Исправленный^5 отчёт по Алгоритмическому Языку Scheme

Ричард Келси, Уильям Клингер и Джонатан Рис (*Редакторы*)

Х. Абельсон
Н. И. Адамс IV
Д. Х. Бартлей
Г. Брукс
Р. К. Дибвиг
Д. П. Фридман
Р. Халстид
С. Хансон
С. Т. Хэйнес
Е. Кольбекер
Д. Оксли
К. М. Питман
Г. Дж. Розас
Г. Л. Стиил Джр.
Г. Дж. Сассмен
М. Уэнд

В память о Роберте Хиебе

Аннотация

Отчёт даёт определяющее описание языка Scheme. Scheme - диалект языка программирования Lisp с хвостовой рекурсией и статической областью видимости, придуманный Гай Льюис Стилом и Джарельдом Джей Сассменом. Он был спроектирован с исключительно понятной и простой семантикой и несколькими способами формирования выражений. Широкое разнообразие парадигм программирования, включающих императивный, функциональный стили, а также стиль обмена сообщениями делают выражения в Scheme удобными.

В разделе введение изложена краткая история языка и данного отчёта. Первые три главы описывают фундаментальные идеи языка и описывают условные обозначения используемые для описания языка и для написания программ в языке.

Главы 4 и 5 описывают синтаксис и семантику, выражений, программ и определений.

Глава 6 описывает встроенные процедуры Scheme, которые включают все действия с данными и примитивы ввода/вывода.

Глава 7 предоставляет формальный синтаксис языка написанный в расширенной БНФ(Бэкус - Наурова форма), вместе с формальной денотационной семантикой. Пример использования языка соответствует формальному синтаксису и семантике.

Отчёт завершается списком справочников и алфавитным указателем.

Содержание

Введение

Языки программирования должны быть спроектированы не нагромождением новых функций поверх других функций, а путём устранения недостатков и ограничений, которые делают появление дополнительных функций необходимым. Scheme демонстрирует, что небольшое количество правил для формирования выражений, без ограничений того, как они составляются, достаточно для того чтобы сформировать практический и эффективный язык программирования, который достаточно гибкий для поддержки большинства парадигм программирования, которые сейчас используются.

Scheme был одним из первых языков программирования, включающий в себя процедуры первого класса в качестве лямбда исчислений, тем самым обеспечивая полезность правил статической области видимости и блочных структур в динамически типизированном языке. Scheme был первым основным диалектом Lisp-а отделяющий процедуры от лямбда выражений и символов, использующий общее лексическое окружение для всех переменных, и определяющий позицию оператора процедурного вызова также, как позицию операнда. Полностью полагаясь на процедурные вызовы к экспресс итерации, Scheme подчёркивает тот факт, что процедуры с хвостовой рекурсией являются по существу операторами `goto`, которые принимают аргументы. Scheme был первым широко используемым языком программирования, имеющий процедуры выхода первого класса, из которых все ранее известные последовательные структуры управления могут быть синтезированы. Следующая версия Scheme представляет концепцию точных и приближенных чисел, расширение стандартной арифметики Common Lisp. Совсем недавно, Scheme стал первым языком программирования поддерживающим гигиенические макросы, которые допускают синтаксис структурного языка для того, чтобы добавить последовательный и надёжный стиль.

Предисловие

Перавое описание Scheme было написано в 1975 [28]. Исправленный доклад появившийся в 1978 году который описывал эволюции языка как реализация MIT был обновлён для поддержки инновационного компилятора. Три особых проекта были запущены в 1981 и 1982 годах для того чтобы использовать варианты языка Scheme для курсов в MIT, Ельском и Индианском университетах. Вводный учебник по компьютерным наукам использующий язык Scheme был опубликован в 1984 году.

Как только Scheme стал более широко используемым локальные диалекты стали более различаться между собой до тех пор пока студенты и исследователи не перестали понимать код написанный на разных сайтах. Поэтому 15 представителей основных реализаций языка Scheme встретились в октябре 1984 года чтобы работать над лучшим и более общепринятым стандартом для Scheme. Их отчёт был опубликован в Массачусетском Технологическом Институте и университете Индианы летом 1985 года. Следующии редакции появились весной 1986 и весной 1988. В настоящем докладе отражены дополнительные изменения согласованные на встрече Xerox PARC в июне 1992 года.

Благодарности

Мы хотим поблагодарить следующих людей за их помощь: Алан Боуден, Майкл Блэр, Джордж Каретте, Энди Кромарте, Павел Кертис, Джефф Дальтон, Оливье Дэнви, Кен Дики, Брюс Дуба, Марк Фили, Энди Фриман, Ричард Габриэль, Йокта Гурсэль, Кен Хаас, Роберт Хиеб, Пол Гудак, Морри Кац, Крис Линдблад, Марк Мейер, Джим Миллер, Джим Филбин, Джон Рамсделл, Майк Шафф, Джонатан Шапиро, Джули Сассмен, Перри Вейгл, Даниэль Вайс, Генри Ву и Озан Йигит. Мы благодарим Кэрол Фессенденома, Даниэля Фридмана и Кристофера Хайнеса за использование текста из справочного руководства Scheme 311 четвёртой версии. Мы благодарим корпорацию Texas Instruments за разрешение использовать текст из TI Scheme Language Reference Manual. Мы охотно признаем влияние справочников для MIT Scheme[17], [T22], Scheme 84[11],Common Lisp[27], and Algol 60[18].

Мы также благодарим Бэтти Декстер за огромные усилия, которые она потратила на переписывание данного доклада в TeX, и Дональда Кнута за проектирование программы, которая причинила Бэтти столько неудобств.

Лаборатория Искусственного Интеллекта Массачусетского Технологического Института, Департамент Компьютерных Наук университета Индианы, Департамент Компьютерных и Информационных Наук университета Орегона и Исследовательский Институт НЭК поддерживали подготовку данного доклада. Поддержку работы МИТ обеспечивало частично Агентство Продвинутых Исследовательских Проектов Министерства обороны США по контракту Управления военно-морских исследований N00014-80-C-0505. Поддержка работы университета Индианы была осуществлена благодаря грантам Национального Научного Фонда NCS 83-04567 и NCS 83-03325.

Глава 1

Обзор Scheme

1.1 Семантика

Данный раздел даёт описание семантики языка Scheme. Детальная неформальная семантика это тема глав с 3 по 6. В качестве справки в разделе 7.2 представлена формальная семантика языка Scheme. Так же как и Algol, Scheme язык программирования со статической областью видимости. Всякое использование переменных связано с лексически явной привязкой этой переменной.

В Scheme есть скрытые в противовес явным типам. Типы скорее связанны со значениями (также называемые объектами), чем с переменными. (Некоторые авторы относят языки со скрытыми типами к слабо- или динамически- типизированным языкам.) примерами других языков со скрытой типизацией могут послужить APL, Snobol и другие диалекты Lisp-а. Языки с явной типизацией (иногда их называют сильно- или статически- типизированными языками) включают в себя Algol 60, Pascal и C.

Все объекты созданные в ходе вычислений Scheme, включая процедуры и предложения, имеют безграничный размер. Ни один объект Sheme не будет когда-либо уничтожен. Причина того что реализации Scheme не выходят(обычно!) из памяти это то, что они имеют право освободить память выделенную под объект только, если они могут доказать что скорее всего объект не будет важен для дальнейших вычислений. Другие языки в которых большинство объектов имеет неограниченный размер включают в себя APL или другие диалекты Lisp. Реализации Scheme должны быть написаны с помощью хвостовой рекурсии. Это обеспечивает выполнение итеративных вычислений в постоянном месте, даже если итеративное вычисление описано синтаксически рекурсивной процедурой. Таким образом с реализацией хвостовой рекурсии итерация может быть выражена с помощью обычного механизма процедурного вызова. Так что специальный итеративный подход полезен только в качестве синтаксического сахара. Смотри раздел 3.5.

Процедуры в Scheme это объекты в полном понимании этого слова. Процедуры могут быть созданы динамически, храниться в структурах данных могут быть возвращены как результат процедур и тому подобное. Другие языки с подобными свойством включают в себя Common Lisp и ML.

Одна из отличительных черт Scheme это то, что продолжения, которые в большинстве языков фигурируют только “за кулисами”, также обладают статусом “первого класса”. Продолжения полезны для реализации широкого разнообразия расширенных управляющих конструкций, включая нелокальные выходы, возвраты и сопрограммы. Смотри раздел 6.4.

Аргументы процедур в Scheme всегда передаются по значению, это означает, что фактические выражения аргументов выполняются перед тем, как передается управление процедуре в независимости от того, нужен результат или нет. ML, C и APL - три других языка, в которых всегда передаются аргументы по значению. Что отличается от семантики ленивых вычислений в Haskell или от семантики вызова по имени в Algol 60, где выражение аргумента не вычисляется до тех пор пока его значение не понадобится процедуре.

Модель арифметики в Scheme спроектирована таким образом, чтобы быть независимым, насколько это возможно, от частных способов представления чисел в компьютере. В Scheme всякое число рациональное, всякое рациональное - вещественное, всякое вещественное - комплексное. Таким образом в Scheme нет разницы между целой и рациональной арифметикой(что так важно для многих языков). Различие между точной арифметикой (что соответствует математическому идеалу) и неточной заключается в приближении. Также как и в Common Lisp, точная арифметика не ограничивается целыми числам.

1.2 Синтаксис

В Scheme как и во многих диалектах языка Lisp используется полностью скобочная префиксная нотация для программ и других данных; грамматика Scheme порождает подъязык внутри языка используемый для данных. Важным следствием этого простого однообразного представления является восприимчивость программ Scheme и данных к обращению другими программами Scheme. К примеру процедура `eval` выполняет программу Scheme представленную в виде данных.

Процедура `read` выполняется синтаксически также как лексическое разложение считанных данных. Процедура `read` парсит входной поток как данные (раздел 7.1.2), не как программу.

Формальный синтаксис языка Scheme описан в разделе 7.1.

1.3 Условные обозначения и терминология

1.3.1 Примитив, библиотека и дополнительные функции

Необходимо чтобы каждая реализация Scheme поддерживала все функции которые помечены *дополнительными*. Реализации могут допускать дополнительные функции Scheme или добавлять расширения, при условии что расширения не противоречат языку описанному здесь. В частности реализации должны поддерживать переносимый код, обеспечивая такой синтаксические вид чтобы он не противоречил соглашениям данного доклада.

Чтобы помочь в понимании и реализации Scheme некоторые функции обозначены, как *библиотечные*. Эти функции легко реализуется с помощью других примитивных функций. Они избыточны в строгом понимании этого слова, но они охватывают основные шаблоны использования и тем самым обеспечивают удобные сокращения.

1.3.2 Ошибочные ситуации и неопределённое поведение

Когда речь идёт об ошибочной ситуации, в данном отчёте используется фраза “сигнализируется ошибка”, чтобы указать, что реализации должны быть обнаружены и необходимо сообщить об ошибке. Если данная формулировка не прозвучала при обсуждении ошибки, то в данной реализации нет необходимости обнаруживать ошибку или сообщать о ней, хоть и настоятельно рекомендуется сделать это. Ошибочные ситуации, реализация которых остается без обнаружения, обычно просто относятся к “ошибке”.

Например, ошибкой является, если процедура принимает аргументы, которые не определены в её объявлении, хотя такого рода доменные ошибки редко упоминаются в данном докладе. Реализации могут расширять принимаемые значения при объявлении процедуры, чтобы включить необходимые аргументы.

В данном отчёте используется фраза “может сообщить о нарушении ограничений реализации”, чтобы обозначить об обстоятельствах при которых реализации разрешено сообщить, что она не может продолжить выполнение данной программы из-за некоторых ограничений наложенных самой реализацией. Ограничения реализации, конечно, обескураживают, но реализации должны сообщить о нарушении ограничений.

Например, реализация может сообщить о нарушении ограничений если не хватает памяти для запуска программы.

Если сказано, что значение выражения “не определено”, то выражение должно быть присвоено некоторому объекту без сообщения об ошибке, однако данное значение зависит от реализации; в данном докладе не говорится о том какое значение должно быть возвращено.

1.3.3 Формат записи

Главы 4 и 6 состоят из записей. Каждая запись описывает одну из функций языка или группу связанных функций, где функцией является либо синтаксическая конструкция, либо встроенная процедура. Запись начинается с одной или более строк заголовков в форме *категория*: *шаблон*

для требуемых, примитивных функций, или

спецификатор категории: *шаблон*

где классификатор либо “библиотечный”, либо “дополнительный”, как определено в секции 1.3.1.

Если *категория* “синтаксическая”, запись описывает тип выражения, и шаблон дает синтаксис типа выражения. Компоненты выражения обозначены синтаксическими переменными, которые написаны с использованием треугольных скобок, к примеру, <выражение>, <переменная>. Синтаксические переменные следует понимать, как обозначение сегментов текста программы. Обозначение

<объект$_1$> …

обозначает ноль или более вхождений <объекта>, и

<объект$_1$> <объект$_2$> …

означает одно или более вхождений <объекта>.

Если *категория* это “процедура”, то запись описывает процедуру, и строка заголовка содержит шаблон для вызова процедуры. Имена аргументов в шаблоне *выделены курсивом*. Таким образом строка заголовка

процедура: `(vector-ref` *`vector k`*`)`

означает, что встроенная процедура `vector-ref` принимает два аргумента вектор *vector* и в точности неотрицательное целое число *k*(смотри ниже). Строка заголовка

процедура: `(make-vector` *`k`*`)`
процедура: `(make-vector` *`k fill`*`)`

означает, что процедура `make-vector` должна принимать либо один, либо два аргумента.

Будет ошибкой если операция будет представлена с аргументом, который не указан в её объявлении. Для краткости, мы будем следовать соглашению, что если имя аргумента также является именем одного из типов перечисленных в разделе 3.2, то этот аргумент должен иметь заданный тип. Например, заголовочная строка для `vector-ref`, приведенная выше, требует чтобы первый аргумент для `vector-ref` был вектором. Приведенные ниже соглашения также подразумевают ограничения типов:

obj	некоторый объект
list, *list*$_1$, … ,*list*$_j$, …	список(смотри раздел 6.3.2)
$z, z_1, \ldots z_j \ldots$	комплексные числа
$x, x_1, \ldots x_j \ldots$	вещественные числа
$y, y_1, \ldots y_j \ldots$	вещественные числа
$q, q_1, \ldots q_j \ldots$	рациональные числа
$n, n_1, \ldots n_j \ldots$	целые
$k, k_1, \ldots k_j \ldots$	в точности неотрицательное целое число

1.3.4 Примеры вычислений

Символ “===>”, который используется в примерах программ должен читаться как “принимает значение”. Например,

(* 5 8) ===> 40

означает, что выражение `(* 5 8)` принимает значения объекта 40. Или более точно: выражение представленное как последовательность символов “`(\* 5 8)`” принимает значение, в данной среде, объекта, который может быть представлен внешне, как последовательность символов “40”. Смотри раздел 3.3, где обсуждается внешнее представление объектов.

1.3.5 Соглашения о наименовании

По соглашению, имена процедур, которые всегда возвращают булево значение обычно заканчиваются “`?`”. Такие процедуры называются предикатами.

По соглашению, имена процедур, которые хранят значения в ранее выделенных местах(смотри раздел 3.4) обычно заканчиваются “!”. Такие процедуры называются мутационными. По договоренности, значения возвращаемые мутационными процедурами неопределено.

По соглашению, знак “`->`” появляется с именами процедур, которые принимают объект одного типа и возвращают аналогичный объект другого типа. Например, `list->vector` принимает список и возвращает вектор, элементы которого такие же как и в списке.

Глава 2

Лексические соглашения

В данном раздели дается неофициальный отчёт о некоторых лексических соглашениях, используемых при написании программ Scheme. Для описания формального синтаксиса смотрите раздел7.1.

Верхние и нижние регситры при написании программ ни чем не отличются, если не считать символьные и строковые константы. Например, идентификатор Foo это тоже самое что идентификатор `FOO`, а `#x1AB` это тоже самое число что и `#X1ab`.

2.1 Идентификаторы

Большинство идентификаторов допустимых в других языках также разрешены в Scheme. Точные правила для формирования идентификаторов меняются в зависимости от реализаций Scheme, но во всех реализациях последовательность букв, цифр и “расширенных алфавитных символов” идентификаторы не могут начинаться с числа. Кроме того, `+`, `-` и `...` являются идентификаторами. Ниже приведены примеры идентификаторов:

```
lambda                      q
list->vector                soup
+                           V17a
<=?                         a34kTMNs
the-word-recursion-has-many -meanings
```

Расширенные алфавитные символы могут использоваться в идентификаторах так, как если бы они были буквами. К расширенным алфавитным относяться следущие символы:

! _ _ & * + - . / : < = > ? @ ^ _ ~

Смотри раздел 7.1.1,где описан формальный синтаксис идентификаторов.

Идентификаторы могут использоваться в двух случаях в программах Scheme:

- Всякий идентификатор может использоваться, как переменная или синтаксическое ключевое слово(смотри раздел 3.1 и 4.3).
- Когда идентификатор как литерал или вместе с литералом(смотри раздел 4.1.2), тогда он обозначается как *символ*(смотри раздел 6.3.3).

2.2 Отступы и коментарии

Пробельные символы это пробелы и переводы строк. (Реализации обычно допускают дополнительные пробельные символы такие, как табуляции или разрыв страницы.) Пробелы используются для улучшения читабельности и необходимы для разделения токенов друг от друга, токеном является неделимая лексическая единица, такая как идентификатор или число, но сам по себе он смысла не несет. Пробелы могут разделять два токена, но сами токенами не являются. Пробелы могут входить в строки, где они уже имеют значение.

Точка с запятой(;) указывает на начало комментария. Комментарий продолжается до конца строки на которой стоит точка с запятой. Scheme не воспринимает комментарии, но конец закомментированной строки отображается как пробел, что предотвращает появление комментария в середине идентификатора или числа.

```
;;; Процедура FACT вычисляет факториал
;;; неотрицательного числа.
(define fact
  (lambda (n)
    (if (= n, 0)
        1         ;Базовый случай: возвращает 1
        (* n (fact (- n 1))))))
```

2.3 Другие обозначения

Описание обозначений, используемых для чисел, смотри в разделе 6.2.

- `. + -` Данные символы используются с числами и могут также находится в любом месте идентификатора, кроме первого символа. Знак плюс или минус разделенный самим же собой также является идентификатором. Разделенный период (не окруженный числом или идентификатором) используется для обозначения пары (смотри раздел 6.3.2), и для того чтобы обозначить остальные параметры в формальном списке параметров (раздел 4.1.4). Разделенная последовательность трех последовательных периодов также является идентификатором.
- `( )` Скобки используются для группировки и обозначения списков (раздел 6.3.2).
- `'` Символ апостроф используется для обозначения буквенных данных (раздел 4.1.2).
- `` ` `` Обратный апостроф используется для обозначения почти-константных данных (раздел 4.2.6).
- `, ,@` Символ запятая и собака последовательность запятая, собака используется вместе с обратной кавычкой (раздел 4.2.6).
- `"` Двойные кавычки используются для выделения строк (раздел 6.3.5).
- `\` Обратный слэш используется в синтаксисе символьных констант (раздел 6.3.4) и как терминальный символ в строковых константах (раздел 6.3.5).
- `[ ] { } |` Левая и правая квадратные и фигурные скобки, а также вертикальная черта зарезервированы для возможных будущих расширений языка.
- `\#` Решетка используется в разных целях в зависимости от символа, который следует за ним.
- `\#t \#f` Булевы константы (раздел 6.3.1).
- `\#\` Вводит символьную константу (раздел 6.3.4).
- `\#(` Вводит векторную константу (раздел 6.3.6). Векторная константа завершается `)`.
- `\#e \#i \#b \#o \#d \#x` Используются для обозначения чисел (раздел 6.2.4).

Глава 3

Базовые концепции

3.1 Переменные, синтаксические ключевые слова, области

Идентификатор может называть тип синтаксиса, или он может называть локацию, где значение должно храниться. Идентификатор, который называет тип синтаксиса, называется синтаксическим ключевым словом и говорит, что данный синтаксис привязан к данной локации. Множество ссылок видимых в данном месте программы, называется окружением, доступным в данном месте. Значение хранимое в том месте, к которому переменная привязана называется значением переменной. В силу избыточности терминологии, переменная иногда обозначает значение или ссылку на значение. Это является неточностью, но как показывает практика редко возникает путаница связанная с этим понятием.

Некоторые типы выражений используются для создания новых видов синтаксиса и привязки синтаксических ключевых слов к этому новому синтаксису, в то же время другие типы выражений создают новые локации и привязывают переменные к этим локациям. Эти типы выражений называются связующими конструктами. Те, которые связывают синтаксические ключевые слова перечислены в разделе 4.3. Самыми главными связующими конструктами переменных являются лямбда выражения, поскольку все другие связующие конструкты переменных могут быть описаны с их помощью. Другие связующие конструкты переменных это выражения `let`, `let*`, `letrec` и `do` (смотри разделы 4.1.4, 4.2.2 и 4.2.4).

Так же как Algol и Pascal, и в отличии от многих других диалектов Lisp за исключением Common Lisp, Scheme статически типизированный язык с блочной структурой. В каждом месте, где идентификатор ограничен в программе образуется *область* программного текста в пределах которой привязка видна. Область определяется особыми связующими конструктами, которые устанавливают связь; если связь установлена, например, с помощью *лямбда* выражений, то её областью является всё *лямбда* выражение. Всякое упоминание идентификатора ссылается на привязку к идентификатору, которая устанавливается в наиболее вложенной области данного контекста. Если нет привязки к идентификатору, область которого содержит контекст, то контекст ссылается на привязку идентификатора на более высоком уровне окружения, если таковая имеется (главы 4 и 6); если привязки идентификатора нет, он называется *несвязным*.

3.2 Дизъюнктность типов

Ни один из объектов не удовлетворяет более чем одному из следующих предикатов:

```
boolean?    pair?
sybol?      number?
char?       string?
vector?     port?
procedure?1 .
```

Данные предикаты определяют типы *boolean*, *pair*, *symbol*, *number*, *char* (или *character*), *string*, *vector*, *port* и *procedure*. Пустой список это специальный объект со своим собственным типом; он не удовлетворяет ни одному из выше перечисленных предикатов.

Несмотря на то что существует отдельный булев тип; Любое значение Scheme может быть использовано как булево значение для условной проверки. Как объясняется в разделе 6.3.1, все значения считаются истинными в такой проверке, кроме #f. В данном докладе используется слово

“истина”, для обозначения любого значения Scheme за исключением #f, и слово “ложь” для обозначения #f.

3.3 Внешние представления

Важной концепцией в Scheme (и Lisp) является *внешнее представление* объекта, как последовательность символов. Например, внешнее представление целого числа 28 это последовательность символов “`28`”, и внешнее представление списка состоящего из целых 8 и 13 это последовательность символов “`(8 13)`”.

Внешнее представление объекта не обязательно уникально. Число 28 также имеет представления `"#e28.000"` и `"#x1c"`, а список из предыдущего параграфа также имеет представления `"( 08 13 )"` и `"(8 . (13 . ()))"` (смотри раздел #6.3.2).

Многие объекты имеют стандартное представление, но некоторые, такие как процедуры, не имеют такового (хотя отдельные имплементации могут определить представления для них).

Внешнее представление может быть написано в программе для получения соответствующего объекта (смотри quote, раздел #4.1.2).

Внешнее представление также может быть использовано для ввода и вывода. Процедура `read` (раздел 6.3.2) парсит внешние представления, а процедура `write`(раздел 6.6.3) генерирует их. Вместе, они обеспечивают элегантный и мощный инструмент ввода/вывода.

Обратите внимание, что последовательность символов `"(+ 2 6)"` не является внешним представлением целого числа 8, даже если это выражение вычисляется как целое число 8; скорее, это внешнее представление списка из трёх элементов, элементы которого символ + и целые числа 2 и 6. Синтаксис Scheme имеет такое свойство, что последовательность символов, которая является выражением также и является внешним представлением некоторого объекта. Это может привести к путанице, поскольку может быть не совсем понятно из контекста предназначена ли данная последовательность символов для обозначения данных или программы, но это также является источником больших возможностей, поскольку это облегчает написание программ таких, как интерпретаторы и компиляторы, которые рассматривают программу как данные (или наоборот).

Синтаксис внешнего представления различных видов объектов сопровождается описанием примитивов для манипуляции объектами в соответствующем разделе главы 6.

3.4 Модель памяти

Переменные и объекты такие, как пары, векторы и строки косвенно обозначают локации или последовательность локаций. Строка, к примеру, означает последовательность локаций символов в строке. (Эти локации не должны соответствовать полному машинному слову.) Новое значение может хранится в одной из этих локаций если использовать процедуру `string-set!`, но строка продолжает означать некоторые локации, как и раньше.

Объект выбранный из локации переменной или процедурой такой как `car`, `vector-ref`, `string-ref` это эквивалент в последовательности `eqv?` (раздел 6.1) последнему объекту хранимому в локации перед выборкой.

Всякая локация помечается, чтобы обозначить используется она или нет. Нет переменных или объектов когда-либо ссылающихся на локацию, которая не используется. Всякий раз, когда в этом докладе говорится о выделение памяти для переменной или объекта подразумевается, что подходящее число локаций выбрано из множества локаций, которые ещё не используются, и выбранные локации помечаются чтобы обозначить, что они сейчас используются перед тем как переменная или объект начнёт указывать на них.

Для многих систем желательно расположение констант (т. е. значений или буквенных выражений) в памяти только для чтения. Для того чтобы выразить это, имеется соглашение представлять, что все объекты являются изменяемыми или неизменяемыми. В таких системах буквенные константы и строки возвращаемые `symbol->string` являются неизменяемыми объектами, в то время как все объекты созданные другими процедурами, перечисленными в этом докладе являются изменяемыми. Будет ошибкой пытаться поместить новое значение в локацию, которая выделена неизменяемым объектом.

3.5 Правильная хвостовая рекурсия

Имплементации Scheme должны быть *правильной хвостовой рекурсией*. Процедурные вызовы, которые выполняются в некотором синтаксическом контексте определённом ниже являются “хвостовыми вызовами”. Имплементация Scheme является правильной хвостовой рекурсией, если она поддерживает неограниченное число активных хвостовых вызовов. Вызов называется *активным*, если вызываемая процедура может выполняться повторно. Обратите внимание, что это включает в себя вызовы, которые могут быть возвращены либо данной континуацией, либо континуацией захваченной ранее с помощью `call-with-current-continuation`, которая вызывается позже. При отсутствии захваченной континуации, вызовы могут возвращать управление максимум один раз и активными вызовами могут быть те, которые ещё не вернули управление. Формальное определение правильной хвостовой рекурсии можно найти в [8].

> *Обоснование*:
>
> Интуитивно, никакого пространства не нужно для активного хвостового вызова, поскольку континуации, которые используются в хвостовом вызове имеют ту же семантику, что и континуации передающиеся процедуре, содержащей вызов. Хотя неправильная имплементация может использовать новую континуацию в вызове, возврат к этой новой континуации может следовать мгновенно за возвратом континуации, передаваемой процедуре. Имплементация правильной хвостовой рекурсии возвращается напрямую континуации.
>
> Правильная хвостовая рекурсия была одной из центральных идей в оригинальной версии Scheme Стила и Сьюзмена. Их первый интерпретатор Sheme реализовал и функции, и акторы. Управление потоком было реализовано с использованием акторов, которые отличались от функций тем, что они передавали их результат другим акторам вместо возврата вызывающему. В терминах данного раздела, каждый актор заканчивается хвостовым вызовом другого актора.
>
> Стил и Сьюзмен позже заметили, что в их интерпретаторе код для работы с акторами был идентичен коду для функций и таким образом не было смысла включать и то и другое в язык.

Хвостовым вызовом является процедурный вызов, который выполняется в хвостовом контексте. Хвостовые контексты определяются индуктивно. Обратите внимание, что хвостовой контекст всегда определяется по отношению к конкретному лямбда выражению.

- Последнее выражение вместе с телом лямбда выражения, обозначенное ниже, как <хвостовое выражение>, выполняется в хвостовом контексте.

```
(lambda <переменные>
            <определение>* <выражение>* <хвостовая рекурсия>)
```

- Если одно из следующих выражений находится в хвостовом контексте, то и подвыражения обозначенные как <хвостовая рекурсия>, находятся в хвостовом контексте. Это правило вытекает из правил грамматики, описанных в главе 7 заменой некоторых вхождений <выражения> <хвостовой рекурсией>. Здесь показаны только те правила, которые содержат хвостовой контекст.

```
(if <выражение> <хвостовая рекурсия> <хвостовая рекурсия>)
(if <выражение> <хвостовая рекурсия>)

(cond <клауза условия>+)
(cond <клауза условия>* (else <хвостовая последовательность>))

(case <выражение>
        <клауза case>+)
(case <выражение>
        <клауза case>*
        (else <хвостовая последовательность>))

(and <выражение>* <хвостовое выражение>)
(or <выражение>* <хвостовое выражение>)

(let (<связующая спецификация>*) <хвостовое тело>)
(let <переменная> (<связующая спецификация>*) <хвостовое тело>)
(let* (<связующая спецификация>*) <хвостовое тело>)
(letrec (<связующая спецификация>*) <хвостовое тело>)

(let-syntax (<синтаксическая спецификация>*) <хвостовое тело>)
(letrec-syntax (<синтаксическая спецификация>*) <хвостовое тело>)

(begin <хвостовая последовательность>)

(do (<спецификация итерации>*)
                          (<проверка> <хвостовая последовательность>)
                    <выражение>*)

где

<клауза условия> ---> (<проверка> <хвостовая последовательность>)
<клауза case> ---> ((<элемент данных>*) <хвостовая последовательность>)

<хвостовое тело> ---> <определение>* <хвостовая последовательность>
<хвостовая последовательность> ---> <выражение>* <хвостовое выражение>
```

- Если условное выражение в хвостовом контексте, имеет клаузу в форме (*<выражение$_1$>* => *<выражение$_2$>*), то (предполагаемый) вызов процедуры который является результатом вычисления значения *<выражение$_2$>* находится в хвостовом контексте. Само *<выражение$_2$>* в хвостовом контексте не находится.

Некоторые встроенные процедуры также требуются для выполнения хвостовых вызовов. Первый аргумент передаётся `apply` и `call-with-current-continuation`, а второй аргумент передаётся`call-with-values`, должен вызываться с помощью хвостового вызова. Аналогичным образом, `eval` должен вычислять свой аргумент, как если бы он находился в хвостовой позиции вместе с процедурой `eval`.

В следующем примере единственный хвостовой вызов это вызов f. Ни один из вызовов g или h не является хвостовым. Ссылка на x находится в хвостовом контексте, но это не вызов, а значит это не хвостовой вызов.

```
(lambda ()
    (if (g)
        (let ((x (h)))
            x)
        (and (g) (f))))
```

> *Замечание:* Имплементации разрешены, но не требуются, для распознавания того, что некоторые не хвостовые вызовы, такие как вызов h выше, могут выполняться, как если бы они были хвостовыми. В примере выше, выражение `let` может быть скомпилировано как хвостовой вызов h. (Возможность h возвращать неизвестное число значений может игнорироваться, поскольку в этом случае результат `let` не указан явно и зависит от реализации.)

Глава 4

Выражения

Типы выражений классифицируются на примитивные и производные. Типы примитивных выражений включают в себя переменные и процедурные вызовы. Производные типы выражений не являются примитивными с семантической точки зрения, но вместо этого могут быть определены через макросы. За исключением `quasiquote`, макро определение которого является сложным, производные выражения классифицируются, как библиотечные функции. Соответствующие определения даны в разделе 7.3.

4.1 Примитивные типы выражений

4.1.1 Ссылки на переменные

синтаксис: *<переменная>*

Выражение состоящее из переменной (раздел 3.1) является ссылкой на переменную. Значение ссылки на переменную это значение, хранимое в локации к которой переменная привязана. Будет ошибкой ссылаться на несвязную переменную.

```
(define x 28)
x           ===>    28
```

4.1.2 Буквенные выражения

синтаксис: `(quote` *<элемент данных>*`)`
синтаксис: `'`*<элемент данных>*
синтаксис: *<постоянная>*

`(quote` *<элемент данных>*`)` вычисляется как <элемент данных>. <Элементом данных> может быть любое внешнее представление объекта (смотри раздел 3.3). Это обозначение используется для включение буквенных констант в код Scheme.

```
(quote a)                       ===>    a
(quote #(a b c))                ===>    (#a b c)
(quote (+ 1 2))                 ===>    (+ 1 2)
```

`(quote <элемент данных>)` можно сократить как `'<элемент данных>`. Два этих обозначения эквиваленты во всех отношениях.

```
'a                          ===>    a
'#(a b c)                   ===>    #(a b c)
'()                         ===>    ()
'(+ 1 2)                    ===>    (+ 1 2)
'(quote a)                  ===>    (quote a)
''a                         ===>    (quote a)
```

Численные константы, строковые константы, символьные константы и булевы константы вычисляются «к самим себе»; нет смысла цитировать их.

```
'"abc"                    ===>    "abc"
"abc"                     ===>    "abc"
'145932                ===>    145932
145932                 ===>    145932
'#t                    ===>    #t
#t                     ===>    #t
```

Как отмечено в разделе 3.4, будет ошибкой изменять константу (т. е. Значение буквенного выражения) используя изменяемые процедуры такие, как `set-car!` или `string-set!`.

4.1.3 Процедурные вызовы

синтаксис: (*<оператор> <операнд$_1$> …*)

Процедурный вызов записывается простым заключением выражения в круглые скобки для вызываемой процедуры и аргументами, которые передаются ей. Оператор и операнд выражений вычисляются (в неопределённом порядке) и полученная процедура принимает полученные аргументы.

```
(+ 3 4)                ===>   7
((if #f + *) 3 4)      ===>   12
```

Некоторые процедуры доступны как значения переменных в первоначальном окружении; например, процедуры сложения и умножения в примере выше значения переменных + и *. Новые процедуры создаются вычислением лямбда выражений (смотри раздел 4.1.4). Процедурные вызовы могу возвращать произвольное число значений (смотри `values` в разделе 6.4). Не считая `values` процедуры доступные в начальном окружении возвращают одно значение или, для таких процедур как `apply`, передающих значения возвращаемые при вызове одному из своих аргументов.

Процедурные вызовы также называются *комбинациями*.

> *Замечание:* Согласно другим диалектам Lisp, порядок выполнения не определён, и оператор и операнд выражения всегда выполняются с одинаковыми правилами выполнения
>
> *Замечание:* Хотя порядок выполнения должным образом не определён, результат любого параллельного вычисления оператора и операнда выражений должен быть последовательным с некоторым последовательном порядком вычисления. Порядок вычисления может быть выбран различным образом для разных процедурных вызовов.
>
> *Замечание:* Во многих диалектах Lisp, пустая комбинация, `()`, является допустимым выражением. В Scheme, комбинации должны иметь как минимум одно подвыражение, то есть `()` не является синтаксически правильным выражением.

4.1.4 Процедуры

синтаксис: `(lambda <формальные определения> <тело>)`

Синтаксис: <Формальные определения> должны быть списком формальных аргументов, как описано выше, а <телом> должна быть последовательность одного или нескольких выражений.

Семантика: Лямбда выражение выполняется как процедура. В результате выполнения лямбда выражения его окружение запоминается, как часть процедуры. Когда после этого процедура вызывается с некоторыми фактическими аргументами, окружение в котором лямбда выражение выполнилось будет расширено новыми локациями за счёт привязок переменных в списке формальных аргументов, соответствующие фактические аргументы будут храниться в этих локациях, и выражения в теле лямбда выражения будут выполняться последовательно в расширенном окружении. Результат последнего выражения в теле будет возвращен как результат процедурного вызова.

```
(lambda (x) (+ x x))                ===>  a procedure
((lambda (x) (+ x x)) 4)            ===>  8
(define reverse-subtract
  (lambda (x y) (- y x)))
(reverse-subtract 7 10)                   ===>  3
(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add 4 6)                                 ===>  10
```

<Формальные определения> должны иметь одну из следующих форм:

- `(`*<переменная$_1$>* `...)`: Процедура принимает фиксированное число аргументов; когда процедура вызывается, аргументы хранятся в привязках к соответствующим переменным.
- <переменная>: Процедура принимает произвольное число элементов; когда процедура вызывается, последовательность фактических аргументов преобразуется в новый список(под который выделяется память), и этот список хранится в привязки <переменной>.
- `(`*<переменная$_1$>* `...` *<переменная$_n$>* `.` *<переменная$_{n+1}$>*`)`: если перед последней переменной стоит точка, процедура принимает *n* или более аргументов, где *n* — это число формальных аргументов перед точкой (процедура должна принимать хотя бы один аргумент). Значение хранимое в привязке последней переменной будет выделено под новый список фактических аргументов оставшихся после того, как все остальные фактические аргументы будут сопоставлены с их формальными аргументами.

Появление <переменной> более одного раза в <формальном определении> будет ошибкой.

```
((lambda x x) 3 4 5 6)                    ===>  (3 4 5 6)
((lambda (x y . z) z)
 3 4 5 6)                                 ===>  (5 6)
```

Всякая процедура созданная в результате выполнения лямбда выражения (концептуально) создаёт локацию памяти, для того чтобы `eqv?` и `eq?` могли работать над процедурами (смотри раздел 6.1).

4.1.5 Условные выражения

синтаксис: `(if <проверка> <следствие> <альтернативная ветка>)`
синтаксис: `(if <проверка> <следствие>)`

Синтаксис: <проверка>, <следствие> и <альтернатива>могут быть произвольными выражениями.

Семантика: выражение `if` выполняется следующим образом: сначала выполняется <проверка>. Если она даёт истинное значение (смотри раздел 6.3.1). выполняется <следствие> и возвращается(-ются) его значение(я). В противном случае выполняется <альтернативная ветка> и возвращается(-ются) его значение(я). Если <проверка> даёт ложное значение и <альтернативная ветка> не указана, то результат выражения не определён.

```
(if (> 3 2) 'yes 'no)                  ===>  yes
(if (> 2 3) 'yes 'no)                  ===>  no
(if (> 3 2)
    (- 3 2)
    (+ 3 2))                           ===>  1
```

4.1.6 Присваивания

синтаксис: `(set! <переменная> <выражение>)`

<Выражение> выполняется, и итоговое значение хранится в локации к которой <переменная> привязана. <Переменная> должна быть привязана либо к некоторой области, содержащей выражение `set!`, либо к верхнему уровню. Результат выражения `set!` не определён.

```
(define x 2)
(+ x 1)                          ===>  3
(set! x 4)                       ===>  unspecified
(+ x 1)                          ===>  5
```

4.2 Производные типы выражений

Конструкты в данном разделе являются гигиеническими, как говорится в разделе 4.3. В справочных целях, раздел 7.3 даёт макро определения, которые преобразуют большинство конструктов, описанных в данном разделе к примитивным конструктам, описанных в предыдущем разделе.

4.2.1 Условные выражения

библиотечный синтаксис: `(cond <клауза1> <клауза2> ...)`

Синтаксис: Всякая <клауза> должно иметь форму:

`(<проверка> <выражение1> ...)`

где <проверка> это произвольное выражение. Также, <клауза> может иметь форму:

`(<проверка> => <выражение>)`

Последняя <клауза> может « клауза иначе», которая имеет форму

`(else` *<выражение$_1$> <выражение$_2$>* `...)`.

Семантика: Выражение `cond` получается вычислением выражений <проверки> успешной <клауз> до тех пор пока одна из них не вычислится как истинное значение (смотри раздел 6.3.1). Когда <проверка> вычисляется, как истинное значение, оставшиеся <выражения> в своей <клаузе> вычисляются в порядке, а результат последнего выражения в <клаузе> возвращается как результат всего выражения `cond`. Если выбранная <клауза> содержит только <проверку> и не содержит <выражений>, то в результате возвращается значение <проверки>. Если выбранная <клауза> использует альтернативную форму =>, то вычисляется <выражение>. Его выражение должно быть процедурой, которая принимает один аргумент; это выражение вызывается со значением <проверки> и значения возвращаемые этой процедурой возвращаются выражением `cond`. Если все <проверки> вычисляются со значением лжи, и нет клаузы `else`, результат условного выражения не определён; если есть клауза `else`, то <выражения> вычисляются, и возвращается(ются) значение(я).

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))                  ===>  greater
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))                     ===>  equal
(cond ((assv 'b '((a 1) (b 2))) => cadr)
      (else #f))                   ===>  2
```

библиотечный синтаксис: `(case` *<ключ> <клауза$_1$> <клауза$_2$>* `...)`

Синтаксис: <Ключом> может быть любое выражение. Всякая <клауза> должна иметь форму:

`((`*<элемент данных$_1$>*`) ...)` *<выражение$_1$> <выражение$_2$>* `...)`,

где каждый <элемент данных> является внешним представлением некоторого объекта. Все <элементы данных> должны быть различны. Последняя <клауза> может быть «клаузой else», которая имеет форму

`(else` *<выражение$_1$> <выражение$_2$>* `...)`.

Семантика: Выражение `case` вычисляется следующим образом. <Ключ> вычисляется, его результат сравнивается с каждым <элементом данных>. Если результат вычисления <ключа> совпадает (в смысле процедуры `eqv?`; смотри раздел 6.1) с <элементом данных>,то выражение в соответствующей <клаузе> вычисляется слева на право и результат(ы) последнего выражения в <клаузе> возвращается, как результат выражения `case`. Если результат вычисления <ключа> отличен от каждого <элемента данных>, то если есть клауза `case`, его выражения вычисляются и результат последнего(последних) результата(результатов) является результатом выражения `case`; в противном случае результат выражения `case` не определён.

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite))            ===>  composite
(case (car '(c d))
  ((a) 'a)
  ((b) 'b))                            ===>  unspecified
(case (car '(c d))
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else 'consonant))                   ===>  consonant
```

библиотечный синтаксис: `(and <проверка1> ...)`

Выражение <проверка> вычисляется слева направо, и значение первого выражения, которое вычисляется как ложь (смотри раздел 6.3.1), возвращается. Все оставшиеся выражения не вычисляются. Если все выражения оказались истинными, будет возвращено значение последнего выражения. Если выражений нет, возвращается `#t`.

```
(and (= 2 2) (> 2 1))                  ===>  #t
(and (= 2 2) (< 2 1))                  ===>  #f
(and 1 2 'c '(f g))                    ===>  (f g)
(and)                                  ===>  #t
```

библиотечный синтаксис: `(or <проверка1> ...)`

<Проверка> вычисляется слева направо, и возвращается результат первого выражения, которое оказалось истинным (смотри раздел 6.3.1). Все оставшиеся выражения не вычисляются. Если все выражения оказываются ложью, возвращается результат последнего. Если выражений нет возвращается `#f`.

```
(or (= 2 2) (> 2 1))                   ===>  #t
(or (= 2 2) (< 2 1))                   ===>  #t
(or #f #f #f)          ===>  #f
(or (memq 'b '(a b c))
    (/ 3 0))
```

4.2.2 Связующие конструкты

Три связующих конструкта `let`, `let*` и `letrec` дают Scheme блочную структуру, как в Algol 60. Синтаксис этих трёх конструктов одинаковый, но они отличаются областью, которую они устанавливают для привязок переменных. В выражении `let`, начальные значения вычисляются перед привязкой всех переменных; привязка и вычисления выполняются последовательно; в то время как в выражении `letrec`, все привязки осуществляются, в то время когда вычисляются их значения, тем самым разрешая взаимно рекурсивные определения.

библиотечный синтаксис: `(let <привязки> <тело>)`

Синтаксис: <Привязки> должны иметь форму

`((<переменная1> <инициализация1>) ...)`,

где каждая <инициализация> является выражением, а <тело> должно быть последовательностью одного или более выражения. Возникнет ошибка, если <переменная> появиться более одного раза в списки привязанных переменных.

Семантика: <инициализация> выполняется в данном окружении (в некотором неопределённом порядке) <переменные> привязываются к новым локациям, запоминая результат, <тело> выполняется

в расширенном окружении, и возвращается(возвращаются) значение(я) последнего выражения <тела>. Всякая привязка <переменной> имеет область <тела>.

```
(let ((x 2) (y 3))
  (* x y))                              ===>  6

(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))                           ===>  35
```

Также смотри именованный `let`, раздел 4.2.4.

библиотечный синтаксис: `(let* <привязки> <тело>)`

Синтаксис: <Привязки> должны иметь форму

`((`*<переменная$_1$>* *<инициализация$_1$>*`) ...)`,

и <тело> должно быть последовательностью одного или более выражения.

Семантика: `Let*` тоже самое что и `let`, но привязки выполняются последовательно слева направо, а область связывания, которая обозначается как `(`*<переменная>* *<инициализация>*`)`, является той частью выражения `let*`, которая справа от привязки. Таким образом вторая привязка осуществляется в окружении, в котором первая видна первая привязка и так далее.

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))                       ===>  70
```

библиотечный синтаксис: `(letrec <привязки> <тело>)`

Синтаксис: <Привязки> имеют форму

`((`*<переменная$_1$>* *<инициализация$_1$>*`) ...)`,

и <тело> должно быть последовательностью одного или более выражения. Если <переменная> появляется более одного раза в списке переменных, которые привязываются, возникнет ошибка.

Семантика: <Переменные> привязываются к новым локациям, которые имеют неопределённые значения, <инициализации> выполняются в полученном окружении (в некотором неопределённом порядке), каждая переменная присваивается к результату соответствующей <инициализации>, <тело> выполняется в полученном окружении, и возвращается значение(я) последнего выражения в <теле>. Каждая привязка <переменной> содержит все выражение `letrec`, как и его область, делая возможным определение взаимно рекурсивных процедур.

```
(letrec ((even?
          (lambda (n)
            (if (zero? n)
                #t
                (odd? (- n 1)))))
         (odd?
          (lambda (n)
            (if (zero? n)
                #f
                (even? (- n 1))))))
  (even? 88))
                        ===>  #t
```

Важным ограничением на использование `letrec` является следующее: должно быть возможно выполнять каждую <инициализацию> без определения или ссылки на значение какой-либо <переменной>. Если это ограничение нарушается, возникает ошибка. Это ограничение необходимо из-за того что в Scheme аргументы передаются по значению, а не по имени. В большинстве случаев использования `letrec` все <инициализации> являются лямбда выражениями и данное ограничение выполняется автоматически.

4.2.3 Последовательное выполнение

<u>библиотечный синтаксис:</u> `(begin` *<выражение$_1$>* *<выражение$_2$>* `...)`

<Выражения> выполняются последовательно слева направо, возвращаются значения последнего выражения. Этот тип выражений используется для случаев последовательного выполнения таких как ввод и вывод.

```
(define x 0)

(begin (set! x 5)
       (+ x 1))                         ===>  6

(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))               ===>  unspecified
  and prints  4 plus 1 equals 5
```

4.2.4 Итерация

<u>библиотечный синтаксис:</u> `(do ((`*<переменная$_1$>* *<инициализация$_1$>* *<шаг$_1$>*`)`

```
    ...)
    (<проверка> <выражение> ...)
<команда> ...)
```

`Do` это итерационный конструкт. Он определяет множество переменных, которые будут привязаны, как они будут инициализироваться в начале, и как они обновляются на каждой итерации. Когда встречается условие завершения, цикл завершается после выполнения <выражений>.

Выражения `do` выполняются следующим образом: Выражения <инициализации> выполняются (в некотором не определённом порядке), <переменные> привязываются к новым локациям, результат выражения <инициализации> храниться в привязках к <переменным>, после этого начинается фаза итерации.

Каждая итерация начинается с выполнения <проверки>; если результат ложь (смотри раздел 6.3.1), выражения <команды> выполняются по порядку для получения результата, выражения <шага> выполняются в некотором неопределённом порядке, <переменные> привязываются к новым локациям, результаты <шагов> записываются в привязки к переменным, после чего начинается следующая итерация.

Если <проверка> вычисляется как истина, то <выражения> вычисляются слева направо и возвращается(возвращаются) значение(я) последнего <выражения>. Если нет ни одного <выражения>, то значение выражения `do` не определено.

Область привязки <переменной> состоит из всего выражения `do`, за исключением <инициализации>. Если <переменная> появляется более одного раза в списке переменных `do`, возникает ошибка.

<Шаг> может быть опущен, в случае чего результат будет тот же, что и в случае если вместо (<переменная> <инициализация>) будет написано (<переменная> <инициализация> <переменная>).

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i i))                  ===>  #(0 1 2 3 4)

(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (sum 0 (+ sum (car x))))
      ((null? x) sum)))                     ===>  25
```

библиотечный синтаксис: `(let` *<переменная>* *<привязки>* *<тело>*`)`

«Именованный `let`» это вариант синтаксиса `let`, который обеспечивает более общий конструкт цикла, чем `do` и может также использоваться для выражения рекурсий. Он имеет тот же синтаксис и семантику, что и обычный `let`, за исключением того, что <переменная> привязывается вместе с <телом> к процедуре, чьи формальные параметры привязаны к переменным и телом которых является <тело>. Таким образом, выполнение <тела> повторить вызвав процедуру, названную <переменной>.

```
(let loop ((numbers '(3 -2 1 6 -5))
           (nonneg '())
           (neg '()))
  (cond ((null? numbers) (list nonneg neg))
        ((>= (car numbers) 0)
         (loop (cdr numbers)
               (cons (car numbers) nonneg)
               neg))
        ((< (car numbers) 0)
         (loop (cdr numbers)
               nonneg
               (cons (car numbers) neg)))))
                ===>  ((6 1 3) (-5 -2))
```

4.2.5 Отложенное вычисление

библиотечный синтаксис: `(delay <выражение>)`

Конструкт `delay` используется вместе с процедурой `force` для реализации *отложенного вычисления* или *вызова по необходимости*. `(delay` <выражение>`)` возвращает объект, называемый обещание, который в некоторый момент в будущем может быть вызван для выполнения <выражения>, и возвращающий итоговое значение. Результат <выражения>, возвращающий множество значение не определён.

Смотри описание процедуры `force` (раздел 6.4) для более полного описания процедуры `delay`.

4.2.6 Квази цитирование

синтаксис: `(quasiquote <шаблон кц>)`
синтаксис: `` `<шаблон кц> ``
// кц — квази цитирование

Выражение «обратного апострофа» или «квази цитирования» полезны для конструирования структуры списка или вектора, когда о желаемой структуре известно заранее большая часть, но не все. Если в <шаблоне КЦ> нет запятых, то результат выполнения <шаблона КЦ> эквивалентен результату выполнения „<шаблон КЦ>. С другой стороны, если вместе с <шаблоном КЦ> используется запятая, выражение, которое следует после запятой, выполняется («раскавычивается») и его результат вставляется в структуру вместо данной запятой и выражения. Если запятая появляется сразу после собаки (@), то последующее выражение должно вычисляться как список; открытие и закрытие круглых скобок списка в таком случае «отбрасывается» и элементы списка вставляются вместо последовательности запятых и собак (@). Запятая вместе с собакой может использоваться только вместе со списком или вектором <шаблонов кц>.

```
`(list ,(+ 1 2) 4)          ===>  (list 3 4)
(let ((name 'a)) \` (list ,name ',name))
                ===>  (list a (quote a))
`(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)
                ===>  (a 3 4 5 6 b)
`(( foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
                ===>  ((foo 7) . cons)
`#(10 5 ,(sqrt 4) ,@(map sqrt '(16 9)) 8)
                ===>  #(10 5 2 4 3 8)
```

Формы квази цитирования могут вкладываться. Подстановки выполняются только для тех компонентов, необрамленных в кавычки, которые появляются в том же уровне вложения, что и самое глубокое цитирование. Уровень вложения увеличивается на единицу внутри каждого успешного квази цитирования, и уменьшается на единицу внутри каждого «раскавычивания»

```
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
                ===>  (a `(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 'x)
      (name2 'y))
  `(a `(b ,,name1 ,',name2 d) e))
                ===>  (a `(b ,x ,'y d) e)
```

Обозначения <шаблон кц> и `(quasiquote` <шаблон кц>`)` одинаковы во всех отношениях. `,` <выражение> идентично `(unqoute` <выражение>`)`. Внешний синтаксис сгенерированный `write` для списка из двух элементов, `car` которого является одним из этих символов, может различаться в зависимости от реализации.

```
(quasiquote (list (unquote (+ 1 2)) 4))
                ===>  (list 3 4)
'(quasiquote (list (unquote (+ 1 2)) 4))
                ===>  `(list ,(+ 1 2) 4)
     i.e., (quasiquote (list (unquote (+ 1 2)) 4))
```

В случае если какой либо из символов `quasiqoute`, `unqoute` или `unquote-splicing` располагается внутри <шаблона кц>, поведение не определено, иначе все происходит по схеме описанной выше.

4.3 Макросы

Программы Scheme могут определять и использовать новые производные типы выражений, называемые макросы.
Программно определённые типы выражений имеют синтаксис:

`(`<ключевое слово> <элемент данных> `...)`

где <ключевым словом> является идентификатор, который однозначно определяет тип выражения. Этот идентификатор называется *синтаксическим ключевым словом* или просто *ключевым словом* макроса. Число <элементов данных>, и их синтаксис, зависит от типа выражения.

Каждый экземпляр макроса называется *использование* макроса. Множество правил, которые определяют как использование макроса расшифровывается в более примитивное выражение называется *преобразователем* макроса.

Конструкция макроса состоит из двух частей:

- Множество выражений используемых для установления того, что данные идентификаторы являются ключевыми словами макроса, связывает их с преобразователями макроса и управляют областью видимости в которой определён макрос, и
- язык шаблонов для определения преобразователей макроса.

Синтаксическое ключевое слово макроса может перекрывать привязки переменных, а привязки локальных переменных могут перекрывать привязки ключевых слов. Все макросы определённые с использованием языка шаблонов являются “гигиеническими” и “референциально прозрачными” и тем самым сохраняют в Scheme лексическую область видимости [14, 15, 2, 7, 9]:

- Если преобразователь макроса вставляет привязку к идентификатору (переменную или ключевое слово), идентификатор в итоге будет переименован во всей своей области видимости для того чтобы избежать конфликтов с другими определениями; смотри раздел 5.2.
- Если преобразователь макроса вставляет свободную ссылку на идентификатор, ссылка относится к привязке, которая была видна в том месте, где был определён преобразователь, несмотря на все локальные привязки, которые могу окружать использование макроса.

4.3.1 Связующие конструкты для синтаксических ключевых слов

`Let-syntax` и `letrec-syntax` аналогичны `let` и `letrec`, но они привязывают синтаксические ключевые слова к преобразователям макросов вместо привязки переменных к локациям, которые содержат значения. Синтаксические ключевые слова могут также быть привязаны к верхнему уровню; смотри раздел 5.3

синтаксис: `(let-syntax` <привязки> <тело>`)`

Синтаксис: <Привязки> имеют форму

`((`<ключевое слово> <определение преобразователя>`) ...)`

Каждое <ключевое слово> является идентификатором, каждое <определение преобразователя> это экземпляр `syntax-rules`, а <тело> должно быть последовательностью одного или более выражений. Если <ключевое слово> используется более одного раза в списке ключевых слов, который будут привязаны, то возникнет ошибка.

Семантика: <Тело> является расширением синтаксической среды, полученное добавлением к синтаксической среде выражение `let-syntax` вместе с макросами, ключевые слова которых являются <ключевыми словами>, привязанными к определенным преобразователям. Каждая привязка <ключевого слова>, содержит область <тела>.

```
(let-syntax ((when (syntax-rules ()
                     ((when test stmt1 stmt2 ...)
                      (if test
                          (begin stmt1
                                 stmt2 ...))))))
  (let ((if #t))
    (when if (set! if 'now))
    if))                                    ===>  now

(let ((x 'outer))
  (let-syntax ((m (syntax-rules () ((m) x))))
    (let ((x 'inner))
      (m))))                                ===>  outer
```

синтаксис: `(letrec-syntax` <привязки> <тело>`)`

Синтаксис: Такой же как и для `let-syntax`

Семантика: <Тело> является расширением синтаксической среды, полученное добавлением к синтаксической среде выражение `let-syntax` вместе с макросами, ключевые слова которых являются <ключевыми словами>, привязанными к определенным преобразователям. Каждая привязка <ключевого слова> содержит <привязки> также как и <тело> вместе со своей областью, поэтому преобразователи могут транслировать выражение в использования макросов, представленных выражением `lertec-syntax`.

```
(letrec-syntax
  ((my-or (syntax-rules ()
            ((my-or) #f)
            ((my-or e) e)
            ((my-or e1 e2 ...)
             (let ((temp e1))
               (if temp
                   temp
                   (my-or e2 ...)))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
        (if even?))
    (my-or x
           (let temp)
           (if y)
           y)))                ===>  7
```

4.3.2 Язык паттернов

<определение преобразователя> имеет следующую форму:

`(syntax-rules` <литералы> <синтаксическое правило> `...)`

Синтаксис: <Литералы> это список идентификаторов, и каждое <синтаксическое правило> должно иметь форму

(<паттерн> <шаблон>)

<Паттерн> в <синтаксическом правиле> это список <паттернов>, который начинается с ключевого слова макроса.

<Паттерном> является либо идентификатор, константа, либо одно из следующего

```
(<паттерн> ...)
(<паттерн> <паттерн> ... . <паттерн>)
(<паттерн> ... <паттерн> <эллипсис>)
#(<паттерн> ...)
#(<паттерн> ... <паттерн> <эллипсис>)
```

и шаблоном является идентификатор, константа, либо одно из следующего

```
(<элемент> ...)
(<элемент> <элемент> ... . <шаблон>)
#(<элемент> ...)
```

где <элемент> это <шаблон>, который может следовать за <эллипсисом>, а <эллипсисом> является идентификатор «…» (который не может использоваться как идентификатор в шаблоне или паттерне).

Семантика: Экземпляр `syntax-rules` создаёт новый преобразователь макроса, связывая последовательность гигиенических правил перезаписи. Использование макросов, ключевые слова которых связаны с преобразователем, которые определяются с помощью `syntax-rules`, сравнивается с паттернами, содержащимися в <синтаксических правилах>, начиная с самого левого <синтаксического правила>. Как только совпадение найдено, использование макроса преобразуется гигиенически в соответствии с шаблоном.

Идентификатор, который появляется в паттерне <синтаксического правила> является *переменной паттерна*, пока он является ключевым словом, с которого начинается паттерн, перечисленный в <литералах> или если он является идентификатором «…». Переменным паттерна соответствуют произвольные входные элементы, эти элементы используются для ссылки на элементы входа в шаблон. Если некоторая переменная паттерна появляется более одного раза в <паттерне>, возникнет ошибка.

Ключевое слово в начале паттерна в <синтаксическом правиле> не включается в сопоставление и не считается переменной паттерна или буквенным идентификатором.

> *Обоснование:* Область видимости ключевого слова определяется выражением или синтаксическим определением, которое связывает его с соответствущим преобразователем макроса. Если ключевое слово было значением переменной паттерна или буквенного идентификатора, то шаблон, который следует за паттерном должен содержаться в области видимости, несмотря на то, с помощью чего привязано слово `let-syntax` или `letrec-syntax`.

Идентификаторы которые возникают в <литералах> интерпретируется, как буквенные идентификаторы, которые сопоставляются с соответствующими подформами входа. Подформа во входе сопоставляется с буквенным идентификатором, если и только если он является идентификатором, и либо его вхождение в выражение макроса и вхождение в определение макроса имеет одну и ту же лексическую привязку, либо два идентификатора эквивалентны друг другу и оба не имеют лексической привязки.

Подпаттерн, который следует за `...` может соответствовать нулевому или более числу элементов входа. Если `...` появиться в <литералах>, возникнет ошибка. Также как и паттерн идентификатор `...` должен следовать за последним элементом непустой последовательности подпаттернов.

Более формально, входная форма F сопоставляется с паттерном P если и только если:

- *P* не литеральный идентификатор; или
- *P* литеральный идентификатор и *F* идентификатор с одной и той же привязкой; или
- *P* является списком (*P*$_1$ ... *P*$_n$) и *F* список из *n* форм, которые соответствуют *P*$_1$ .. *P*$_n$ соответственно; или
- *P* это неподходящий список (*P*$_1$ *P*$_2$ ... *P*$_n$. *P*$_{n+1}$) и *F* это список или неподходящий список из n или более форм, которые соответствуют *P*$_1$ … *P*$_n$, и *n*-ый “`cdr`” которых соответствует *P*$_{n+1}$; или
- P это вектор в форме `#(`*P*$_1$ ... *P*$_n$ *P*$_{n+1}$ `<эллипсис>)`, где <эллипсис> это идентификатор `...`, а *F* это вектор *n* или более форм, первые *n* элементов которого соответствуют *P*$_1$ … *P*$_n$, и каждый оставшийся элемент *F* соответствует *P*$_{n+1}$; или
- *P* это элемент данных и *F* эквивалентно *P* в смысле процедуры `equal?`.

Использование ключевого слова макроса в области видимости привязки в выражении, которое не соответствует какому-либо паттерну является ошибкой.

Когда использование макроса преобразуется в соответствии с шаблоном соответствующего <синтаксического правила>, переменные паттерна, которые встречаются в шаблоне заменяются подформами, они сопоставляются во входе. Переменные паттерна, которые встречаются в подпатернах, следующие за одним или более экземплярами идентификатора `...` разрешены только в подшаблонах, которые следуют за множеством экземпляров `...`. Они заменяются в выходе всеми подформами, которые соответствуют подформам на входе, распределяясь, как указано. Если выход не может быть сформирован, как определено выше, возникнет ошибка.

Идентификаторы которые используются в шаблонах, но не являются переменными паттерна или идентификатором `...` вставляются в выход литеральных идентификаторов. Если литеральный идентификатор вставляется, как свободный идентификатор, то он относиться к привязке этого идентификатора вместе с областью видимости экземпляра syntax-rules. Если литеральный идентификатор вставляется как связный идентификатор, то в результате он переименовывается, чтобы предотвратить случайные захваты свободных идентификаторов.

Например, если `let` и `cond` определены также как в разделе 7.3, то они являются гигиеническими (как и требуется) и следующее выражение не является ошибкой.

```
(let ((=> #f))
  (cond (#t => 'ok)))                    ===> ok
```

Преобразователь макроса для `cond` распознает `=>` как локальную переменную, а значит выражение, и не распознает как идентификатор верхнего уровня `=>`, который преобразователь макроса обрабатывает как синтаксическое ключевое слово. Таким образом данный пример разворачивается в

```
(let ((=> #f))
  (if #t (begin => 'ok)))
```

вместо

```
(let ((=> #f))
  (if #t (begin => 'ok)))
```

который получится в результате неправильного вызова процедуры.

Глава 5

Структура программы

5.1 Программы

Программа Scheme состоит из последовательности выражений, определений и синтаксических определений. Выражения описаны в главе 4; определения и синтаксические определение являются предметом остальной части данной главы.

Программы обычна хранятся в файле или вводятся интерактивно в запущенную систему Scheme, хотя возможны и другие варианты; вопросы пользовательского интерфейса не входят в данный доклад. (В самом деле, Scheme может быть также полезен, как система обозначений для выражения вычислительных методов даже при отсутствии механической имплементации.)

Определения и синтаксические определения возникающие на верхнем уровне программы могут быть интерпретированы декларативно. Они вызывают привязки, которые будут созданы на верхнем уровне окружения или модифицировать значение существующих привязок верхнего уровня. Выражения фигурирующие на верхнем уровне программы интерпретируются императивно; они выполняются, когда программа вызывается или загружается, и обычно выполняют некоторые виды инициализации.

На верхнем уровне программы `(begin` <форма$_1$> `...)` эквивалентна последовательности выражений, определений и определений синтаксиса, которые формируются тело выражения `begin`.

5.2 Определения

Определения являются корректными в некоторых, но не всех, контекстах, где разрешены выражения. Они корректны только на верхнем уровне <программы> и в начале <тела>.

Определение может иметь одну из следующих форм:

- `(define` <переменная> <выражение>`)`
- `(define` (<переменная> <формальные аргументы>) <тело>`)`

  <Формальные определения> могут быть либо последовательностью нулевого или более числа переменных, либо последовательностью одной или более переменной, следующих за разделённой пробелом точкой и другой переменной (такой как лямбда выражение). Данная форма эквивалентна

  ```
  (define <переменная>
      (lambda (<формальные аргументы>) <тело>)).
  ```

- `(define (<переменная> . <формальные аргументы>) <тело>)`

  <Формальным аргументом> должна быть единственная переменная. Данная форма эквивалентна

```
(define <переменная>
    (lambda <формальные аргументы> <тело>))
```

5.2.1 Определения верхнего уровня

На верхнем уровне программы, определение

```
(define <переменная> <выражение>)
```

по существу имеет тот эффект, что и выражение присваивания

```
(set! <переменная> <выражение>)
```

если <переменная> привязана. Однако, если <переменная> не связана, определение будет привязывать <переменные> к новой локации перед выполнением присваивания, в тоже время, при применении set! к несвязной переменной возникнет ошибка.

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3)                                ===>  6
(define first car)
(first '(1 2))                          ===>  1
```

Некоторые имплементации Scheme используют начальное окружение, в котором все возможные переменные переменные привязаны к локациям, большинство из которых содержат неопределённые значения. Определения верхнего уровня в таких имплементациях полностью эквивалентны присваиваниям.

5.2.2 Внутренние определения

Определения могут встречаться в начале <тела> (которое является телом выражений `lambda`, `let`, `let*`, `letrec`, `let-syntax` или `letrec-syntax` или определением соответствующей формы). Такие определения называются *внутренними определениями*, как противоположность определений верхнего уровня, описанных выше. Переменная определённая внутренним определением локальна внутри <тела>. То есть <переменная> является привязанной, а не присвоенной, и областью привязки является все <тело>. Например,

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))                        ===>  45
```

<Тело> содержащее внутренние определения может всегда быть преобразовано в полностью эквивалентное выражение `letrec`. Например, выражение `let` в примере выше, эквивалентное

```
(let ((x 5))
  (letrec ((foo (lambda (y) (bar x y)))
           (bar (lambda (a b) (+ (* a b) a))))
    (foo (+ x 3))))
```

Так же как и для эквивалентного выражения `letrec`, должно быть возможно вычислять каждое <выражение> всех внутренних определений <тела> без присваивания соответствующего значения какой-либо определённой <переменной>.

Везде, где может появиться внутреннее определение, выражение `(begin` <определение$_1$> `...)` эквивалентно последовательности определений, которые формируют тело `begin`.

5.3 Определения синтаксиса

Определения синтаксиса могут использоваться только на верхнем уровне <программы>. Они имеют следующую форму:

```
(define-syntax <ключевое слово> <определение преобразователя>)
```

<Ключевым словом> является идентификатор, а <определением преобразователя> должен быть экземпляр `syntax-rules`. Окружение синтаксиса верхнего уровня расширяется привязкой <ключевого слова> к определённому преобразователю.

Не существует аналога `define-syntax` для внутренних определений.

Хотя макросы могут быть развёрнуты в определения и определения синтаксиса в некотором контексте, который разрешён для них, возникнет ошибка, если определение или определение синтаксиса перекрывает синтаксическое ключевое слово, значение которого необходимо для определения того является ли некоторая форма в группе форм, которые содержат экранирующее определение, по факту определением, или если определение внутреннее, необходимо определить связь между группой и выражением, которое следует за группой.

```
(define define 3)

(begin (define begin list))

(let-syntax
  ((foo (syntax-rules ()
          ((foo (proc args ...) body ...)
           (define proc
             (lambda (args ...)
               body ...))))))
  (let ((x 3))
    (foo (plus x y) (+ x y))
    (define foo x)
    (plus foo x)))
```

Глава 6

Стандартны процедуры

Это глава описывает встроенные процедуры Scheme. Начальное (или «верхнего уровня») окружение Scheme запускается с некоторым количеством переменных привязанных к локациям, содержащим полезные значения, большинство из которых являются примитивными процедурами, которые манипулируют данными. Например, переменная `abs`, привязана к (локации изначально содержащей) процедуре одного аргумента, которая вычисляет модуль значения числа, а переменная + привязана к процедуре, которая вычисляет суммы. Встроенные процедуры, которые могут быть легко написаны в терминах других встроенных процедур помечаются как «библиотечные процедуры».

Программа может использовать определения верхнего уровня для привязки к какой-либо переменной. Она может впоследствии изменять эти привязки с помощью присваивания (смотри 4.1.6). Эти операции не изменяют поведения встроенных процедур Scheme. Изменение какой-либо привязки верхнего уровня, которая не представлена определением, может привести к непредсказуемому поведению встроенных процедур.

6.1 Предикаты эквивалентности

Предикат это процедура, которая всегда возвращает булево значение (#t или #f). *Предикатом эквивалентности* является вычислительный аналог математического отношения эквивалентности (оно симметрично, рефлексивно и транзитивно). Из всех предикатов эквивалентности описанных в это разделе, `eq?` является самым лучшим и понятным, а `equal?` является самым грубым. `Eqv?` чуть менее понятный чем `eq?`.

процедура: `(eqv?` obj_1 obj_2`)`

Процедура `eqv?` определяет полезное отношение эквивалентности между объектами. Вкратце, она возвращает `#t`, если obj_1 и obj_2, как правило, рассматриваются как один и тот же объект. Это отношение остаётся слегка не раскрытым, однако следующая неполная спецификация `eqv?` выполняется для всех имплементаций Scheme.

Процедура `eqv?` возвращает `#t`, если:

- obj_1 и obj_2 оба являются `#t` или оба `#f`.
- obj_1 и obj_2 символы и

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
            ===>  #t
```

> *Замечание:* Предполагается, что ни obj_1, ни obj_2 не являются «непечатаемыми символами», что подразумевает раздел 6.3.3. Данный отчёт не предполагает определение поведения `eqv?` для имплементационно-зависимых расширений.

- obj_1 и obj_2 оба являются числами, и численно равны (смотри =, раздел 6.2), и либо оба - точные, либо - приближенные.
- obj_1 и obj_2 являются одинаковыми символами в смысле процедуры `char=?` (раздел 6.3.4).
- obj_1 и obj_2 оба являются пустым списком.
- obj_1 и obj_2 являются парами, векторами или строками, которые указывают на одну и ту же локацию в памяти (раздел 3.4).
- obj_1 и obj_2 являются процедурами, теги локаций которых, совпадают (раздел 4.1.4).

Процедура `eqv?` возвращает `#f`, если:

- obj_1 и obj_2 разных типов (раздел 3.2).
- один из obj_1 и obj_2 имеет значение `#t`, а другой - `#f`.
- obj_1 и obj_2 - символы, но

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
            ===>  #f
```

- один из obj_1 и obj_2 является точным числом, а другой — неточным.
- obj_1 и obj_2 являются числами для которых процедура = возвращает `#f`.
- obj_1 и obj_2 являются символами, для которых процедура `char=?`. возвращает `#f`.
- один из obj_1 и obj_2 является пустым списком, а другой — нет.
- obj_1 и obj_2 это пары, векторы или строки, которые указывают на разные локации.
- obj_1 и obj_2 являются процедурами, которые могут вести себя по разному (возвращают разные значения или имеют различный побочный эффект) для некоторых аргументов.

```
(eqv? 'a 'a)                                ===>  #t
(eqv? 'a 'b)                                ===>  #f
(eqv? 2 2)                                  ===>  #t
(eqv? '() '())                              ===>  #t
(eqv? 100000000 100000000)                  ===>  #t
(eqv? (cons 1 2) (cons 1 2))                ===>  #f
(eqv? (lambda () 1)
      (lambda () 2))                        ===>  #f
(eqv? #f 'nil)                             ===>  #f
(let ((p (lambda (x) x)))
  (eqv? p p))                                    ===>  #t
```

Следующие примеры иллюстрируют случаи, в которых вышеперечисленные правила не полностью определяют поведение `eqv?`. Все то же самое может быть сказано о случаях в которых значение возвращаемое `eqv?` должно быть булевым.

```
(eqv? "" "")                    ===>  unspecified
(eqv? '#() '#())                ===>  unspecified
(eqv? (lambda (x) x)
      (lambda (x) x))           ===>  unspecified
(eqv? (lambda (x) x)
      (lambda (y) y))           ===>  unspecified
```

Следующее множество примеров показывают использование `eqv?` с процедурами, которые имеют локальное состояние. `Gen-counter` должен возвращать различные процедуры каждый раз, поскольку у каждой процедуры есть свой собственный внутренний счётчик. `Gen-closer`, с другой тороны, каждый раз возвращает эквивалентные процедуры, так как локальное состояние не влияет на значение или на побочный эффект процедур.

```
(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-counter)))
  (eqv? g g))                    ===>  #t
(eqv? (gen-counter) (gen-counter))
                                 ===>  #f
(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-loser)))
  (eqv? g g))                    ===>  #t
(eqv? (gen-loser) (gen-loser))
                                 ===>  unspecified

(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
         (g (lambda () (if (eqv? f g) 'both 'g))))
  (eqv? f g))
                                 ===>  unspecified

(letrec ((f (lambda () (if (eqv? f g) 'f 'both)))
         (g (lambda () (if (eqv? f g) 'g 'both))))
  (eqv? f g))
                                 ===>  #f
```

Поскольку при изменении константных объектов (которые возвращаются литеральными выражениями) возникает ошибка, имплементациям разрешено, хоть и не требуется, разделять структуру между константами, где необходимо. Таким образом, значение `eqv?` для констант иногда являются имплементационно зависимыми.

```
(eqv? '(a) '(a))                     ===>  unspecified
(eqv? "a" "a")                       ===>  unspecified
(eqv? '(b) (cdr '(a b)))             ===>  unspecified
(let ((x '(a)))
  (eqv? x x))                        ===>  #t
```

> *Обоснование:* Определение `eqv?` упомянутое выше предоставляет имплементациям широкую свободу действий для процедур и литералов; имплементации могут обнаруживать или не обнаруживать те, две процедуры или два литерала, которые эквивалентны друг другу, а также могут принимать решения объединять или не объединять в одно представление два эквивалентных объекта, используя один и тот же указатель или частичный указатель для представления обоих объектов.

процедура: (eq? obj_1 obj_2)

Eq? то же самое что и eqv? за исключением того, что в некоторых случаях он способен распознавать различия лучше, чем их распознает eqv?.

Гарантировано, что Eq? и eqv? ведут себя одинаково для символов, булевых переменных, пустого списка, пар, процедур, непустых строк и непустых векторов. Поведение eq? для чисел и символов зависит от имплементации, но оно всегда будет возвращать либо истину, либо ложь, и будет возвращать истинное значение только, когда eqv? также возвращать истину. Поведение eq? может также отличаться от eqv? на пустых векторах и пустых строках.

```
(eq? 'a 'a)                              ===>  #t
(eq? '(a) '(a))                          ===>  unspecified
(eq? (list 'a) (list 'a))                ===>  #f
(eq? "a" "a")                            ===>  unspecified
(eq? "" "")                              ===>  unspecified
(eq? '() '())                            ===>  #t
(eq? 2 2)                                ===>  unspecified
(eq? #\A #\A)          ===>  unspecified
(eq? car car)                            ===>  #t
(let ((n (+ 2 3)))
  (eq? n n))               ===>  unspecified
(let ((x '(a)))
  (eq? x x))               ===>  #t
(let ((x '#()))
  (eq? x x))               ===>  #t
(let ((p (lambda (x) x)))
  (eq? p p))               ===>  #t
```

> *Обоснование:* Обычно возможно использовать eq? куда более эффективней чем eqv?, например, для простого сравнения указателей, вместо некоторых других способов более сложных операций. Одна из причин это то, что может оказаться невозможным вычисление значения eqv? для двух чисел за константное время, в то время как eq?, реализованный через сравнение указателей, всегда выполняется за константное время. Eq? может использоваться также как eqv? в приложениях, которые используют процедуры для имплементации объектов с внутренним состоянием, поскольку он подчиняется тем же ограничениям, что и eqv?.

библиотечная процедура: (equal? obj_1 obj_2)

Equal? Рекурсивно сравнивает содержимое пар, векторов, строк, применяя eqv? к объектам другого типа, таким как числа и символы. Практическим принципом этого предиката является то, что обычно объекты считаются эквивалентными, если они печатаются одинаково. Equal? может не завершиться, если его аргументами являются циклические структуры данных.

```
(equal? 'a 'a)                           ===>  #t
(equal? '(a) '(a))                       ===>  #t
(equal? '(a (b) c)
        '(a (b) c))                      ===>  #t
(equal? "abc" "abc")                     ===>  #t
(equal? 2 2)                             ===>  #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))              ===>  #t
(equal? (lambda (x) x)
        (lambda (y) y))          ===>  unspecified
```

6.2 Числа

Сообщество Lisp традиционно пренебрегает численными вычислениями. До Common Lisp не было аккуратно продуманной стратегии для организации численного вычисления, и за исключением системы MacLisp [20], прилагалось недостаточно усилий для эффективного выполнения численного кода. Данный доклад признает превосходную работу сообщества Common Lisp и принимает их рекомендации. В некотором смысле, данный доклад упрощает и обобщает их предложения в манере соответствующей целями Scheme.

Важно отличать математические числа от чисел Scheme, которые только пытаются смоделировать их, для реализации Scheme чисел используется машинное представление, а условные обозначения используются для написания их. В данном докладе используются типы *number*, *complex*, *real*, *rational* и *integer* по отношению и к математическим, и к Scheme числам. Машинные представления такие, как с фиксированной точкой и с плавающей точкой называют как *fixnum* и *flonum*.

6.2.1 Численные типы

Математически, числа могут быть организованы в башню подтипов, в которой каждый уровень является подмножеством уровня выше последнего:

```
number
    complex
    real
    rational
    integer
```

Например, число 3 это целое. Более того, 3 также является рациональным, вещественным и комплексным. Тоже самое выполняется и для Scheme чисел, которые моделируют число 3. Для Scheme чисел, эти типы определяются предикатами `number?`, `complex?`, `real?`, `rational?` и `integer?`.

Не существует простого отношения между числовыми типами и их представлениями внутри компьютера. Хотя большинство имплементаций Scheme предлагают как минимум два различных представления 3, эти два различных представления определяют одно и то же целое.

Численные операции Scheme обрабатывают числа, как абстрактные данные, настолько не зависимо от их представления, насколько возможно. Хотя реализации Scheme могут использовать fixnum (числа с фиксированной запятой), flonum (числа с плавающей запятой), и возможно другие представления для чисел, для обычного программиста, пишущего простую программу, это может быть не очевидно.

Однако необходимо различать числа, которые представляются точно от тех, которые могут таковыми не являться. Например, должны быть точно известны индексы в структурах данных, поскольку должны быть известны некоторые коэффициенты многочлена в символической алгебраической системе. С другой стороны, результаты вычислений по существу неточны, а иррациональные числа могут приближаться рациональными и более того неточными приближениями. Для того чтобы отловить случаи использования неточных чисел, где необходимы точные, в Scheme есть чёткое разделение на точные и неточные числа. Это различие ортогонально размерности типа.

6.2.2 Точность

Числа в Scheme являются либо точными, либо неточными. Число является точным, если оно записано как точная константа или получено из точных чисел с использованием только точных операций. Число является неточным, если оно записано как неточная константа и если оно получено с использованием неточных ингредиентов, или если оно получено с использованием неточных операций. Таким образом неточность это «заразительное» свойство числа. Если две имплементации выдают точные результаты для вычислений, которые не включают неточных промежуточных результатов, два конечных результата будут математически эквивалентными. В общем случае это правило не выполняется для вычислений, включающих в себя неточные числа, поскольку могут использоваться методы приближения такие, как арифметика с плавающей точкой, но каждая имплементация обязана сделать результат как можно ближе к математически идеальному результату.

Рациональные операции такие, как + должны всегда выдавать точные результаты при передаче точных аргументов. Если операция неспособна выдать точный результат, то она либо может сообщить о нарушении ограничения имплементации, либо оставить свой результат в неточной форме. Смотри раздел 6.2.3.

За исключением `inexact->exact`, операции описанные в данном разделе должны в общем случае возвращать неточные результаты, когда передаются какие-либо неточные аргументы. Однако, операция может возвращать точный результат, если может доказать, что на значение результата не влияет неточность аргументов. Например, умножение любого числа на точный ноль может выдавать в качестве результата точный ноль, даже если другие аргументы являются неточными.

6.2.3 Ограничения имплементации

Имплементации Scheme не требуют от имплементаций полной башни подтипов, описанной в разделе 6.2.1, но имплементации должны реализовывать связное подмножества, не противоречащее целям имплементации и духу языка Scheme. Например, иимплементация в которой все числа являются вещественными может быть весьма полезна.

Имплементации могут также допускать только ограниченный диапазон чисел какого-либо типа, что является пунктом списка требований данного раздела. Допустимый диапазон для точных чисел некоторого типа может отличаться от допустимого диапазона для неточных чисел данного типа. Например, имплементации, которые используют числа с плавающей запятой для представления всех неточных вещественных чисел могут допускать практически неограниченный диапазон точных целых и рациональных чисел, ограничивая при этом диапазон неточных вещественных чисел (а значит, диапазон неточных целых и рациональных) динамическим диапазоном формата чисел с плавающей запятой. Более того, разрывы между представляемыми неточными целыми и рациональными могут оказаться очень большими в имплементациях, в которых возникает ограничения такого диапазона.

Любая имплементация Scheme должна поддерживать точные целые по всему диапазону чисел, которые могут быть использованы для индексов списка, векторов и строк или могут быть результатам вычисления длины списка, вектора или строки. Процедуры `length`, `vector-length`, `string-length` должны возвращать точные целые, и при использовании любого другого, но не точного индекса возникнет ошибка. Более того, любое константное целое использующееся, как диапазон индексов, если выражается с помощью синтаксиса целого точного числа, в самом деле будет считано как целое, несмотря на все ограничения имплементаций, которые могут применяться вне данного диапазона. И наконец, процедуры, перечисленные ниже, будут всегда возвращать в качестве результата точные целые числа, при условии, что все их аргументы являются точными целыми и ожидаемый результат математически представим как точное целое число в рамках данной реализации:

```
+         -           *
quotient  remainder   modulo
max       min         abs
numerator denominator gcd
lcm       floor       ceilng
truncate  round       rationalize
expt
```

В реализациях рекомендуется, но не обязательно, придерживаться точных целых и точных рациональных чисел неограниченного размера и точности, и реализовывать вышеперечисленные процедуры и процедуру `/` таким образом, чтобы они всегда возвращали точные результаты для точных аргументов. Если одна из этих процедур не может предоставить точный результат при передаче ей точных аргументов, то может возникнуть либо сообщение о нарушении ограничений имплементации, либо может её результат может принудительно вернуть неточное число. Такое принуждение позже может привести к ошибке.

Имплементация может использовать числа с плавающей запятой и другие приближенные стратегии представления для неточных чисел. В данном докладе рекомендуется, однако не обязательно то, что IEEE 32-битные и IEEE 64-битные стандарты чисел с плавающей запятой, используются в имплементациях, которые используют представление чисел с плавающей запятой, и те имплементации, которые используют другие представления должны быть больше или равны достижимой точности используемой данные стандарты чисел с плавающей запятой 12.

В частности имплементации, которые используют числа с плавающей запятой должны соответствовать следующим правилам: Результат числа с плавающей запятой должен быть представим с как минимум большей точностью чем точность, которая используются для выражения данной операции любых из неточных аргументов. Желательно (но не обязательно) для потенциально неточных операций таких, как `sqrt`, при применении их к аргументам, выдавать точные результаты всякий раз, когда это возможно (например, для квадратного корня точного числа 4 должно возвращаться точное число 2). Однако, если точное число вычисляется так, что его результат является неточным (как в случае с `sqrt`) , и если результат представим как число с плавающей запятой, то должен использоваться самый точный формат приближения числа с плавающей запятой на сколько это возможно; но если результат представим каким-либо другим способом, то представление должно иметь настолько небольшое приближение насколько допускает формат приближения числа с плавающей запятой.

Несмотря ни на что Scheme, допускает множество уже написанных нотаций для чисел, любая отдельная имплементация может поддерживать только некоторые из них. Например, имплементация в которой все числа являются вещественными не должна поддерживать прямые и диаметрально противоположные нотации для комплексных чисел. Если некоторая имплементация сталкивается с точной численной константой, которая непредставима как точное число, то о она может либо вывести сообщение о нарушении ограничения имплементации, либо может как ни в чем не бывало представить константу константу как неточное число.

6.2.4 Синтаксис числовых констант

Синтаксис написанных представлений для чисел формально описан в разделе 7.1.1. Обратите внимание, данный случай важен для числовых констант.

Число может быть записано в двоичном, восьмеричном иди шестнадцатеричном виде с помощью префикса основания разрядного префикса. Разрядные префиксы это `#b` (двоичный), `#o` (восьмеричный), `#d` (десятичный) и `#x` (шестнадцатеричный). При отсутствии разрядного префикса, число считается представленным в десятичном виде.

Числовая константа может быть определена, как точная или неточная с помощью префикса. Префиксы `#e` для точного, и `#i` для неточного. Префикс точности может появляться перед или после используемого разрядного префикса. Если записанное представление числа имеет префикс точности, константа может быть либо неточной, либо точной. Она является точной, если она содержит точку в десятичной дроби, экспоненту или символ “#” на месте цифры, иначе она является точным. В системах с неточными числами различного приближения может быть полезным указывать точность константы. Для этой цели, численные константы могут записываться с меткой экспоненты, которая указывает желаемую точность неточного представления. Буквы `s`, `f`, `d` и `l` указывают использование приближения `short`, `single`, `double` и `long` соответственно. (Когда имеется меньше четырёх внутренних неточных представлений, спецификации размера 4 отображаются на те которые доступны. Например, имплементация с двумя внутренними представлениями отображают `short` и `single` вместе, также как `long` и `double`.) Кроме того, метка экспоненты e указывает приближение по умолчанию для данной имплементации. Приближение по умолчанию имеет по меньшей мере приближение `double`, но имплементация может попросить разрешить это значение по умолчанию устанавливать пользователем.

```
3.14159265358979F0
        Round to single --- 3.141593
0.6L0
        Extend to long --- .600000000000000
```

6.2.5 Числовые операции

Мы отсылаем читателя к разделу 1.3.3 для краткого изложения соглашений об именовании используемых для указания ограничений типов аргументов числовых подпрограмм. Примеры, которые используются в данном разделе, предполагают что любая числовая константа написанная с использованием точных обозначений представляется, как точное число в полном понимании этого слова. Некоторые примеры также предполагают, что некоторые числовые константы написанные с использованием неточной нотации могут быть представлены без потери точности; неточные константы выбранные таким образом, что имплементации, которые используют числа с плавающей запятой для представления неточных чисел скорее всего будут являться верными.

процедура: `(number? obj)`
процедура: `(complex? obj)`
процедура: `(real? obj)`
процедура: `(rational? obj)`
процедура: `(integer? obj)`

Данные предикаты числовых типов могут применяться к любому типу аргументов, включая те, которые не являются числами. Они возвращают `#t`, если объект именованного типа, иначе они возвращают `#f`. Обычно, если предикат типа данного числа является истинным, то все предикаты типов, включающие данный тип, являются также истинными для данного числа. Соответственно, если предикат типа для числа принимает ложное значение, то все предикаты типа нижнего уровня также принимают значение ложь для данного числа. Если z неточное комплексное число, то `(real? z)` является истиной, если и только если `(zero? (imag-part z))` является истинным. Если x неточное вещественное число, то `(integer? x)` является истинным, если и только если выражение `(= x (round x))` является истинным.

```
(complex? 3+4i)                 ===>  #t
(complex? 3)                    ===>  #t
(real? 3)                       ===>  #t
(real? -2.5+0.0i)               ===>  #t
(real? #e1e10)                  ===>  #t
(rational? 6/10)                ===>  #t
(rational? 6/3)                 ===>  #t
(integer? 3+0i)                 ===>  #t
(integer? 3.0)                  ===>  #t
(integer? 8/4)                  ===>  #t
```

> *Замечание:* Поведение этих предикатов типов для неточных чисел являются ненадёжным, поскольку любая неточность может повлиять на результат.
>
> *Замечание:* Во многих имплементациях процедура `rational?` вернёт тоже что и `real?`, а процедура `complex?` вернёт тоже самое, что `number?`, но нетипичные имплеменатции могут представлять нерациональные числа точно или могут расширить систему счисления для поддержки некоторых видов не комплексных чисел.

процедура: `(exact? z)`
процедура: `(inexact? z)`

Вышеперечисленные предикаты обеспечивают количественную проверку точности. Для любого числа в Scheme, в точности один из данных предикатов является истинным.

процедура: `(= z1 z2 z3 ...)`
процедура: `(< x1 x2 x3 ...)`
процедура: `(> x1 x2 x3 ...)`
процедура: `(<= x1 x2 x3 ...)`
процедура: `(>= x1 x2 x3 ...)`

Эти процедуры возвращают `#t`, если их аргументы являются (соответственно): равными, монотонно возрастающими, монотонно убывающими, монотонно не убывающими или монотонно не возрастающими.

Данные предикаты должны быть транзитивными.

> *Замечание:* Традиционные имплементации данных предикатов в Lisp-подобных языках не являются транзитивными.
>
> *Замечание:* Хоть и сравнение неточных чисел с помощью данных предикатов не является ошибкой, результат может быть ненадёжным, поскольку небольшая неточность может повлиять на результат; это особенно сказывается на процедурах `=` и `zero?`. Когда возникают сомнения обратитесь к числовому обозревателю.

библиотечная процедура: `(zero? z)`
библиотечная процедура: `(positive? x)`
библиотечная процедура: `(negative? x)`
библиотечная процедура: `(odd? n)`
библиотечная процедура: `(even? n)`

Числовые предикаты перечисленные выше проверяют число на конкретное свойство, возвращая `#t` или `#f`. Смотри замечание выше.

библиотечная процедура: `(max` x_1 x_2 `...)`
библиотечная процедура: `(min` x_1 x_2 `...)`

Данные процедуры возвращают максимум или минимум своих аргументов.

```
(max 3 4)                    ===>  4    ; exact
(max 3.9 4)                  ===>  4.0  ; inexact
```

Замечание: Если какой либо аргумент является неточным, то результат также будет неточным (если процедура не сможет доказать, что неточность не настолько большая, чтобы повлиять на результат, который возможен только в случае нетипичных реализаций). Если `min` или `max` используется для сравнения чисел различной точности, и численное значение результата не может быть представлено как неточное число без потери точности, то процедура может сообщить о нарушении ограничения имплементации.

процедура: `(+` z_1 `...)`
процедура: `(*` z_1 `...)`

Данные процедуры возвращают сумму или произведение своих аргументов.

```
(+ 3 4)                      ===>  7
(+ 3)                        ===>  3
(+)                          ===>  0
(* 4)                        ===>  4
(*)                          ===>  1
```

процедура: `(-` z_1 z_2`)`
процедура: `(-` z`)`
необязательная процедура: `(-` z_1 z_2 `...)`
процедура: `(/` z_1 z_2`)`
процедура: `(/` z`)`
необязательная процедура: `(/` z_1 z_2 `...)`

В случае с двумя или более аргументами, данные процедуры возвращают разность или частное аргументов, ассоциативные слева. Если передаётся один аргумент, они возвращают аддитивную или мультипликативную инверсию этого аргумента.

```
(- 3 4)                      ===>  -1
(- 3 4 5)                    ===>  -6
(- 3)                        ===>  -3
(/ 3 4 5)                    ===>  3/20
(/ 3)                        ===>  1/3
```

библиотечная процедура: `(abs` x`)`

`Abs` возвращает модуль значения аргумента.

```
(abs -7)                     ===>  7
```

процедура: `(quotient` n_1 n_2`)`
процедура: `(remainder` n_1 n_2`)`
процедура: `(modulo` n_1 n_2`)`

Эти процедуры реализуют теоретико-числовое (целое) деление. n_2 должно быть ненулевым. Все три процедуры возвращают целые числа. Если n_1/n_2 целое:

```
(quotient n1 n2)          ===> n1/n2
(remainder n1 n2)         ===> 0
(modulo n1 n2)            ===> 0
```

Если n_1/n_2 не целое:

```
(quotient n1 n2)          ===> nq
(remainder n1 n2)         ===> nr
(modulo n1 n2)            ===> nm
```

где n_q это n_1/n_2 округлённое ближе к нулю, $0 < |n_r| < |n_2|$, $0 < |n_m| < |n_2|$, n_r и n_m отличаются от n_1 умножением на n_2, n_r имеет тот же знак, что и n_1, а n_m имеет тот же знак, что и n_2.

Из всего этого можно заключить, что для целых n_1 и n_2 при n_2 неравным 0, выражение

```
(= n1 (+ (* n2 (quotient n1 n2))
         (remainder n1 n2)))
                                    ===>  #t
```

делает все числа участвующие в данном вычислении точными.

```
(modulo 13 4)                   ===>  1
(remainder 13 4)                ===>  1

(modulo -13 4)                  ===>  3
(remainder -13 4)               ===>  -1

(modulo 13 -4)                  ===>  -3
(remainder 13 -4)               ===>  1

(modulo -13 -4)                 ===>  -1
(remainder -13 -4)              ===>  -1

(remainder -13 -4.0)            ===>  -1.0  ; inexact
```

библиотечная процедура: `(gcd` n_1 `...)`
библиотечная процедура: `(lcm` n_1 `...)`

Данные процедуры возвращают наибольший общий делитель и наименьшее общее кратное своих аргументов. Результат всегда неотрицательный.

```
(gcd 32 -36)                    ===>  4
(gcd)                           ===>  0
(lcm 32 -36)                    ===>  288
(lcm 32.0 -36)                  ===>  288.0  ; inexact
(lcm)                           ===>  1
```

процедура: `(numenator q)`
процедура: `(denumenator q)`

Данные процедуры возвращают числитель или знаменатель своих аргументов; результат вычисляется так, как если бы аргумент был представлен как несократимая дробь. Знаменатель всегда положительный. Знаменатель равный 0, считается за 1.

```
(numerator (/ 6 4))          ===>  3
(denominator (/ 6 4))          ===>  2
(denominator
  (exact->inexact (/ 6 4)))         ===> 2.0
```

процедура: `(floor x)`
процедура: `(ceiling x)`
процедура: `(truncate x)`
процедура: `(round x)`

Вышеперечисленные процедуры возвращают целые числа. `Floor` возвращает наибольшее целое число не большее *x*. `Ceiling` возвращает наименьшее целое не большее *x*. `Truncate` возвращает целое, ближайшее к *x*, модуль значения которого не больше чем модуль *x*. `Round` возвращает ближайшее к *x* целое, округляемое к чётному, когда *x* является половиной расстояния между двумя числами.

> *Обоснование:* `Round` округляет до чётного для согласования со стандартным правилом округления, определённым стандартом чисел с плавающей точкой IEEE.
>
> *Замечание:* Если аргумент данных процедур неточен, то результат также будет неточным. Если требуется точное значение, результат должен передаваться процедуре `inexact->exact`.

```
(floor -4.3)                 ===>  -5.0
(ceiling -4.3)               ===>  -4.0
(truncate -4.3)              ===>  -4.0
(round -4.3)                 ===>  -4.0

(floor 3.5)                  ===>  3.0
(ceiling 3.5)                ===>  4.0
(truncate 3.5)               ===>  3.0
(round 3.5)                  ===>  4.0  ; inexact

(round 7/2)                  ===>  4    ; exact
(round 7)                    ===>  7
```

библиотечная процедура `(rationalize x y)`

`Rationalize` возвращает *простейшее* рациональное число, отличающееся от *x* не более чем на *y*. Говорят, что рациональное число r_1 *проще* чем другое рациональное число r_2, если $r_1 = p_1 / q_1$ и $r_2 = p_2 / q_2$ (r_1 и r_2 - несократимые дроби), $|p_1| \leq |p_2|$ и $|q_1| \leq |q_2|$. Так 3/5 проще, чем 4/7. Пусть и не все рациональные сравнимы с помощью данного отношения порядка (например для дробей 2/7 и 3/5), но любой интервал содержит число, которое проще другого числа из того же интервала (число 2/5 лежит между 2/7 и 3/5 в смысле простейших). Обратите внимание, что 0 = 0/1 проще любого другого рационального.

```
(rationalize
  (inexact->exact .3) 1/10)          ===> 1/3    ; exact
(rationalize .3 1/10)                ===> #i1/3  ; inexact
```

процедура: `(exp z)`
процедура: `(log z)`
процедура: `(sin z)`
процедура: `(cos z)`
процедура: `(tan z)`
процедура: `(asin z)`
процедура: `(acos z)`
процедура: `(atan z)`
процедура: `(atan y x)`

Данные процедуры являются частью любой типичной имплементации, которая поддерживает общие вещественные числа; они вычисляют обычные трансцендентные функции. `Log` вычисляет натуральный логарифм *z* (не по основанию десятичного логарифма). `Asin`, `acos` и `atan` вычисляют арксинус (sin^{-1}), арккосинус (cos^{-1}) и арктангенс (tan^{-1}) соответственно. Вариант `atan` с двумя операциями вычисляет `(angle (make-rectangular` *x y*`))` (смотри ниже), в том числе для имплементаций, которые не поддерживают комплексные числа.

$$sin^{-1}\ z = -i\ log\ (i\ z\ +\ (1\ -\ z^2)^{1/2})$$

$$cos^{-1}\ z = \pi\ /\ 2 - sin^{-1}\ z$$

$$tan^{-1}\ z = (log\ (1\ + i\ z)\ -\ log\ (1\ -\ i\ z))\ /\ (2\ i)$$

Определения данные выше вытекают из статьи [27], которая в свою очередь приводит к [19]; обратитесь к данным источникам для более детального обсуждения ветвлений, граничных условий и реализации данных функций. При возможности, данные процедуры возвращают вещественный результат для вещественного аргумента.

процедура: `(sqrt z)`

Возвращает главный квадратный корень *z*. Результатом может быть либо положительная вещественная часть, либо нулевая вещественная часть и неотрицательная мнимая.

процедура: `(expt` z_1 z_2`)`

Возвращает z_1 возведенное в степень z_2. Для $z_1 \neq 0$.

$$z_1^{z_2} = e^{z_2\ log\ z_1}$$

0^z равно 1, если $z = 0$, и 0 в противном случае.

процедура: `(make-rectangular` x_1 x_2`)`
процедура: `(make-polar` x_3 x_4`)`
процедура: `(real-part` z`)`
процедура: `(imag-part` z`)`
процедура: `(magnitude` z`)`
процедура: `(angle` z`)`

Данные процедуры являются частью любой имплементации, которая поддерживает общие комплексные числа. Пусть x_1, x_2, x_3 и x_4 - вещественные числа, а z - комплексное, такие что

$$z = x_1 + x_2\, i = x_3 \cdot e^{i\, x_4}$$

Тогда

```
(make-rectangular x1 x2)              ===> z
(make-polar x3 x4)                 ===> z
(real-part z)                          ===> x1
(imag-part z)                          ===> x2
(magnitude z)                          ===> |x3|
(angle z)                              ===> xangle
```

где $-\pi < x_{angle} \leq \pi$ с $x_{angle} = x_4 + 2\pi\, n$ для некоторого целого n.

> *Обоснование:* `Magnitude` является тем же самым, что и `abs` для вещественного аргумента, но `abs` может использоваться во всех имплементациях, в то время как `magnitude` необходим только в тех имплементациях, которые поддерживают общие комплексные числа.

процедура: `(exact->inexact` z`)`
процедура: `(inexact->exact` z`)`

`Exact-inexact` возвращает неточное представление z. Возвращаемое значение это неточное число, которое численно близко к аргументу. Если точный аргумент не имеет достаточно близкого неточного эквивалента, то будет сообщено о нарушении ограничения имплементации.

`Inexact->exact` возвращает точное представление числа z. Возвращаемым значением является точное число, которое численно близко к аргументу. Если неточный аргумент не имеет достаточно близкого точного эквивалента, то будет сообщено о нарушении ограничения имплементации.

Данные процедуры реализуют естественное отношение один-к-одному между точными и неточными целыми во всем имплементационно-зависимом диапазоне. Смотри раздел 6.2.3.

6.2.6 Числовой ввод и вывод

процедура: `(number->string` *`z`*`)`
процедура: `(number->string` *`z radix`*`)`

Radix должен быть точным целым 2, 8, 10 или 16. Если *radix* опущен, по умолчанию он считается равным 10. Процедура `number->string` принимает число и основание системы счисления и возвращает в качестве строки внешнее представление данного числа в данном системе счисления такой, что выражение

```
(let ((number number)
      (radix radix))
  (eqv? number
        (string->number (number->string number
                                        radix)
                        radix)))
```

является истинным. Если ни один из возможных результатов этого вражения не даёт истину, то возникает ошибка.

Если *z* является неточным числом с основанием, и результат выражения может содержать число с десятичной точкой, то результат (процедуры) будет содержать десятичную точку и будет выражаться минимальным количеством цифр (не считая экспоненту и ведущих нулей) необходимым для того, чтобы выражение выше стало истинным [3, 5]; если это невозможно, то формат результата является неопределённым.

Результат возвращаемый процедурой `number->string` никогда не содержит явного префикса основания.

> *Замечание:* Ошибочная ситуация может возникнуть, когда *z* не является комплексным числом или является таковым, но с нерациональной вещественной или мнимой частью.
>
> *Обоснование:* Если *z* неточное число представленное с использованием плавающей запятой, а его основание 10, то выражение выше, будет естественно удовлетворять результату содержащему десятичную точку. Благодаря случаю с неопределённым значением, имеют место бесконечности, NaN-ы и представления чисел без плавающей запятой.

процедура: `(string->number` *`string`*`)`
процедура: `(string->number` *`string radix`*`)`

Возвращают число, или его максимально точное представление данное строкой *string*. *Radix* должно быть точным целым, 2, 8, 10 или 16. Если *radix* указан, то он является стандартным основанием, которое может быть записано с помощью явного префикса в *string* (например, `"#o177"`). Если *radix* не указан, то стандартным основанием считается 10. Если *string* представлено в синтаксически не правильной нотации числа, то `string->number` возвращает `#f`.

```
(string->number "100")                ===>  100
(string->number "100" 16)             ===>  256
(string->number "1e2")                ===>  100.0
(string->number "15##")               ===>  1500.0
```

> *Замечание:* Диапазон допустимых значений `string->number` может быть ограничен имплементацией следующими способами. `String->number` может возвращать `#f` в том случае, если *string* содержит явный префикс основания. Если все числа, которые поддерживаются имплементацией, являются вещественными, то `string->number` может вернуть `#f` всякий раз, когда *string* использует полярную или прямоугольную нотацию для комплексных чисел. Если все числа целые, `string->number` может вернуть `#f` в том

случае, если используется дробная нотация. Если все числа точные, `string->number` может вернуть `#f` в случае, когда используется маркер экспоненты или явный префикс точности, или если `#` появляется на месте цифры. Если все неточные числа являются целыми, `string->number` может вернуть `#f` всякий раз, когда используется десятичная точка.

6.3 Другие типы данных

Данный раздел описывает операции на некоторых не числовых типах данных Scheme: булевы типы, пары, списки, обозначения, символы и векторы.

6.3.1 Булевы типы

Стандартные объекты булева типа для значений истина и ложь используют запись `#t` и `#f`. Большое значение имеют объекты, которые условные выражения Scheme (`if`, `cond`, `and`, `or`, `do`) трактуют как истина или ложь. Фраза “истинное занчение” (или иногда просто “истина”) означает, что некоторый объект рассматривается условными выражениями как истина, а фраза “ложное значение” (или “ложь”) означает, что некоторый объект обрабатывается условными выражениями как истина.

Из всех стандартных значений Scheme, только `#f` вычисляется как ложь в условном выражении. За исключением `#f` все стандартные значения Scheme, включая `#t`, пары, пустой список, символы, числа, строки, векторы и процедуры вычисляются как истина.

> *Замечание:* Программисты привыкшие к другим диалектам Lisp-а, должны осознавать, что в Scheme `#f` и пустой список отличается от символа `nil`.

Булевы константы вычисляются к самим себе, так что нет необходимости цитировать их в программах.

```
#t                 ===>  #t
#f                 ===>  #f
'#f                ===>  #f
```

библиотечная функция: `(not` *obj*`)`

`Not` возвращает `#t`, если объект является ложью, и возвращает `#f` в противном случае.

```
(not #t)           ===>  #f
(not 3)                  ===>  #f
(not (list 3))           ===>  #f
(not #f)           ===>  #t
(not '())                ===>  #f
(not (list))             ===>  #f
(not 'nil)               ===>  #f
```

библиотечная функция: `(boolean?` *obj*`)`

`Boolean?` возвращает `#t`, если `obj` это `#f` или `#t`, и `#f` в противном случае.

```
(boolean? #f)          ===>  #t
(boolean? 0)                 ===>  #f
(boolean? '())               ===>  #f
```

6.3.2 Пары и списки

Пара это структура записи с двумя полями, которые называются car и cdr(по историческим причинам). Пары создаются с помощью процедуры `cons`. К полям car и cdr можно получить доступ с помощью процедур `car` и `cdr`. К полям car и cdr можно присвоить значение с помощью процедур `set-car!` и `set-cdr!`.

Пары в первую очередь используются для представления списков. Список может быть рекурсивно определен либо как пустой список, либо как пара, cdr которой является список. Точнее, множество списков определяется как наименьшее множество *X* такое, что

- Пустой список содержится в *X*.
- Если *список* содержится в *X*, то некоторая пара, поле cdr которой содержит *список* также содержится в *X*.

Объектами в полях car последовательных пар списка являются элементы этого списка. Например, список из двух элементов является парой, car которого является первым элементом, а cdr которого является парой, car которой является второй элемент, и cdr которой является пустым списком. Длина списка это число элементов, которое равно числу пар этого списка.

Пустой список - это специальный объект со своим собственным типом (он не является парой); у него нет элементов, а длина его равна нулю.

> *Замечание:* Определения выше подразумевают, что все списки имеют конечную длину и заканчиваются пустым списком.

Наиболее общим обозначением (внешним представлением) для пар в Scheme является “точечная” нотация `(`c_1 `.` c_2`)`, где c_1 это значение поля car, а c_2 это значение поля cdr. Например, `(4 . 5)` это пара, car которой равен 4, и cdr которой равен 5. Обратите внимание, что `(4 . 5)` это внешнее представление пары, а не выражение которое вычисляется как пара.

Для списков может быть использована более удобная нотация: элементы списка просто заключаются в круглые скобки и разделяются пробелами. Пустой список записывается как `()`. Например,

```
(a b c d e)
```

и

```
(a . (b . (c . (d . (e . ()))))
```

являются эквивалентными нотациями для списка символов.

Цепочка пар, не заканчивающаяся пустым списком, называется *неправильным списком*. Обратите внимание, что неправильный список не является списком. Список и точечная нотация могут быть скомбинированы для представления неправильного списка:

```
(a b c . d)
```

что эквивалентно

```
(a . (b . (c . d)))
```

Является ли данная пара списком зависит от того, что храниться в поле cdr. Когда используется процедура `set-cdr!`, объект может быть в один момент списком, а в другой - нет:

```
(define x (list 'a 'b 'c))
(define y x)
y                               ===>  (a b c)
(list? y)                       ===>  #t
(set-cdr! x 4)                  ===>  unspecified
x                               ===>  (a . 4)
(eqv? x y)                      ===>  #t
y                               ===>  (a . 4)
(list? y)                       ===>  #f
(set-cdr! x x)                  ===>  unspecified
(list? x)                       ===>  #f
```

Вместе с выражением лиетарала и представлениями объектов считанных с помощью процедуры `read`, формы `'`<элемент данных>, `` `<элемент данных>, `,`<элемент данных> и `,@`<элемент данных> обозначают двух элементные списки, первые элементы которых это символы `qoute`, `qouasiquote`, `unquote` и `unquote-splicing` соответственно. А второй элемент в каждом случае это <элемент данных>. Это соглашение принято для того, чтобы каждая программа могла быть представлена как множество списков. То есть, в соответствии с грамматикой Scheme, каждое <выражение> также является <элементом данных> (смотри раздел 7.1.2). Между прочим, это позволяет использование процедуры `read` для структурного анализа программ Scheme. Смотри раздел 3.3.

процедура: `(pair? obj)`

`Pair?` возвращает `#t`, если *obj* это пара, иначе - `#f`.

```
(pair? '(a . b))                ===>  #t
(pair? '(a b c))                ===>  #t
(pair? '())                     ===>  #f
(pair? '#(a b))                 ===>  #f
```

процедура: `(cons` obj_1 obj_2`)`

Возвращает заново выделенную пару, car которой это obj_1, а cdr которй это obj_2. Гарантировано, что данная пара будет отлична (в смысле `eqv?`) от любого другого существующего объекта.

```
(cons 'a '())                   ===>  (a)
(cons '(a) '(b c d))            ===>  ((a) b c d)
(cons "a" '(b c))               ===>  ("a" b c)
(cons 'a 3)                     ===>  (a . 3)
(cons '(a b) 'c)                ===>  ((a b) . c)
```

процедура: `(car` *pair*`)`

Возвращает содержимое поля car *pair*. Обратите внимание, что при попытке получить car пустого списка возникнет ошибка.

```
(car '(a b c))                  ===>  a
(car '((a) b c d))              ===>  (a)
(car '(1 . 2))                  ===>  1
(car '())                       ===>  error
```

процедура:`(cdr` *pair*`)`

Возвращает содержимое поля cdr *pair*. Обратите внимание, что при попытке получить cdr пустого списка возникнет ошибка.

```
(cdr '((a) b c d))              ===>  (b c d)
(cdr '(1 . 2))                  ===>  2
(cdr '())                       ===>  error
```

процедура: `(set-car!` *pair obj*`)`

Сохраняет *obj* в поле car *pair*. Значение возвращаемое `set-car!` не определено.

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)                    ===>  unspecified
(set-car! (g) 3)                    ===>  error
```

процедура: `(set-cdr!` *pair obj*`)`

Сохраняет *obj* в поле cdr *pair*. Значение возвращаемое `set-cdr!` не определено.

библиотечная процедура: `(caar` *pair*`)`
библиотечная процедура: `(cadr` *pair*`)`
…
библиотечная процедура: `(cdddar` *pair*`)`
библиотечная процедура: `(cddddr` *pair*`)`

Данные процедуры являются композициями car и cdr, где например, `caddr` может быть определён с помощью выражения

```
(define caddr (lambda (x) (car (cdr (cdr x))))).
```

Произвольные композиции предоставлены до четвертого уровня глубины. В сумме есть двадцать восемь процедура данного типа.

библиотечная процедура: `(null? obj)`

Возвращает `#t`, если *obj* является пустым списком, иначе возвращает `#f`.

библиотечная процедура: `(list? obj)`

Возвращает `#t` если *obj* является списком, в противном случае возвращает `#f`. По определению все списки имеют конечную длину и заканчиваются пустым списком.

```
(list? '(a b c))            ===>  #t
(list? '())                 ===>  #t
(list? '(a . b))            ===>  #f
(let ((x (list 'a)))
  (set-cdr! x x)
  (list? x))                ===>  #f
```

библиотечная процедура: `(list obj ...)`

Возвращает заново выделенный список состоящей из своих аргументов.

```
(list 'a (+ 3 4) 'c)                    ===>  (a 7 c)
(list)                                  ===>  ()
```

библиотечная процедура: `(length list)`

Возвращает длину *list*.

```
(length '(a b c))                       ===>  3
(length '(a (b) (c d e)))               ===>  3
(length '())                            ===>  0
```

библиотечная процедура: `(length list ...)`

Возвращает список состоящий из элементов первого *list* следующего за списком элементами других *list*-ов.

```
(append '(x) '(y))                      ===>  (x y)
(append '(a) '(b c d))                  ===>  (a b c d)
(append '(a (b)) '((c)))                ===>  (a (b) (c))
```

Под итоговый список всегда выделяется заново память, за исключением того случая, когда данный список разделяет структуру с последним аргументом *list*. Последним аргументом на самом деле может быть любой объект; если последний аргументы не является правильным списком, то результатом данной процедуры будет неправильный список.

```
(append '(a b) '(c . d))                ===>  (a b c . d)
(append '() 'a)                         ===>  a
```

библиотечная процедура: `(reverse` *`list`*`)`

Возвращает заново выделенный список состоящий из элементов *list*, расположенных в обратном порядке.

```
(reverse '(a b c))                      ===>  (c b a)
(reverse '(a (b c) d (e (f))))
               ===>  ((e (f)) d (b c) a)
```

библиотечная процедура: `(list-tail` *`list k`*`)`

Возвращает подсписок списка *list* полученный пропуском первых *k* элементов. Если список содержит меньше *k* элементов возникнет ошибка. `List-tail` может быть определен с помощью выражения

```
(define list-tail
  (lambda (x k)
    (if (zero? k)
        x
        (list-tail (cdr x) (- k 1)))))
```

библиотечная процедура: `(list-ref` *`list k`*`)`

Возвращает *k*-ый элемент списка *list*. (Это тоже самое, что car `(list-ref` *list k*`)`.) Если *list* имеет меньше *k* элементов, возникнет ошибка.

```
(list-ref '(a b c d) 2)                          ===>  c
(list-ref '(a b c d)
          (inexact->exact (round 1.8)))
               ===>  c
```

библиотечная процедура: `(memq` *`obj list`*`)`
библиотечная процедура: `(memv` *`obj list`*`)`
библиотечная процедура: `(member` *`obj list`*`)`

Данные процедуры возвращают перывй подсписок *list*, car котрого это *obj*, где подсписки *list* это непустые возвращаемые `(list-tail` *list k*`)` для *k* меньшего длины *list*. Если *obj* не содержится в *list*, то возвращается `#f` (не пустой список). `Memq` использует `eq?` для сравнения *obj* с элементами *list*, в то время как `memv` использует `eqv?`, а `member?` использует `equal?`.

```
(memq 'a '(a b c))                      ===>  (a b c)
(memq 'b '(a b c))                      ===>  (b c)
(memq 'a '(b c d))                      ===>  #f
(memq (list 'a) '(b (a) c))             ===>  #f
(member (list 'a)
        '(b (a) c))                     ===>  ((a) c)
(memq 101 '(100 101 102))               ===>  unspecified
(memv 101 '(100 101 102))               ===>  (101 102)
```

библиотечная процедура `(assq obj alist)`
библиотечная процедура `(assv obj alist)`
библиотечная процедура `(assoc obj alist)`

`Alist` (для “ассоциативного списка”) должен быть списком пар. Данные процедуры находят первую пару в *alist*, поле car которой это *obj*, и возвращают эту пару. Если в *alist* нет пары, содержащей *obj* в car, то возвращается `#f` (а не пустой список). `Assq` использует `eq?` для сравнения *obj* с полями car пар в *alist*, `assv` - `eqv?`, а `assoc` - `equal?`.

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)             ===>  (a 1)
(assq 'b e)             ===>  (b 2)
(assq 'd e)             ===>  #f
(assq (list 'a) '(((a)) ((b)) ((c))))
                        ===>  #f
(assoc (list 'a) '(((a)) ((b)) ((c))))
                                    ===>  ((a))
(assq 5 '((2 3) (5 7) (11 13)))
                                    ===>  unspecified
(assv 5 '((2 3) (5 7) (11 13)))
                                    ===>  (5 7)
```

> *Обоснование:* Хотя они и используются обычно как предикаты, `memq`, `memv`, `member`, `assq`, `assv` и `assoc` не содержат вопрос в их именах, поскольку они возвращают более полезные значения, чем просто `#t` или `#f`.

6.3.3 Знаки

Знаки это объекты, полезность которых основывается на том факте, что два знака идентичны (в смысле `eqv?`), если и только если их имена записываются по буквам одинаково. Это именно то свойство, которое необходимо для представления идентификаторов в программах, и поэтому большинство имплементаций Scheme используют их внутри для данной цели. Знаки полезны для многих других приложений; например, они могут использоваться так, как используются перечисленные значения в Pascal.

Правила для записи знака точно такие же ка правила для записи идентификаторов; смотри разделы 2.1 и 7.1.1.

Гарантировано, что любой знак, который возвращается как часть выражения литерала или считывается с использованием процедуры `read`, а затем выводиться с помощью процедуры `write`, будет повторно считан, как тот же знак (в смысле `eqv?`). Процедура `string->symbol`, с другой стороны, может создать знаки, для которых эта инвариантность не будет выполняться, поскольку их имена содержат специальные символы или буквы в нестандартном регистре.

Замечание: Некоторые имплементации Scheme имеют функцию, известную как “слэшифифкация” для того, чтобы гарантировать инвариантность чтения/записи для всех знаков, но исторически наиболее важное использование этой функции компенсировало недостаток строкового типа данных

Некоторые имплементации также имеют “неинтегрированные знаки”, которые устраняют инвариантность чтения/записи даже в имплементациях с слэшификацией, и также генерируют исключения из правила, что два знака одинаковы, если и только если их имена записываются одинаково.

процедура: `(symbol? obj)`

Возвращает `#t`, если *obj* является знаком, иначе - `#f`.

```
(symbol? 'foo)                  ===>  #t
(symbol? (car '(a b)))          ===>  #t
(symbol? "bar")                 ===>  #f
(symbol? 'nil)                  ===>  #t
(symbol? '())                   ===>  #f
(symbol? #f)             ===>  #f
```

процедура: `(symbol->string symbol)`

Возвращает в качестве строки название знака. Если знак был частью объекта, который был возвращен как значение выражения литерала (раздел 4.1.2) или с помощью вызова процедуры `read`, и его имя содержит алфавитные символы, то возвращаемая строка будет содержать символы в стандартном регистре, предпочитаемом данной имплементацией - некоторые имплементации предпочитаются врехний регистр, другие - нижний. Если процедура `string->symbol` возвращает знак, то регистр символов в возвращаемой строке будет таким же, как регистр в строке, которая была передана процедуре `string->symbol`. Возникнет ошибка, если применять изменяющие процедуры такие, как `string-set!`, к строка возвращаемым данной процедурой.

Следующие примеры предполагают, что стандартным регистром имелпементации является нижний регистр:

```
(symbol->string 'flying-fish)
                                        ===>  "flying-fish"
(symbol->string 'Martin)                ===>  "martin"
(symbol->string
   (string->symbol "Malvina"))
                                        ===>  "Malvina"
```

процедура: `(string->symbol string)`

Возвращает зеак, имя которого *string*. Данная процедура может создавать знаки с именами, содержащими специальные символы или буквы записанные в нестандартном регистре, но обычно это не самая лучшая идея создавать такие знаки, поскольку в некторых имплементациях Scheme они могут быть считаны по другому. Смотри `symbol->string`.

Следующие примеры предполагают, что стандартным регистром имплементации является нижний регистр.

```
(eq? 'mISSISSIppi 'mississippi)
                ===>  #t
(string->symbol "mISSISSIppi")
                ===>  the symbol with name "mISSISSIppi"
(eq? 'bitBlt (string->symbol "bitBlt"))
                ===>  #f
(eq? 'JollyWog
     (string->symbol
       (symbol->string 'JollyWog)))
                ===>  #t
(string=? "K. Harper, M.D."
          (symbol->string
            (string->symbol "K. Harper, M.D.")))
                ===>  #t
```

6.3.4 Символы

Символы это объекты, которые представляются печатаемыми символами такими, как буквы и цифры. Символы записываются с помощью нотации `#\` <символ> или `#\`<имя символа>. Например:

```
#\a ; символ в нижнем регистре
#\A ; символ в верхнем регистре
#\( ; левая круглая скобка
#\  ; пробельный символ
#\space ; предпочитаемый способ записи пробела
#\newline   ; символ новой строки
```

Регистр определяется в `#\` <символе>, а не в `#\` <символьном имени>. Если <символ> в `#\` <символ> является алфавитным, то символ, который следует за <символом> должен быть разделительным символом таким, как пробел или круглые скобки. Это правило разрешает произвольный регистр, где, например, последовательность символов “`#\space`” может быть представлением пробельного символа, либо представлением символа “`#\s`”, которое следует за представлением символа “`pace`”.

Символы записанные в нотации `#\` являются само-выполняющимися. То есть, они не могут быть цитироваться в программах. Некоторые процедуры, которые работают с символами игнорируют различие между верхним и нижним регистрами. Имена процедур, игнорирующих регистр, содержат “`-ci`” (для “нечувствительности к регистру”).

процедура: `(char?` *obj*`)`

Возвращает `#t`, если *obj* - символ, и `#f` в противном случае.

процедура: `(char=?` $char_1$ $char_2$`)`
процедура: `(char<?` $char_1$ $char_2$`)`
процедура: `(char>?` $char_1$ $char_2$`)`
процедура: `(char<=?` $char_1$ $char_2$`)`
процедура: `(char>=?` $char_1$ $char_2$`)`

Данные процедуры выполняются полное сравнение множества символов. Гарантировано, что во время этого сравнения:

- Символы верхнего регистра упорядочены. Например, `(char<? #\A #\B)` вернет `#t`.
- Символы нижнего регистра упорядочены. Например, `(char<? #\a #\b)` вернет `#t`.
- Цифры упорядочены. Например, `(char<? #\0 #\9)` вернет `#t`.
- Либо все цифры предшествуют буквам верхнего регистра, либо наоборот.
- Либо все цифры предшествуют буквам нижнего регистра, либо наоборот.

Некоторые имплементации могут обобщить данные процедуры, принимая более двух аргументов, как соответствующие числовые предикаты.

библиотечная процедура: `(char-ci=? `$char_1$ $char_2$`)`
библиотечная процедура: `(char-ci<? `$char_1$ $char_2$`)`
библиотечная процедура: `(char-ci>? `$char_1$ $char_2$`)`
библиотечная процедура: `(char-ci<=? `$char_1$ $char_2$`)`
библиотечная процедура: `(char-ci>=? `$char_1$ $char_2$`)`

Данные процедуры идентичны `char=?` и им подобным, но они воспринимают буквы верхнего и нижнего регистров как одинаковые. Например, `(char-ci=? #\A #\a)` вернет `#t`. Некоторые имплементации могут обобщить данные процедуры, принимая более двух аргументов, как с соответствующими числовыми предикатами.

библиотечная процедура: `(char-alphabetic? `*char*`)`
библиотечная процедура: `(char-numeric? `*char*`)`
библиотечная процедура: `(char-whitespace? `*char*`)`
библиотечная процедура: `(char-upper-case? `*letter*`)`
библиотечная процедура: `(char-lower-case? `*letter*`)`

Данные процедуры возвращают `#t`, если их аргументы являются буквенными, численными, пробельными, верхнего или нижнего регистров числами, иначе они возвращают `#f`. Следующие замечания, которые являются спецификой множества символов ASCII, предназначены только, как руководство: Буквенными символами являются 52 буквы в нижнем и верхнем регистре. Числовые символы это десять десятичных цифр. Пробельными символами являются пробел, табуляция, перевод строки, разрыв страницы и перевод каретки.

процедура: `(char->integer `*char*`)`
процедура: `(integer->char `*n*`)`

Принимая символ, `char->integer` возвращает представление точного целого числа данного символа. Принимая точное целое, которое является образом символа, отображение `char->integer`, `integer->char` возвращает символ. Данные процедуры реализуют сохраняющий порядок изоморфизм между множеством символов с заданным отношением порядка `char<=?` и некоторым подмножеством целых с отношением порядка `<=`. То есть, если

`(char<=? `*a b*`)  ===> #t` и `(<= `*x y*`)  ===> #t`

и *x*, *y* лежат в области допустимых значений процедуры `integer->char`, то

```
(<= (char->integer a)
    (char->integer b))                ===>  #t

(char<=? (integer->char x)
         (integer->char y))             ===>  #t
```

библиотечная процедура: `(char-upcase `*char*`)`
библиотечная процедура: `(char-downcase `*char*`)`

Данные процедуры возвращают символ $char_2$ такой, что `(char-ci=? `*char* $char_2$`)`. Кроме того, если *char* является буквой, то результат процедуры `char-upcase` будет в верхнем регистре, а результат процедуры `char-downcase` в нижнем.

6.3.5 Строки

Строки это последовательность символов. Строки записываются, как последовательность символов, заключенная в двойные кавычки("). Двойные кавычки могут быть записаны внутри строки только если, они отделены обратным слэшом (‘\’), как, например, в строке

```
"The word \"recursion\" has many meanings."
```

Обратный слэш может быть записан в строке только, если перед ним стоит другой обратный обратный слэш. Scheme не определяет результат обратного слэша в строке, которая не следует за двойной кавычкой или обратным слэшом.

Строковая константа может быть продолжена на следующей строке, но точное содержание такой строки не определено. *Длина* строки это число символов, которые в ней содержаться. Это число является точным, неотрицательным целым, которое фиксируется при создании строки. *Корректными индексами* называют точные неотрицательные целые числа, меньшие чем длина строки. Первый символ строки имеет индекс 0, второй - 1 и так далее.

Под такими фразами, как “символы *строки* начинающиеся с индекса *старт* и заканчивающиеся индексом *конец*” подразумевается то, что индекс *старт* включен, а индекс *конец* не включен. Таким образом, если *старт* и *конец* совпадают, то данная строка является нулевой, а если *старт* является нулем, а *конец* - длиной *строки*, то подразумевается вся строка.

Некоторые процедуры, которые оперируют со строками игнорируют различие между верхним и нижнем регистром. Та версии процедур, которые игнорирует регистр, содержат “`-ci`” в своих именах.

процедура: `(string? obj)`

Возвращает `#t`, если *obj* является строкой, и `#f` - в противном случае.

процедура: `(make-string k)`
процедура: `(make-string k char)`

`Make-string` возвращает заново выделенную строку длины *k*. Если передан *char*, то все элементы строки будут инициализированы как *char*, иначе содержимое *string* не определено.

библиотечная процедура: `(string char ...)`

Возвращает заново выделенную строку состоящую из аргументов.

процедура: `(string-length string)`

Возвращает число символов в данной строке *string*.

процедура: `(string-ref string k)`

k должно быть корректным индексом *string*. `String-ref` возвращает *k* символов *string*, используя индекс начинающийся с нуля.

процедура: (string-set! *string k char*)

k должно быть корректным индексом *string*. String-set! сохраняет *char* в *k*-ый элемент *string* и возвращает неопределённое значение.

```
(define (f) (make-string 3 #\*))
(define (g) "***")
(string-set! (f) 0 #\?)          ===>  unspecified
(string-set! (g) 0 #\?)          ===>  error
(string-set! (symbol->string 'immutable)
             0
             #\?)          ===>  error
```

библиотечная процедура: (string=? $string_1$ $string_2$)
библиотечная процедура: (string-ci=? $string_1$ $string_2$)

Возвращает #t, если две строки одинаковой длины и содержат одинаковые символы на одинаковых позициях, иначе возвращает #f. String-ci=? обрабатывает буквы верхнего и нижнего регистра так, как будто бы они одни и те же, а string=? воспринимает буквы верхнего и нижнего регистр как различные.

библиотечная процедура: (string<? $string_1$ $string_2$)
библиотечная процедура: (string>? $string_1$ $string_2$)
библиотечная процедура: (string<=? $string_1$ $string_2$)
библиотечная процедура: (string>=? $string_1$ $string_2$)
библиотечная процедура: (string-ci<? $string_1$ $string_2$)
библиотечная процедура: (string-ci>? $string_1$ $string_2$)
библиотечная процедура: (string-ci<=? $string_1$ $string_2$)
библиотечная процедура: (string-ci>=? $string_1$ $string_2$)

Данные процедуры являются лексикографическими расширениями строкам соответствующих отношений порядка заданных на символах. Например, string<? это лексикографическое отношение порядка строк, индуцированное отношением char<? на символах. Если две строки различной длины, но их подстроки меньшей длины совпадают, то строка мньшей длины считается лексикографически меньше, чем строка, которая длиннее.

Имплементации могут обобщить эти процедуры и процедуры string=? и string-ci=? на число аргументов большее двух, как в случае соответствующих числовых предикатов.

библиотечная процедура: (substring *string start end*)

String должна быть строкой, а *start* и *end* должны быть точными целыми, удовлетворяющие

$0 \leq start \leq end \leq$ (string-length *string*).

Substrings возвращает заново выделенную строку, сформированную из символов *string* начиная с индекса *start* (включительно) и заканчивая индексом *end* (не включительно).

библиотечная процедура: (string-append *string* ...)

Возвращает заново выделенную строку символы которой формируются конкатенацией данных строк.

библиотечная процедура: (string->list *string*)
библиотечная процедура: (string->list *string*)

String->list возвращает заново выделенный список символов, которые формируются из заданной строки. List->string возвращает заново выделенную строку, сформированную из символов списка

list. `String->list` и `list->string` являются инверсиями в смысле `equal?`.

библиотечная процедура: `(string-copy` *`string`*`)`

Возвращает заново выделенную копию данной *string*.

библиотечная процедура: `(string-fill!` *`string char`*`)`

Сохраняет *char* в каждый элемент данной строки и возвращает неопределённое значение.

6.3.6 Векторы

Векторы это неоднородные структуры, элементы которых индексируются целыми числами. Обычно, вектор занимает меньше места, чем список той же длины, и среднее время необходимое для доступа к произвольному элементу у вектора обычно меньше, чем у списка.

Длиной вектора является количество элементов, которые в нем содержаться. Это число является не отрицательным целым, которое фиксируется в момент создания вектора. *Корректными индексами* вектора называют точные неотрицательные числа, меньшие длины вектора. Первый элемент вектора имеет нулевой индекс, а последний элемент индексируется числом на единицу меньшим длины вектора.

Векторы записываются с использованием нотации `#(`*`obj ...`*`)`. Например, вектор длины 3, содержащий элемент 0 по нулевому индексу, список `(2 2 2 2)` в качестве элемента с индексом 1, и `"Anna"` в качестве 2 элемента может быть записан следующим образом:

```
#(0 (2 2 2 2) "Anna")
```

Обратите внимание, что это внешнее представление вектора, а не выражения вычисляющегося к вектору. Как и списочные константы, векторные константы можно цитировать:

```
'#(0 (2 2 2 2) "Anna")
               ===>  #(0 (2 2 2 2) "Anna")
```

процедура: `(vector? obj)`

Возвращает `#t`, если *obj* является вектором, и `#f` в противном случае.

процедура: `(make-vector` *`k`*`)`
процедура: `(make-vector` *`k fill`*`)`

Возвращает заново выделенный вектор из *k* элементов. Если задан второй элемент, то каждый элемент инициализируется как *fill*. Иначе начальное значение каждого элемента не определено.

библиотечная процедура: `(vector obj ...)`

Возвращает заново выделенный вектор, элементы которого содержат данные аргументы. Аналогична `list`.

```
(vector 'a 'b 'c)                       ===>  #(a b c)
```

процедура: `(vector-length vector)`

Возвращает точное целое - количество элементов в *vector*.

процедура: `(vector-ref vector k)`

k должно быть корректным индексом *vector*. `Vector-ref` возвращает содержимое *k*-го элемента *vector*.

```
(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
```

процедура: `(vector-set! vector k obj)`

k должно быть корректным индексом *vector*-а. `Vector-set!` записывает *obj* *k*-ым элементом *vector*. Значение возвращаемое `vector-set!` не определено.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec)
               ===>  #(0 ("Sue" "Sue") "Anna")

(vector-set! '#(0 1 2) 1 "doe")
               ===>  error  ; constant vector
```

библиотечная процедура: `(vector->list vector)`
библиотечная процедура: `(list->vector list)`

`Vector->list` возвращает заново выделенный список объектов, состоящий из элементов *vector*. `List->vector` возвращает заново выделенный вектор, инициализированный из элементами списка *list*.

библиотечная процедура: `(vector-fill! vector fill)`

Записывает *fill* в каждый элемент *vector*. Значение, возвращаемое `vector-fill!` не определено.

6.4 Управляющие функции

Данная глава описывает множество прирitивных процедур, которые управляют потоком выполнения программы специальным способом. Предикат `procedure?` также описан здесь.

процедура: `(procedure? obj)`

Возвращает `#t` если *obj* это процедура, иначе возвращает `#f`.

```
(procedure? car)                    ===>  #t
(procedure? 'car)                   ===>  #f
(procedure? (lambda (x) (* x x)))
                                    ===>  #t
(procedure? '(lambda (x) (* x x)))
                                    ===>  #f
(call-with-current-continuation procedure?)
                                    ===>  #t
```

процедура: `(apply proc arg1 ... args)`

`Proc` должен быть процедурой, а `args` должен быть списком. Вызывает *proc* с фактическими аргументами, состоящими из элементов списка `(append (list arg1 ...) args)`.

```
(apply + (list 3 4))                        ===>  7

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))

((compose sqrt *) 12 75)                        ===>  30
```

библиотечная процедура: `(map proc list1 list2 ...)`

list-ы должны быть списками, а *proc* должен быть процедурой, принимающей множество аргументов *list* и возвращающей единственное значение. Если более передается более одного списка *list*, то они все должны быть одинаковой длины. `Map` применяет *proc* к каждой элементу списков *list* и возвращает список результатов сохраняя порядок. Динамический порядок, в котором *proc* применяется к элементам списков *list* не определен.

```
(map cadr '((a b) (d e) (g h)))
               ===>  (b e h)

(map (lambda (n) (expt n n))
     '(1 2 3 4 5))
               ===>  (1 4 27 256 3125)

(map + '(1 2 3) '(4 5 6))                   ===>  (5 7 9)

(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b)))                          ===>  (1 2) or (2 1)
```

библиотечная процедура: `(for-each` *proc* $list_1$ $list_2$ `...)`

Элементы `for-each` аналогичны аргументам `map`, но `for-each` вызывает *proc* для получение побочного действия, а не для получения его значений. В отличии от `map`, гарантировано, что `for-each` вызовет *proc* для элементов списков *list* в определенном порядке, начиная с первого(-ых) элемента(-ов) и заканчивая последним(и), и значение, возвращаемое процедурой `for-each`, не определено.

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
              (vector-set! v i (* i i)))
            '(0 1 2 3 4))
  v)                                        ===>  #(0 1 4 9 16)
```

библиотечная процедура: `(force` *promise*`)`

Принудительно получает значение *promise* (сотри `delay`, раздел 4.2.5). Если не было обещания вычислять какое-либо значение, то значение вычисляется и возвращается. Значение обещания кэшируются (или “запоминается”), то есть если оно запрашивается второй раз, возвращается значение, которое было вычислено ранее.

```
(force (delay (+ 1 2)))           ===>  3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))
                                        ===>  (3 3)

(define a-stream
  (letrec ((next
            (lambda (n)
              (cons n (delay (next (+ n 1)))))))
    (next 0)))
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))

(head (tail (tail a-stream)))
                                        ===>  2
```

`Force` и `delay` главным образом предназначены для программ написанных в функциональном стиле. Следующие примеры предоставлены не для того чтобы показать хороший стиль программирования, а для того чтобы обозначить такое свойство, что только одно значение вычисляется по обещанию, не важно сколько раз оно запрашивается.

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p                               ===>  a promise
(force p)                       ===>  6
p                               ===>  a promise, still
(begin (set! x 10)
       (force p))               ===>  6
```

Здесь приведена возможная реализация `delay` и `force`. Здесь обещания реализуются, как процедуры без аргументов, а `force` просто вызывает свой аргумент.

```
(define force
  (lambda (object)
    (object)))
```

Мы определяем выражение

```
(delay <выражение>)
```

так, чтобы оно означало то же, что и вызов процедуры

```
(make-promise (lambda () <выражение>))
```

следующим образом

```
(define-syntax delay
  (syntax-rules ()
    ((delay expression)
     (make-promise (lambda () expression))))),
```

где `make-promise` определяется следующим образом:

```
(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()
        (if result-ready?
            result
            (let ((x (proc)))
              (if result-ready?
                  result
                  (begin (set! result-ready? #t)
                         (set! result x)
                         result)))))))))
```

> Обоснование: Обещание может ссылаться на его собственное значение, как в примере выше. Выполнение такого обещания может привести к тому, что обещание выполниться второй раз, перед тем как значение вычислится. Это усложняет определение `make-promise`.

Поддерживается данной семантики `delay` и `force` поддерживается в различных имплементациях:

- Вызов `force` для объекта, который не является обещанием может просто вернуть данный объект.
- Может оказаться, что не существует никаких способов, с помощью которых обещание может быть функционально отделено от принудительного значения. То есть, выражение по типу следующего может вычисляться либо как `#t`, либо как `#f`, в зависимости от имплементации:

```
(eqv? (delay 1) 1)                  ===>  unspecified
(pair? (delay (cons 1 2)))          ===>  unspecified
```

- Некоторые имплементации могут реализовывать “неявное принуждение”, где значение обещания принудительно запрашивается примитивным процедурами такими, как `cdr` и `+`:

  ```
  (+ (delay (* 3 7)) 13)          ===>  34
  ```

процедура: `(call-with-current-continuation` *`proc`*`)`

`Proc` должна быть процедурой одного аргумента. Процедура `call-with-current-continuation` упаковывает текущую континуацию (смотри обоснование выше) как “переходную процедуру” и передает ее в качестве аргумента процедуре *proc*. Переходная процедура это процедура Scheme, которая если вызывается позднее, то текущая процедура приостанавливается какая бы континуация не выполнялась в данный момент и вместо нее используется континауция, которая выполнялась, когда данная переходная процедура была создана. Вызов переходной процедуры может привести к инициированию работы переходников (небольшого участка кода, выполняющего некоторое преобразование) *перед* и *после* установленных с помощью `dynamic-word`.

Переходная процедура принимает такое же число аргументов, как континуация к изначальному вызову `call-with-current-continuation`. За исключением континуациq созданных с помощью процедуры `call-with-values`, все континуации принимают в точности одно значение. В том случае если ни одно или более одного значения передается континуации созданной с помощью `call-with-values`, результат не определён.

Переходная процедура, которая передается *proc* имеет неограниченный размер, как и любая другая процедура в Scheme. Она может записываться в переменных или структурах данных и может вызываться столько раз, сколько потребуется.

Следующие примеры показывают только самые распространенные способы в которых `call-with-current-continuation` используется. Если все настоящие использования были бы такими простыми, как эти примеры, не было бы смысла в такой мощной процедуре, как `call-with-current-continuation`.

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #t))                                  ===>  -3

(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        (letrec ((r
                  (lambda (obj)
                    (cond ((null? obj) 0)
                          ((pair? obj)
                           (+ (r (cdr obj)) 1))
                          (else (return #f))))))
          (r obj))))))

(list-length '(1 2 3 4))                    ===>  4

(list-length '(a b . c))                    ===>  #f
```

Обоснование:

Обычно `call-with-current-continuation` используют для стрктурированных, нелокальных выходов из циклов или тел процедур, но по факту, `call-with-current-continuation` очень полезна для реализации широкого множества расширенных структур управления.

Когда выражение Scheme выполняется, речь идет о континуации желающей получить результат выражения. Континуации представляют полую (по умолчанию) функцию для вычислений. Если выражение выполняется на верхнем уровне, например, то континуация может получить результат, вывести его на экран, запросить следующий ввод, вычислить его и так до бесконечности. Большую часть времени, континуации включают в себя действия определенные пользовательским кодом, как в континуации, которая получает результат, умножает его на значение, которое храниться в локальной переменной, добавляет семь и выводит результат на верхний уровень для того чтобы вывести на экран. Обычно, эти повсеместные континуации остаются на заднем плане и программисты особо о них не задумываются. В редком случае, однако, программисту может понадобиться работать с континуациями напрямую. `Call-with-current-continuation` позволяет Scheme программистам сделать это создав процедуру, которая только действует, как текущая континуация.

Большинство языков программирования объединяет одну или более переходных структур такими именами, как `exit`, `return` и даже `goto`. Однако, в 1965 году, Питер Ландин [16] придумал переходный оператор общего назначения, называемый J-оператор. Джон Рейнальдс [24] описал более простую, но такую же мощную конструкцию в 1972. Специальная форма `catch` описанная Сьюзменом и Стиллом в 1975 на докладе по Scheme, являлась точно такой же, как конструкция Рэйнольда, хотя ее название пришло из более общей конструкции в MacLisp. Несколько реализаторов Scheme заметили, что полная мощь конструкции `catch` может быть обеспечена процедурой, а не специальными синтаксическими конструкциями, а название `call-with-current-continuation` было принято в 1982. Это название описательного типа, но многие люди считают, что данное название слишком длинное, и поэтому используют `call/cc` вместо него.

процедура: `(values` *`obj`* `...)`

Передает все свои аргументы соответствующей континуации. За исключением континуаций, созданных процедурой `call-with-current-continuation`, все континуации принимают в точности одно занчение. `Values` можно определить следующим образом:

```
(define (values . things)
  (call-with-current-continuation
    (lambda (cont) (apply cont things))))
```

процедура: `(call-with-values` *`producer consumer`*`)`

Вызывает свой аргумент *producer* без каких либо значений и континуация, которая принимает некоторые значения, вызывает процедуру *consumer* с данными значениями в качестве аргументов. Континуацией, вызывающей *consumer*, является континуация вызова `call-with-values`.

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
                                                    ===>  5

(call-with-values * -)                              ===>  -1
```

процедура: `(dynamic-wind` *`before thunk after`*`)`

Вызывает *thunk* (переходник) без аргументов, возвращая результат(ы) этого вызова. *before* и *after* вызываются также без аргументов, как того требуют следующие правила (заметьте, что в отсутствие вызовов континуаций, перехваченных с помощью `call-with-current-continuation` три аргумента вызываются в порядке, один за другим) *Before* вызывается, всякий раз, когда выполнение входит в динамический период вызова *thunk*, а *after* вызывается, когда оно выходит из этого динамического периода. Динамическое период процедурного вызова это интервал между тем, когда вызов инициируется и тем, когда он возвращает управление. В Scheme, из-за `call-with-current-continuation`, динамический период вызова может не быть связанным с единственным периодом времени. Он определяется следующим образом:

- Динамический период начинается в момент выполнения начала выполнения тела процедуры.
- Динамический период также начинается, когда выполнение находится не в динамическом периоде, а континуация призывает то, что было захвачено (используя `call-with-current-continuation`) во время динамического периода.
- Он выходит, когда вызванная процедура завершается.
- Он также выходит, когда выполнение в динамическом периоде и континуация призывает то, что было захвачено в не динамический период.

Если происходит второй вызов `dynamic-wind` во время динамического периода вызова *thunk*, после чего континуация призывается таким образом, что *after*-ы из этих двух вызовов `dynamic-wind` оба будут вызваны, то *before* связанное с первым (выходным) вызовом `dynamic-wind` вызывается первым.

Если второй вызов `dynamic-wind` происходит во время динамического периода вызова *thunk* и после этого континуация призывается так, что *before* из этих двух призывов `dynamic-word` будут оба вызваны, то *before*, связанный с первым (выходным) вызовом `dynamic-wind` будет вызван первым.

Если призыв континуации требует вызова *before* из одного вызова `dynamic-word` и *after* из другого, то *after* вызывается первым.

Результат использования захваченной континуации для входа в или выхода из динамического периода вызова *before* или *after* не определен.

```
(let ((path '())
      (c #f))
  (let ((add (lambda (s)
               (set! path (cons s path)))))
    (dynamic-wind
      (lambda () (add 'connect))
      (lambda ()
        (add (call-with-current-continuation
               (lambda (c0)
                 (set! c c0)
                 'talk1))))
      (lambda () (add 'disconnect)))
    (if (< (length path) 4)
        (c 'talk2)
        (reverse path))))

              ===> (connect talk1 disconnect
              connect talk2 disconnect)
```

6.5 `Eval`

процедура: (eval *expression environment-specifier*)

Выполняет *expression* в определенном окружении и возвращает его значение. *Expression* должно быть корректным выражением Scheme, представленным как данные, а *environment-specifier* должен быть значением возвращаемым одной из трех процедур, описанных выше. Имплементации могут расширять `eval` разрешая программы без выражений (определения) в качестве первого аргумента и разрешая другие переменные, в качестве окружений, с тем ограничением, что `eval` не позволяет создавать новые привязки к окружениям, связанным с `null-environment` или `scheme-report-environment`.

```
(eval '(* 7 3) (scheme-report-environment 5))
                                                    ===>  21

(let ((f (eval '(lambda (f x) (f x x))
               (null-environment 5))))
  (f + 10))
                                                    ===>  20
```

процедура: (scheme-report-environment *version*)
процедура: (null-environment *version*)

Version должно являться в точности целым числом 5, в соответствии с версией Scheme данного доклада (Исправленный[5] Отчет по Scheme). `Scheme-report-environment` возвращает спецификатор среды, который являются пустым, не считая всех привязок определенных в данном докладе, которые либо необходимы, либо необязательны и поддерживаются данной имплементацией.

Другие значения *version* могут использоваться для определения окружений, соответствующих последним имплементация сообщит об ошибке, если версия не является ни 5, ни каким другим значением, поддерживаемым данной имплементацией.

Побочное действие присваивания (во время использования `eval`) переменной привязанной в окружении `scheme-report-environment` может быть неизменяемым.

дополнительная процедура: `(interaction-environment)`

Данная процедура возвращает спецификатор для окружения, которое содержит имплементационно-зависимые привязки, обычно надстройкой последних перечисленных в докладе. Суть в том, что данная процедура возвращает окружение в котором имплементация может быть выполнять выражения введенные пользователем динамически.

6.6 Ввод и вывод

6.6.1 Порты

Порты представляют устройства ввода и вывода. В Scheme, входной порт это объект Scheme, который может передавать символы через команду, в то время как порт вывода это объект Scheme, который может принимать символы.

библиотечная процедура: `(call-with-input-file` *`string proc`*`)`
библиотечная процедура: `(call-with-output-file` *`string proc`*`)`

String должен быть строкой, называющей имя файла, а *proc* должен быть процедурой, которая принимает один аргумент. Для `call-with-input-file` файл должен уже существовать, для `call-with-output-file`, действие не определено, если файл уже существует. Данные процедуры вызывают *proc* с одним аргументом: порт получается при открытием названного файла ввода или вывода. Если файл не может быть открыт, сообщается об ошибке. Если процедура *proc* возвращает управление, порт автоматически закрывается и процедурой *proc* возвращаются аргументы. Если *proc* не возвращает управление, то порт не будет автоматически закрыт до тех пор, пока не возможно доказать, что данный порт никогда больше не будет использоваться для операций чтения или записи.

> *Обоснование:* Из-за того, что процедуры возврата Scheme имеют неограниченное размер, возможен выход из данной континуации, но позже возврат назад. Если бы имплементации было разрешено закрывать порт на любом выходе из данной континуации, то было бы невозможно писать переносимый код с использованием `call-with-current-continuation`, `call-with-input-file` или `call-with-output-file`.

процедура: `(input-port?` *`obj`*`)`
процедура: `(output-port?` *`obj`*`)`

Возвращает `#t`, если *obj* является портом ввода или вывода соответственно, иначе возвращает `#f`.

процедура: `(current-input-port)`
процедура: `(current-output-port)`

Возвращает текущие порты ввода или вывода установленные по умолчанию.

дополнительная процедура: `(with-input-from-file` *`string thunk`*`)`
дополнительная процедура: `(with-output-from-file` *`string thunk`*`)`

String дожна быть строкой, называющей файл, а *proc* должна быть процедурой не принимающей аргументы. Для процедуры `with-input-from-file` всегда должен существовать файл; для процедуры `with-output-from-file`, если файл уже существует, побочное действие не определено. Файл открывается для ввода или вывода, входной или выходной порт, связанный с ним определяет значение по умолчанию возвращаемое `current-input-port` или `current-output-port` (и используется `(read)`, `(write` *`obj`*`)` и так далее), а *thunk* вызывается без аргументов. Когда *thunk* возвращает управление, порт закрывается и предыдущие значение восстанавливается. `With-input-from-file` и `with-output-to-file` возвращают значение(-я) полученные *thunk*. Если процедура возврата используется для возврата из континуации данных процедур, их поведение является имплементационно-зависимым.

процедура: `(open-input-file` *`filename`*`)`

Принимает в качестве строки имя существующего файла и возвращает входной порт, отвечающий за доставку символов из файла. Если файл не может быть открыт, будет будет передан сигнал об ошибке.

процедура: `(open-output-file` *`filename`*`)`

Принимает название выходного файла, который будет создан, в качестве строки и возвращает выходной порт, отвечающий за запись символов в новый файл с данным именем. Если файл не может быть открыт, будет просигнализировано об ошибке. Если файл с данным именем уже существует побочное действие не определено.

процедура: `(close-input-port` *`port`*`)`
процедура: `(close-output-port` *`port`*`)`

Закрывают файлы связанные с портом *port*, делая *port* не способным к доставке или получению символов. Данные операции не имеют никакого эффекта, если файл уже закрыт. Возвращаемое значение не определено.

6.6.2 Ввод

библиотечная процедура: `(read)`
библиотечная процедура: `(read` *`port`*`)`

`Read` преобразует внешнее представление объектов Scheme в сами объекты. То есть он парсит не терминальные <элементы данных> (смотри раздел 7.1.2 и 6.3.2). `Read` возвращает следующий объект, который можно распарсить из данного входного порта *port*, обновляя *port*, чтобы тот указывал на первый символ следующий за символом в конце внешнего представления данного объекта.

Если встречается конец файла во входном порту перед каким-либо символом, который может начинать объект, то возвращается объекта конца файла. Порт остается открытым, а дальнейшие попытки считать также приведут к возврату объекта конца файла. Если конец файла встретился после начала внешнего представления объекта, но внешнее представление не завершилось, а значит не парсится, то будет сообщено об ошибке.

Аргумент *port* может быть опущен, в случае чего он считается равным стандартному значению, которое возвращает `current-input-port`. Будет ошибкой считывать из закрытого порта.

процедура: `(read-char)`
процедура: `(read-char` *`port`*`)`

Возвращает следующий символ доступный из входного порта *port*, обновляя *port*, передвигая его указатель до следующего символа. Если символы больше не доступны, возвращается объект конца файла. *Port* может быть опущен, в случае чего ему присваивается стандартное значение, которое возвращает `current-input-port`.

процедура: `(peek-char)`
процедура: `(peek-char` *`port`*`)`

Возвращает следующий доступный символ из входного порта *port*, *не* обновляя *port*, сдвигая указатель до следующего символа. Если больше нет доступных символов, возвращается объект конца файла. *Port* может быть опущен, в случае чего будет использовано значение по умолчанию, возвращаемое `corrent-input-port`.

> *Замечание:* Значение возвращаемое при вызове `peek-chair` такое же, как значение, которое было бы возвращено при вызове `read-char` с таким же портом *port*. Единственное различие это то, что последующий вызов `read-char` или `peek-char` на данном порту *port* вернет значение, возвращаемое предыдущим вызовом `peek-char`. В частности, вызов `peek-char` на интерактивном порту может повиснуть в ожидании ввода, если повиснет вызов `read-char`.

процедура: `(eof-object? obj)`

Возвращает `#t`, если *obj* является объектом конца файла, иначе - `#f`. Точное множество объектов конца файла может изменяться в зависимости от имплементации, но в любом случае ни какой объект конца файла никогда не будет объектом, который может быть считан с использованием `read`.

процедура: `(char-ready?)`
процедура: `(char-ready?` *`port`*`)`

Возвращает `#t`, если символ во входном порту *port* готов и возвращает `#f` в противном случае. Если `char-ready` возвращает `#t`, то гарантировано, что следующая операция `read-char` на данном порту *port* не зависнет. Если *port* указывает на конец файла, то `char-ready?` вернет `#t`. *Port* может быть опущен, в случае чего он равен стандартному значению, возвращаемому `current-input-port`.

> *Обоснование:* `char-ready?` существует для того, чтобы у программ была возможность принимать символы от интерактивных портов, не застревая в ожидании ввода. Любой редактор ввода, связанный с такими портами должен обеспечить, чтобы символы, существование которых был проверено `char-ready?` не может быть стерто. Если `char-ready?` вернул `#f` в конце файла, порт в конце файла ни чем не будет отличаться от интерактивного порта, который не имеет готовых символов.

6.6.3 Вывод

библиотечная процедура: `(write` *`obj`*`)`
библиотечная процедура: `(write` *`obj port`*`)`

Выводит записанное представление *obj* в заданный *port*. Строки, которые появляются в записанном представлении заключаются в двойные кавычки, а внутри этих строк обратный слэш и двойные кавычки отделются обратным слэшем. Символы объектов записанные с использованием нотации `#\` . `Write` возвращает не определенное значение. Аргумент *port* может быть опущен, тогда его значение по умолчанию совпадает со значением, возвращаемым процедурой `current-output-port`.

библиотечная процедура: `(display` *`obj`*`)`
библиотечная процедура: `(display` *`obj port`*`)`

Записывает представление *obj* по заданному порту *port*. Строки, которые возникают в записанном представлении не заключаются в двойные кавычки, и никакие символы не выделяются в данных строках. Символьные объекты записываются в представлении так, как если бы они были записаны с помощью `write-chare` вместо `write`. `Display` возвращает не определенное значение. Аргумент *port* может быть опущен, тогда оно совпадает со стандартным значением, которок возвращается `current-output-port`.

> *Обоснование:* Процедура `write` предназначена для создания машинно-читаемого вывода, а `display` для создания человеко-читаемого вывода. Имплементации, которые позволяют “слэшификацию” знаков могут с некоторой вероятностью использовать `write`, но не `display` для слэшификации особых символах в знаках.

библиотечная процедура: `(newline)`
библиотечная процедура: `(newline` *`port`*`)`

Записывает конец строки в *port*. Именно таким образом, как это является корректным для данной операционной системы. Возвращает неопределенное значение. Аргумент *port* может быть опущен, в случае чего он считается равным по умолчанию значению, которое возвращается `current-output-port`.

процедура: `(write-char` *`char`*`)`
процедура: `(write-char` *`char port`*`)`

Записывает символ *char* (а не его внешнее представление) в заданный *port* и возвращает неопределенное значение. Аргумент *port* может быть опущен, в случае чего он считается равным по умолчанию значению, которое возвращает `current-output-port`.

6.6.4 Системный интерфейс

Вопросы системного интерфейса в целом выходят за рамки данного доклада. Однако, следующие операции заслуживают внимания и поэтому будут изложены в данном разделе.

дополнительная процедура: `(load` *`filename`*`)`

Filename должен быть строкой, содержащей название существующего файла, содержащего исходный код Scheme. Процедура `load` считывает выражения и определения из файла и выполняют их последовательно. Будет ли напечатан результаты выражений или нет не определено. Процедура `load` не влияет на значения возвращаемые процедурой `current-input-port` и `current-ouput-port`. `Load` возвращает неопределенное значение.

> *Обоснование:* Для портируемости, `load` должен оперировать с файлами исходного кода. Его оперирование другими типами файлов непременно варьируется в зависимости от разных имплементаций.

дополнительная процедура: `(transcript-on` *`filename`*`)`
дополнительная процедура: `(transcript-off)`

Filename должен быть строкой, содержащей название выходного файла, который будет создан. Результатом выполнения `transcript-on` является открытие указанного файла для вывода, и становиться причиной последовательных записей взаимодействия между пользователем и системой Scheme, которые записываются в файл. Запись заканчивается вызовом `transcript-off`, который закрывает файл записей. Только одна запись может выполняться в произвольный момент времени, хоть и некоторые имплементации могут игнорировать это ограничение. Значения, возвращаемые данными процедурами не определено.

Глава 7

Формальный синтаксис и семантика

Данная глава дает формальное описание того, что было описано раннее неформально в предыдущих главах данного раздела.

7.1 Формальный синтаксис

Данный раздел предоставляет формальный синтаксис Scheme, написанный в расширенной БНФ.

Все пробелы в грамматике использованы для разборчивости. Регистр не различается; например, `#x1A` и `#X1a` эквивалентны. <пустота> обозначает пустую строку.

Следующие расширения БНФ используются для того чтобы сделать описание более кратким: * (сущность) означает ноль или более вхождений ; а + означает как минимум одну <сущность>.

7.1.1 Лексическая структура

Данный подраздел описывает как отдельные токены (идентификаторы, числа и так далее) формируются из последовательности символов. Следующие секции описывают, как выражения и программы формируются из последовательности токенов.

(внутренний пробельный токен) может встречаться с любой стороны какого-либо токена, но не внутри него.

Токены, которые требуют неявного завершения (идентификаторы, числа, символы и точка) могут завершаться каким-либо (разделителем), но нет никакой необходимости в чем-либо другом

Следующие пять символов зарезервированы для последующего расширения языка: `[ ] { } |`

⟨identifier⟩ ⟶ ⟨initial⟩ ⟨subsequent⟩*
    | ⟨peculiar identifier⟩
⟨initial⟩ ⟶ ⟨letter⟩ | ⟨special initial⟩
⟨letter⟩ ⟶ a | b | c | ... | z

⟨special initial⟩ ⟶ ! | $ | % | & | * | / | : | < | =
    | > | ? | ^ | _ | ~
⟨subsequent⟩ ⟶ ⟨initial⟩ | ⟨digit⟩
    | ⟨special subsequent⟩
⟨digit⟩ ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨special subsequent⟩ ⟶ + | - | . | @
⟨peculiar identifier⟩ ⟶ + | - | ...
⟨syntactic keyword⟩ ⟶ ⟨expression keyword⟩
    | else | => | define
    | unquote | unquote-splicing
⟨expression keyword⟩ ⟶ quote | lambda | if
    | set! | begin | cond | and | or | case
    | let | let* | letrec | do | delay
    | quasiquote

⟨variable⟩ ⟶ ⟨any ⟨identifier⟩ that isn't
                also a ⟨syntactic keyword⟩⟩

⟨boolean⟩ ⟶ #t | #f
⟨character⟩ ⟶ #\ ⟨any character⟩
    | #\ ⟨character name⟩
⟨character name⟩ ⟶ space | newline

⟨string⟩ ⟶ " ⟨string element⟩* "
⟨string element⟩ ⟶ ⟨any character other than " or \⟩

Следующие правила , , , , и должны применяться для R = 2, 8, 10 и 16. Для , и нет никаких правил, которые означают, что числа содержащии точки или экспоненты должны быть в десятичной системе счисления.

⟨num R⟩ ⟶ ⟨prefix R⟩ ⟨complex R⟩
⟨complex R⟩ ⟶ ⟨real R⟩ | ⟨real R⟩ @ ⟨real R⟩
    | ⟨real R⟩ + ⟨ureal R⟩ i | ⟨real R⟩ - ⟨ureal R⟩ i
    | ⟨real R⟩ + i | ⟨real R⟩ - i
    | + ⟨ureal R⟩ i | - ⟨ureal R⟩ i | + i | - i
⟨real R⟩ ⟶ ⟨sign⟩ ⟨ureal R⟩
⟨ureal R⟩ ⟶ ⟨uinteger R⟩
    | ⟨uinteger R⟩ / ⟨uinteger R⟩
    | ⟨decimal R⟩
⟨decimal 10⟩ ⟶ ⟨uinteger 10⟩ ⟨suffix⟩
    | . ⟨digit 10⟩+ #* ⟨suffix⟩
    | ⟨digit 10⟩+ . ⟨digit 10⟩* #* ⟨suffix⟩
    | ⟨digit 10⟩+ #+ . #* ⟨suffix⟩
⟨uinteger R⟩ ⟶ ⟨digit R⟩+ #*
⟨prefix R⟩ ⟶ ⟨radix R⟩ ⟨exactness⟩
    | ⟨exactness⟩ ⟨radix R⟩

7.1.2 Внешние представления

(элемент данных) это то, что процедура `read` (раздел 6.6.2) успешно парсит. Обратите внимание, что любая строка, которая парсится как (выражение) также будет распарсена, как элемент данных .

```
⟨datum⟩ ⟶ ⟨simple datum⟩ | ⟨compound datum⟩
⟨simple datum⟩ ⟶ ⟨boolean⟩ | ⟨number⟩
    | ⟨character⟩ | ⟨string⟩ | ⟨symbol⟩
⟨symbol⟩ ⟶ ⟨identifier⟩
⟨compound datum⟩ ⟶ ⟨list⟩ | ⟨vector⟩
⟨list⟩ ⟶ (⟨datum⟩*) | (⟨datum⟩+ . ⟨datum⟩)
    | ⟨abbreviation⟩
⟨abbreviation⟩ ⟶ ⟨abbrev prefix⟩ ⟨datum⟩
⟨abbrev prefix⟩ ⟶ ' | ` | , | ,@
⟨vector⟩ ⟶ #(⟨datum⟩*)
```

7.1.3 Выражения

```
⟨expression⟩ ⟶ ⟨variable⟩
    | ⟨literal⟩
    | ⟨procedure call⟩
    | ⟨lambda expression⟩
    | ⟨conditional⟩
    | ⟨assignment⟩
    | ⟨derived expression⟩
    | ⟨macro use⟩
    | ⟨macro block⟩

⟨literal⟩ ⟶ ⟨quotation⟩ | ⟨self-evaluating⟩
⟨self-evaluating⟩ ⟶ ⟨boolean⟩ | ⟨number⟩
    | ⟨character⟩ | ⟨string⟩
⟨quotation⟩ ⟶ '⟨datum⟩ | (quote ⟨datum⟩)
⟨procedure call⟩ ⟶ (⟨operator⟩ ⟨operand⟩*)
⟨operator⟩ ⟶ ⟨expression⟩
⟨operand⟩ ⟶ ⟨expression⟩

⟨lambda expression⟩ ⟶ (lambda ⟨formals⟩ ⟨body⟩)
⟨formals⟩ ⟶ (⟨variable⟩*) | ⟨variable⟩
    | (⟨variable⟩+ . ⟨variable⟩)
⟨body⟩ ⟶ ⟨definition⟩* ⟨sequence⟩
⟨sequence⟩ ⟶ ⟨command⟩* ⟨expression⟩
⟨command⟩ ⟶ ⟨expression⟩

⟨conditional⟩ ⟶ (if ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩)
⟨test⟩ ⟶ ⟨expression⟩
⟨consequent⟩ ⟶ ⟨expression⟩
⟨alternate⟩ ⟶ ⟨expression⟩ | ⟨empty⟩
```

7.1.4 Квази цитирования

Следующее грамматика для выражений квази цитирования не являются контекстно-свободными. Они представлены, как способ генерации бесконечного числа правил создания. Представьте копию следующих правил для $D = 1, 2, 3, \ldots$. D указывает на глубину вложения.

⟨quasiquotation⟩ ⟶ ⟨quasiquotation 1⟩
⟨qq template 0⟩ ⟶ ⟨expression⟩
⟨quasiquotation D⟩ ⟶ '⟨qq template D⟩
    | (quasiquote ⟨qq template D⟩)
⟨qq template D⟩ ⟶ ⟨simple datum⟩
    | ⟨list qq template D⟩
    | ⟨vector qq template D⟩
    | ⟨unquotation D⟩
⟨list qq template D⟩ ⟶ (⟨qq template or splice D⟩*)
    | (⟨qq template or splice D⟩+ . ⟨qq template D⟩)
    | '⟨qq template D⟩
    | ⟨quasiquotation $D + 1$⟩
⟨vector qq template D⟩ ⟶ #(⟨qq template or splice D⟩*)
⟨unquotation D⟩ ⟶ ,⟨qq template $D - 1$⟩
    | (unquote ⟨qq template $D - 1$⟩)
⟨qq template or splice D⟩ ⟶ ⟨qq template D⟩
    | ⟨splicing unquotation D⟩
⟨splicing unquotation D⟩ ⟶ ,@⟨qq template $D - 1$⟩
    | (unquote-splicing ⟨qq template $D - 1$⟩)

В (квази цитированиях) (список шаблонов квази цитирования) может иногда возникать путаница между (закрывающей кавычкой) и (скошенной закрывающей кавычкой). Интерпретирование как или берет старшинство.

7.1.5 Преобразователи

⟨transformer spec⟩ ⟶
    (syntax-rules (⟨identifier⟩*) ⟨syntax rule⟩*)
⟨syntax rule⟩ ⟶ (⟨pattern⟩ ⟨template⟩)
⟨pattern⟩ ⟶ ⟨pattern identifier⟩
    | (⟨pattern⟩*)
    | (⟨pattern⟩+ . ⟨pattern⟩)
    | (⟨pattern⟩* ⟨pattern⟩ ⟨ellipsis⟩)
    | #(⟨pattern⟩*)
    | #(⟨pattern⟩* ⟨pattern⟩ ⟨ellipsis⟩)
    | ⟨pattern datum⟩
⟨pattern datum⟩ ⟶ ⟨string⟩
    | ⟨character⟩
    | ⟨boolean⟩
    | ⟨number⟩
⟨template⟩ ⟶ ⟨pattern identifier⟩
    | (⟨template element⟩*)
    | (⟨template element⟩+ . ⟨template⟩)
    | #(⟨template element⟩*)
    | ⟨template datum⟩
⟨template element⟩ ⟶ ⟨template⟩
    | ⟨template⟩ ⟨ellipsis⟩
⟨template datum⟩ ⟶ ⟨pattern datum⟩
⟨pattern identifier⟩ ⟶ ⟨any identifier except ...⟩
⟨ellipsis⟩ ⟶ ⟨the identifier ...⟩

7.1.6 Программы и определения

```
⟨program⟩ ⟶ ⟨command or definition⟩*
⟨command or definition⟩ ⟶ ⟨command⟩
    | ⟨definition⟩
    | ⟨syntax definition⟩
    | (begin ⟨command or definition⟩+)
⟨definition⟩ ⟶ (define ⟨variable⟩ ⟨expression⟩)
    | (define (⟨variable⟩ ⟨def formals⟩) ⟨body⟩)
    | (begin ⟨definition⟩*)
⟨def formals⟩ ⟶ ⟨variable⟩*
    | ⟨variable⟩* . ⟨variable⟩
⟨syntax definition⟩ ⟶
    (define-syntax ⟨keyword⟩ ⟨transformer spec⟩)
```

7.2 Формальная семантика

Данный раздел предоставляет формально описание семантики для примитивных выражений Scheme и выбранных встроенных процедур. Концепции и обозначения используемые здесь описаны в [29]; нотация прорезюмирована ниже:

$\langle \ldots \rangle$	sequence formation
$s \downarrow k$	kth member of the sequence s (1-based)
$\#s$	length of sequence s
$s \S t$	concatenation of sequences s and t
$s \dagger k$	drop the first k members of sequence s
$t \rightarrow a, b$	McCarthy conditional "if t then a else b"
$\rho[r/i]$	substitution "ρ with r for i"
r in D	injection of r into domain D
$r \mid$ D	projection of r to domain D

Причина по которой выражение континуаций принимают последовательность значений вместо единственных значений это то, что формальная обработка процедурных вызовов упрощается и возвращает значения несколько раз.

Булев флаг связанный с парами, векторами и строками вернет истину для изменяемых объектов и ложь для неизменяемых. Порядок вычисления в вызове не определен. Здесь мы имитируем это произвольными перестановками *permute* и *unpermute*, которые должны быть инверсиями, аргументов в вызове пере и после их выполнением. Это не совсем правильно, так как предполагается, некорректно, что этот порядок выполнения не изменяется на протяжении программы (для произвольного количество переданных элементов), но это более близкое приближение задуманной семантики, чем елси бы было вычисление слева-на-право.

Выделитель памяти *new* является имплементационно-зависимым, но он должен подчиняться следующей аксиоме: если $new\ \sigma \in L$, то$\sigma\ (new\ \sigma \mid L) \downarrow 2 = false$

Данное определение $\mathcal{K}$ опущено, поскольку точное определение $\mathcal{K}$ может усложнить семантику, делая ее не особо интересной.

Если P это программа в которой все переменные определены перед тем как стать привязанными или присвоенными, под P будет подразумеваться

$\mathcal{E}[\![((\texttt{lambda}\ (\mathrm{I}^*)\ \mathrm{P}')\ \langle\texttt{undefined}\rangle\ \ldots)]\!]$

, где I* это последовательность переменных определенных в P, P' это последовательность выражений полученных заменой всех определений в P на присваивание, это выражение, которое вычисляется как *undefined*, а $\mathcal{E}$ это семантическая функция, которая присваивает значение выражениям.

7.2.1 Абстрактный синтаксис

$\mathrm{K} \in \mathrm{Con}$	constants, including quotations
$\mathrm{I} \in \mathrm{Ide}$	identifiers (variables)
$\mathrm{E} \in \mathrm{Exp}$	expressions
$\Gamma \in \mathrm{Com} = \mathrm{Exp}$	commands

$$
\begin{aligned}
\mathrm{Exp} \longrightarrow\ & \mathrm{K} \mid \mathrm{I} \mid (\mathrm{E}_0\ \mathrm{E}^*) \\
& \mid (\texttt{lambda}\ (\mathrm{I}^*)\ \Gamma^*\ \mathrm{E}_0) \\
& \mid (\texttt{lambda}\ (\mathrm{I}^*\ .\ \mathrm{I})\ \Gamma^*\ \mathrm{E}_0) \\
& \mid (\texttt{lambda}\ \mathrm{I}\ \Gamma^*\ \mathrm{E}_0) \\
& \mid (\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2) \mid (\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1) \\
& \mid (\texttt{set!}\ \ \mathrm{I}\ \mathrm{E})
\end{aligned}
$$

7.2.2 Уравнения домена

$\alpha \in \mathrm{L}$		locations
$\nu \in \mathrm{N}$		natural numbers
T	$= \{\mathit{false}, \mathit{true}\}$	booleans
Q		symbols
H		characters
R		numbers
E_p	$= \mathrm{L} \times \mathrm{L} \times \mathrm{T}$	pairs
E_v	$= \mathrm{L}^* \times \mathrm{T}$	vectors
E_s	$= \mathrm{L}^* \times \mathrm{T}$	strings
M	$= \{\mathit{false}, \mathit{true}, \mathit{null}, \mathit{undefined}, \mathit{unspecified}\}$	miscellaneous
$\phi \in \mathrm{F}$	$= \mathrm{L} \times (\mathrm{E}^* \to \mathrm{K} \to \mathrm{C})$	procedure values
$\epsilon \in \mathrm{E}$	$= \mathrm{Q} + \mathrm{H} + \mathrm{R} + \mathrm{E}_p + \mathrm{E}_v + \mathrm{E}_s + \mathrm{M} + \mathrm{F}$	expressed values
$\sigma \in \mathrm{S}$	$= \mathrm{L} \to (\mathrm{E} \times \mathrm{T})$	stores
$\rho \in \mathrm{U}$	$= \mathrm{Ide} \to \mathrm{L}$	environments
$\theta \in \mathrm{C}$	$= \mathrm{S} \to \mathrm{A}$	command continuations
$\kappa \in \mathrm{K}$	$= \mathrm{E}^* \to \mathrm{C}$	expression continuations
A		answers
X		errors

7.2.3 Семантические функции

$$\mathcal{K} : \mathrm{Con} \to \mathtt{E}$$
$$\mathcal{E} : \mathrm{Exp} \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}$$
$$\mathcal{E}^* : \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}$$
$$\mathcal{C} : \mathrm{Com}^* \to \mathtt{U} \to \mathtt{C} \to \mathtt{C}$$

Определение $\mathcal{K}$ умышленно опущено.

$$\mathcal{E}[\![\mathrm{K}]\!] = \lambda\rho\kappa\,.\; \mathit{send}\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa$$

$$\begin{aligned}\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\kappa\,.\; &\mathit{hold}\,(\mathit{lookup}\;\rho\;\mathrm{I})\\ &\quad(\mathit{single}(\lambda\epsilon\,.\;\epsilon = \mathit{undefined} \to\\ &\qquad\mathit{wrong}\text{ “undefined variable”},\\ &\qquad\mathit{send}\;\epsilon\;\kappa))\end{aligned}$$

$$\begin{aligned}&\mathcal{E}[\![(\mathrm{E}_0\ \mathrm{E}^*)]\!] =\\ &\quad\lambda\rho\kappa\,.\;\mathcal{E}^*(\mathit{permute}(\langle\mathrm{E}_0\rangle \S \mathrm{E}^*))\\ &\qquad\rho\\ &\qquad(\lambda\epsilon^*\,.\;((\lambda\epsilon^*\,.\;\mathit{applicate}\,(\epsilon^* \downarrow 1)\,(\epsilon^* \dagger 1)\,\kappa)\\ &\qquad\quad(\mathit{unpermute}\;\epsilon^*)))\end{aligned}$$

$$\begin{aligned}&\mathcal{E}[\![(\mathtt{lambda}\ (\mathrm{I}^*)\ \Gamma^*\ \mathrm{E}_0)]\!] =\\ &\quad\lambda\rho\kappa\,.\;\lambda\sigma\,.\\ &\qquad\mathit{new}\,\sigma \in \mathrm{L} \to\\ &\qquad\quad\mathit{send}\,(\langle\mathit{new}\,\sigma\,|\,\mathrm{L},\\ &\qquad\qquad\lambda\epsilon^*\kappa'\,.\;\#\epsilon^* = \#\mathrm{I}^* \to\\ &\qquad\qquad\qquad\mathit{tievals}(\lambda\alpha^*\,.\;(\lambda\rho'\,.\;\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa'))\\ &\qquad\qquad\qquad\qquad(\mathit{extends}\;\rho\;\mathrm{I}^*\;\alpha^*))\\ &\qquad\qquad\qquad\qquad\epsilon^*,\\ &\qquad\qquad\qquad\mathit{wrong}\text{ “wrong number of arguments”}\rangle\\ &\qquad\qquad\text{in E})\\ &\qquad\quad\kappa\\ &\qquad\quad(\mathit{update}\,(\mathit{new}\,\sigma\,|\,\mathrm{L})\;\mathit{unspecified}\,\sigma),\\ &\qquad\mathit{wrong}\text{ “out of memory”}\;\sigma\end{aligned}$$

$$\begin{aligned}&\mathcal{E}[\![(\mathtt{lambda}\ (\mathrm{I}^*\ .\ \mathrm{I})\ \Gamma^*\ \mathrm{E}_0)]\!] =\\ &\quad\lambda\rho\kappa\,.\;\lambda\sigma\,.\\ &\qquad\mathit{new}\,\sigma \in \mathrm{L} \to\\ &\qquad\quad\mathit{send}\,(\langle\mathit{new}\,\sigma\,|\,\mathrm{L},\\ &\qquad\qquad\lambda\epsilon^*\kappa'\,.\;\#\epsilon^* \geq \#\mathrm{I}^* \to\\ &\qquad\qquad\qquad\mathit{tievalsrest}\\ &\qquad\qquad\qquad\quad(\lambda\alpha^*\,.\;(\lambda\rho'\,.\;\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa'))\\ &\qquad\qquad\qquad\qquad\quad(\mathit{extends}\;\rho\;(\mathrm{I}^* \S \langle\mathrm{I}\rangle)\;\alpha^*))\\ &\qquad\qquad\qquad\quad\epsilon^*\\ &\qquad\qquad\qquad\quad(\#\mathrm{I}^*),\\ &\qquad\qquad\qquad\mathit{wrong}\text{ “too few arguments”}\rangle\text{ in E})\\ &\qquad\quad\kappa\\ &\qquad\quad(\mathit{update}\,(\mathit{new}\,\sigma\,|\,\mathrm{L})\;\mathit{unspecified}\,\sigma),\\ &\qquad\mathit{wrong}\text{ “out of memory”}\;\sigma\end{aligned}$$

$$\mathcal{E}[\![(\mathtt{lambda}\ \mathrm{I}\ \Gamma^*\ \mathrm{E}_0)]\!] = \mathcal{E}[\![(\mathtt{lambda}\ (.\ \mathrm{I})\ \Gamma^*\ \mathrm{E}_0)]\!]$$

$$\begin{aligned}&\mathcal{E}[\![(\mathtt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!] =\\ &\quad\lambda\rho\kappa\,.\;\mathcal{E}[\![\mathrm{E}_0]\!]\;\rho\;(\mathit{single}(\lambda\epsilon\,.\;\mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa,\\ &\qquad\qquad\qquad\qquad\mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa))\end{aligned}$$

$$\mathcal{E}[\![(\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1)]\!] =$$
$$\quad \lambda\rho\kappa\,.\ \mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\ \mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa,$$
$$\qquad\qquad\qquad\qquad \mathit{send}\,\mathit{unspecified}\,\kappa))$$

Здесь и в другом месте, некоторое полученное значение отличное от *undefined* может быть использовано вместо *unspecified*.

$$\mathcal{E}[\![(\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1)]\!] =$$
$$\quad \lambda\rho\kappa\,.\ \mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\ \mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa,$$
$$\qquad\qquad\qquad\qquad \mathit{send}\,\mathit{unspecified}\,\kappa))$$

$$\mathcal{E}[\![(\texttt{set!}\ \mathrm{I}\ \mathrm{E})]\!] =$$
$$\quad \lambda\rho\kappa\,.\ \mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(\mathit{single}(\lambda\epsilon\,.\ \mathit{assign}\,(\mathit{lookup}\,\rho\,\mathrm{I})$$
$$\qquad\qquad\qquad\qquad\qquad\qquad \epsilon$$
$$\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{send}\,\mathit{unspecified}\,\kappa)))$$

$$\mathcal{E}^*[\![\,]\!] = \lambda\rho\kappa\,.\ \kappa\langle\,\rangle$$

$$\mathcal{E}^*[\![\mathrm{E}_0\,\mathrm{E}^*]\!] =$$
$$\quad \lambda\rho\kappa\,.\ \mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}(\lambda\epsilon_0\,.\ \mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\,(\lambda\epsilon^*\,.\ \kappa\,(\langle\epsilon_0\rangle\,\S\,\epsilon^*))))$$

$$\mathcal{C}[\![\,]\!] = \lambda\rho\theta\,.\ \theta$$

7.2.4 Вспомогательные функции

$$\mathit{lookup} : \mathtt{U} \to \mathrm{Ide} \to \mathtt{L}$$
$$\mathit{lookup} = \lambda\rho\mathrm{I}\,.\ \rho\mathrm{I}$$

$$\mathit{extends} : \mathtt{U} \to \mathrm{Ide}^* \to \mathtt{L}^* \to \mathtt{U}$$
$$\mathit{extends} =$$
$$\quad \lambda\rho\mathrm{I}^*\alpha^*\,.\ \#\mathrm{I}^* = 0 \to \rho,$$
$$\qquad\qquad \mathit{extends}\,(\rho[(\alpha^* \downarrow 1)/(\mathrm{I}^* \downarrow 1)])\,(\mathrm{I}^* \dagger 1)\,(\alpha^* \dagger 1)$$

$$\mathit{wrong} : \mathtt{X} \to \mathtt{C} \qquad \text{[implementation-dependent]}$$

$$\mathit{send} : \mathrm{E} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{send} = \lambda\epsilon\kappa\,.\ \kappa\langle\epsilon\rangle$$

$$\mathit{single} : (\mathrm{E} \to \mathtt{C}) \to \mathtt{K}$$
$$\mathit{single} =$$
$$\quad \lambda\psi\epsilon^*\,.\ \#\epsilon^* = 1 \to \psi(\epsilon^* \downarrow 1),$$
$$\qquad\qquad \mathit{wrong}\ \text{“wrong number of return values”}$$

$$\mathit{new} : \mathtt{S} \to (\mathtt{L} + \{\mathit{error}\}) \qquad \text{[implementation-dependent]}$$

$$\mathit{hold} : \mathtt{L} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{hold} = \lambda\alpha\kappa\sigma\,.\ \mathit{send}\,(\sigma\,\alpha \downarrow 1)\kappa\sigma$$

$$\mathit{assign} : \mathtt{L} \to \mathrm{E} \to \mathtt{C} \to \mathtt{C}$$
$$\mathit{assign} = \lambda\alpha\epsilon\theta\sigma\,.\ \theta(\mathit{update}\,\alpha\epsilon\sigma)$$

$$\mathit{update} : \mathtt{L} \to \mathrm{E} \to \mathtt{S} \to \mathtt{S}$$
$$\mathit{update} = \lambda\alpha\epsilon\sigma\,.\ \sigma[\langle\epsilon, \mathit{true}\rangle/\alpha]$$

$$\mathit{tievals} : (\mathtt{L}^* \to \mathtt{C}) \to \mathrm{E}^* \to \mathtt{C}$$
$$\mathit{tievals} =$$
$$\quad \lambda\psi\epsilon^*\sigma\,.\ \#\epsilon^* = 0 \to \psi\langle\,\rangle\sigma,$$
$$\qquad \mathit{new}\,\sigma \in \mathtt{L} \to \mathit{tievals}(\lambda\alpha^*\,.\ \psi(\langle \mathit{new}\,\sigma \mid \mathtt{L}\rangle\,\S\,\alpha^*))$$
$$\qquad\qquad\qquad\qquad\qquad\qquad (\epsilon^* \dagger 1)$$
$$\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L})(\epsilon^* \downarrow 1)\sigma),$$
$$\qquad\quad \mathit{wrong}\ \text{“out of memory”}\,\sigma$$

$$
\begin{array}{l}
\mathit{tievalsrest} : (\mathrm{L}^* \to \mathrm{C}) \to \mathrm{E}^* \to \mathrm{N} \to \mathrm{C} \\
\mathit{tievalsrest} = \\
\quad \lambda\psi\epsilon^*\nu \,.\, \mathit{list}\,(\mathit{dropfirst}\,\epsilon^*\nu) \\
\qquad\qquad (\mathit{single}(\lambda\epsilon \,.\, \mathit{tievals}\,\psi\,((\mathit{takefirst}\,\epsilon^*\nu)\,\S\,\langle\epsilon\rangle)))
\end{array}
$$

$$
\mathit{dropfirst} = \lambda l n \,.\, n = 0 \to l, \mathit{dropfirst}\,(l \dagger 1)(n-1)
$$

$$
\mathit{takefirst} = \lambda l n \,.\, n = 0 \to \langle\,\rangle, \langle l \downarrow 1\rangle\,\S\,(\mathit{takefirst}\,(l \dagger 1)(n-1))
$$

$$
\begin{array}{l}
\mathit{truish} : \mathrm{E} \to \mathrm{T} \\
\mathit{truish} = \lambda\epsilon \,.\, \epsilon = \mathit{false} \to \mathit{false}, \mathit{true}
\end{array}
$$

$\mathit{permute} : \mathrm{Exp}^* \to \mathrm{Exp}^*$ [implementation-dependent]

$\mathit{unpermute} : \mathrm{E}^* \to \mathrm{E}^*$ [inverse of *permute*]

$$
\begin{array}{l}
\mathit{applicate} : \mathrm{E} \to \mathrm{E}^* \to \mathrm{K} \to \mathrm{C} \\
\mathit{applicate} = \\
\quad \lambda\epsilon\epsilon^*\kappa \,.\, \epsilon \in \mathrm{F} \to (\epsilon \mid \mathrm{F} \downarrow 2)\epsilon^*\kappa, \mathit{wrong}\ \text{“bad procedure”}
\end{array}
$$

$$
\begin{array}{l}
\mathit{onearg} : (\mathrm{E} \to \mathrm{K} \to \mathrm{C}) \to (\mathrm{E}^* \to \mathrm{K} \to \mathrm{C}) \\
\mathit{onearg} = \\
\quad \lambda\zeta\epsilon^*\kappa \,.\, \#\epsilon^* = 1 \to \zeta(\epsilon^* \downarrow 1)\kappa, \\
\qquad\qquad \mathit{wrong}\ \text{“wrong number of arguments”}
\end{array}
$$

$$
\begin{array}{l}
\mathit{twoarg} : (\mathrm{E} \to \mathrm{E} \to \mathrm{K} \to \mathrm{C}) \to (\mathrm{E}^* \to \mathrm{K} \to \mathrm{C}) \\
\mathit{twoarg} = \\
\quad \lambda\zeta\epsilon^*\kappa \,.\, \#\epsilon^* = 2 \to \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\kappa, \\
\qquad\qquad \mathit{wrong}\ \text{“wrong number of arguments”}
\end{array}
$$

$$
\begin{array}{l}
\mathit{list} : \mathrm{E}^* \to \mathrm{K} \to \mathrm{C} \\
\mathit{list} = \\
\quad \lambda\epsilon^*\kappa \,.\, \#\epsilon^* = 0 \to \mathit{send}\,\mathit{null}\,\kappa, \\
\qquad\qquad \mathit{list}\,(\epsilon^* \dagger 1)(\mathit{single}(\lambda\epsilon \,.\, \mathit{cons}\langle\epsilon^* \downarrow 1, \epsilon\rangle\kappa))
\end{array}
$$

$$
\begin{array}{l}
\mathit{cons} : \mathrm{E}^* \to \mathrm{K} \to \mathrm{C} \\
\mathit{cons} = \\
\quad \mathit{twoarg}(\lambda\epsilon_1\epsilon_2\kappa\sigma \,.\, \mathit{new}\,\sigma \in \mathrm{L} \to \\
\qquad\qquad\qquad (\lambda\sigma' \,.\, \mathit{new}\,\sigma' \in \mathrm{L} \to \\
\qquad\qquad\qquad\qquad \mathit{send}(\langle \mathit{new}\,\sigma \mid \mathrm{L}, \mathit{new}\,\sigma' \mid \mathrm{L}, \mathit{true}\rangle \\
\qquad\qquad\qquad\qquad\qquad \text{in E}) \\
\qquad\qquad\qquad\qquad\qquad \kappa \\
\qquad\qquad\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma' \mid \mathrm{L})\epsilon_2\sigma'), \\
\qquad\qquad\qquad\qquad \mathit{wrong}\ \text{“out of memory”}\sigma') \\
\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma \mid \mathrm{L})\epsilon_1\sigma), \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“out of memory”}\sigma)
\end{array}
$$

$$
\begin{array}{l}
\mathit{less} : \mathrm{E}^* \to \mathrm{K} \to \mathrm{C} \\
\mathit{less} = \\
\quad \mathit{twoarg}(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathrm{R} \wedge \epsilon_2 \in \mathrm{R}) \to \\
\qquad\qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathrm{R} < \epsilon_2 \mid \mathrm{R} \to \mathit{true}, \mathit{false})\kappa, \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“non-numeric argument to <”})
\end{array}
$$

$$
\begin{array}{l}
\mathit{add} : \mathrm{E}^* \to \mathrm{K} \to \mathrm{C} \\
\mathit{add} = \\
\quad \mathit{twoarg}(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathrm{R} \wedge \epsilon_2 \in \mathrm{R}) \to \\
\qquad\qquad\qquad \mathit{send}\,((\epsilon_1 \mid \mathrm{R} + \epsilon_2 \mid \mathrm{R})\ \text{in E})\kappa, \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“non-numeric argument to +”})
\end{array}
$$

```
car : E* → K → C
car =
  onearg(λεκ . ε ∈ Ep → hold(ε | Ep ↓ 1)κ,
                urong "non-pair argument to car")

cdr : E* → K → C      [similar to car]

setcar : E* → K → C
setcar =
  twoarg(λε1ε2κ . ε1 ∈ Ep →
                (ε1 | Ep ↓ 3) → assign(ε1 | Ep ↓ 1)
                                      ε2
                                      (send unspecified κ),
                urong "immutable argument to set-car!",
                urong "non-pair argument to set-car!")

eqv : E* → K → C
eqv =
  twoarg(λε1ε2κ . (ε1 ∈ M ∧ ε2 ∈ M) →
                   send(ε1 | M = ε2 | M → true, false)κ,
                (ε1 ∈ Q ∧ ε2 ∈ Q) →
                   send(ε1 | Q = ε2 | Q → true, false)κ,
                (ε1 ∈ H ∧ ε2 ∈ H) →
                   send(ε1 | H = ε2 | H → true, false)κ,
                (ε1 ∈ R ∧ ε2 ∈ R) →
                   send(ε1 | R = ε2 | R → true, false)κ,
                (ε1 ∈ Ep ∧ ε2 ∈ Ep) →
                   send((λp1p2 . ((p1 ↓ 1) = (p2 ↓ 1)∧
                                  (p1 ↓ 2) = (p2 ↓ 2)) → true,
                                  false)
                        (ε1 | Ep)
                        (ε2 | Ep))
                        κ,
                (ε1 ∈ Ev ∧ ε2 ∈ Ev) → ...,
                (ε1 ∈ Es ∧ ε2 ∈ Es) → ...,
                (ε1 ∈ F ∧ ε2 ∈ F) →
                   send((ε1 | F ↓ 1) = (ε2 | F ↓ 1) → true, false)
                        κ,
                   send false κ)

apply : E* → K → C
apply =
  twoarg(λε1ε2κ . ε1 ∈ F → valueslist⟨ε2⟩(λε* . applicate ε1ε*κ),
                   urong "bad procedure argument to apply")>

valueslist : E* → K → C
valueslist =
  onearg(λεκ . ε ∈ Ep →
                cdr⟨ε⟩
                   (λε* . valueslist
                          ε*
                          (λε* . car⟨ε⟩(single(λε . κ(⟨ε⟩ § ε*))))),
              ε = null → κ⟨ ⟩,
                urong "non-list argument to values-list")
```

```
cwcc : E* → K → C     [call-with-current-continuation]
cwcc =
  onearg(λεκ . ε ∈ F →
                (λσ . new σ ∈ L →
                       applicate ε
                                 ⟨⟨new σ | L, λε*κ' . κε*⟩ in E⟩
                                 κ
                                 (update(new σ | L)
                                        unspecified
                                        σ),
                       wrong "out of memory" σ),
                wrong "bad procedure argument")

values : E* → K → C
values = λε*κ . κε*

cwv : E* → K → C     [call-with-values]
cwv =
  twoarg(λε1ε2κ . applicate ε1 ⟨⟩(λε* . applicate ε2 ε*))
```

7.3 Производные типы выражений

Данный раздел дает макро определения для производных типов выражений с точки зрения примитивных типов выражений (литерал, переменная, вызов, `lambda`, `id`, `set!`). Смотри раздел 6.4, где указано возможное определение `delay`.

```scheme
(define-syntax cond
  (syntax-rules (else =>)
    ((cond (else result1 result2 ...))
     (begin result1 result2 ...))
    ((cond (test => result))
     (let ((temp test))
       (if temp (result temp))))
    ((cond (test => result) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           (result temp)
           (cond clause1 clause2 ...))))
    ((cond (test)) test)
    ((cond (test) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           temp
           (cond clause1 clause2 ...))))
    ((cond (test result1 result2 ...))
     (if test (begin result1 result2 ...)))
    ((cond (test result1 result2 ...)
           clause1 clause2 ...)
     (if test
         (begin result1 result2 ...)
         (cond clause1 clause2 ...)))))


(define-syntax case
  (syntax-rules (else)
    ((case (key ...)
       clauses ...)
     (let ((atom-key (key ...)))
       (case atom-key clauses ...)))
    ((case key
       (else result1 result2 ...))
     (begin result1 result2 ...))
    ((case key
```

```
      ((atoms ...) result1 result2 ...))
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)))
    ((case key
       ((atoms ...) result1 result2 ...)
       clause clauses ...)
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)
         (case key clause clauses ...)))))


(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and test) test)
    ((and test1 test2 ...)
     (if test1 (and test2 ...) #f))))


(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or test) test)
    ((or test1 test2 ...)
     (let ((x test1))
       (if x x (or test2 ...))))))


(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))
    ((let tag ((name val) ...) body1 body2 ...)
     ((letrec ((tag (lambda (name ...)
                      body1 body2 ...)))
        tag)
      val ...))))


(define-syntax let*
  (syntax-rules ()
    ((let* () body1 body2 ...)
     (let () body1 body2 ...))
    ((let* ((name1 val1) (name2 val2) ...)
       body1 body2 ...)
     (let ((name1 val1))
       (let* ((name2 val2) ...)
         body1 body2 ...)))))
```

Следующий макрос letrec использует знак на месте выражения, которое возвращает то, что при попытке записи в локацию получить значение из данной локации, приводит к ошибке (таких выражений не определено в Scheme). Для генерации временных имен, необходимых для избежания указания порядка в котором значения выполняются используется трюк. Он может быть выполнен с использованием вспомогательного макроса.

```
(define-syntax letrec
  (syntax-rules ()
    ((letrec ((var1 init1) ...) body ...)
     (letrec "generate_temp_names"
       (var1 ...)
       ()
       ((var1 init1) ...)
       body ...))
    ((letrec "generate_temp_names"
       ()
```

```
   (temp1 ...)
   ((var1 init1) ...)
   body ...)
 (let ((var1 <undefined>) ...)
   (let ((temp1 init1) ...)
     (set! var1 temp1)
     ...
     body ...)))
((letrec "generate_temp_names"
   (x y ...)
   (temp ...)
   ((var1 init1) ...)
   body ...)
 (letrec "generate_temp_names"
   (y ...)
   (newtemp temp ...)
   ((var1 init1) ...)
   body ...))))
```

```
(define-syntax begin
  (syntax-rules ()
    ((begin exp ...)
     ((lambda () exp ...)))))
```

Следующее альтернативное расширение `begin` не использует способность записывать более одного выражение в тело лямбда выражения. В любом случе, обратите внимание, что данные правила применяются только, если тело `begin` не содержит определений.

```
(define-syntax begin
  (syntax-rules ()
    ((begin exp)
     exp)
    ((begin exp1 exp2 ...)
     (let ((x exp1))
       (begin exp2 ...)))))
```

Следующее определение `do` использует трюк для расширения клауз переменных. Как в случае с `letrec`, указанном выше, вспомогательный макрос может также сработать. Выражение `(if #f #f)` используется для того чтобы добиться неопределенного значения.

```
(define-syntax do
  (syntax-rules ()
    ((do ((var init step ...) ...)
         (test expr ...)
         command ...)
     (letrec
       ((loop
         (lambda (var ...)
           (if test
               (begin
                 (if #f #f)
                 expr ...)
               (begin
                 command
                 ...
                 (loop (do "step" var step ...)
                       ...))))))
       (loop init ...)))
    ((do "step" x)
     x)
    ((do "step" x y)
     y)))
```

Заметки

Изменения языка

Данный раздел перечисляет изменения, которые были сделаны в Scheme, после публикации “Исправленного^4 доклада”6.

- Сейчас доклад является надмножеством IEEE стандарта Scheme[13] (#sec13): имплементации удовлетворяющие данному докладу будут также удовлетворять данному стандарту. Для этого требуются следующие изменения:
- Пустой список теперь вычисляется, как истина.
- Классификация функций на основные и неосновные была убрана. Сейчас существует три класса встроенных процедур: примитивные, библиотечные и дополнительные. Дополнительные процедуры это `load`, `with-input-from-file`, `with-ouput-to-file`, `transcript-on`, `traanscript-off` и `interaction-environment`, и `-`, `/` с более чем двуся аргументами. Ничего из этого нет в IEEE стандарте.
- Программы позволяют переопределять встроенные процедуры. При этом если это сделать, то поведение других встроенных процедура не измениться.
- *Port* был добавлен в список не пересекающихся типов.
- Макро дополнения были удалены. Теперь макросы верхнего уровня являются частью главного части данного доклада. Правила перезаписи для производных выражений были заменены макро определениями. Идентификаторы не зарезервированы.
- `Syntax-rules` теперь допускают шаблоны векторов.
- Возвращаются множественные значения, были добавлены `eval` и `dynamic-wind`
- Вызовы, которые должны быть реализованы в стиле правильной хвостовой рекурсии объявляются явно.
- `@` может быть использована в пределах идентификатора. `|` зарезервирована для для возможных будущих расширений.

Дополнительный материал

Репозиторий Scheme в Интернете

```
http://www.cs.indiana.edu/scheme-repository/
```

содержит расширенную библиографию Schem, такую как друменты, программы, имплементации и другие материалы, которые относяться к Scheme.

Пример

`Integrate-system` интегрирует систему

$$y_k' = f_k(y_1, y_2, \ldots, y_n), \quad k = 1, \ldots, n$$

дифференциальных уравнений методом Ранга-Кутта.

Параметр `system-derivate` это функция, которая принимает состояние системы (вектор значений для переменных состояния $y_1, \ldots, y_n$) и создает систему производных (значения y_1', …, y_n'). Параметр `initial-store` предоставляет начальное состояние системы, а h это предполагаемая в начальный момент длина шага интегрирования.

Значение, возвращаемое `integrate-system`, это бесконечный поток состояний системы.

```
(define integrate-system
  (lambda (system-derivative initial-state h)
    (let ((next (runge-kutta-4 system-derivative h)))
      (letrec ((states
                (cons initial-state
                      (delay (map-streams next
                                          states)))))
        states))))
```

`Runge-Kutta-4` принимает функцию f, которая создает систему производных из системы состояния. `Runge-Kutta-4` создает функцию, которая принимает состояние системы и создает новую.

```
(define runge-kutta-4
  (lambda (f h)
    (let ((*h (scale-vector h))
          (*2 (scale-vector 2))
          (*1/2 (scale-vector (/ 1 2)))
          (*1/6 (scale-vector (/ 1 6))))
      (lambda (y)
        ;; y is a system state
        (let* ((k0 (*h (f y)))
               (k1 (*h (f (add-vectors y (*1/2 k0)))))
               (k2 (*h (f (add-vectors y (*1/2 k1)))))
               (k3 (*h (f (add-vectors y k2)))))
          (add-vectors y
            (*1/6 (add-vectors k0
                               (*2 k1)
                               (*2 k2)
                               k3)))))))))
```

```
(define elementwise
  (lambda (f)
    (lambda vectors
      (generate-vector
        (vector-length (car vectors))
        (lambda (i)
          (apply f
                 (map (lambda (v) (vector-ref  v i))
                      vectors)))))))
(define generate-vector
  (lambda (size proc)
    (let ((ans (make-vector size)))
      (letrec ((loop
                (lambda (i)
                  (cond ((= i size) ans)
                        (else
                         (vector-set! ans i (proc i))
                         (loop (+ i 1)))))))
        (loop 0)))))
(define add-vectors (elementwise +))
(define scale-vector
  (lambda (s)
    (elementwise (lambda (x) (* x s)))))
```

`Map-streams` это аналог `map`: он применяет первый аргумент (процедуру) ко всем элементам второго аргумента (потока).

```
(define map-streams
  (lambda (f s)
    (cons (f (head s))
          (delay (map-streams f (tail s))))))
```

Бесконечные потоки реализуются как пары, car которых хранит первый элемент потока, а cdr которых хранит обещание доставить остальную часть потока.

```
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))
```

Следующие выкладки иллюстрируют использование `integrate-system` в интегрирующей системы.

$$C\frac{dU_C}{dt} = -i_L - \frac{v_C}{R}$$

$$L\frac{di_L}{dt} = v_C$$

который моделирует затухающий осциллятор

```
(define damped-oscillator
  (lambda (R L C)
    (lambda (state)
      (let ((Vc (vector-ref state 0))
            (Il (vector-ref state 1)))
        (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
                (/ Vc L))))))
(define the-states
  (integrate-system
     (damped-oscillator 10000 1000 .001)
     '#(1 0)
     .01))
```

Библиография

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. Structure and Interpretation of Computer Programs, second edition. MIT Press, Cambridge, 1996.

[2] Alan Bawden and Jonathan Rees. Syntactic closures. In Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming, pages 86-95.

[3] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In Proceedings of the ACM SIGPLAN ’96 Conference on Programming Language Design and Implementation, pages 108-116.

[4] William Clinger, editor. The revised revised report on Scheme, or an uncommon Lisp. MIT Artificial Intelligence Memo 848, August 1985. Also published as Computer Science Department Technical Report 174, Indiana University, June 1985.

[5] William Clinger. How to read floating point numbers accurately. In Proceedings of the ACM SIGPLAN ’90 Conference on Programming Language Design and Implementation, pages 92-101. Proceedings published as SIGPLAN Notices 25(6), June 1990.

[6] William Clinger and Jonathan Rees, editors. The revised4 report on the algorithmic language Scheme. In ACM Lisp Pointers 4(3), pages 1-55, 1991.

[7] William Clinger and Jonathan Rees. Macros that work. In Proceedings of the 1991 ACM Conference on Principles of Programming Languages, pages 155-162.

[8] William Clinger. Proper Tail Recursion and Space Efficiency. To appear in Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation, June 1998.

[9] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. Lisp and Symbolic Computation 5(4):295-326, 1993.

[10] Carol Fessenden, William Clinger, Daniel P. Friedman, and Christopher Haynes. Scheme 311 version 4 reference manual. Indiana University Computer Science Technical Report 137, February 1983. Superseded by [11].

[11] D. Friedman, C. Haynes, E. Kohlbecker, and M. Wand. Scheme 84 interim reference manual. Indiana University Computer Science Technical Report 153, January 1985.

[12] IEEE Standard 754-1985. IEEE Standard for Binary Floating-Point Arithmetic. IEEE, New York, 1985.

[13] IEEE Standard 1178-1990. IEEE Standard for the Scheme Programming Language. IEEE, New York, 1991.

[14] Eugene E. Kohlbecker Jr. Syntactic Extensions in the Programming Language Lisp. PhD thesis, Indiana University, August 1986.

[15] Eugene E. Kohlbecker Jr., Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, pages 151-161.

[16] Peter Landin. A correspondence between Algol 60 and Church’s lambda notation: Part I. Communications of the ACM 8(2):89-101, February 1965.

[17] MIT Department of Electrical Engineering and Computer Science. Scheme manual, seventh edition. September 1984.

[18] Peter Naur et al. Revised report on the algorithmic language Algol 60. Communications of the ACM 6(1):1-17, January 1963.

[19] Paul Penfield, Jr. Principal values and branch cuts in complex APL. In APL ’81 Conference Proceedings, pages 248-256. ACM SIGAPL, San Francisco, September 1981. Proceedings published as APL Quote Quad 12(1), ACM, September 1981.

[20] Kent M. Pitman. The revised MacLisp manual (Saturday evening edition). MIT Laboratory for Computer Science Technical Report 295, May 1983.

[21] Jonathan A. Rees and Norman I. Adams IV. T: A dialect of Lisp or, lambda: The ultimate software tool. In Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, pages 114-122.

[22] Jonathan A. Rees, Norman I. Adams IV, and James R. Meehan. The T manual, fourth edition. Yale University Computer Science Department, January 1984.

[23] Jonathan Rees and William Clinger, editors. The revised3 report on the algorithmic language Scheme. In ACM SIGPLAN Notices 21(12), pages 37-79, December 1986.

[24] John Reynolds. Definitional interpreters for higher order programming languages. In ACM Conference Proceedings, pages 717-740. ACM, 1972.

[25] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, January 1978.

[26] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[27] Guy Lewis Steele Jr. Common Lisp: The Language, second edition. Digital Press, Burlington MA, 1990.

[28] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349, December 1975.

[29] Joseph E. Stoy. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, 1977.

[30] Texas Instruments, Inc. TI Scheme Language Reference Manual. Preliminary version 1.0, November 1985.