вызова фукнций или размещений. Трюк конечно, когда знаете что результат операции будет fixnum, и здесь может быть пара ошибок.

More instructions could be added here over time.

All of these operations place their result in their first operand, *dst*.

`add` *s8:dst s8:a s8:b* [Instruction]
Add *a* to *b*.

`add/immediate` *s8:dst s8:src c8:imm* [Instruction]
Add the unsigned integer *imm* to the value in *src*.

`sub` *s8:dst s8:a s8:b* [Instruction]
Subtract *b* from *a*.

`sub/immediate` *s8:dst s8:src s8:imm* [Instruction]
Subtract the unsigned integer *imm* from the value in *src*.

`mul` *s8:dst s8:a s8:b* [Instruction]
Multiply *a* and *b*.

`div` *s8:dst s8:a s8:b* [Instruction]
Divide *a* by *b*.

`quo` *s8:dst s8:a s8:b* [Instruction]
Divide *a* by *b*.

`rem` *s8:dst s8:a s8:b* [Instruction]
Divide *a* by *b*.

`mod` *s8:dst s8:a s8:b* [Instruction]
Compute the modulo of *a* by *b*.

`ash` *s8:dst s8:a s8:b* [Instruction]
Shift *a* arithmetically by *b* bits.

`logand` *s8:dst s8:a s8:b* [Instruction]
Compute the bitwise `and` of *a* and *b*.

`logior` *s8:dst s8:a s8:b* [Instruction]
Compute the bitwise inclusive `or` of *a* with *b*.

`logxor` *s8:dst s8:a s8:b* [Instruction]
Compute the bitwise exclusive `or` of *a* with *b*.

`logsub` *s8:dst s8:a s8:b* [Instruction]
Place the bitwise `and` of *a* and the bitwise `not` of *b* into *dst*.

9.3.7.13 Инструкции Встроенных Байтовых Векторов

Операторы байтовых векторов точно соответствуют тому, что может сделать текущее оборудование, поэтому они сделаны что быть встроенными в инструкции ВМ, обеспечивая четкий путь для возможной нативной компиляции. Без этого программам Scheme понадобятся другие примитивы для доступа к сырым(необработанным) данным - но эти примитивы так же хороши как и другие.

`bv-length` *s12:dst s12:src* [Instruction]
Store the length of the bytevector in *src* in *dst*, as an unboxed unsigned 64-bit integer.

`bv-u8-ref` *s8:dst s8:src s8:idx* [Instruction]
`bv-s8-ref` *s8:dst s8:src s8:idx* [Instruction]
`bv-u16-ref` *s8:dst s8:src s8:idx* [Instruction]
`bv-s16-ref` *s8:dst s8:src s8:idx* [Instruction]
`bv-u32-ref` *s8:dst s8:src s8:idx* [Instruction]
`bv-s32-ref` *s8:dst s8:src s8:idx* [Instruction]
`bv-u64-ref` *s8:dst s8:src s8:idx* [Instruction]
`bv-s64-ref` *s8:dst s8:src s8:idx* [Instruction]
`bv-f32-ref` *s8:dst s8:src s8:idx* [Instruction]
`bv-f64-ref` *s8:dst s8:src s8:idx* [Instruction]
Fetch the item at byte offset *idx* in the bytevector *src*, and store it in *dst*. All accesses use native endianness.

The *idx* value should be an unboxed unsigned 64-bit integer.

The results are all written to the stack as unboxed values, either as signed 64-bit integers, unsigned 64-bit integers, or IEEE double floating point numbers.

`bv-u8-set!` *s8:dst s8:idx s8:src* [Instruction]
`bv-s8-set!` *s8:dst s8:idx s8:src* [Instruction]
`bv-u16-set!` *s8:dst s8:idx s8:src* [Instruction]
`bv-s16-set!` *s8:dst s8:idx s8:src* [Instruction]
`bv-u32-set!` *s8:dst s8:idx s8:src* [Instruction]
`bv-s32-set!` *s8:dst s8:idx s8:src* [Instruction]
`bv-u64-set!` *s8:dst s8:idx s8:src* [Instruction]
`bv-s64-set!` *s8:dst s8:idx s8:src* [Instruction]
`bv-f32-set!` *s8:dst s8:idx s8:src* [Instruction]
`bv-f64-set!` *s8:dst s8:idx s8:src* [Instruction]
Store *src* into the bytevector *dst* at byte offset *idx*. Multibyte values are written using native endianness.

The *idx* value should be an unboxed unsigned 64-bit integer.

The *src* values are all unboxed, either as signed 64-bit integers, unsigned 64-bit integers, or IEEE double floating point numbers.

9.3.7.14 Арифметика Распакованных Целых Чисел

Guile поддерживает два типа распакованных целых чисел: беззнаковое 64-битное целое, и со знаковом 64-битное целое. Guile предпочитает целые числа без знака, в том смысле. что компилятор Guile поддерживает их лучше, а на виртуальной машине больше операций,которые с ними работают. Тем не менее, целые числа со знаком поддерживаются, по крайней мере, для того, чтобы позволить `bv-s64-ref` и соответствующим инструкциям избегать упаковки значений.

`scm->u64` *s12:dst s12:src* [Instruction]
Распаковка значения SCM из *src* в беззнаковое 64-битное целое, и поместить результат в *dst*. Если значение *src* не является точным целым числом в 64-битном диапазоне, сигнализировать об ошибке.

`u64->scm` *s12:dst s12:src* [Instruction]
Упаковать беззнаковое целое 64-битное число из *src* в заначение SCM и разметсит результат в *dst*. Результат может быть fixnum или bignum.

`load-u64` *s24:dst au32:high-bits au32:low-bits* [Instruction]
Загрузить 64-битрное значение сформированное объединением *high-bits* и *low-bits*, и записать результат в *dst*.

`scm->s64` *s12:dst s12:src* [Instruction]
`s64->scm` *s12:dst s12:src* [Instruction]
`load-s64` *s24:dst as32:high-bits as32:low-bits* [Instruction]
Как и `scm->u64`, `u64->scm`, и `load-u64`, но для 64 битных целых чисел со знаком.

Иногда компилятор может знать, что нам понадобиться токль подмножество бит в целом числе. В этом случае мы можем иногда распаковывать целое число, даже если оно может быть вне диапазона.

`scm->u64/truncate` *s12:dst s12:src* [Instruction]
Взять заначение SCM в *dst* и `logand` с помощью `(1- (ash 1 64))`. Поместить распакованный результат в *dst*.

`br-if-u64-=` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
`br-if-u64-<` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
`br-if-u64-<=` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
Если для распакованного целого 64 битное численного значения из *a* выполненos `=`, `<`, or `<=` по отношению к целому 64 битному числовому значению из *b*, сответственно, добавить смещение *offset* к ip(текущему значению указателя программ)

`br-if-u64-=-scm` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
`br-if-u64-<-scm` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
`br-if-u64-<=-scm` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
Если для распакованного беззнакового целого 64-битного значения из *a* выполнено `=`, `<`, или `<=` по отношению к значению SCM из *b*, соответственно, добавить смещение *offset* к ip(текущему значению указателя программы).

`uadd` *s8:dst s8:a s8:b* [Instruction]
`usub` *s8:dst s8:a s8:b* [Instruction]
`umul` *s8:dst s8:a s8:b* [Instruction]
Подобно `add`, `sub`, и `mul`, за искючением операндов в качестве распакованных беззнаковых 64-битных целых, и получают такой же результат. Результат будет молча обрезан до 64 бит.

`uadd/immediate` *s8:dst s8:a c8:b* [Instruction]
`usub/immediate` *s8:dst s8:a c8:b* [Instruction]
`umul/immediate` *s8:dst s8:a c8:b* [Instruction]
Подобно `uadd`, `usub`, и `umul`, за исключением второго операнда, это непосредственное 8-битное целое число.

`ulogand` *s8:dst s8:a s8:b* [Instruction]
`ulogior` *s8:dst s8:a s8:b* [Instruction]
`ulogxor` *s8:dst s8:a s8:b* [Instruction]
`ulogsub` *s8:dst s8:a s8:b* [Instruction]
Подобно `logand`, `logior`, `logxor`, и `logsub`, но работает с распакованным беззнаковым 64-битным целым числом.

`ulsh` *s8:dst s8:a s8:b* [Instruction]
Сдвиг беззнакового 64битного целого числа из *a* влево на *b* бит, результат также беззнаковое 64 битное целое. Обрезается до 64 бит и записывается в *dst* как распакованное значение. Только нижние 6 бит из *b* используются.

`ursh` *s8:dst s8:a s8:b* [Instruction]
Тоже что и `ulsh`, но сдвиг вправо.

`ulsh/immediate` *s8:dst s8:a c8:b* [Instruction]
`ursh/immediate` *s8:dst s8:a c8:b* [Instruction]
Тоже что и `ulsh` и `ursh`, но `b` рассматривается как непосредственное 8-битное беззнаковое целое.

9.3.7.15 Арифметика Распакованных чисел с плавающей точкой

`scm->f64` *s12:dst s12:src* [Instruction]
Распаковать значение SCM в *src* в формат IEEE двойной точности, разместив результат в *dst*. Если значение *src* не является вещественным числом, выдать сигнал ошибки.

`f64->scm` *s12:dst s12:src* [Instruction]
Упокавать двойное IEEE из *src* в значение SCM и разместить результат в *dst*.

`load-f64` *s24:dst au32:high-bits au32:low-bits* [Instruction]
Загрузить 64-битное значение, сформированное путем объединения старших бит(*high-bits*) и младших бит(*low-bits*) и записать результат в *dst*.

`fadd` *s8:dst s8:a s8:b* [Instruction]
`fsub` *s8:dst s8:a s8:b* [Instruction]

`fmul` *s8:dst s8:a s8:b* [Instruction]
`fdiv` *s8:dst s8:a s8:b* [Instruction]
Подобно `add`, `sub`, `div`, и `mul`, за исключением того, что операнды являются распакованными двойными числами с плавающей точкой формата IEEE и получают подобное число.

`br-if-f64-=` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
`br-if-f64-<` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
`br-if-f64-<=` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
`br-if-f64->` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
`br-if-f64->=` *s24:a x8:_ s24:b b1:invert x7:_ l24:offset* [Instruction]
Если для значения распакованного двойного в формате IEEE в *a* выполнено `=`, `<`, `<=`, `>`, or `>=` по отношению к значению распакованного двойного в формате IEEE из *b*, ТО соотвественно , добавить смещение *offset* к текущей ip (указателю выплняемой инструкции) pointer.

9.4 Компиляция в код Виртуальной Машины

Компиляторы! Само слово вызывает волнение и срах даже среди опытных практиков. Но компилятор - это просто программа: черезвычайно хакерская вещь. Эта секция направлена на то, чтобы описать компилятор Guile таким образом, чтобы заинтересованные Scheme хакеры могли чувствовать себя удобно при чтении и расширении его.

См. Раздел 6.18 [Read/Load/Eval/Compile], страница 404, если вы потерялись, и вы просто хотите знать, как скомпилировать ваш `.scm` файл.

9.4.1 Башня Компилятора

Guile компилятор довольно прост – это компиляторы *compilers*, если выражаться более точно. Guile определяет башню языков, начиная со Scheme и постепенно упрощая до языков которые напоминают набор инструкций ВМ (см. Раздел 9.3.7 [Instruction Set], страница 851).

Каждый язык знает, как скомпилировать до следующего, поэтому каждый шаг прост и понятен. Кроме того, этот набор языков не жестко закодирован в Guile, поэтому можно пользователю добавлять новые языки высокого уровня, новые проходы, или даже раличные цели компиляции.

Языки зарегистрированы в модуле, `(system base language)`:

```
(use-modules (system base language))
```

Они зарегистрированы в форме `define-language`.

`define-language` [*#:name*] [*#:title*] [*#:reader*] [*#:printer*] [Scheme Syntax]
[*#:parser=#f*] [*#:compilers='()*] [*#:decompilers='()*] [*#:evaluator=#f*]
[*#:joiner=#f*] [*#:for-humans?=#t*]
[*#:make-default-environment=make-fresh-user-module*]
Определяет язык.

Этот синтаксис определяет объект `<language>`, связаный с именем *name* в текущй среде. Кроме того, язык будет добавлен в глобальный набор языков. Например, это определения языка для Scheme:

```
(define-language scheme
  #:title "Scheme"
  #:reader      (lambda (port env) ...)
  #:compilers   `((tree-il . ,compile-tree-il))
  #:decompilers `((tree-il . ,decompile-tree-il))
  #:evaluator (lambda (x module) (primitive-eval x))
  #:printer write
  #:make-default-environment (lambda () ...))
```

Интересная вещь заключается в том, что языки определены таким образом, что они представляют единый интерфейс к циклу read-eval-print loop. Это позволяет пользователю изменять текущий язык REPL:

```
scheme@(guile-user)> ,language tree-il
Happy hacking with Tree Intermediate Language!  To switch back, type `,L scheme'.
tree-il@(guile-user)> ,L scheme
Happy hacking with Scheme!  To switch back, type `,L tree-il'.
scheme@(guile-user)>
```

Язык можно найти по имени, так как показано выше.

`lookup-language` *name* [Scheme Procedure]

Ищет язык по имени *name*, автоматически загружая его, если необходимо.

Язык автоматически загружается путем поиска пермеменной с именем *name* в модуле с именем `(language` *name* `spec)`.

Объект языка будет возвращен, или `#f` если не существует языка с таким именем.

Определение языков таким образом позволяет нам программно определять необходимые шаги для компиляции кода с одного языка на другой.

`lookup-compilation-order` *from to* [Scheme Procedure]

Рекурсивно пересекает множество языков, с которыми можно компилировать ища в глубину, и возвращает первый путь, который может преобразовывать *from* в *to*. Возвращает `#f` если путь не найден.

Эта функция запоминает свои результаты в кеше, который становиться недействительным последующими вызовами `define-language`, поэтому он должен быть доволно быстрым.

Существует понятие “текущего языка(current language)”, который содержиться в параметре `current-language`, определенном в ядре модуля `(guile)`. Этот язык обычно представляет собой Scheme, и может быть переустановлен пользователем. Интерфейс компиляции во время выполнения (см. Раздел 6.18 [Read/Load/Eval/Compile], страница 404) так же позволяет выбирать другие языки источника и цели.

Обычная башня языков при компиляции Scheme выглядит следующим образом:

- Scheme
- Tree Intermediate Language (Tree-IL)
- Continuation-Passing Style (CPS)

- Bytecode

Как обсуждалось ранее (см. Раздел 9.3.6 [Object File Format], страница 849), байт-код(bytecode) находиться в формате ELF, готовый к сериализации(последовательной записи) на диск. Но при компиляции Scheme во время выполнения, вам требуется значение Scheme: например, при компиляции процедуры. По этой причине, чтобы не сломать абстракцию, Guile определяет фальшивый язык в нижней части башни языков:

- Value

Компиляция в язык `value` загружает байткод в процедуру, превращает холодные байты в горячий код.

Возможно, эту странность можно объяснить примером: `compile-file` по умолчанию компилирует в байт-код(bytecode), потому что он создает объектный код который должен жить в бесплодном мире вне времени выполнения Guile; но фукция `compile` по умолчанию компилирут в `value`, послкольку этот результат возвращается в мир Guile.

Действительно, процесс компиляции ожет распространяться через эти разные миры на неопределенный срок, как показано в следующем quine(прогр. выдающая на выход точную копию своего исходного текста):

```
((lambda (x) ((compile x) x)) '(lambda (x) ((compile x) x)))
```

9.4.2 Компилятор Scheme

Задача компилятора Scheme состоит в том, чтобы развернуть все макросы и все конструкции Scheme до самых примитивных выражений. Определение “примитивного выражения” дается инвентаризацией конструкций, предоставляемых Tree-IL, целевом языке компилятора Scheme: вызовы процедур, условные выражения, лексические ссылки и т.д. Они описаны более подробно в следующем разделе.

Сложная и интересная вещь о компиляторе Scheme-to-Tree-IL заключается в том, что он полностью реализует расширение(развертку) макросов. Поскольку расширитель макросов должен пробегать по всему исходному коду для расширения макросов, он мог бы также провести анализ в это же время, непосредственно формируя выражения Tree-IL.

Поскольку этот компилятор фактически является расширителем макросов, он расширяем! Любой макрос, который записывает пользователь, становиться частью компилятора.

Макрорасширитель Scheme-to-Tree-IL может быть вызван с ипользованием общей процедуры компиляции `compile`:

```
(compile '(+ 1 2) #:from 'scheme #:to 'tree-il)
⇒
#<tree-il (call (toplevel +) (const 1) (const 2))>
```

`(compile` *foo* `#:from 'scheme #:to 'tree-il)` полностью эквивалентен вызову макрорасширителя `(macroexpand` *foo* `'c '(compile load eval))`. См. Раздел 6.10.9 [Macro Expansion], страница 297. `compile-tree-il`, это процедура используемая в `compile` для получения `'tree-il`, представляет собой обертку вокруг `macroexpand`,

чтобы его вызов соответствовал общей форме процедур компилятора в языковой башне Guile.

Процедуры компиляции принимают три аргумента: выражение, среду(окружение) и как опцию список ключевых слов. Они возвращают три значения: скомпилированное выражение, соответствующее среде целевого языка, и "среда продолжения (continuation environment)". Скомпилированное выражение и среда будут служить в качестве входных данных для компилятора языка следущего уровня. "Среда продолжения" может быть использована для компиляции другого выражения из того же языка источника внутри одного модуля.

Например, вы можете скомпилировать выражение, `(define-module (foo))`. Это приведет построению вражения и среды на языке Tree-IL. Но если вы компилируете второе(и последующие) выражения, вы хотели бы получить эффект от компиляции предыдущего выражения учитываемый во время компиляции, результат которой пользователь помещает в модуль `(foo)`. Это и есть назначение "среды продолжения(continuation environment)"; вы должны передать ее в качестве среды при компиляции последующих выражений.

Для Scheme, среда представляет собой модуль. По умолчанию, процедуры `compile` и `compile-file` компилирутся в новом модуле, так что связанные переменные и макросы введенные в выражении компилируются отдельно:

```
(eq? (current-module) (compile '(current-module)))
⇒ #f

(compile '(define hello 'world))
(defined? 'hello)
⇒ #f

(define / *)
(eq? (compile '/) /)
⇒ #f
```

Аналогично, измения `current-reader` fluid (см. Раздел 6.18.6 [Loading], страница 413) изолированы:

```
(compile '(fluid-set! current-reader (lambda args 'fail)))
(fluid-ref current-reader)
⇒ #f
```

Тем не менее, имея компилятор и *compilee*, общее пространство имен может быть достигнуто явной передачей `(current-module)` в качестве среды компиляции:

```
(define hello 'world)
(compile 'hello #:env (current-module))
⇒ world
```

9.4.3 Tree-IL

Tree Intermediate Language (Tree-IL) это структурированный промежуточный язык, который близок к выразительной способности Scheme. Это развернутая, предварительно проанализированная Scheme.

Tree-IL является "структурированным(structured)" в том смысле, что его представление основано на записях, а не S-выражениях. Это дает жесткость языку, которые обеспечивает компиляцию на язык нижнего уровня, требуя ограниченного набора преобразований. Например, тип Tree-IL `<const>` это запись с двумя полями `src` и `exp`. Экземпляры этого типа созданы с помощью `make-const`. Поля этого типа доступны через процедуры `const-src` и `const-exp`. Существует также предикат, `const?`. См. Раздел 6.6.17 [Records], страница 234, для дальнейшей информации о записях.

Все типы Tree-IL имеют слот `src`, который содержит информацию о местоположении источника для выражения. Эта информаия, если она присутствует, будет оставлена в скомпилированом объектном коде, позволяя трассировщику вызовов показывать информацию об исходном коде процедур. Формат `src` такой же, как и возвращаемый функцией Guile `source-properties`. См. Раздел 6.26.2 [Source Properties], страница 498, для получения дополнительной информации.

Хотя объекты Tree-IL представлены внутренне с использованием записей, существует также эквивалент S-выражений внешнего представления для каждого типа Tree-IL. Например, S-выражение представления выражения `#<const src: #f exp: 3>` был бы:

```
(const 3)
```

Пользоватеи могут запрограммировать этот формат непосредственно на REPL:

```
scheme@(guile-user)> ,language tree-il
Happy hacking with Tree Intermediate Language!  To switch back, type `,L scheme'.
tree-il@(guile-user)> (call (primitive +) (const 32) (const 10))
⇒ 42
```

Поля `src` остаются вне внешнего представления.

Можно создавать объекты Tree-IL из своих внешних представлений посредством вызова `parse-tree-il`, читателя для Tree-IL. Если подключить какую либо исходную информацию вводимого S-выражения, она будет распространена на результирующее выражение Tree-IL. Это вероятно, самый простой способ скомпилировать выражение в Tree-IL: просто сделайте соответствующее внешнее представление в формате S-выражения, и пусть `parse-tree-il` позаботиться об остальном.

`<void>` *src* [Scheme Variable]
`(void)` [External Representation]

Пустое выражение. На практике, оно эквивалентно выражению Scheme `(if #f #f)`.

`<const>` *src exp* [Scheme Variable]
`(const` ***exp***`)` [External Representation]

Константа.

`<primitive-ref>` *src name* [Scheme Variable]
`(primitive` ***name***`)` [External Representation]

Ссылка на "примитив(primitive)". Примитив - это процедура, которая при компиляции превращается в код операции(open-coded). Например, выражения `cons` обычно распознаются как примитивные, так что скомпилируются до одной инструкции.

Компиляция Tree-IL обычно начинается с прохода, кторый разрешает некоторые выражения `<module-ref>` и `<toplevel-ref>` в выражения `<primitive-ref>`. Фактический проход компиляции имеет особые случаи для интерпретации некотоырых вызов в определенные примитивы, например подобные `apply` или `cons`.

`<lexical-ref>` *src name gensym* [Scheme Variable]
`(lexical` *name gensym*`)` [External Representation]
Ссылка на лексически связанную переменную. Имя(*name*) это оригинальное имя переменной в исходной программе. *gensym* это уникальный идентификатор для этой переменной.

`<lexical-set>` *src name gensym exp* [Scheme Variable]
`(set!` (*lexical* *name gensym*`)` *exp*`)` [External Representation]
Выполняет лексическое связывание переменной.

`<module-ref>` *src mod name public?* [Scheme Variable]
`(@` *mod name*`)` [External Representation]
`(@@` *mod name*`)` [External Representation]
Ссылка на переменную в определенном(указанном) модуле. *mod* должна быть именем модуля, например: `(guile-user)`.

Если *public?* истинно, переменная с именем *name* будет видна в открытом(внешнем) интерфейся модуля *mod*, и сериализована с помощью `@`; иначеона будет рассматриваться как внутренняя связанная переменная и сериализироваться с помощью `@@`.

`<module-set>` *src mod name public? exp* [Scheme Variable]
`(set!` (@ *mod name*`)` *exp*`)` [External Representation]
`(set!` (@@ *mod name*`)` *exp*`)` [External Representation]
Установка переменной в указанном модуле.

`<toplevel-ref>` *src name* [Scheme Variable]
`(toplevel` *name*`)` [External Representation]
Ссылка на переменную(верхнего уровня) из текущей процедуры модуля.

`<toplevel-set>` *src name exp* [Scheme Variable]
`(set!` (*toplevel* *name*`)` *exp*`)` [External Representation]
Устанавливает(связывает с выржением выдающим значение) переменную(верхнего уровня) в текущей процедуре модуля.

`<toplevel-define>` *src name exp* [Scheme Variable]
`(define` *name exp*`)` [External Representation]
Определяет новую переменную верхнего уровня в текущей процедуре модуля.

`<conditional>` *src test then else* [Scheme Variable]
`(if` *test then else*`)` [External Representation]
Условие. Обратите внимание что *else* является обязательным.

`<call>` *src proc args* [Scheme Variable]
`(call` *proc . args*`)` [External Representation]
Вызов процедуры.

`<primcall>` *src name args* [Scheme Variable]
`(primcall` *name . args*`)` [External Representation]

Вызов примитива Эквивалент `(call (primitive` *name*`) . args)`. Эту конструкцию более удобно создавать и анализировать, чем `<call>`.

В рамках процесса компиляции экземпляры `(call (primitive` *name*`) . args)` преобразуются в primcalls.

`<seq>` *src head tail* [Scheme Variable]
`(seq` *head tail*`)` [External Representation]

Последовательность. Семантика заключается в том, что сначала вычисляется *head*(голова) и любые результирующие значения игнорируются. Затем вычисляется *tail*(хвост), указанный в позиции tail.

`<lambda>` *src meta body* [Scheme Variable]
`(lambda` *meta body*`)` [External Representation]

Замыкание. *meta* это ассоциированный список свойств(значения с именами) для процедуры. *body* это одиночное выражение Tree-IL типа `<lambda-case>`. Поскольку предложение `<lambda-case>` может быть альтернативной цепочкой предложений, это означает, что у Tree-IL's `<lambda>` есть выразительность языка Scheme `case-lambda`.

`<lambda-case>` *req opt rest kw inits gensyms body alternate* [Scheme Variable]
`(lambda-case ((`*req opt rest kw inits gensyms*`) body)` [External Representation]
`[`*alternate*`])`

Единичное предложение `case-lambda`. Лямбда(`lambda`) выражение на Scheme рассматривается как `case-lambda` с одним предложением.

req - это список необходимых процедуре аргуменов, как символов *opt* - это список необязательных аргументовis, или `#f` если нет необязательных аргументов. *rest* это имя остальных аргументов, или `#f`.

kw это список формы, `(`*allow-other-keys?* `(`*keyword name var*`) ...)`, где *keyword* это ключевое слово соответствующее аргументу с именем *name*, и соответствующим gensym значением *var*. *inits* это выражения tree-il соответствующие всем необязательным или ключевым аргументам, вычисляемых для связи переменных со значениями которые не предоставляются вызывающей процедурой. Каждое выражение *init* вычисляется в лексическом контексте ранее связанных переменных, с лева на право.

gensyms список gensyms соответствующий всем аргументам: сначала все необходимые аргументы, затем необязательные, если они есть затем все аргументы ключевые слова.

body это тело предложения. Если процедура вызывается с соответствующим числом аргументов, тело(*body*) вычисляется в хвостовой позиции. В противном случае , если есть *alternate*, оно должно быть выражением `<lambda-case>`, представляющим следующее предложение которое надо попробовать вычислить. Если нет *alternate*, вызывается сигнал ошибки wrong-number-of-arguments.

`<let>` *src names gensyms vals exp* [Scheme Variable]
`(let` ***names gensyms vals exp***`)` [External Representation]
Лексическое связывание, как и в Scheme `let`. *names* это оригинальные именя привязываемых имен, *gensyms* это gensyms соответствующие именам *names*, и *vals* это выражения Tree-IL для получения значений. *exp* это единичное выражение Tree-IL.

`<letrec>` *in-order? src names gensyms vals exp* [Scheme Variable]
`(letrec` ***names gensyms vals exp***`)` [External Representation]
`(letrec*` ***names gensyms vals exp***`)` [External Representation]
Версия `<let>` которая создает рекурсивные связи, подобые Scheme `letrec`, или `letrec*` если *in-order?* истинно.

`<prompt>` *escape-only? tag body handler* [Scheme Variable]
`(prompt` ***escape-only? tag body handler***`)` [External Representation]
Динамический запрос. Вставляет подсказку именуемую *tag*(являющуюся выражением), продолжающуюся выполнением *body*(также выражение). Если в этом запросе произойдет внезапное прерывание, управление передается процедуре *handler*(также выражение, которое должно быть процедурой). Первым аргументом процедуры handler будут захваченные продолжения, последующие все значения переданные в abort. Если *escape-only?* истинно, обработчк handler должен быть `<lambda>` с единственным выражением тела `<lambda-case>` без необязательных аргументов или аргументов ключевых слов, и не альтернативой, и чей первый аргумент не указан. См. Раздел 6.13.5 [Prompts], страница 323, для получения дополнительной информацией.

`<abort>` *tag args tail* [Scheme Variable]
`(abort` ***tag args tail***`)` [External Representation]
Отмена до ближайшего prompt с именем *tag*(являющеимся выражением). *args* должен быть списком выражений для передачи обработчику handler указанному в prompt, и *tail* должен быть выражением которое будет вычислять(обрабатывать) список дополнительных аргументов. abort сохраняет части продолжения, которые позже могут быть востановлены, что приведет к вычислению в выражении `<abort>` некоторого количества значений.

Существуют две конструкции Tree-IL, которые обычно не генерируются высокоуровневым компилятором, но вместо этого генерирутся во время оптимизации source-to-source и прохождения анализа, что и делает компилятор Tree-IL. Пользователи не должны генерировать эти выражения напрямую, если только не чувствуют себя очень умными, поскольку прохождение анализа по умолчанию будет генерировать их по мере необходимости.

`<let-values>` *src names gensyms exp body* [Scheme Variable]
`(let-values` ***names gensyms exp body***`)` [External Representation]
Подобно Scheme's `receive` – привязывает значения возвращаемые путем вычисления `exp` `lambda`-подобной связи описанной в *gensyms*. То есть, *gensyms* может быть неправильным списком.

`<let-values>` - это оптимизация вызова `<call>` для примитива, `call-with-values`.

`<fix>` *src names gensyms vals body* [Scheme Variable]
`(fix` *names gensyms vals body*`)` [External Representation]

Как и `<letrec>`, но только для *vals*, котрые не заданы `lambda` выражениями.

`fix` это оптимизация of `letrec` (and `let`).

Tree-IL - это удобная цель компиляции из исходных языков. Это может быть удобно как средство оптимизации, хотя CPS обычно лучше. Сила Tree-IL что он не фиксирует порядок вычислений, поэтому немного облегчает движение кода.

Выполнение оптимизационного прохода в Tree-IL включает:

- Open-coding (превращение toplevel-refs в primitive-refs, и вызов примитивов primcalls)
- Частичное вычислениен (включая вложения, copy propagation, и constant folding)

9.4.4 Continuation-Passing Style

Continuation-passing style (CPS) в Guile является основным промежуточным языком, преодолевающим разрыв между языками для людей и языками для машин. CPS дает имя каждой части программы: каждой контрольной точке и каждому промежуточному значению. Это создает отличную среду для рассуждения о программах, которая является основной задачей компилятора.

9.4.4.1 Введение в CPS

Рассмотрим следующее выражение Scheme:

```
(begin
  (display "The sum of 32 and 10 is: ")
  (display 42)
  (newline))
```

Выделим все подвыражения в этом выражении, анотируя их уникальнми метками.

```
(begin
  (display "The sum of 32 and 10 is: ")
  |k1      k2
  k0
  (display 42)
  |k4      k5
  k3
  (newline))
  |k7
  k6
```

Каждая из этих меток идентифицирует точку в программе. Одна метка может быть продолжением другой метки. Например, продолжение `k7` это `k6`. Это связано с тем, что после вычисления значения `newline`, выполняемое выражением помеченным `k7`, мы продолжим применять его в `k6`.

Какое выражение имеет `k0` в качестве продолжения? Это либо выражение помеченное как `k1` либо выражение помеченное как `k2`. Scheme не имеет фиксированного порядка вычисления аргументов. Хотя она гарантирует, что они будут вычисляться в определенном порядке. В отличи от общей Scheme, continuation-passing style(стиль

продолжене-передача) делает порядок вычисления явным. В Guile, этот выбор делают компиляторы языков более высокого-уровня.

Предположим, что порядок вычисления слева направо. В этом случае продолжением `k1` будет `k2`, и продолжением `k2` будет `k0`.

Для выбранного примера, мы готовы привести пример CPS в Scheme:

```
(lambda (ktail)
  (let ((k1 (lambda ()
              (let ((k2 (lambda (proc)
                          (let ((k0 (lambda (arg0)
                                      (proc k4 arg0))))
                            (k0 "The sum of 32 and 10 is: "))))
                (k2 display))))
        (k4 (lambda _
              (let ((k5 (lambda (proc)
                          (let ((k3 (lambda (arg0)
                                      (proc k7 arg0))))
                            (k3 42)))))
                (k5 display))))
        (k7 (lambda _
              (let ((k6 (lambda (proc)
                          (proc ktail))))
                (k6 newline)))))
    (k1))
```

Взрыв Священного кода, Бэтман! Что со всеми лямбдами? Действительно, CPS по своей природе гораздо более подробный, чем промежуточные языки “прямого-стиля” подобные Tree-IL. В тоже время, CPS проще, чем полная Scheme, потому что он делает вещи более явными.

В исходной программе, выражение помеченное`k0` является фактически контекстом. Любые возвращаемые значения игнорируются. В Scheme, этот факт не выражен явно. В CPS, мы видим это явно, отмечая что продолжение `k4`, принимает любое количество значений и игнорирует их. Сравнивая его с `k2`, которое принимает одно значение, мы можем сказать что `k1` является “значением” контекста. Аналогично `k6` находится в хвостовом(конечном) контексте относительно всего представленного выражения, поскольку его продолжение это хвостовое продолжение, `ktail`. CPS делает эти детали очевидными и дает им имена.

9.4.4.2 CPS в Guile

Язык CPS Guile состоит из продолжений(*continuations*). Продолжение это снабженная меткой точка программы. Если вы привыкли к традиционным компиляторам, думайте о продолжении как о тривиальном базовом блоке. Программм представляет собой “суп” из продолжений, представляемый в виде карты меток к продолжениям.

Подобно базовым блокам, каждое продолжение осущетсвляет только одну функцию. Некоторые продолжения являются специальными, такие как продолжения соответствующие точке входа в функцию, или продолжения представляющие хвост функции. Другие содержат термы(*term*). Терм содержит выражение (*expression*), которое вычисляет ноль или больше значений. Терм также описывает продолжение, которому он будет передавать свои значения. Некоторые термы, такие как условные ветви, могут продожиться в одно из некоторого числа продолжений.

Метки продолжений представляют собой малые целые числа. Это упрощает сортировку и группировку их в множества. Всякий раз, когда терм ссылается на продолжение, он делает это по имения, просто записывая метку продолжения. Метки продолжений уникальны среди множества меток в программе.

Переменные также именуются малыми целыми числами. Имена переменных уникальны среди множетсва переменных в программе.

Например, простое продолжение, которое получает два значения и объединяет их, такое как это, использует форму `match` из модуля `(ice-9 match)`:

```
(match cont
  (($ $kargs (x-name y-name) (x-var y-var)
      ($ $continue k src ($ $primcall '+ (x-var y-var))))
   (format #t "Add ~a and ~a and pass the result to label ~a"
           x-var y-var k)))
```

Здесь мы видим наиболее распространенный вид продолжения,`$kargs`, который привязывает некоторое число значений к переменным, а завем вычисляет терм.

`$kargs` *names vars term* [CPS Continuation]

Связь входных значений с переменными *vars*, с оригинальными именами *names*, и вычисление *term*.

Имена(*names*) `$kargs` предназначены только для отладки и в конечном итоге будут оставлены в объектном файле для использования отладчиком.

Выражение *term* в `$kargs` всегда продолжение(`$continue`), которое вычисляется как выражение и продолжает продолжение.

`$continue` *k src exp* [CPS Term]

Вычисление выражения *exp* и передача полученных значений (если они есть) в продолжение с меткой *k*. Исходная информация связанная с выражением, может быть найдена в *src*, который является ассоциативным списком(alist), как в `source-properties` или быть `#f` если нет связанного источника.

Существует несколько видов выражений. Выше вы видите пример `$primcall`.

`$primcall` *name args* [CPS Expression]

Perform the primitive operation identified by `name`, a well-known symbol, passing it the arguments *args*, and pass all resulting values to the continuation. The set of available primitives includes all primitives known to Tree-IL and then some more; see the source code for details.

Переменные, которые использует `$primcall`, или любое другое выражение, должны быть определены перед вычислением выражения. Эквивалентным способом сказать это является то, что предшествующее `$kargs` продолжение, которое связывает переменные используемые выражением должны доминировать*dominate* над продолжением, которое использует выражение: определения преобладают в использовании. Это условие тривиально удовлетворены в примере выше, но в целом для определения набора переменных, которые находятся в “области” для доступа данного термина, вам нужно провести анализ потока, чтобы увидеть, какие продолжения доминируют термин. Переменные, которые входят в область охвата, - это те переменные, которые определены в продолжениях, которые доминируют над термином.

Вот список видов выражений в языке CPS Guile, кроме того `$primcall` который уже описан. Напомним, что все выражения завернуты в `$continue`, который указывает их продолжение.

`$const` *val* [CPS Expression]
: Продолжение с постоянным значением *val*.

`$prim` *name* [CPS Expression]
: Продолжение процедуры, которая реализует примитивную операцию названную *name*.

`$call` *proc args* [CPS Expression]
: Вызвать *proc* с аргументами *args*, и передать все значения в продолжение. *proc* и элементы списка *args* должны быть именами переменных. Продолжение идентифицируемое термином *k* должно быть `$kreceive` или экземпляром `$ktail`.

`$values` *args* [CPS Expression]
: Передача значений, указанных в списке *args* в продолжение.

`$branch` *kt exp* [CPS Expression]
: Вычисление выражения ветвления *exp*, и продолжение *kt* с нулевым значением если тест возвращает истину. В противном случае продолжается продолжение с именем `$continue` во внешнем терме.

  Только определенные выражения действительны в *$branch*. Компиляция `$branch` исключает выделение пространства для тестовой переменной, поэтому выражение должно быть вычислено без временного значения. На практике это условие верно для `$primcall` для `null?`, `=`, и аналогичных примитивов, которые имеют соответствующие операции ВМ `br-if-`*`foo`*; см исходный код для полной информации. Если есть сомнения, привяжите тестовое выражение к переменной и ветвь в выражение `$values`, ссылающееся на эту переменную. Оптимизатор должен вставить ссылку, если это возможно.

`$prompt` *escape? tag handler* [CPS Expression]
: Помещает приглашение в стек идентифицируемое именем переменной *tag*, выйти из которого можно только если *escape?* равно истине, и продлжиться с нулевым значением. Если тело выполниния программы прервано этим запросом, управление продолжиться в обработчике с меткой *handler*, который должен быть продолжением `$kreceive`. Само приглашение выдается позже вызовом primcalls `pop-prompt`.

Существуют два подязыка CPS, высокоуровневый *higher-order CPS* и первого порядка *first-order CPS*. Различие их в том, что в высокоуровневом CPS существуют выражения `$fun` и `$rec`, которые связывают функции или взаимно-рекурсивные функции в неявной области их использования. Трансформация Guile высокоуровневого CPS в CPS первого порядка путем *closure conversion*, которое выбирает представление для всех замыканий и которое организует доступ к свободным переменным через неявный закрытый параметр, который передается каждому вызову функции.

`$fun` *body* [CPS Expression]
Продолжение с процедурой. Имя *body* это точка входа функции, которая должна быть `$kfun`. Этот вид выражения действителен только в высокоуровневом CPS, которым является язык CPS до вызова closure conversion.

`$rec` *names vars funs* [CPS Expression]
Продолжение с набором взаимно-рекурсивных процедур, обозначеных именами *names*, *vars*, и *funs*. *names* это список символов, *vars* это список имен переменных (уникальные целые числа), и *funs* это список значений `$fun`. Заметим, что продолжение `$kargs` также должно определять связки *names*/*vars*.

Проход contification попытается преобразовать функции, объявленные в `$rec` в локальные продолжения. Любые оставшиеся экземпляры `$fun` позже удаляются проходом closure conversion. По умолчанию, замыкание представлено как объект построенный выражением `$closure`.

`$closure` *label nfree* [CPS Expression]
Создает замыкание, которое присоединятся к коду в продолжении с именем *label* с пространством имен свободных переменных *nfree*. Переменные будут инициализированные позже через вызов primcalls `free-set!`. Этот вид выражения является частью CPS первого порядка.

Если для замыкания можно доказать что оно никогда не выходит за пределы своей области, тогда может быть выбрано другое более легковесное представление. Кроме того, если известны все точки вызова, closure conversion принудительно преобразует вызовы в низкоуровневые, опустив `$call` до `$callk`.

`$callk` *label proc args* [CPS Expression]
Как и `$call`, но для случая, когда цель вызова, известно находитьсяв томже компилируемом модуле. *label* должно означать продолжение `$kfun` в программе. В этом случае *proc* является просто дополнительным аргуменом, поскольку он не используется для определения цели вызова во время выполнения.

На этом этапе мы описали термины, выражения и наиболее распространенные виды продолжений, `$kargs`. `$kargs` используется, когда предшествующие продолжения могут передать значения, в которых требуется их продолжение. Например, если `$kargs` продолжение *k* связывает переменную *v*, и компилятор решает выделить *v* слот 6, все предшественники *k* должны помещать значение для *v* в слот 6 перед переходом к *k*. Одна ситуация, в которой это не возможно это получение значения из вызова функции. Guile имеет соглашение о вызовах функций, которое в настоящее время помещает возвращаемые значения в стек. Продолжение вызова должно проверить, что количество значений возвращаемых функцией, соответствует ожидаемому числу значений и затем необходимо перетасовать или собрать эти значения для именованных переменных. `$kreceive` обозначает этот вид продолжения.

`$kreceive` *arity k* [CPS Continuation]
Получает значения в стеке. Разбирает их в соответствии с арностью(*arity*), а затем приступает к разбору значений `$kargs` продолжения помеченного *k*. В качестве ограничения, характерного для `$kreceive`, арность(*arity*) может содержать только необходимые и остальные аргументы.

`$arity` это вспомогательная структура данных, используемая `$kreceive`, а также `$kclause`, описанная ниже.

`$arity` *req opt rest kw allow-other-keys?* [CPS Data]

Тип данных, обявляющий арность(arity). *req* и *opt* - списки имен источников требуемых и необязательных аргументов, соответственно. *rest* являетс либо либо исходным именем переменной rest, либо `#f`, если эта арность(arity) не принимает дополнительные значения. *kw* - это список форм вида `((keyword name var) ...)`, описывающий аргументы ключевого слова. *allow-other-keys?* установлено в истину если другие ключевые слова разрешены, иначе ложь.

Обратите внимание, что все эти имена, за исключением *var* в списке *kw*, являются исходными именами, а не уникальными именами переменных.

Кроме того, сущетсвтуют три вида продолжений, которые используются только в внутри функций.

`$kfun` *src meta self tail clauses* [CPS Continuation]

Объявляет точку входа в функцию. *src* является исходной информацией для объявления процедуры, а *meta* это ассоциативный список(alist) метаданных, как описано выше в Tree-IL's `<lambda>`. *self* это переменная связанная с названной процедурой и которая может испольоваться для ссылки на саму себя. *tail* это метка `$ktail` для данной функции, соответствующая продолжению реализующему хвост. *clause* это метка первого `$kclause` для первого предложения `case-lambda` в функции или иначе `#f`.

`$ktail` [CPS Continuation]

Хвост(окончание) продолжения.

`$kclause` *arity cont alternate* [CPS Continuation]

Предложение функции с заданной арностью(arity). Применение функции с совместимым набором фактичеких аргументов будет продолжаться до продолжения с меткой *cont*, а экземпляр `$kargs` пердставляет тело предложения. Если аргументы не совместимы, переходит к альтернативе (*alternate*), которая представляет собой `$kclause` для следующего предложения, или `#f`, если нет следующего предложения.

9.4.4.3 Построение CPS

В отличии от Tree-IL, язык CPS построен, чтобы быть сконструированным и деконструированным с абстратными макросами, а не через процедурные конструкторы или аксессоры, или вместо вычисления S-выражений.

Деконструкция и сопоставление обрабатываются надлежащим образом формой `match` из модуля `(ice-9 match)`. См. Раздел 7.7 [Pattern Matching], страница 720. Конструкция обрабатывается набором взаимосвязанных макросов: `build-term`, `build-cont`, и `build-exp`.

В следующих определениях интерфейсов рассматриваются термы(`term`) и выражения(`exp`), которые будут построены `build-term` или `build-exp`, соответственно. Рассмотрим любое другое имя, которое будет вычисленно как выражение Scheme. Многие из этих форм распознают `unquote` в некоторых контекстах, чтобы объединять с ранее построенным значением; см. спецификации ниже для получения полной информации.

`build-term` *,val* [Scheme Syntax]
`build-term` (*$continue k src exp*) [Scheme Syntax]
`build-exp` *,val* [Scheme Syntax]
`build-exp` (*$const val*) [Scheme Syntax]
`build-exp` (*$prim name*) [Scheme Syntax]
`build-exp` (*$branch kt exp*) [Scheme Syntax]
`build-exp` (*$fun kentry*) [Scheme Syntax]
`build-exp` (*$rec names syms funs*) [Scheme Syntax]
`build-exp` (*$closure k nfree*) [Scheme Syntax]
`build-exp` (*$call proc* (*arg ...*)) [Scheme Syntax]
`build-exp` (*$call proc args*) [Scheme Syntax]
`build-exp` (*$callk k proc* (*arg ...*)) [Scheme Syntax]
`build-exp` (*$callk k proc args*) [Scheme Syntax]
`build-exp` (*$primcall name* (*arg ...*)) [Scheme Syntax]
`build-exp` (*$primcall name args*) [Scheme Syntax]
`build-exp` (*$values* (*arg ...*)) [Scheme Syntax]
`build-exp` (*$values args*) [Scheme Syntax]
`build-exp` (*$prompt escape? tag handler*) [Scheme Syntax]
`build-cont` *,val* [Scheme Syntax]
`build-cont` (*$kargs* (*name ...*) (*sym ...*) *term*) [Scheme Syntax]
`build-cont` (*$kargs names syms term*) [Scheme Syntax]
`build-cont` (*$kreceive req rest kargs*) [Scheme Syntax]
`build-cont` (*$kfun src meta self ktail kclause*) [Scheme Syntax]
`build-cont` (*$kclause ,arity kbody kalt*) [Scheme Syntax]
`build-cont` (*$kclause* (*req opt rest kw aok?*) *kbody*) [Scheme Syntax]
Создают CPS термы(term), выражения, или продолжения(continuation).

Есть еще несколько различных интерфейсов

`make-arity` *req opt rest kw allow-other-keywords?* [Scheme Procedure]
Процедурный конструктор для арных(`$arity`) объектов.

`rewrite-term` *val* (*pat term*) ... [Scheme Syntax]
`rewrite-exp` *val* (*pat exp*) ... [Scheme Syntax]
`rewrite-cont` *val* (*pat cont*) ... [Scheme Syntax]
Сопоставляет *val* с серией шаблонов *pat...*, используя `match`. Тело соответствия должно быть шаблоном в синтаксисе `build-term`, `build-exp`, или `build-cont`, соответственно.

9.4.4.4 Суп CPS

Мы описываем программы на языке CPS Guile как своего рода “суп” потому что все продолжения в программе смешиваются в один и тот же “банк”, так сказать, без явных указаний относительно того, какая есть функции или области действия. Программа в CPS это карта из помеченных продолжений к значениям продолжений. Как обсуждалось во введении, метка продолжения это целое число. Никакая метка не может быть отритцательной.

В качестве условного обозначения, метка 0 должна отображать продолжение `$kfun` на точку входа в программу, которая должна быть функцией без аргументов. Тело

функции состоит из помеченных продолжений, доступных из точки входа в функцию. Программа может ссылаться на другие функции, либо через `$fun` и `$rec` в CPS высокого порядка, либо через `$closure` и `$callk` CPS первого порядка. Программа логически содержит все продолжения всех функций достижимых из входной функции. Проход компилятора может оставить недостижимые продолжения в программе; последующие проходы компилятора должны гарантировать, что их преобразвания и анализ учитывают только достижимые продолжения. Это нормально, хотя если трансформация пробегает все продолжения, если включение недостижимых продолжений не влияет на преобразования живых продолжений.

Сам "суп" реализован как *intmap*, функциональный массив, специализированный для сопоставления целых ключей. Intmaps связывает целые со значениями любого типа. В настоящее время intmaps частная структура данных, используемая только фазой компилятора CPS. Чтобы работать с intmaps, загрузиет модуль `(language cps intmap)`:

```
(use-modules (language cps intmap))
```

Intmaps это функциональная структура данных, поэтому у нее нет конструктора как такового: можно просто начать с пустой intmap и добавлять в нее записи.

```
(intmap? empty-intmap) ⇒ #t
(define x (intmap-add empty-intmap 42 "hi"))
(intmap? x) ⇒ #t
(intmap-ref x 42) ⇒ "hi"
(intmap-ref x 43) ⇒ error: 43 not present
(intmap-ref x 43 (lambda (k) "yo!")) ⇒ "yo"
(intmap-add x 42 "hej") ⇒ error: 42 already present
```

`intmap-ref` и `intmap-add` являются ядром интерфейса intmap. Есть также `intmap-replace`, которая заменяет значение связанное с данным ключом, требует что бы ключ уже присутствовал в intmap, и `intmap-remove`, который удаляет ключ из intmap.

Intmaps имеет древовидную структуру, которая хорошо подходит для операций с множествами такими как объединение и пересечение, поэтому существуют также двоичные процедуры `intmap-union` и `intmap-intersect`. Если результат эквивалентен любому аргументу, этот аргумент возвращается как есть; таким образом, можно определить, вызвала ли заданная операция новый езультат просто выполнив проверку с помощью `eq?`. Это делает полезным intmaps при вычислении фиксированных точек(fixed points).

Если ключ присутствует в обоих intmaps и связанные значения не совпадают в смысле `eq?`, результирующее значение определяется процедурой "meet", которая является необязательным последним аргументом `intmap-union`, `intmap-intersect`, а также `intmap-add`, `intmap-replace`, и аналогичных функций. Процедура meet будет вызываться с двумя значениями и должна возвращать пересекающееся или объединенное значение определяемое домен-специвфичным способом. Если нет соответствующей процедуры, имеющаяся по умолчанию процедура meet вызывает ошибку.

Чтобы пройти по набору значений в intmap, есть процедуры `intmap-next` и `intmap-prev`. Например, если intmap *x* имеет одно отображение записи 42 для нескольких значений, мы бы получили:

```
(intmap-next x) ⇒ 42
(intmap-next x 0) ⇒ 42
(intmap-next x 42) ⇒ 42
(intmap-next x 43) ⇒ #f
(intmap-prev x) ⇒ 42
(intmap-prev x 42) ⇒ 42
(intmap-prev x 41) ⇒ #f
```

Существует также процедура `intmap-fold`, которая складывает(folds) по ключам и значениям в intmap от минимального до максимального значения, и `intmap-fold-right` делающая тоже самое в противоположном направлении. Эти процедуры могут принимать 3 начальных значений. Количество значений, которые складываются процедурой возвращаются как количество начальных значений.

```
(define q (intmap-add (intmap-add empty-intmap 1 2) 3 4))
(intmap-fold acons q '()) ⇒ ((3 . 4) (1 . 2))
(intmap-fold-right acons q '()) ⇒ ((1 . 2) (3 . 4))
```

Когда запись в intmap обновляется (удаляется, добавляется или изменяется), новый intmap создает разделяемую структуру с исходным intmap. Эта операция гарантирует, что результат существующих вычислений не зависит от будущих вычислений: никаких изменений невидимых для пользовательского кода. Это отличное свойство в структуре данных компилятора, поскольку оно позволяет удерживать копию программы перед преобразованием и использовать ее, пока мы строим пост-преобразование программы. Обновление intmap это операция порядка O(log n) от размера intmap.

Однако, затраты на размещение O(log n) иногда слишком велики, особенно в тех случаях, когда мы знаем, что мы можем просто обновить intmap на месте. В качестве примера, скажем у нас есть intmap отображающий целые числа от 1 до 100 в целые числа от 42 до 141. Предположим, что мы хотим преобразовать это отображение, добавив 1 к каждому значению. Уже существует эффективная процедура `intmap-map` в модуле `(language cps utils)`, но если бы мы не знали об этом, мы могли бы делать так:

```
(define (intmap-increment map)
  (let lp ((k 0) (map map))
    (let ((k (intmap-next map k)))
      (if k
          (let ((v (intmap-ref map k)))
            (lp (1+ k) (intmap-replace map k (1+ v))))
          map))))
```

Обратите внимание, что промежуточные значения, созданные методом `intmap-replace`, полностью невидимы в программе – нужен только последний результат значения `intmap-replace`. rest может совместно использовать состояние с последним, чтобы мы могли обновить его. Guile позволяет этот вид интерфейса через *transient intmaps*, вдохновленный переходным интерфейсом Замыканий. (`http://clojure.org/transients`).

Процедуры `intmap-add!` и `intmap-replace!` изменяющие входные данные возвращают переходный intmaps. Если одна из этих процедур изменяющих входные данные вызывается с постоянным intmap, создается новый переходный intmap is. Это опе-

рация O(1). Во всех других отношениях интерфейс подобен их постоянной копии, `intmap-add` и `intmap-replace`. Если процедура меняющая входные данные вызывается с переходным intmap, intmap изменяется на месте и возвращается одно и тоже значение(значение изменных входных данных).

Если на временном intmap вызывается сохраняющая входные данные операция, такая как `intmap-add`, изменяемая субструктура затем помечается как постояная, и `intmap-add` запускается на новой постоянной структуре intmap совместного использования, не имеющей пометки переходного состояния. Изменение перходного состояния приводит к необходимому копированию, чтобы обеспечить это изменение, но если часть его подструктуры уже "принадлежит" им, копирование этой структуры больше требуется.

Мы можем использовать переходные initmap чтобы сделать `intmap-increment` более эффективным. Они изменяют элементы имеющие пометки **следующим образом**.

```
(define (intmap-increment map)
  (let lp ((k 0) (map map))
    (let ((k (intmap-next map k)))
      (if k
          (let ((v (intmap-ref map k)))
            (lp (1+ k) (intmap-replace! map k (1+ v))))
          (persistent-intmap map)))))
```

Обязательно пометьте результат как постоянный, используя процедуру `persistent-intmap`, чтобы предотвратить утечку памяти в другой части программы. Для дополнительной паранои, вы можете вызывать `persistent-intmap` для входной initmap, чтобы убедиться, что если она уже была временной, изменения в теле `intmap-increment` не повлияют на входящее значение.

Таким образом, программы в CPS представляют собой intmaps, значения которого являются продолжениями. См. исходный код `(language cps utils)` для ряда полезных возможностей для работы со значениями CPS.

9.4.4.5 Компиляция CPS

Компиляция CPS в Guile состоит из трех этапов: преобразование, оптимзация, и генерация кода.

Преобразование CPS это процесс взятия высокоуровнего языка и компиляция его в CPS. Исходные языки могут делать это напрямую, или они могут преобразовываться в Tree-IL (что, вероятно проще) и позже Tree-IL преобразуется в CPS. Переход через Tree-IL имеет преимущество выполнения оптимзационного этапа, как частичного выполнения. Кроме того, компилятор из Tree-IL в CPS обрабатывает преобразование присваивания, в котором назначаются локальные переменные (в Tree-IL, локальные переменные, которые являются `<lexical-set>`) преобразуются в значения указанные в куче. См. Раздел 9.3.4 [Variables and the VM], страница 847.

После преобразования CPS, Guile запускает некоторые проходы оптимизации над CPS. Большинство оптимизаций в Guile делается на языке CPS. Одним из основных исключений является частичное выполнение, которое по историческим причинам сделано на Tree-IL.

Основная оптимизация, выполняемая на CPS это contification, в котором функции, которые всегда вызываются с тем же продолжением, включаются непосредственно в тело функции. Это открывает пространство для большей оптимизации и превращает вызовы процедур в `goto`. Оно может также делать петли из гнезд рекурсивных функций. Guile также уничтожает мертвый код, устраняет код общегоподвыражения, пилинг(peeling) цикла и инвариантного кода, а также диапазон и тип вывода.

Остальная часть проходов оптимизации это очистка и канонизация (canonicalizations). CPS заполняет разрыв между языками высокого уровня и байт-кодом низкого уровня, что позволяет выразить процесс компиляции как трансформацию исходного кода в исходный код. Таков случай для преобразования замыкания, в котором ссылки на переменные, свободные в функции, преобразуютя в ссылки замыкания, и в которых функция преобразуется в замыкание. Есть еще несколько проходов, чтобы гарантировать, что единственные primcalls, оставшиеся в термах, это те, которые имеют соответствующие инструкции на виртуальной машине, и что их продолжения ожидают правильное количество значений.

Наконец, в завершении компилятор CPS выдает байт-код для каждой функции, одной за другой. Для этого он определяет набор живых переменных во всех точках функции. Използуя эту информацию, он выделяет слоты в стеке для каждой переменной, так что переменная может жить в одном слоте все время своей жизни, без перетасовки(перемещения). (Конечно, переменные с непересекающимся временем жизни могут использовать совместно слот) Наконец в завершении, генерируется код, как правило, только для одной виртуальной машины, для каждого продолжения в функции.

9.4.5 Байт Код

Как упоминалось ранее, Guile компилирует весь код в байт-код и что байт код содержиться в ELF образе. См. Раздел 9.3.6 [Object File Format], страница 849, для дополнительной информации о использовании в Guile формата ELF.

Чтобы создать образ байт-кода, Guile предоставляет ассемблер и компоновщик.

Ассемблер, определяется в модуле `(system vm assembler)`, он имеет относительно прямолинейную струкуру внешнего императивного интерфейса. Он предоставляет функцию `make-assembler` для создания экземпляра ассемблера и набор процедур `emit-`*`inst`* для генерации инструкций каждого типа.

Процедуры `emit-`*`inst`* фактически генерируются во время компиляции из машиночитаемого описания ВМ. За некоторыми исключениями для определенных типов операндов каждый опперанд генерирующей процедуры соответствует операнду соответствующей инструкции.

Рассмотрим `vector-length`, из см. Раздел 9.3.7.9 [Miscellaneous Instructions], страница 863. Она задокументирована как:

`vector-length` *u12:dst u12:src* [Instruction]

Поэтому генерирующая процедура имеет вид:

`emit-vector-length` *asm dst src* [Scheme Procedure]

Все генерирующие процедуры получают ассемблер в качестве первого аргумента и не возвращают никакого полезного значения.

Типы аргументов зависят от типов операндов. См. Раздел 9.3.7 [Instruction Set], страница 851. Большинство из них представляют собой целые числа в ограниченном диапазоне, хотя метки обычно выражаются как непрозрачные символы.

Есть также несколько макрокоманд.

`emit-label` *asm label* [Scheme Procedure]
Определяет метку в текущей программной точке.

`emit-source` *asm source* [Scheme Procedure]
Связывает исходный код(*source*) с текущей точкой программы.

`emit-cache-current-module!` *asm module scope* [Scheme Procedure]
`emit-cached-toplevel-box` *asm dst scope sym bound?* [Scheme Procedure]
`emit-cached-module-box` *asm dst module-name sym public? bound?* [Scheme Procedure]
Макро инструкции реализующие кэширование переменных верхнего уровня. Первая принимает текущий модуль, слоте *module*, и связывает его с указанным местоположением кэша в переменной *scope*. Вторая принимает *scope*, и разрешает(находит) переменную. См. Раздел 9.3.7.2 [Top-Level Environment Instructions], страница 854. Последней не требуется модуль кэширования, вместо этого она получает имя модуля напрямую.

`emit-load-constant` *asm dst constant* [Scheme Procedure]
Загружает исходные константы *constant* Scheme в *dst*.

`emit-begin-program` *asm label properties* [Scheme Procedure]
`emit-end-program` *asm* [Scheme Procedure]
Отмечает границы процедуры, с указанной меткой *label* и мета данными *properties*.

`emit-load-static-procedure` *asm dst label* [Scheme Procedure]
Загружает процедуру с указанной меткой *label* в локальную переменную *dst*. Данная макро-инструкция должна использоваться только для процедур без свободных переменных — т.е процдур не являющихся замыканиями.

`emit-begin-standard-arity` *asm req nlocals alternate* [Scheme Procedure]
`emit-begin-opt-arity` *asm req opt rest nlocals alternate* [Scheme Procedure]
`emit-begin-kw-arity` *asm req opt rest kw-indices allow-other-keys? nlocals alternate* [Scheme Procedure]
`emit-end-arity` *asm* [Scheme Procedure]
Разделительные предложения процедуры.

`emit-br-if-symbol` *asm slot invert? label* [Scheme Procedure]
`emit-br-if-variable` *asm slot invert? label* [Scheme Procedure]
`emit-br-if-vector` *asm slot invert? label* [Scheme Procedure]
`emit-br-if-string` *asm slot invert? label* [Scheme Procedure]
`emit-br-if-bytevector` *asm slot invert? label* [Scheme Procedure]
`emit-br-if-bitvector` *asm slot invert? label* [Scheme Procedure]
TC7-специфичные инструкции для тестирования и ветвления. TC7 это 7-битный код который является частью типов объектов кучи. См. Раздел 9.2.5 [The SCM

Type in Guile], страница 839. Также см, См. Раздел 9.3.7.6 [Branch Instructions], страница 859.

Линкер - сложный зверь. Хакеры, заинтересованные в том, чтобы узнать как он работает, читайте серию статей Ian Lance Taylor о линкерах. Поиск в интернете должен найти их без труда. С точки зрения пользователя, есть только один признак для контроля: будет ли получен результирующий образ для записи в файл или нет. Если пользователь передает `#:to-file? #t` как опцию компилятора (см. Раздел 9.4.2 [The Scheme Compiler], страница 872), компоновщик будет выравнивать результирующие сегменты на границах страниц, в противном случае - нет.

`link-assembly` *asm #:page-aligned?=#t* [Scheme Procedure]
Связывает ELF образ, и возвращает байт-вектор. Если *page-aligned?* установлено в истину, Guile будет выравнивать сегменты с разными разрешениями на границах размера страниц, чтобы максимизировать совместное разделение кода между различными процессами. В противном случае заполнение минимизируется, для минимизации использования адресного пространства.

Для записи на диск, просто используйте `put-bytevector` из `(ice-9 binary-ports)`.

Компиляция объектного кода на фальшивый язык, `value`, выполняется путем загрузки объектного кода(objcode) в программу, а затем выполняет его относительно среды компиляции. Обычно среда передается через компилятор прозрачно, но пользователи могут также указать среду компиляции как модуль. Процедуры загрузки образов можно найти в модуле `(system vm loader)`:

```
(use-modules (system vm loader))
```

`load-thunk-from-file` *file* [Scheme Variable]
`scm_load_thunk_from_file` (*file*) [C Function]
Загружает объектный код из файла с именем *file*. Файл будет отображаться в паметь через функцию `mmap`, так что это очень быстрая операция.

`load-thunk-from-memory` *bv* [Scheme Variable]
`scm_load_thunk_from_memory` (*bv*) [C Function]
Загрузка объектного кода из байт-вектора. Данные будут скопированы из байтового вектора в порядке обечивающем правильное выравнивание встроенных значений Scheme.

Кроме того, есть процедуры для поиска ELF образа для заданного указателя, или списка всех отображенных(загруженных) ELF образов:

`find-mapped-elf-image` *ptr* [Scheme Variable]
Учитывая целочисленное значение *ptr*, ищет и возвращает ELF образ который содержит указатель, как байтовый вектор. Если изобржение не найдено, возвращает `#f`. Эта процедура в основном используется отладчиком и другими интроспективными инструментами.

`all-mapped-elf-images` [Scheme Variable]
Возвращает все отображенные(загруженные) ELF образы, как список байт-векторов.

9.4.6 Написание Новых Высоко-Уровневых Языков.

Чтобы интегрировать новый язык *lang* в систему компилятора Guile, нужно создать модуль (`language` *`lang`* `spec`) содержащий определение языка и ссылающийся на анализатор, компилятор и другие процедуры обрабатывающие его. Иерархия модулей в (`language brainfuck`) определяет очень базовую реализацию Brainfuck предназначенную для того чтобы служить легко понятным примером о том как это сделать. См. например `http://en.wikipedia.org/wiki/Brainfuck` для дополнительной информации о языке Brainfuck.

9.4.7 Расширение Компилятора

В этот момент мы отходим от безличного тона остальной части руководства. Признайте это: если вы внимательно изучили руководство по внутреннему компилятору, вы являетесь наркоманом! Возможно курс в вашем университете оставил вас без внимания, или, возможно, вы всегда испытывали желание взломать святость компьютерных наук: компилятор! Хорошо, что вы в хорошей компании и в хорошем положении. Компилятор Guile нуждается в вашей помощи.

Существует множество возможностей для улучшения компилятора Guile. Вероятно, наиболее важное улучшение, по скорости, будет представлять собой некоторую форму нативной компиляции, как просто во время выполнения, так и предварительной. Это можно сделать разными способами. Вероятно самая простая стратегия была бы расширить скомпилированную процедуру структурой включающей указатель на вектор исполняемого кода, и скомпилировать из байт кода исполняемый код во время выполнения, после того как процедура вызывается определенное количество раз.

Название игры сбор урожая низкорослых фруктов на основе профилирования программы, запуск программы представляющих интерес под профилировщиком системного уровня определение того, какие улучшения даст самый удачный buck. Это действительно доходит до того, что нативная компиляция - следующий шаг.

Компилятору также нужна помощь на верхнем уровне, улучшая Scheme которая, как известно, также понимает R6RS, и добавляя новые компиляторы высокого уровня. У нас есть JavaScript и Emacs Lisp в основном завершены, Lua тоже бы неплохо, да любой язык поразивший ваше воображение, тоже будет приветствоваться.

Компиляторы предназначены для взлома, а не для восхищения или жалоб. Доберитесь до него!

Приложение A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ascii without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

Этот индекс содержит понятия, ключевые слова и не-Scheme имена для некоторых функций, чтобы упростить поиск нужных разделов

!

#

(

.

/

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Procedure Index

Этот алфавитный список всех процедур и макросов в Guile. Он также включает макросы Autoconf Guile.

При поиске конкретной процедуры, пожалуйста смотрие ее Scheme имя, а также ее Си имя. Имя Си может быть построено из имени Scheme простым преобразованием описанным в разделе См. Раздел 6.1 [API Overview], страница 107.

(Предметный указатель не существует)

Variable Index

Этот алфавитный список всех важных переменных и констатн в Guile.

При поиске определенной перменной или константы, пожалуйста посмотрите ее имя Scheme, а также ее имя в Си. Имя Си может быть получено по имени Scheme простым преобразованием, описанным в разделе См. Раздел 6.1 [API Overview], страница 107.

(Предметный указатель не существует)

Type Index

Это алфавитный укзатель всех важных типов данных, определенных в Справочном Руководстве Guile.

(Предметный указатель не существует)

R5RS Index

(Предметный указатель не существует)