

Revised⁶ Report on the Algorithmic Language Scheme

MICHAEL SPERBER

R. KENT DYBVIG, MATTHEW FLATT, ANTON VAN STRAATEN

(Editors)

RICHARD KELSEY, WILLIAM CLINGER, JONATHAN REES

(Editors, Revised⁵ Report on the Algorithmic Language Scheme)

ROBERT BRUCE FINDLER, JACOB MATTHEWS

(Authors, formal semantics)

26 September 2007

SUMMARY

The report gives a defining description of the programming language Scheme. Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including functional, imperative, and message passing styles, find convenient expression in Scheme.

This report is accompanied by a report describing standard libraries [?]; references to this document are identified by designations such as “library section” or “library chapter”. It is also accompanied by a report containing non-normative appendices [?]. A fourth report gives some historical background and rationales for many aspects of the language and its libraries [?].

The individuals listed above are not the sole authors of the text of the report. Over the years, the following individuals were involved in discussions contributing to the design of the Scheme language, and were listed as authors of prior reports:

Hal Abelson, Norman Adams, David Bartley, Gary Brooks, William Clinger, R. Kent Dybvig, Daniel Friedman, Robert Halstead, Chris Hanson, Christopher Haynes, Eugene Kohlbecker, Don Oxley, Kent Pitman, Jonathan Rees, Guillermo Rozas, Guy L. Steele Jr., Gerald Jay Sussman, and Mitchell Wand.

In order to highlight recent contributions, they are not listed as authors of this version of the report. However, their contribution and service is gratefully acknowledged.

We intend this report to belong to the entire Scheme community, and so we grant permission to copy it in whole or in part without fee. In particular, we encourage implementors of Scheme to use this report as a starting point for manuals and other documentation, modifying it as necessary.

CONTENTS

Introduction

Описание языка

1 Введение в Scheme

1.1	Основные типы	6
1.2	Выражения	7
1.3	Переменные и привязка	7
1.4	Определения	7
1.5	Формы	8
1.6	Процедуры	8
1.7	Вызовы процедур и синтаксические ключевые слова	8
1.8	Присваивание	8
1.9	Производные формы и макросы	9
1.10	Синтаксические данные и значения datum	9
1.11	Продолжения	9
1.12	Библиотеки	10
1.13	Программы верхнего уровня	10

2 Уровни требований

3 Числа

3.1	Числовая башня	11
3.2	Точность	11
3.3	Fixnums and flonums	11
3.4	Требования к реализациям	11
3.5	Бесконечность и NaN	12
3.6	Распознавание -0.0	12

4 Лексический и datum-синтаксис

4.1	Нотация	13
4.2	Лексический синтаксис	13
4.3	Datum-синтаксис	17

5 Семантические концепции

5.1	Программы и библиотеки	18
5.2	Переменные, ключевые слова и регионы	18
5.3	Исключительные ситуации	19
5.4	Проверка аргументов	19
5.5	Нарушения синтаксиса	20
5.6	Безопасность	20
5.7	Булевы значения	20
5.8	Несколько возвращаемых значений	20
5.9	Неопределённое поведение	21
5.10	Модель памяти	21
5.11	Чистая хвостовая рекурсия	21
5.12	Динамический экстенд и динамическое окружение	21

6 Entry format

6.1	Syntax entries	22
6.2	Procedure entries	22
6.3	Implementation responsibilities	23
6.4	Other kinds of entries	23
6.5	Equivalent entries	23
6.6	Evaluation examples	23
6.7	Naming conventions	24

7 Библиотеки

7.1	Библиотечная форма	24
7.2	Уровни экспорта и импорта	27
7.3	Примеры	28

8 Top-level programs

8.1	Top-level program syntax	29
8.2	Top-level program semantics	29

9 Primitive syntax

9.1	Primitive expression types	29
9.2	Macros	30

10 Expansion process

11 Base library

11.1	Base types	32
11.2	Definitions	32
11.3	Bodies	33
11.4	Expressions	33
11.5	Equivalence predicates	37
11.6	Procedure predicate	39
11.7	Arithmetic	39
11.8	Booleans	46
11.9	Pairs and lists	47
11.10	Symbols	48
11.11	Characters	49
11.12	Strings	49
11.13	Vectors	50
11.14	Errors and violations	51
11.15	Control features	52
11.16	Iteration	53
11.17	Quasiquote	54
11.18	Binding constructs for syntactic keywords	55
11.19	Macro transformers	56
11.20	Tail calls and tail contexts	57

Appendices

A	Formal semantics	59
A.1	Background	59
A.2	Grammar	59
A.3	Quote	61
A.4	Multiple values	61
A.5	Exceptions	62
A.6	Arithmetic and basic forms	62
A.7	Lists	63
A.8	Eqv	63
A.9	Procedures and application	63
A.10	Call/cc and dynamic wind	64
A.11	Letrec	64
A.12	Underspecification	65
B	Sample definitions for derived forms	65
C	Additional material	67
D	Example	68
E	Language changes	69

INTRODUCTION

Языки программирования должны разрабатываться не путём накопления функциональных возможностей, а путём удаления слабостей и ограничений, приводящих к кажущейся необходимости дополнительных средств. Scheme демонстрирует, что очень небольшого количества правил для построения выражений без ограничений на их составления достаточно для формирования практического и эффективно-го языка программирования, который является достаточно гибким, чтобы поддержать большинство основных парадигм программирования, используемых в настоящее время.

Scheme был одним из первых языков программирования, который включает полноценные процедуры, как в лямбда-исчислении, таким образом доказывая полноценность статических правил области видимости и блочной структуры в языке с динамической типизацией. Scheme был первым основным диалектом, Lisp, чтобы отличить процедуры от лямбда-выражений и символов, чтобы использовать единственное лексическое окружение для всех переменных, и чтобы вычислить позицию оператора вызова процедуры таким же образом как позицию операнда. Полагаясь полностью на запросы процедуры выражать повторение, Scheme подчеркнул факт, что рекурсивные хвостом запросы процедуры являются по существу *gotos* теми аргументами прохода. Scheme был первым широко используемым языком программирования, который охватит первоклассные процедуры спасения, из которых могут синтезироваться все предварительно известные последовательные структуры контроля-управления. Последующая версия Scheme вводила понятие-концепцию точных и неточных объектов числаномера, расширение родовой арифметики ОбщегоОбычного Лиспа. Позже, Scheme стал первым языком программирования, который поддержит гигиеническое макроопределениемакрос, которые разрешают синтаксису языка с блочной структурой быть расширенным в последовательной и надёжной манере.

Руководящие принципы

Чтобы помогать вести усилие по стандартизации, редакторы приняли ряд принципов, представленных ниже. Как язык Scheme, определенный в *Revised⁵ Report on the Algorithmic Language Scheme* [?] язык, описанный в этом сообщении предназначен:

- позвольте программистам читать код друг друга, и позволять разработку портативных программ, которые могут быть выполнены в любой соответствующей реализации Scheme;
- derive its power from simplicity, a small number of generally useful core syntactic forms and procedures, and no unnecessary restrictions on how they are composed;
- allow programs to define new procedures and new hygienic syntactic forms;
- support the representation of program source code as data;

- make procedure calls powerful enough to express any form of sequential control, and allow programs to perform non-local control operations without the use of global program transformations;
- allow interesting, purely functional programs to run indefinitely without terminating or running out of memory on finite-memory machines;
- allow educators to use the language to teach programming effectively, at various levels and with a variety of pedagogical approaches; and
- allow researchers to use the language to explore the design, implementation, and semantics of programming languages.

In addition, this report is intended to:

- allow programmers to create and distribute substantial programs and libraries, e.g., implementations of Scheme Requests for Implementation, that run without modification in a variety of Scheme implementations;
- support procedural, syntactic, and data abstraction more fully by allowing programs to define hygiene-bending and hygiene-breaking syntactic abstractions and new unique datatypes along with procedures and hygienic macros in any scope;
- allow programmers to rely on a level of automatic run-time type and bounds checking sufficient to ensure type safety; and
- allow implementations to generate efficient code, without requiring programmers to use implementation-specific operators or declarations.

While it was possible to write portable programs in Scheme as described in *Revised⁵ Report on the Algorithmic Language Scheme*, and indeed portable Scheme programs were written prior to this report, many Scheme programs were not, primarily because of the lack of substantial standardized libraries and the proliferation of implementation-specific language additions.

In general, Scheme should include building blocks that allow a wide variety of libraries to be written, include commonly used user-level features to enhance portability and readability of library and application code, and exclude features that are less commonly used and easily implemented in separate libraries.

The language described in this report is intended to also be backward compatible with programs written in Scheme as described in *Revised⁵ Report on the Algorithmic Language Scheme* to the extent possible without compromising the above principles and future viability of the language. With respect to future viability, the editors have operated under the assumption that many more Scheme programs will be written in the future than exist in the present, so the future programs are those with which we should be most concerned.

Acknowledgements

Many people contributed significant help to this revision of the report. Specifically, we thank Aziz Ghuloum and André van Tonder for contributing reference implementations of the library system. We thank Alan Bawden, John Cowan, Sebastian Egner, Aubrey Jaffer, Shiro Kawai, Bradley Lucier, and André van Tonder for contributing insights on language design. Marc Feeley, Martin Gasbichler, Aubrey Jaffer, Lars T Hansen, Richard Kelsey, Olin Shivers, and André van Tonder wrote SRFIs that served as direct input to the report. Marcus Crestani, David Frese, Aziz Ghuloum, Arthur A. Gleckler, Eric Knael, Jonathan Rees, and André van Tonder thoroughly proofread early versions of the report.

We would also like to thank the following people for their help in creating this report: Lauri Alanko, Eli Barzilay, Alan Bawden, Brian C. Barnes, Per Bothner, Trent Buck, Thomas Bushnell, Taylor Campbell, Ludovic Courtès, Pascal Costanza, John Cowan, Ray Dillinger, Jed Davis, J.A. “Biep” Durieux, Carl Eastlund, Sebastian Egner, Tom Emerson, Marc Feeley, Matthias Felleisen, Andy Freeman, Ken Friedenbach, Martin Gasbichler, Arthur A. Gleckler, Aziz Ghuloum, Dave Gurnell, Lars T Hansen, Ben Harris, Sven Hartrumpf, Dave Herman, Nils M. Holm, Stanislav Ievlev, James Jackson, Aubrey Jaffer, Shiro Kawai, Alexander Kjeldaas, Eric Knael, Michael Lenaghan, Felix Klock, Donovan Kolbly, Marcin Kowalczyk, Thomas Lord, Bradley Lucier, Paulo J. Matos, Dan Muresan, Ryan Newton, Jason Orendorff, Erich Rast, Jeff Read, Jonathan Rees, Jorgen Schäfer, Paul Schlie, Manuel Serrano, Olin Shivers, Jonathan Shapiro, Jens Axel Søgaard, Jay Sulzberger, Pinku Surana, Mikael Tillenius, Sam Tobin-Hochstadt, David Van Horn, André van Tonder, Reinder Verlinde, Alan Watson, Andrew Wilcox, Jon Wilson, Lynn Winebarger, Keith Wright, and Chongkai Zhu.

We would like to thank the following people for their help in creating the previous revisions of this report: Alan Bawden, Michael Blair, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Bruce Duba, Marc Feeley, Andy Freeman, Richard Gabriel, Yekta Gürsel, Ken Haase, Robert Hieb, Paul Hudak, Morry Katz, Chris Lindblad, Mark Meyer, Jim Miller, Jim Philbin, John Ramsdell, Mike Shaff, Jonathan Shapiro, Julie Sussman, Perry Wagle, Daniel Weise, Henry Wu, and Ozan Yigit.

We thank Carol Fessenden, Daniel Friedman, and Christopher Haynes for permission to use text from the Scheme 311 version 4 reference manual. We thank Texas Instruments, Inc. for permission to use text from the *TI Scheme Language Reference Manual* [?]. We gladly acknowledge the influence of manuals for MIT Scheme [?], T [?], Scheme 84 [?], Common Lisp [?], Chez Scheme [?], PLT Scheme [?], and Algol 60 [?].

We also thank Betty Dexter for the extreme effort she put into setting this report in \TeX , and Donald Knuth for designing the program that caused her troubles.

The Artificial Intelligence Laboratory of the Massachusetts Institute of Technology, the Computer Science Department of Indiana University, the Computer and Information Sciences Department of the University of Oregon, and the NEC Research Institute supported the preparation of this report. Support for

the MIT work was provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505. Support for the Indiana University work was provided by NSF grants NCS 83-04567 and NCS 83-03325.

ОПИСАНИЕ ЯЗЫКА

1. Введение в Scheme

В данной главе описано введение в семантику Scheme. Целью данного введения является объяснение фундаментальных концепций языка, достаточных для облегчения понимания последующих глав работы, организованных в виде справочного руководства. Поэтому данный краткий обзор не является полным введением в язык, и при этом он не всегда точен в некоторых аспектах или не всегда нормативен.

После Algol Scheme является языком программирования со статическими областями видимости. Каждое использование переменной ассоциировано с лексически явным связыванием этой переменной.

Scheme имеет неявные, в отличие от декларативных, типы[?]. Типы связаны с объектами (также называемыми значениями), а не с переменными. (Некоторые авторы называют языки с неявными типами нетипизированными, слабо типизированными или динамически типизированными языками.) Другими языками с неявными типами являются Python, Ruby, Smalltalk, а также другие диалекты Lisp. Языки с декларативными типами (иногда называемые строго типизированными или статически типизированными языками) включают Algol 60, C, C#, Java, Haskell и ML.

Все объекты, созданные в процессе вычисления Scheme, включая процедуры и продолжения, имеют неограниченный экстенд. Ни один объект Scheme никогда не разрушается. Причина того, что реализации Scheme полностью не расходуют (обычно!) память, состоит в том, что им разрешается возвращать память, занятую объектом, если они смогут выяснить, что объект не имеет возможности влиять ни на одно будущее вычисление. Другие языки, в которых большинство объектов имеет неограниченный экстенд, включают C#, Java, Haskell, большинство диалектов Lisp, ML, Python, Ruby и Smalltalk.

Реализации Scheme должны подчиняться правилам хвостовой рекурсии. Это даёт возможность выполнения итеративного вычисления в постоянном пространстве, даже если итеративное вычисление описано синтаксически рекурсивной процедурой. Таким образом, при реализации поддержки хвостовой рекурсии, итерация может быть выражена с помощью обычной техники вызова процедуры, так, чтобы специальные итеративные конструкции были применимы только в качестве синтаксического сахара.

Scheme был одним из первых языков, поддерживающих процедуры как объекты сами по себе. Процедуры могут динамически создаваться, сохраняться в структурах данных, возвращаться как результаты процедур, и так далее. Другие языки с этими свойствами включают Common Lisp, Haskell, ML, Ruby и Smalltalk.

Одной из отличительных особенностей Scheme является то, что продолжения, которые в большинстве других языков функционируют только внутренне, также имеют "первоклассный" статус. Первоклассные продолжения полезны для реализации широкого разнообразия передовых управляющих конструкций, включая нелокальные выходы, бектрекинг и сопрограммы.

Выражения аргументов вызова процедуры в Scheme вычисляются перед получением процедурой управления независимо от необходимости для процедуры результата вычисления. C, C#, Common Lisp, Python, Ruby и Smalltalk - другие языки, которые всегда вычисляют выражения аргументов перед вызовом процедуры. Это отличается от семантики ленивого вычисления Haskell или семантики вызова по имени Algol 60, где выражение аргумента не вычисляется, если процедура не нуждается в его значении.

Арифметическая модель Scheme предоставляет богатый набор численных типов и операций с ними. Кроме того, она различает *точные* и *неточные* числовые объекты: По существу, точный числовой объект точно соответствует числу, а неточный числовой объект является результатом вычисления, которое повлекло за собой округление или другие ошибки.

1.1. Основные типы

Программы Scheme оперируют *объектами*, которые также называются *значениями*. Объекты Scheme организованы в наборы значений, которые называются *типами*. В этой секции дан краткий обзор существенно важных типов языка Scheme. Больше типов описано в последующих главах.

Note: Поскольку Scheme латентно типизирован, использование термина *тип* в данной работе отличается от использования термина в контексте других языков, особенно с явно объявляемой типизацией.

Булевые Булевый тип является истинностным значением и может быть true или false. Объект "false" в Scheme записывается #f. Объект "true" записывается #t. В большинстве мест, где ожидается истинностное значение, однако, любой объект, отличный от #f интерпретируется как true.

Числовые Scheme поддерживает множество числовых типов данных, включая объекты, представляющие целые числа произвольной точности, рациональные числа, сложные числа и приближённые числа различных видов. В Главе 3 дан краткий обзор структуры башни численных типов Scheme.

Знаковые Символы Scheme по большей части соответствуют текстовым символам. Более точно, они изоморфны к скалярным величинам стандарта Unicode.

Строковые Строки являются конечными последовательностями символов фиксированной длины и, таким образом, представляют произвольные тексты Unicode.

Символьные Символ является объектом, представляющим строку, *имя* символа. В отличие от строки, два символа, имена которых записаны в том же порядке, никоим образом не различаются. Символы полезны для многих применений; например, они могут использоваться, перечислимые значения пути используются на других языках.

Пары и списки Пара является структурой данных с двумя компонентами. Наиболее общее применение пар относится к представлению (отдельно связанных) списков, где первый компонент ("car") представляет первый элемент списка, а второй компонент ("cdr") - остальную часть списка. Scheme также имеет отдельный пустой список, который является последним cdr в цепочке пар, формирующих список.

Векторы Векторы, как и списки, являются линейными структурами данных, представляющими конечные последовательности произвольных объектов. Поскольку к элементам списка получают доступ последовательно через цепь пар, представляющих его, к элементам вектора обращаются целыми индексами. Таким образом, векторы предпочтительнее списков для произвольного доступа к элементам.

Процедуры В Scheme процедуры являются значениями.

1.2. Выражения

Важнейшими элементами кода Scheme являются *выражения*. Выражения могут быть *вычислены*, порождая *значение*. (Фактически, любое количество значений—см. раздел 5.8.) Самые фундаментальные выражения - литеральные выражения:

#t	⇒	#t
23	⇒	23

Эта форма записи означает, что выражение #t вычисляется как #t, то есть, значение для "true", и что выражение 23 вычисляется к численному объекту, представляющему число 23.

Составные выражения формируются помещением круглых скобок вокруг своих подвыражений. Первое подвыражение идентифицирует операцию; остальные подвыражения являются операндами операции:

(+ 23 42)	⇒	65
(+ 14 (* 23 42))	⇒	980

В начале этого примера + является именем встроенной операции сложения, а 23 и 42 - операндами. Выражение (+ 23 42) читается как "сумма 23 и 42". Составные выражения могут вкладываться—следующий пример читается как "сумма 14 и произведения 23 и 42".

Как показывают эти примеры, составные выражения в Scheme всегда записываются с помощью одной и той же префиксной нотации. Следствием является необходимость круглых скобок для указания структуры. Следовательно, "лишние" круглые скобки, которые часто допускаются в математической нотации, а также во многих языках программирования, в Scheme не позволяют.

Как и во многих других языках, пробельные символы (включая конец строки) являются не значащими, когда они отделяют подвыражения выражений, и могут использоваться для указания структуры.

1.3. Переменные и привязка

Scheme позволяет привязывать идентификаторы к содержащим значения ячейкам памяти. Такие идентификаторы называются переменными. Во многих случаях, определенно когда значение ячейки памяти никогда не изменяется после её создания, полезно думать о переменной как привязанной к значению непосредственно.

(let ((x 23)		
(y 42))		
(+ x y))	⇒	65

В данном случае выражение, начинающееся с **let** является конструкцией привязки. В окружённой скобками структуре после **let** перечислены переменные совместно с выражениями: переменная **x** совместно с 23 и переменная **y** совместно с 42. Выражение **let** связывает **x** с 23 и **y** с 42. Эти привязки доступны в теле выражения **let**, (+ x y), и только там.

1.4. Определения

Переменные, связанные выражением **let**, являются *локальными*, так как их связывания видимы только в теле **let**. Scheme также позволяет создавать связывания верхнего уровня для идентификаторов следующим образом:

(define x 23)		
(define y 42)		
(+ x y)	⇒	65

(Они фактически являются "верхним уровнем" в теле программы верхнего уровня или библиотеки; см. секцию 1.12 ниже.)

Первые две заключенные в скобки структуры являются *определениями*; они создают привязки верхнего уровня, связывая **x** с 23, а **y** с 42. Определения не являются выражениями и не могут находиться там, где может находиться выражение. Кроме того, определение не имеет значения.

Привязки подчиняются лексической структуре программы: при наличии нескольких одноименных привязок переменная соотносится с ближайшей к ней на пути изнутри наружу привязкой, начиная от её появления в программе, и с привязкой верхнего уровня, если локальная привязка не может быть найдена на этом пути:

(define x 23)
(define y 42)
(let ((y 43))

```
(+ x y))           ⇒ 66

(let ((y 43))
  (let ((y 44))
    (+ x y)))      ⇒ 67
```

1.5. Формы

Хотя определения не являются выражениями, составные выражения и определения имеют схожую синтаксическую структуру:

```
(define x 23)
(* x 2)
```

При этом первая линия содержит определение, а следующая - выражение, данное различие основывается на связывании **define** и *****. На чисто синтаксическом уровне обе являются *формами*, а *форма* является обобщённым названием синтаксической части программы Scheme. В частности, **23** является *подформой* формы **(define x 23)**.

1.6. Процедуры

Определения могут также использоваться для определения процедур.

```
(define (f x)
  (+ x 42))

(f 23)           ⇒ 65
```

Процедура, несколько упрощённо, является абстракцией выражения посредством объектов. В первом определении примера определена процедура, названная **f**. (Обратите внимание на круглые скобки вокруг **f x**, обозначающие, что это - определение процедуры.) Выражение **(f 23)** является вызовом процедуры, приблизительно означающим "вычислить **(+ x 42)** (тело процедуры) с **x**, привязанным к **23**".

Поскольку процедуры являются объектами, их можно передавать в другие процедуры:

```
(define (f x)
  (+ x 42))

(define (g p x)
  (p x))

(g f 23)         ⇒ 65
```

В этом примере тело **g** вычисляется с **p**, привязанным к **f**, и **x**, привязанным к **23**, что эквивалентно **(f 23)** и вычисляется как **65**.

Фактически многие предопределённые операции Scheme обеспечиваются не синтаксисом, а переменными, значениями которых являются процедуры. Операция **+**, например, приобретающая специальную синтаксическую трактовку во многих других языках, в Scheme является всего лишь регулярным идентификатором, связанным с процедурой, складывающей числовые объекты. То же самое касается и *****, и многих других:

```
(define (h op x y)
  (op x y))

(h + 23 42)       ⇒ 65
(h * 23 42)       ⇒ 966
```

Определения процедур - не единственный способ создания процедур. **lambda**-выражение создаёт новую процедуру в качестве объекта без необходимости указания имени:

```
((lambda (x) (+ x 42)) 23) ⇒ 65
```

Всё выражение в этом примере является вызовом процедуры; **(lambda (x) (+ x 42))** вычисляется как процедура, принимающая одиночный числовой объект и добавляющая к нему 42.

1.7. Вызовы процедур и синтаксические ключевые слова

Хотя и **(+ 23 42)**, и **(f 23)**, и **((lambda (x) (+ x 42)) 23)** являются примерами вызовов процедур, выражения **lambda** и **let** - нет. Это потому что **let**, хоть и идентификатор, но не переменная, а *синтаксическое ключевое слово*. Форма, содежащая синтаксическое ключевое слово в качестве своего первого подвыражения, подчиняется специальным правилам, определяемым ключевым словом. Идентификатор **define** в определении также является синтаксическим ключевым словом. Следовательно, определения также не являются вызовами процедур.

Правилами для ключевого слова **lambda** определено, что первая подформа является списком параметров, а остальные подформы - телом процедуры. В выражении **let** первая подформа является списком спецификаций привязки, а остальные подформы образуют тело выражений.

Обычно вызовы процедур можно отличить от таких *специальных форм*, с помощью поиска синтаксического ключевого слова в первом положении формы: если в первом положении не содержится синтаксического ключевого слова, выражение является вызовом процедуры. (Так называемый *макрос идентификатора* позволяет создавать другие виды специальных форм, но сравнительно редко.) Набор синтаксических ключевых слов Scheme является довольно небольшим, что обычно делает эту задачу довольно простой. Возможно, однако, создание новых привязок для синтаксических ключевых слов; см. секцию 1.9 ниже.

1.8. Присваивание

Переменные Scheme, связанные с определениями или с выражениями **let** или **lambda**, привязываются фактически не непосредственно к объектам, определённым в соответствующих привязках, а к адресам памяти, содержащим эти объекты. Содержимое по этим адресам впоследствии может быть деструктивно изменено с помощью *присваивания*:

```
(let ((x 23))
  (set! x 42)
  x)           ⇒ 42
```


В данном случае тело выражения **let** состоит из двух вычисляемых последовательно выражений со значением финального выражения, принимающего значение всего выражения **let**. Выражение **(set! x 42)**, является присваиванием, указывающим "заменить объект по адресу, на который указывает **x**, на 42". Таким образом, предыдущее значение **x** - 23 изменяется на 42.

1.9. Производные формы и макросы

Большинство специальных форм, определённых в данной работе, могут быть приведены к более простым специальным формам. Например, выражение **let** может быть приведено к вызову процедуры и выражению **lambda**. Следующие два выражения эквивалентны:

```
(let ((x 23)
      (y 42))
  (+ x y))           ⇒ 65
```

```
((lambda (x y) (+ x y)) 23 42)
  ⇒ 65
```

Специальные формы, такие, как выражения **let**, называются *производными формами*, так как их семантика может быть получена из того или иного вида форм синтаксическим преобразованием. Некоторые определения процедур также являются производными формами. Следующие два определения эквивалентны:

```
(define (f x)
  (+ x 42))
```

```
(define f
  (lambda (x)
    (+ x 42)))
```

В программе Scheme имеется возможность создания своих собственных производных форм путём связывания синтаксических ключевых слов с макросами:

```
(define-syntax def
  (syntax-rules ()
    ((def f (p ...) body)
     (define (f p ...)
       body))))
```

```
(def f (x)
  (+ x 42))
```

Конструкция **define-syntax** определяет, что заключённая в скобки структура, соответствующая шаблону **(def f (p ...) body)**, где **f**, **p** и **body** - переменные шаблона, приводится к **(define (f p ...) body)**. Таким образом, форма **def**, находящаяся в примере, приводится к:

```
(define (f x)
  (+ x 42))
```

Возможность создания новых синтаксических ключевых слов делает Scheme чрезвычайно гибким и выразительным, что позволяет большинству особенностей, встроенных в другие языки, быть производными формами в Scheme.

1.10. Синтаксические данные и значения datum

Подмножество объектов Scheme называется *значениями datum*. Оно включает булевы, численные объекты, знаки, символы и строки, равно как списки и векторы, элементы которых являются данными. Каждое значение datum может быть представлено в текстовом виде как *синтаксический datum*, который может записываться и считываться без потери информации. Значение datum может быть представлено различными синтаксическими данными. Кроме того, каждое базисное значение может быть тривиально приведено к литеральному выражению в программе путём добавления ' к соответствующему синтаксическому datum:

'23	⇒ 23
'#t	⇒ #t
'foo	⇒ foo
'(1 2 3)	⇒ (1 2 3)
'#(1 2 3)	⇒ #(1 2 3)

' , показанный в предыдущих примерах, не нужен для представлений численных объектов или булевых переменных. Синтаксический базис **foo** представляет символ с именем "foo", а 'foo - литеральное выражение с этим символом в качестве его значения. **(1 2 3)** - синтаксический базис, представляющий список с элементами 1, 2 и 3, а '(1 2 3) - литеральное выражение с этим списком в качестве его значения. Аналогично, **#(1 2 3)** - синтаксический базис, представляющий вектор с элементами 1, 2 и 3, а '#(1 2 3) - соответствующий литерал.

Синтаксические данные являются расширенным набором форм Scheme. Таким образом, данные могут использоваться, для представления форм Scheme в виде объектов данных. В частности, символы могут использоваться для представления идентификаторов.

'(+ 23 42)	⇒ (+ 23 42)
'(define (f x) (+ x 42))	⇒ (define (f x) (+ x 42))

Это облегчает написание программ, работающих с исходным кодом Scheme, в частности, интерпретаторов и программных преобразователей.

1.11. Продолжения

Всякий раз, когда выражение Scheme оценено есть *продолжение*, ожидая результат выражения. Продолжение представляет все будущее (по умолчанию) для вычисления. Например, произвольно, продолжение 3 в выражении

```
(+ 1 3)
```

добавляет 1 к нему. Обычно эти встречающиеся повсюду продолжения скрыты на заднем плане, и программисты особо не заботятся о них. В редких случаях, однако, программист, возможно, должен иметь дело с продолжениями явно. Процедура **call-with-current-continuation** (см. секцию 11.15) позволяет программистам Scheme делать это, создавая процедуру, которая восстанавливает текущее продолжение. Процедура **call-with-current-continuation** принимает процедуру, вызывает её немедленно с аргументом, который является *аварийной процедурой*

. Эту аварийную процедуру можно тогда вызвать с аргументом, который становится результатом вызова **call-with-current-continuation**. Таким образом, аварийная процедура оставляет ее собственное продолжение, и восстанавливает продолжение вызова к **call-with-current-continuation**.

В следующем примере, аварийная процедура, представляющая продолжение, которое добавляет 1 к его аргументу, привязана к **escape**, и затем вызывается с 3 как аргумент. Продолжение вызова **escape** оставлено, и вместо этого эти 3 передают к продолжению, которое добавляет 1:

```
(+ 1 (call-with-current-continuation
      (lambda (escape)
        (+ 2 (escape 3))))))
⇒ 4
```

Аварийная процедура имеет неограниченный экстенд: Она может быть вызвана после вызова захватившего его продолжения, и это можно вызвать несколько раз. Это делает **call-with-current-continuation** значительно более мощным чем типичные нелокальные управляющие конструкции, типа исключений в других языках.

1.12. Библиотеки

Код Scheme может быть организован в компонентах, которые называются *библиотеками*. Каждая библиотека содержит определения и выражения. Она может импортировать определения из других библиотек и экспортировать определения в другие библиотеки.

Следующая библиотека называется (**hello**), экспортирует определение, которое называется **hello-world**, и импортирует основную библиотеку (см. главу 11), и простая библиотека I/O (см. библиотечную секцию ??). Экспорт **hello-world** является процедурой, которая показывает **Hello World** на отдельной линии:

```
(library (hello)
  (export hello-world)
  (import (rnrs base)
          (rnrs io simple))
  (define (hello-world)
    (display "Hello World")
    (newline)))
```

1.13. Программы верхнего уровня

Программа Scheme активизируется посредством *программы верхнего уровня*. Как и библиотека, программа верхнего уровня содержит импорт, определения и выражения, и определяет точку входа для выполнения. Таким образом, программа верхнего уровня определяет, через замкнутое выражение библиотек, которые это импортирует, программу Scheme.

Следующая программа верхнего уровня получает первый аргумент из командной строки через процедуру **command-line** из библиотеки (**rnrs programs (6)**) (см. библиотечную главу ??). Это тогда открывает файл с помощью **open-file-input-port** (см. секцию ??, приводя к *порт*,

т.е. к связи с файлом как источником данных, и вызывает процедуру **get-bytes-all** получения содержимого файла в виде двоичных данных. Это тогда использует **put-bytes** для вывода содержимого файла в стандартный вывод:

```
#!r6rs
(import (rnrs base)
        (rnrs io ports)
        (rnrs programs))
(put-bytes (standard-output-port)
  (call-with-port
    (open-file-input-port
      (cadr (command-line)))
    get-bytes-all)))
```

2. Уровни требований

Ключевые слова, “must”, “must not”, “should”, “should not”, “recommended”, “may”, и “optional” в данной работе должны интерпретироваться как описано в RFC 2119 [?]. А именно:

must Это слово означает, что инструкция является абсолютным требованием спецификации.

must not Эта фраза означает, что инструкция является абсолютным запрещением спецификации.

should Это слово, или прилагательное “recommended”, означают, что в особых обстоятельствах могут существовать веские причины для игнорирования инструкции, но перед выбором иной линии поведения последствия должны быть осознаны и взвешены.

should not Эта фраза, или фраза “not recommended”, означают, что в особых обстоятельствах могут существовать веские причины для принятия функционирования инструкции, но перед выбором описываемой инструкцией линии поведения последствия должны быть осознаны и взвешены.

may Это слово, или прилагательное “optional” означают, что элемент является действительно дополнительным.

В частности, “should” в данной работе иногда используется для обозначения обстоятельств, лежащих вне спецификации данной работы, но не обнаруживаемых реализацией на практике; см. секцию 5.4. При таких обстоятельствах конкретная реализация может позволить программисту игнорировать рекомендацию в данной работе и даже продемонстрировать адекватное функционирование. Однако, поскольку данная работа не специфицирует функционирование, такие программы могут быть неимплементируемыми, то есть, их выполнение может привести к различным результатам в разных реализациях.

Кроме того, иногда используемая в данной работе фраза “not required” означает отсутствие абсолютного требования.

3. Числа

В данной главе описывается модель чисел Scheme. Важно различать математические числа, представляющие их объекты Scheme, машинные представления, используемые для реализации чисел, и используемые для записи чисел нотации. В данной работе термин *число* относится к математическому числу, а термин *числовой объект* - к объекту Scheme, представляющему число. В данной работе используются типы *complex*, *real*, *rational* и *integer* для обращения и к математическим числам, и к числовым объектам. Типы *fixnum* и *flonum* относятся к специальным подмножествам числовых объектов, как определяется общими машинными представлениями и объясняется ниже.

3.1. Числовая башня

Числа могут быть систематизированы башней подмножеств, каждый уровень которой является подмножеством верхнего уровня:

```
number
complex
real
rational
integer
```

Например, 5 является целым числом. Поэтому 5 также является рациональным, действительным и комплексным. То же верно и для числовых объектов, представляющих 5.

Числовые объекты организованы как соответствующая башня подтипов, определённых предикатами **number?**, **complex?**, **real?**, **rational?**, и **integer?**; см. секцию 11.7.4. Числовые объекты целого также называются *объектами целого*.

Не существует очевидной взаимосвязи между подмножеством, содержащим число, и его представлением в компьютере. Например, целое число 5 может иметь несколько представлений. Операции с числами в Scheme интерпретируют числовые объекты как абстрактные данные, столь же независимые от их представления, насколько возможно. Хотя реализация Scheme может использовать множество различных представлений чисел, это не должно быть очевидно случайному программисту, пишущему простые программы.

3.2. Точность

Следует различать числовые объекты, о которых известно, что они точно соответствуют числу, и числовые объекты, вычисление которых повлекло за собой округление или другие ошибки. Например, операциям индексации структур данных может потребоваться знание точного индекса, как и некоторым операциям с коэффициентами полинома в системе

символьной алгебры. С другой стороны, результаты измерений являются неточными по определению, и иррациональные числа могут быть аппроксимированы рациональной, и поэтому неточной, аппроксимацией. Для обнаружения использования чисел, известных только приблизительно, там, где требуются точные числа, Scheme явно различает *точные* числовые объекты от *неточных*. Это различие независимо от измерения типа.

Числовой объект является точным, если он является значением точного числового литерала или был получен из точных числовых объектов с помощью только точных операций. Точные числовые объекты соответствуют математическим числам явным образом.

В свою очередь, числовой объект является неточным, если он является значением неточного числового литерала или был получен из неточных числовых объектов числа, или с помощью неточных операций. Таким образом неточность передаётся непосредственно.

Точная арифметика безопасна в следующем смысле: Если точные числовые объекты передаются любой из арифметических процедур, описанных в секции 11.7.1, и возвращается точный числовой объект, результат математически корректен. Это в общем случае не верно для вычислений с использованием неточных числовых объектов, так как могут использоваться методы аппроксимации, типа арифметики с плавающей запятой, но это уже является обязанностью каждой реализации - выдать результат как можно ближе практически к математически идеальному результату.

3.3. Fixnums and flonums

fixnum является точным объектом целого, который лежит в пределах конкретного зависимого от реализации поддиапазона точного объекта целого. (В секции ?? описана библиотека для вычислений с *fixnums*.) Аналогично, каждая реализация должно обозначать подмножество своих неточных вещественных числовых объектов как *flonums*, и преобразовывать конкретные внешние представления в *flonums*. (В секции ?? описана библиотека для вычислений с *flonums*.) Обратите, что не подразумевается, что реализация должна использовать представления с плавающей запятой.

3.4. Требования к реализациям

Реализации Scheme должны поддерживать числовые объекты для всей башни подтипов, приведённых в секции 3.1. Кроме того, реализации должны поддерживать точные объекты целого точные рациональные числовые объекты практически неограниченного размера и точности, и реализовывать конкретные процедуры (перечисленные в 11.7.1) с тем, чтобы они всегда возвращали точные результаты при предоставлении точных аргументов. (“Практически неограниченный” означает, что размер и точность таких чисел должны ограничиваться только размером доступной памяти.)

Реализации могут поддерживать только ограниченный диапазон неточных числовых объектов любого типа, подчиняясь требованиям данной секции. Например, реализация может ограничить диапазон неточных вещественных числовых объектов (и, следовательно, диапазон неточных целых и рациональных числовых объектов) динамическим диапазоном формата `flonum`. Кроме того, зазоры между неточными целыми объектами и `rational`s, вероятно, будут очень большими в такой реализации при приближении к границам этого диапазона.

В реализации может применяться плавающая запятая и другие приближённые способы представления неточных чисел. В данной работе рекомендуется, но не требуется, чтобы реализации, использующие представления с плавающей запятой, следовали стандартам плавающей запятой IEEE, а реализации, использующие другие представления, имели соответствующую или превышающую достижимую при применении стандартов с плавающей запятой [?] точность.

В частности, реализации, использующие представления с плавающей запятой, должны соблюдать следующие правила: результат с плавающей запятой должен быть представлен с точностью, по крайней мере не менее используемой для выражения любого неточного аргумента в данной операции. При применении потенциально неточных операций, типа `sqrt` к точным аргументам должны выдаваться точные ответы всегда, когда возможно (например, квадратный корень точного 4 должен быть точным 2). Однако это не является требованием. Если, с другой стороны, точным числовым объектом управляют с целью получения неточного результата (как `sqrt`), и если результат представляется с плавающей запятой, должен использоваться самый точный доступный формат с плавающей запятой; но если результат представлен неким иным способом, тогда, представление должно иметь точность, по крайней мере не менее самого точного доступного формата с плавающей запятой.

Отказ от применения неточных числовых объектов со слишком большим для представления в реализации значением или значащей частью является заботой программистов.

3.5. Бесконечность и NaN

Некоторые реализации Scheme, например, придерживающиеся стандартов IEEE с плавающей точкой, различают специальные числовые объекты, называемые *положительная бесконечность*, *отрицательная бесконечность* и *NaN*.

Положительная бесконечность рассматривается как неточный вещественный (но не рациональный) числовой объект, представляющий неопределённое число, большее, чем числа, представляемые всеми рациональными числовыми объектами. Отрицательная бесконечность рассматривается как неточный вещественный (но не рациональный) числовой объект, представляющий неизвестное число, меньшее, чем числа, представляемые всеми рациональными числами.

NaN рассматривается как неточный вещественный (но не рациональный) числовой объект, настолько неопределённый, что он может представлять любое вещественное число, включая положительную или отрицательную бесконечность, и даже может быть больше положительной бесконечности или меньше отрицательной бесконечности.

3.6. Распознавание -0.0

Некоторые реализации Scheme, например, придерживающиеся стандартов IEEE с плавающей точкой, различают числовые объекты 0.0 и -0.0, то есть, положительный и отрицательный неточный ноль. В данной работе в ряде случаев специфицируется функционирование конкретных арифметических операций с такими числовыми объектами. Такие спецификации записываются совместно с “если -0.0 распознан” или “реализация, различающая -0.0”.

4. Лексический и datum-синтаксис

Синтаксис кода Scheme организован на трёх уровнях:

1. *Лексический синтаксис*, описывающий, как текст программы разбивается на последовательность лексем,
2. *Datum-синтаксис*, сформулированный в терминах лексического синтаксиса, структурирующий последовательность лексем в виде последовательности *синтаксических данных*, где синтаксический datum является рекурсивно структурированным элементом,
3. *Программный синтаксис*, сформулированный в терминах синтаксиса считывания, задающий дальнейшую структуру и наполняющий смысловым содержанием синтаксические данные.

Синтаксические данные (называемые также *внешними представлениями*) одновременно служат как формой записи объектов, так и библиотекой Scheme (`rnrs io ports (6)`) (секция библиотек ??), предоставляющей процедуры `get-datum` и `put-datum` для чтения и записи синтаксических данных, преобразовывающие их из текстового представления в соответствующие объекты, и наоборот. Каждый синтаксический datum представляет соответствующее *datum-значение*. Синтаксический datum может использоваться в программе для получения соответствующего datum-значения с помощью `quote` (см. секцию 11.4.1).

Исходный текст Scheme состоит из синтаксических данных и (незначащих) комментариев. Синтаксические данные в исходном тексте Scheme называются *формами*. (Форма, вложенная в другую форму, называется *подформой*.) Следовательно, синтаксис Scheme обладает свойством, что любая последовательность символов, являющаяся формой, является также и синтаксическим datum, представляющей некоторый объект. Это может привести к замешательству, так как из контекста может быть не ясно, предназначена ли данная последовательность символов для представления объектов или текста программы. Это - также источник мощи, так как это облегчает написание программ, типа интерпретаторов или компиляторов, интерпретирующих программы в качестве объектов (или наоборот).

Datum-значение может иметь несколько различных внешних представлений. Например, и “`#e28.000`”, и “`#x1c`” являются синтаксическими данными, представляющими точный целый объект 28, а синтаксические данные “`(8 13)`”, “`(08 13)`”, “`(8 . (13 . ()))`” представляют список, содержащий точные целые объекты 8 и 13. Синтаксические данные, представляющие равные объекты (в смысле `equal?`;

см. секцию 11.5), всегда эквивалентны как формы программы.

Вследствие точного соответствия синтаксических данных и datum-значений в данной работе термин *datum* иногда используется и для синтаксического datum, и для datum-значения, когда точный смысл очевиден из контекста.

Реализации запрещено расширять лексический или datum-синтаксис, за одним исключением: синтаксис **#!**<identifier> при любом <identifier> (см. секцию 4.2.4), отличном от **r6rs**, не должен считаться синтаксическим нарушением, а специфические идентификаторы с префиксом **#!** могут использоваться в качестве флагов, указывающих на наличие в последующем вводе расширений лексического стандарта или datum-синтаксиса. Синтаксис **#!r6rs** может применяться для указания, что последующий ввод записан с лексическим и datum-синтаксисом, описанным в данной работе. В противном случае **#!r6rs** считается комментарием; см. секцию 4.2.3.

4.1. Нотация

Формальный синтаксис Scheme записан в расширенной BNF. Нетерминальные символы окружены угловыми скобками. Для нетерминальных имён регистр не имеет значения.

Все пробелы в грамматике применяются для удобочитаемости. <Empty> обозначает пустую строку.

Для более лаконичного описания используются следующие расширения BNF: <thing>* означает ноль или более вхождений <thing>, а <thing>+ означает не менее одного <thing>.

Некоторые нетерминальные имена относятся к одноимённым скалярным значениям Unicode: <character tabulation> (U+0009), <linefeed> (U+000A), <carriage return> (U+000D), <line tabulation> (U+000B), <form feed> (U+000C), <carriage return> (U+000D), <space> (U+0020), <next line> (U+0085), <line separator> (U+2028) и <paragraph separator> (U+2029).

4.2. Лексический синтаксис

Лексический синтаксис определяет, как символьная последовательность разбивается на последовательность лексем, пропуская незначимые части, типа комментариев и пробелов. Полагается, что символьная последовательность является текстом согласно стандарту Unicode [?]. Некоторые лексемы, типа идентификаторов, представлений числовых объектов, строки и т.д., лексического синтаксиса являются синтаксическими данными в datum-синтаксисе и, таким образом, представляют объекты. Помимо формального описания синтаксиса, в этой секции также описывается, какие datum-значения представляют эти синтаксические данные.

Лексический синтаксис в описании комментариев содержит прямую ссылку на <datum>, описанную как часть datum-синтаксиса. Будучи комментариями, однако, такие <datum> не играют существенную роль в синтаксисе.

Регистр является значащим, за исключением представлений booleans, числовых объектов и шестнадцатеричных чисел, определяющих скалярные значения Unicode. Например, **#x1A** и **#X1a** эквивалентны. А вот идентификатор **Foo** отличается от идентификатора **FOO**.

4.2.1. Формальное описание

<Interlexeme space> может находиться с любой стороны лексемы, но не внутри лексемы

<Identifier>, ., <number>, <character> и <boolean> могут заканчиваться <delimiter> или концом ввода.

Следующие два знака зарезервированы для будущих расширений языка: { }

```

<lexeme> → <identifier> | <boolean> | <number>
          | <character> | <string>
          | ( | ) | [ | ] | # ( | #vu8 ( | ' | □ | , | , @ | .
          | #' | #□ | #, | #, @
<delimiter> → ( | ) | [ | ] | " | ; | #
              | <whitespace>
<whitespace> → <character tabulation>
               | <linefeed> | <line tabulation> | <form feed>
               | <carriage return> | <next line>
               | <any character whose category is Zs, Zl, or Zp>
<line ending> → <linefeed> | <carriage return>
               | <carriage return> <linefeed> | <next line>
               | <carriage return> <next line> | <line separator>
<comment> → ; <all subsequent characters up to a
              <line ending> or <paragraph separator>
              | <nested comment>
              | # ; <interlexeme space> <datum>
              | # ! r6rs
<nested comment> → # | <comment text>
                  <comment cont>* | #
<comment text> → <character sequence not containing
                  # | or | #>
<comment cont> → <nested comment> <comment text>
<atmosphere> → <whitespace> | <comment>
<interlexeme space> → <atmosphere>*

<identifier> → <initial> <subsequent>*
              | <peculiar identifier>
<initial> → <constituent> | <special initial>
           | <inline hex escape>
<letter> → a | b | c | . . . | z
           | A | B | C | . . . | Z
<constituent> → <letter>
               | <any character whose Unicode scalar value is greater than
                 127, and whose category is Lu, Ll, Lt, Lm, Lo, Mn,
                 Nl, No, Pd, Pc, Po, Sc, Sm, Sk, So, or Co>

```

$\langle \text{special initial} \rangle \rightarrow ! | \$ | \% | \& | * | / | : | < | =$
 $| > | ? | ^ | _ | \sim$
 $\langle \text{subsequent} \rangle \rightarrow \langle \text{initial} \rangle | \langle \text{digit} \rangle$
 $| \langle \text{any character whose category is Nd, Mc, or Me} \rangle$
 $| \langle \text{special subsequent} \rangle$
 $\langle \text{digit} \rangle \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
 $\langle \text{hex digit} \rangle \rightarrow \langle \text{digit} \rangle$
 $| a | A | b | B | c | C | d | D | e | E | f | F$
 $\langle \text{special subsequent} \rangle \rightarrow + | - | . | @$
 $\langle \text{inline hex escape} \rangle \rightarrow \backslash x \langle \text{hex scalar value} \rangle ;$
 $\langle \text{hex scalar value} \rangle \rightarrow \langle \text{hex digit} \rangle^+$
 $\langle \text{peculiar identifier} \rangle \rightarrow + | - | \dots | \rightarrow \langle \text{subsequent} \rangle^*$
 $\langle \text{boolean} \rangle \rightarrow \#t | \#T | \#f | \#F$
 $\langle \text{character} \rangle \rightarrow \# \backslash \langle \text{any character} \rangle$
 $| \# \backslash \langle \text{character name} \rangle$
 $| \# \backslash x \langle \text{hex scalar value} \rangle$
 $\langle \text{character name} \rangle \rightarrow \text{nul} | \text{alarm} | \text{backspace} | \text{tab}$
 $| \text{linefeed} | \text{newline} | \text{vtab} | \text{page} | \text{return}$
 $| \text{esc} | \text{space} | \text{delete}$
 $\langle \text{string} \rangle \rightarrow " \langle \text{string element} \rangle^* "$
 $\langle \text{string element} \rangle \rightarrow \langle \text{any character other than " or \} \rangle$
 $| \backslash a | \backslash b | \backslash t | \backslash n | \backslash v | \backslash f | \backslash r$
 $| \backslash " | \backslash \backslash$
 $| \backslash \langle \text{intraline whitespace} \rangle \langle \text{line ending} \rangle$
 $| \langle \text{intraline whitespace} \rangle$
 $| \langle \text{inline hex escape} \rangle$
 $\langle \text{intraline whitespace} \rangle \rightarrow \langle \text{character tabulation} \rangle$
 $| \langle \text{any character whose category is Zs} \rangle$

A $\langle \text{hex scalar value} \rangle$ represents a Unicode scalar value between 0 and $\#x10FFFF$, excluding the range $[\#xD800, \#xDFFF]$.

The rules for $\langle \text{num } R \rangle$, $\langle \text{complex } R \rangle$, $\langle \text{real } R \rangle$, $\langle \text{ureal } R \rangle$, $\langle \text{uinteger } R \rangle$, and $\langle \text{prefix } R \rangle$ below should be replicated for $R = 2, 8, 10$, and 16 . There are no rules for $\langle \text{decimal } 2 \rangle$, $\langle \text{decimal } 8 \rangle$, and $\langle \text{decimal } 16 \rangle$, which means that number representations containing decimal points or exponents must be in decimal radix.

$\langle \text{number} \rangle \rightarrow \langle \text{num } 2 \rangle | \langle \text{num } 8 \rangle$
 $| \langle \text{num } 10 \rangle | \langle \text{num } 16 \rangle$
 $\langle \text{num } R \rangle \rightarrow \langle \text{prefix } R \rangle \langle \text{complex } R \rangle$
 $\langle \text{complex } R \rangle \rightarrow \langle \text{real } R \rangle | \langle \text{real } R \rangle @ \langle \text{real } R \rangle$
 $| \langle \text{real } R \rangle + \langle \text{ureal } R \rangle i | \langle \text{real } R \rangle - \langle \text{ureal } R \rangle i$
 $| \langle \text{real } R \rangle + \langle \text{naninf} \rangle i | \langle \text{real } R \rangle - \langle \text{naninf} \rangle i$
 $| \langle \text{real } R \rangle + i | \langle \text{real } R \rangle - i$
 $| + \langle \text{ureal } R \rangle i | - \langle \text{ureal } R \rangle i$
 $| + \langle \text{naninf} \rangle i | - \langle \text{naninf} \rangle i$
 $| + i | - i$
 $\langle \text{real } R \rangle \rightarrow \langle \text{sign} \rangle \langle \text{ureal } R \rangle$
 $| + \langle \text{naninf} \rangle | - \langle \text{naninf} \rangle$
 $\langle \text{naninf} \rangle \rightarrow \text{nan.0} | \text{inf.0}$
 $\langle \text{ureal } R \rangle \rightarrow \langle \text{uinteger } R \rangle$
 $| \langle \text{uinteger } R \rangle / \langle \text{uinteger } R \rangle$
 $| \langle \text{decimal } R \rangle \langle \text{mantissa width} \rangle$
 $\langle \text{decimal } 10 \rangle \rightarrow \langle \text{uinteger } 10 \rangle \langle \text{suffix} \rangle$
 $| . \langle \text{digit } 10 \rangle^+ \langle \text{suffix} \rangle$
 $| \langle \text{digit } 10 \rangle^+ . \langle \text{digit } 10 \rangle^* \langle \text{suffix} \rangle$
 $| \langle \text{digit } 10 \rangle^+ . \langle \text{suffix} \rangle$

$\langle \text{uinteger } R \rangle \rightarrow \langle \text{digit } R \rangle^+$
 $\langle \text{prefix } R \rangle \rightarrow \langle \text{radix } R \rangle \langle \text{exactness} \rangle$
 $| \langle \text{exactness} \rangle \langle \text{radix } R \rangle$
 $\langle \text{suffix} \rangle \rightarrow \langle \text{empty} \rangle$
 $| \langle \text{exponent marker} \rangle \langle \text{sign} \rangle \langle \text{digit } 10 \rangle^+$
 $\langle \text{exponent marker} \rangle \rightarrow e | E | s | S | f | F$
 $| d | D | l | L$
 $\langle \text{mantissa width} \rangle \rightarrow \langle \text{empty} \rangle$
 $| | \langle \text{digit } 10 \rangle^+$
 $\langle \text{sign} \rangle \rightarrow \langle \text{empty} \rangle | + | -$
 $\langle \text{exactness} \rangle \rightarrow \langle \text{empty} \rangle$
 $| \#i | \#I | \#e | \#E$
 $\langle \text{radix } 2 \rangle \rightarrow \#b | \#B$
 $\langle \text{radix } 8 \rangle \rightarrow \#o | \#O$
 $\langle \text{radix } 10 \rangle \rightarrow \langle \text{empty} \rangle | \#d | \#D$
 $\langle \text{radix } 16 \rangle \rightarrow \#x | \#X$
 $\langle \text{digit } 2 \rangle \rightarrow 0 | 1$
 $\langle \text{digit } 8 \rangle \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7$
 $\langle \text{digit } 10 \rangle \rightarrow \langle \text{digit} \rangle$
 $\langle \text{digit } 16 \rangle \rightarrow \langle \text{hex digit} \rangle$

4.2.2. Окончание строки

В Scheme окончание строки является значащим в однострочных комментариях (см. секцию 4.2.3) и внутри строковых литералов. В исходном тексте Scheme любое окончание строки в $\langle \text{line ending} \rangle$ означает конец строки. Кроме того, двухсимвольное окончание строки $\langle \text{carriage return} \rangle \langle \text{linefeed} \rangle$ и $\langle \text{carriage return} \rangle \langle \text{next line} \rangle$ считается одним окончанием строки.

В строковом литерале $\langle \text{line ending} \rangle$ без \backslash перед ним обозначает символ `linefeed`, являющийся стандартным символом конца строки Scheme.

4.2.3. Пробельные символы и комментарии

Пробельными символами являются пробелы, обратные переводы строк, переводы каретки, символы табуляции, переводы страницы, линейные табуляции и любой другой символ, категория которого - Zs, Zl, или Zp. Пробельные символы используются для улучшения удобочитаемости и при необходимости отделения лексем друг от друга. Пробельные символы могут находиться между любыми двумя лексемами, но не внутри лексемы. Пробельный символ может также находиться в строке, где он является значащим.

Лексический синтаксис включает несколько форм комментариев. Во всех случаях, комментарии невидимы, в Scheme, за исключением того, что они действуют как разделители, таким образом, например, комментарий не может находиться в середине идентификатора или представления числового объекта.

Точка с запятой (;), указывает начало строки комментария. Комментарий продолжается до конца строки, на которой находится точка с запятой.

Другим способом указания комментария является добавления к <datum> (см. секцию 4.3.1) префикса #; , возможно с <interlexeme space> перед <datum>. Комментарий состоит из приставки комментария #; и <datum>. Такая нотация полезна для “комментирования” секций кода.

Блочные комментарии могут быть обозначены соответствующим образом вложенными парами #| и |#.

```
#|
  The FACT procedure computes the factorial
  of a non-negative integer.
|#
(define fact
  (lambda (n)
    ;; base case
    (if (= n 0)
        #; (= n 1)
        1      ; identity of *
        (* n (fact (- n 1))))))
```

Лексема #!r6rs, которая означает, что текст следующей далее программы написан с лексическим и data-синтаксисом, описанным в данной работе, также иным образом интерпретируется как комментарий.

4.2.4. Идентификаторы

Большинство идентификаторов, допустимых в других языках программирования, также допустимы и в Scheme. В общем случае, последовательность букв, цифр, и “расширенных алфавитных символов” является идентификатором, если она начинается с символа, с которого не может начинаться представление числового объекта. Кроме того, идентификаторами являются +, - и . . . , равно как и последовательность букв, цифр и расширенных алфавитных символов, начинающаяся с двухсимвольной последовательности ->. Некоторые примеры идентификаторов:

```
lambda      q      soup
list->vector  +      v17a
<=          a34kTMNs  ->-
the-word-recursion-has-many-meanings
```

Расширенные алфавитные символы могут использоваться внутри идентификаторов, как если бы они были буквами. Расширенными алфавитными символами являются:

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

Кроме того, в идентификаторах могут использоваться все символы, чьи скалярные значения Unicode больше, чем 127, и чьей категорией Unicode является Lu, Ll, Lt, Lm, Lo, Mn, Mc, Me, Nd, Nl, No, Pd, Pc, Po, Sc, Sm, Sk, So, или Co. Кроме того, любой символ может использоваться в идентификаторе, если он специфицирован через <inline hex escape>. Например, идентификатор Н\x65;11o - точно такой же, как и идентификатор Hello, а идентификатор \x3BB; - точно такой же, как и идентификатор λ.

В программе Scheme любой идентификатор может использоваться как переменная или как синтаксическое ключевое словосyntactic keyword (см. секции 5.2 и 9.2) . Любой идентификатор может также использоваться как синтаксический datum, в этом случае он представляет *символ* (см. секцию 11.10).

4.2.5. Booleans

Стандартные булевы объекты для true и false имеют внешние представления #t и #f.

4.2.6. Символы

Символы представляются с помощью нотации #\<character> или #\<character name> или #\x<hex scalar value>.

Например:

#\a	lower case letter a
#\A	upper case letter A
#\ (left parenthesis
#\	space character
#\nul	U+0000
#\alarm	U+0007
#\backspace	U+0008
#\tab	U+0009
#\linefeed	U+000A
#\newline	U+000A
#\vtab	U+000B
#\page	U+000C
#\return	U+000D
#\esc	U+001B
#\space	U+0020
	preferred way to write a space
#\delete	U+007F
#\xFF	U+00FF
#\x03BB	U+03BB
#\x00006587	U+6587
#\λ	U+03BB
#\x0001z	&lexical exception
#\λx	&lexical exception
#\alarmx	&lexical exception
#\alarm x	U+0007
	followed by x
#\Alarm	&lexical exception
#\alert	&lexical exception
#\xA	U+000A
#\xFF	U+00FF
#\xff	U+00FF
#\x ff	U+0078
	followed by another datum, ff

#\x(ff)	U+0078 followed by another datum, a parenthesized ff
#\ (x)	&lexical exception
#\ (x	&lexical exception
#\ ((x)	U+0028 followed by another datum, parenthesized x
#\x00110000	&lexical exception out of range
#\x000000001	U+0001
#\xD800	&lexical exception in excluded range

(Нотация **&lexical exception** означает, что рассматриваемая строка является лексическим нарушением синтаксиса.)

Регистр является значащим в **#\<character>** и в **#\<character name>**, но не в **#\x<hex scalar value>**. За **<character>** должен находиться **<delimiter>** или конец ввода. Это правило разрешает различные неоднозначные случаи с участием названных символов, требуя, например, чтобы последовательность символов **"#\space"** интерпретировалась как символ пробела, а не как символ **"#\s"** и следующим за ним идентификатором **"pace"**.

Note: Нотация **#\newline** сохранена с целью обратной совместимости. Её использование устарело; вместо неё должна использоваться **#\linefeed**.

4.2.7. Строки

Строка представляется последовательностью символов, окружённых двойными кавычками (**"**). Внутри строкового литерала различные управляющие последовательности представляют символы, кроме самих себя. Управляющие последовательности всегда начинаются с обратного слеша (****):

- **\a** : alarm, U+0007
- **\b** : backspace, U+0008
- **\t** : character tabulation, U+0009
- **\n** : linefeed, U+000A
- **\v** : line tabulation, U+000B
- **\f** : formfeed, U+000C
- **\r** : return, U+000D
- **\"** : doublequote, U+0022
- **** : backslash, U+005C
- **\<intra-line whitespace><line ending>**
 <intra-line whitespace> : nothing
- **\x<hex scalar value>**; : specified character (note the terminating semi-colon).

Эти управляющие последовательности регистрочувствительны, за исключением того, что алфавитные цифры **<hex scalar value>** могут быть заглавными или строчными.

Любой другой символ в строке после обратного слеша является нарушением синтаксиса. За исключением окончания строки, любой символ в строковом литерале вне управляющей последовательности, не являющийся двойными кавычками, обозначает самого себя. Например, односимвольный строковый литерал **"\lambda"** (двойные кавычки, строчная **lambda**, двойные кавычки) представляет ту же самую строку, как и **"\x03bb;"**. Окончание строки, находящееся не за обратным слешем, обозначает символ **linefeed**.

Примеры:

"abc"	U+0061, U+0062, U+0063
"\x41;bc"	"Abc" ; U+0041, U+0062, U+0063
"\x41; bc"	"A bc" U+0041, U+0020, U+0062, U+0063
"\x41bc;"	U+41BC
"\x41"	&lexical exception
"\x;"	&lexical exception
"\x41bx;"	&lexical exception
"\x00000041;"	"A" ; U+0041
"\x0010FFFF;"	U+10FFFF
"\x00110000;"	&lexical exception out of range
"\x000000001;"	U+0001
"\xD800;"	&lexical exception in excluded range
"A	
bc"	U+0041, U+000A, U+0062, U+0063 if no space occurs after the A

4.2.8. Числа

Синтаксис внешних представлений числовых объектов формально описан правилом **<number>** формальной грамматики. Во внешних представлениях числовых объектов регистр является незначащим.

Представление числового объекта может быть записано в двоичном, восьмеричном, десятичном или шестнадцатеричном виде при помощи приставки основания. Приставками основания являются **#b** (двоичная), **#o** (восьмеричная), **#d** (десятичная) и **#x** (шестнадцатеричная). Без приставки основания представление числового объекта предполагается выраженным в десятичном виде.

Представление числового объекта может указываться точным или неточным с помощью приставки. Приставками являются **#e** для точного и **#i** для неточного. Приставка точности может находиться до или после любой используемой приставки основания. Если представление числового объекта не содержит приставки точности, константа является неточной, если она содержит десятичную точку, экспоненту или непустую ширину мантийсы; в противном случае она является точной.

В системах с неточными числовыми объектами переменной точности может быть полезно определение точности кон-

станты. С этой целью представления числовых объектов могут записываться с экспоненциальным маркером, указывающим желаемую точность неточного представления. Буквы **s**, **f**, **d** и **l** указывают на использование точности *short*, *single*, *double* и *long* соответственно. (Если существует менее четырёх внутренних неточных представлений, четыре спецификации размера преобразовываются в доступные. Например, реализация с двумя внутренними представлениями может преобразовывать вместе *short* и *single*, а также *long* и *double*) Кроме того, маркер точности **e** указывает точность по умолчанию для реализации. Точность по умолчанию имеет значение по крайней мере не менее точности *double*, но реализации могут позволять пользователю устанавливать данное умолчание.

```
3.1415926535898F0
    Round to single, perhaps 3.141593
0.6L0
    Extend to long, perhaps .6000000000000000
```

Представление числового объекта с непустой шириной мантиисы $x|p$ представляет наилучшее двоичное приближение с плавающей точкой x с помощью p -битной значащей части. Например, **1.1|53** является представлением наилучшего приближения 1.1 в IEEE двойной точности. Если x является внешним представлением неточного действительного числового объекта, не содержащим вертикальной черты, его численное значение должно вычисляться так, как если бы оно имело ширину мантиисы 53 или более.

Реализации, использующие двоичные представления с плавающей точкой действительных числовых объектов, должны представлять $x|p$ с помощью p -битной значащей части, если это удобно, или с большей точностью, если p -битная значащая часть не удобна, или с наибольшей доступной точностью, если p или более бит значащей части не удобны в реализации.

Note: Точность значащей части не следует путать с количеством бит, используемых для представления значащей части. В стандартах IEEE с плавающей точкой, например, старший значащий бит значащей части является неявным при одинарной и двойной точности, и явным при расширенной точности. На математическую точность не влияет, является ли этот бит явным или неявным. В реализациях, использующих двоичную плавающую точку, точность по умолчанию может быть вычислена путём вызова следующей процедуры:

```
(define (precision)
  (do ((n 0 (+ n 1))
      (x 1.0 (/ x 2.0)))
    ((= 1.0 (+ 1.0 x)) n)))
```

Note: Если основным представлением с плавающей точкой является двойная точность IEEE, $|p$, суффикс не должен всегда быть пропущенным: Денормализованные числа с плавающей точкой имеют пониженную точность, и поэтому их внешние представления должны содержать суффикс $|p$ с фактической шириной значащей части.

Литералы **+inf.0** и **-inf.0** представляют положительную и отрицательную бесконечность соответственно. Литерал **+nan.0** представляет NaN, являющийся результатом **(/ 0.0 0.0)**, и может также представлять другие NaN.

Если x является внешним представлением неточного действительного числового объекта и не содержит вертикальной черты и экспоненциального маркера кроме **e**, неточным действительным числовым объектом, представляющим его, является *flonum* (см. секцию библиотек ??). Некоторые или все другие внешние представления неточных действительных числовых объектов могут также представлять *flonums*, но это не является требованием данной работы.

4.3. Datum-синтаксис

Datum-синтаксис описывает синтаксис синтаксических данных в терминах последовательности (лексем) как определено в лексическом синтаксисе.

Синтаксические данные включают данные лексем, описанные в предыдущих секциях, а также следующие конструкции для формирования составных данных:

- пары и списки, заключённые в () или [] (см. секцию 4.3.2)
- векторы (см. секцию 4.3.3)
- байтовые векторы (см. секцию 4.3.4)

4.3.1. Формальное описание

Следующая грамматика описывает синтаксис синтаксических данных в терминах лексем различных видов, определённых в грамматике в секции 4.2:

```
<datum> → <lexeme datum>
          | <compound datum>
<lexeme datum> → <boolean> | <number>
                  | <character> | <string> | <symbol>
<symbol> → <identifier>
<compound datum> → (list) | <vector> | <bytevector>
(list) → (<datum>*) | [<datum>*]
        | (<datum>+ . <datum>) | [<datum>+ . <datum>]
        | <abbreviation>
<abbreviation> → <abbrev prefix> <datum>
<abbrev prefix> → ' | □ | # | , | @
                  | #' | #□ | #, | #, @
<vector> → # (<datum>*)
<bytevector> → #vu8 (<u8>*)
<u8> → <any <number> representing an exact
        integer in {0, ..., 255}>
```

4.3.2. Пары и списки

Данные списков и пар, представляющие значения пар и списков (см. секцию 11.9), представляются с помощью круглых или квадратных скобок. Соответствие пар квадратных скобок, находящихся в правилах (list), эквивалентно соответствию пар круглых скобок.

Наиболее общей нотацией для пар Scheme как синтаксических данных является “точечная” нотация $(\langle datum_1 \rangle . \langle datum_2 \rangle)$, где $\langle datum_1 \rangle$ является представлением значения *car*, а $\langle datum_2 \rangle$ - значения поля *cdr*. Например, **(4 . 5)** является парой, *car* которой - 4, а *cdr* - 5.

Для списков может использоваться более чёткая нотация: элементы списка просто заключаются в круглые скобки и разделяются пробелами. Пустой список представляется, как **()**. Например,

(a b c d e)

и

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

являются эквивалентными формами записи списка символов

Общее правило гласит, что если за точкой следует открывающаяся круглая скобка, точка, открывающаяся круглая скобка и соответствующая ей закрывающаяся круглая скобка во внешнем представлении могут быть пропущены.

Последовательность символов “**(4 . 5)**” является внешним представлением пары, а не выражением, которое вычисляется как пара. Аналогично, последовательность символов “**(+ 2 6)**” не является внешним представлением целого числа 8, даже при том, что она является выражением (на языке библиотеки **(rnrns base (6))**), вычисляемым как целое число 8; наоборот, это синтаксический *datum*, представляющий трёх-элементный список, элементами которого являются символ **+** и целые числа 2 и 6.

4.3.3. Векторы

Векторные данные, представляющие вектора объектов (см. секцию 11.13), представляются с помощью нотации **#(<datum> ...)**. Например, вектор с длиной 3, содержащий числовой объект для нуля в элементе 0, список **(2 2 2 2)** в элементе 1 и строку “**Анна**” в элементе 2, может быть представлен следующим образом:

#(0 (2 2 2 2) “Анна”)

Это внешнее представление вектора, а не выражение, вычисляемое как вектор.

4.3.4. Байтовые векторы

Байт-векторные данные, представляющие байтовые вектора (см. библиотечную главу ??), представляются с помощью нотации **#vu8(<u8> ...)**, где $\langle u8 \rangle$ представляет октет байтового вектора. Например, байтовый вектор с длиной 3, содержащий октеты 2, 24 и 123, может быть представлен следующим образом:

#vu8(2 24 123)

Это внешнее представление байтового вектора, а не выражение, вычисляемое как байтовый вектор.

4.3.5. Сокращения

'<datum>
□<datum>
,<datum>
,@<datum>
#'<datum>
#□<datum>
#,<datum>
#,@<datum>

Каждый из них является сокращением:

'<datum> for (quote <datum>),
□<datum> for (quasiquote <datum>),
,<datum> for (unquote <datum>),
,@<datum> for (unquote-splicing <datum>),
#'<datum> for (syntax <datum>),
#□<datum> for (quasisyntax <datum>),
#,<datum> for (unsyntax <datum>), and
#,@<datum> for (unsyntax-splicing <datum>).

5. Семантические концепции

5.1. Программы и библиотеки

Программа Scheme состоит из *программы верхнего уровня* совместно с набором *библиотек*, каждая из которых определяет часть программы, связанную с другими частями посредством явно указываемых экспорта и импорта. Библиотека состоит из ряда спецификаций экспорта и импорта, а также тела, состоящего из определений и выражений. Программа верхнего уровня похожа на библиотеку, но не имеет спецификаций экспорта. В главах 7 и 8 описаны синтаксис и семантика библиотек и программ верхнего уровня соответственно. В главе 11 описана основная библиотека, в которой определено большинство конструкций, традиционно ассоциированных со Scheme. В отдельной работе [?] описаны различные *стандартные библиотеки*, предоставляемые системой Scheme.

Деление на основную библиотеку и прочие стандартные библиотеки является прикладным, а не конструктивным. В частности, некоторые средства, обычно реализуемые как “примитивы” компилятором или системой во время выполнения, а не в терминах других стандартных процедур или синтаксических форм, не являются частью основной библиотеки, а определены в отдельных библиотеках. Примеры включают библиотеки *fixnums* и *flonums*, библиотеки исключений и условий, а также библиотеки для записей.

5.2. Переменные, ключевые слова и регионы

В теле библиотеки или программы верхнего уровня идентификатор может именовать или вид синтаксиса, или ячейку памяти, где может храниться значение. Идентификатор, именующий вид синтаксиса, называется *ключевым словом*, или *синтаксическим ключевым словом*, и считается *привязанным*

к этому виду синтаксиса (или, в случае синтаксической абстракции, *преобразователем*, транслирующим синтаксис в более примитивные формы; см. секцию 9.2). Идентификатор, именуемый ячейку памяти, называется *переменной* и считается *привязанным* к этой ячейке памяти. К каждой точке программы верхнего уровня или библиотеке привязан конкретный, постоянный набор идентификаторов. Набор таких идентификаторов, набор *видимого связывания*, называется *окружением*, действующим в данной точке.

Одни формы используются для создания синтаксических абстракций и связывания ключевых слов с преобразователями для этих новых синтаксических абстракций, в то время как другие формы создают новые ячейки памяти и связывают переменные с этими ячейками. Все эти формы обобщённо называются *конструкциями привязки*. Некоторые конструкции привязки принимают форму *определений*, в то время как другие являются выражениями. За исключением экспортируемых библиотечных привязок, привязка, созданная определением, видима только внутри тела, в котором находится определение, например, в теле библиотеки, в программе верхнего уровня или в выражении **lambda**. Экспортируемые библиотечные привязки также видимы внутри тел библиотек и программ верхнего уровня, импортирующих их (см. главу 7).

Выражения, связывающие переменные, включают формы **lambda**, **let**, **let***, **letrec**, **letrec***, **let-values**, и **let*-values** из основной библиотеки (см. секции 11.4.2, 11.4.6). Из них **lambda** является самой фундаментальной. Определения переменных, находящиеся внутри тела такого выражения, или внутри тела библиотеки или программы верхнего уровня, интерпретируются как набор привязок **letrec***. Кроме того, для тел библиотеки, к переменным, экспортируемым из библиотеки, можно обратиться, импортируя программы верхнего уровня и библиотеки.

Выражения, привязывающие ключевые слова, включают формы **let-syntax** и **letrec-syntax** (см. секцию 11.18). Форма **define** (см. секцию 11.2.1) является определением, создающим привязку переменной (см. секцию 11.2.1), а форма **define-syntax** - определением, создающим привязку ключевого слова (см. секцию 11.2.2).

Scheme – язык со статическими областями видимости и блочной структурой. Каждой позиции в программе верхнего уровня или тела библиотеки, где привязан идентификатор, соответствует *регион* кода, внутри которого видима привязка. Регион определяется конкретной конструкцией привязки, устанавливающей привязку; если привязка установлена, например, выражением **lambda**, её регионом является всё выражение **lambda**. Каждое упоминание идентификатора обращается к привязке идентификатора, установленной самым внутренним из регионов, содержащих её применение. Если применение идентификатора находится в позиции, где ни одно из близлежащих выражений не содержит привязки идентификатора, применение может обратиться к привязке, установленной определением или импортом из верхнего уровня окружающей библиотеки, или программой верхнего уровня (см. главу 7). Если для идентификатора не существует привязки, он называется *несвязанным*.

5.3. Исключительные ситуации

В данной работе рассматривается разнообразие исключительных ситуаций, среди которых нарушения синтаксиса, нарушения спецификаций процедур, нарушения ограничений реализации и исключительные ситуации в окружении. При обнаружении реализацией исключительной ситуации *возбуждается исключение*, что означает вызов специальной процедуры, называемой *текущим обработчиком исключений*. Программа может также возбудить исключение и перепреопределить текущий обработчик исключений; см. библиотечную секцию ??.

При возбуждении исключения порождается объект, описывающий характер исключительной ситуации. В работе используется система состояний, описанная в библиотечной секции ??, для описания исключительных ситуаций классификацией их типами состояний.

Некоторые исключительные ситуации допускают продолжение программы, если обработчик исключений предпримет соответствующее действие. Соответствующие исключения называются *продолжаемыми*. В большинстве исключительных ситуаций, описанных в данной работе, переносимые программы не могут зависеть от продолжаемых в месте обнаружения ситуации исключений. Для таких исключений обработчик исключений, вызванный исключением, не должен возвращаться. В некоторых случаях, однако, продолжение допустимо, и обработчик может возвращаться. См. библиотечную секцию ??.

Реализации должны возбудить исключение в случае невозможности продолжения корректного выполнения корректной программы из-за некоторого *ограничения реализации*. Например, реализация, не поддерживающая бесконечность, должна возбудить исключение с типом состояния **implementation-restriction**, при вычислении выражения, результатом которого может быть бесконечность.

Некоторые возможные ограничения реализации, типа нехватки представлений для NaN и бесконечностей (см. секцию 11.7.2) предугадываются в соответствии с данной работой, и реализации, как правило, должны возбуждать исключение соответствующего типа состояния, если они сталкиваются с такой ситуацией.

В данной работе применение фразы “исключение возбуждено” синонимично с “исключение должно быть возбуждено”. В данной работе используется фраза “исключение с типом состояния *i*”, для указания, что объект, порождаемый с исключением, является объектом состояния указанного типа. Фраза “продолжаемое исключение возбуждено”, указывает исключительную ситуацию, разрешающую возврат обработчика исключений.

5.4. Проверка аргументов

Многие процедуры, определённые в данной работе или в качестве части стандартной библиотеки, накладывают ограничения на принимаемые ими аргументы. Обычно процедура принимает только конкретное количество и конкретные типы аргументов. Аналогично, многие синтаксические

формы накладывают ограничения на значения результатов вычислений одной или более своих подформ. Эти ограничения подразумевают обязанности и программиста, и реализации. А именно, программист обязан гарантировать, что значения действительно удовлетворяют описанным в спецификации ограничениям. Реализация должна проверить, что ограничения спецификации действительно выполнены в том смысле, что предоставление возможности успешного завершения указанной операции является разумным, возможным и необходимым. Обязанности реализации определены более подробно в главе 6 и повсюду в данной работе.

Необходимо отметить, что у реализации не всегда имеется возможность полной проверки сформулированных в спецификации ограничений. Например, если операция определена для приёма процедуры с конкретными свойствами, проверка этих свойств неразрешима в принципе. Аналогично, некоторые операции принимают как списки, так и процедуры, которые вызывают эти операции. Так как списки могут быть видоизменены процедурами с помощью библиотеки (**rnrs mutable-pairs (6)**) (см. библиотечную главу ??), аргумент, являвшийся списком при запуске операции, может перестать быть списком во время выполнения операции. К тому же, процедура может перейти к другому продолжению, что не позволит операции произвести больше проверок. Требовать от операции проверки, что аргумент является списком после каждого вызова такой процедуры было бы непрактично. Кроме того, некоторым операциям, принимающим только списки, для выполнения своей функции необходимо частичное прохождение по данным спискам; требование реализации проходить остаток списка для проверки выполнения всех указанных ограничений может нарушить разумные условия производительности. По этим причинам обязанности программистов могут превышать обязанности проверки реализацией.

При обнаружении реализацией нарушения ограничения аргумента она должна возбудить исключение с типом состояния `&assertion` совместимым с безопасностью выполнения образом, как описано в секции 5.6.

5.5. Нарушения синтаксиса

Подформы специальной формы обычно должны подчиняться некоторым синтаксическим ограничениям. Поскольку формы могут быть представлены макро-разворачиванием, возможно, не завершённым, вопрос их подчинения указанным ограничениям неразрешим в принципе.

При окончании макро-разворачивания, однако, реализации должны обнаруживать нарушения синтаксиса. *Нарушение синтаксиса* - ошибка относительно синтаксиса библиотечных тел, тел верхнего уровня, или “syntax” записей в спецификации базовой библиотеки или стандартных библиотек. Кроме того, попытка присваивания неизменяемой переменной (то есть, переменным, экспортируемым библиотекой; см. секцию 7.1) также считается нарушением синтаксиса.

Если форма верхнего уровня или библиотеки в программе не является синтаксически корректной, реализация должна возбудить исключение с типом состояния `&syntax`, и не должна позволить начать выполнение такой программы или библиотеки верхнего уровня.

5.6. Безопасность

Стандартные библиотеки, экспорт которых описан в данном документе, называются *безопасными библиотеками*. Библиотеки и программы верхнего уровня, импортирующие только из безопасных библиотек, также называются безопасными.

Как определено данным документом, язык программирования Scheme безопасен в следующем смысле: выполнение безопасной программы верхнего уровня не может происходить настолько неверно, чтобы привести к аварийному завершению или продолжению выполнения, функционирование которого противоречит семантике, описанной в данном документе, если исключение не возбуждено.

Нарушения ограничений реализации должны возбуждать исключение с типом состояния `&implementation-restriction`, как должны все нарушения и ошибки, которые в противном случае угрожали бы целостности системы таким образом, что это могло бы привести к выполнению, противоречащему семантике, описанной в данном документе.

Вышеупомянутые свойства безопасности гарантируются только для программ верхнего уровня и библиотек, которые называются безопасными. В частности, реализации могут обеспечивать доступ к опасным библиотекам способами, которые не могут гарантировать безопасность.

5.7. Булевы значения

Хотя существует отдельный булевый тип, любое значение Scheme может использоваться в качестве булевого для проверки условия. В проверке условия все значения считаются истинными в таком тесте, за исключением `#f`. В данной работе используется слово “true” для обращения к любому значению Scheme, кроме `#f`, и слово “false” для обращения к `#f`.

5.8. Несколько возвращаемых значений

Выражение Scheme может вычисляться, как произвольное конечное количество значений. Эти значения передаются продолжению выражения.

Не все продолжения принимают несколько значений. Например, продолжение, принимающее аргумент вызова процедуры, гарантированно примет ровно одно значение. Результат передачи любого другого количества значений такому продолжению не определён. Процедура **call-with-values**, описанная в секции 11.15, позволяет создать продолжения, принимающие указанное количество возвращаемых значений. Если количество возвращаемых значений, переданных продолжению, созданному вызовом **call-with-values**,

не принято его потребителем, переданным в этом вызове, возбуждается исключение. Более полное описание количества значений, принимаемых различными продолжениями и последствия передачи неожиданного количества значений приведено в описании процедуры **values** в секции 11.15.

Во множестве форм основной библиотеки содержатся последовательности выражений в качестве подформ, вычисляемых последовательно, со всеми возвращаемыми ими значениями, кроме последнего, не учитываемого выражения. Продолжения, не учитывающие эти значения, принимают любое количество значений.

5.9. Неопределённое поведение

Если сообщается, что выражение “возвращает неопределённые значения”, выражение должно вычисляться без возбуждения исключения, но возвращаемые значения зависят от реализации; в данной работе явно не оговаривается, сколько или какие значения должны возвращаться. Программисты не должны полагаться на конкретное количество возвращаемых значений или непосредственно конкретных значений.

5.10. Модель памяти

Переменные и объекты типа пар, векторов, байтовых векторов, строк, хэш-таблиц и записей, неявно адресуются к областям памяти или последовательностям областей памяти. Строка, например, содержит столько областей памяти, сколько символов в строке. (Эти области памяти не обязаны соответствовать полному машинному слову.) Новое значение может быть сохранено в одной из этих областей памяти с помощью процедуры **string-set!**, но строка содержит те же области памяти, как и прежде.

Объект, считанный из области памяти обращением к переменной или процедурой типа **car**, **vector-ref** или **string-ref**, эквивалентен в смысле **eqv?** (секция 11.5) последнему сохранённому в области памяти перед считыванием объекту.

Каждая область памяти помечается признаком её использования. Ни переменная, ни объект, никогда не адресуются к неиспользуемой области памяти. При каждом упоминании в данной работе памяти, выделяемой для переменной или объекта, имеется в виду, что соответствующее количество областей памяти выбрано из множества неиспользуемых областей памяти, и выбранные области памяти помечаются признаком их использования перед обращением к ним переменной или объекта.

Хранить константы (то есть значения литеральных выражений) целесообразно в памяти только для чтения. Чтобы выразить это, удобно представить, что каждый объект, адресующийся к области памяти, связан с флагом, указывающим, является ли объект изменяемым или неизменяемым. Литеральные константы, строки, возвращаемые **symbol->string**, записи без изменяемых полей и другие значения, явно определённые неизменяемыми являются неизменяемыми объектами, в то время как все объекты, созданные другими процедурами, перечисленными в данной работе - изменяемыми.

Попытка сохранить новое значение в области памяти, адресуемое неизменяемым объектом, должна возбудить исключение с типом состояния **&assertion**.

5.11. Чистая хвостовая рекурсия

Реализации Scheme должны быть *чистыми хвост-рекурсивными*. Вызовы процедур, находящихся в определённых синтаксических контекстах, называемых *хвостовыми контекстами*, являются *хвостовыми вызовами*. Реализация Scheme обладает свойством чистой хвостовой рекурсии, если она поддерживает неограниченное количество активных хвостовых вызовов. Вызов является *активным*, если вызываемая процедура может все еще возвращаться. Отметим, что это включает регулярные возвращения так же как возвращения через продолжения, захваченные ранее **call-with-current-continuation**, вызываемые позже. В отсутствии захваченных продолжений, вызовы могли возвратиться максимум один раз, и активными будут вызовы, которые еще не возвратились. Формальное определение чистой хвостовой рекурсии может быть найдено в газете Клайнджера [?]. Правила идентификации хвостовых вызовов в конструкциях из библиотеки (**rnrs base (6)**) описаны в секции 11.20.

5.12. Динамический экстенд и динамическое окружение

Для вызова процедуры, время между тем, когда это начато и когда это возвращается, называют его *динамическим экстендом*. В Scheme, **call-with-current-continuation** (секция 11.15) позволяет вернуться динамическую экстенд после того, как ее вызов процедуры возвратился. Таким образом, динамический экстенд вызова может не быть единственным, связанным периодом времени.

Some operations described in the report acquire information in addition to their explicit arguments from the *dynamic environment*. For example, **call-with-current-continuation** accesses an implicit context established by **dynamic-wind** (section 11.15), and the **raise** procedure (library section ??) accesses the current exception handler. The operations that modify the dynamic environment do so dynamically, for the dynamic extent of a call to a procedure like **dynamic-wind** or **with-exception-handler**. When such a call returns, the previous dynamic environment is restored. The dynamic environment can be thought of as part of the dynamic extent of a call. Consequently, it is captured by **call-with-current-continuation**, and restored by invoking the escape procedure it creates.

6. Entry format

The chapters that describe bindings in the base library and the standard libraries are organized into entries. Each entry describes one language feature or a group of related features, where a feature is either a syntactic construct or a built-in procedure. An entry begins with one or more header lines of the form

template

The *category* defines the kind of binding described by the entry, typically either “syntax” or “procedure”. An entry may specify various restrictions on subforms or arguments. For background on this, see section 5.4.

category

6.1. Syntax entries

If *category* is “syntax”, the entry describes a special syntactic construct, and the template gives the syntax of the forms of the construct. The template is written in a notation similar to a right-hand side of the BNF rules in chapter 4, and describes the set of forms equivalent to the forms matching the template as syntactic data. Some “syntax” entries carry a suffix (*expand*), specifying that the syntactic keyword of the construct is exported with level 1. Otherwise, the syntactic keyword is exported with level 0; see section 7.2.

Components of the form described by a template are designated by syntactic variables, which are written using angle brackets, for example, $\langle \text{expression} \rangle$, $\langle \text{variable} \rangle$. Case is insignificant in syntactic variables. Syntactic variables stand for other forms, or sequences of them. A syntactic variable may refer to a non-terminal in the grammar for syntactic data (see section 4.3.1), in which case only forms matching that non-terminal are permissible in that position. For example, $\langle \text{identifier} \rangle$ stands for a form which must be an identifier. Also, $\langle \text{expression} \rangle$ stands for any form which is a syntactically valid expression. Other non-terminals that are used in templates are defined as part of the specification.

The notation

$\langle \text{thing}_1 \rangle \dots$

indicates zero or more occurrences of a $\langle \text{thing} \rangle$, and

$\langle \text{thing}_1 \rangle \langle \text{thing}_2 \rangle \dots$

indicates one or more occurrences of a $\langle \text{thing} \rangle$.

It is the programmer’s responsibility to ensure that each component of a form has the shape specified by a template. Descriptions of syntax may express other restrictions on the components of a form. Typically, such a restriction is formulated as a phrase of the form “ $\langle x \rangle$ must be a ...”. Again, these specify the programmer’s responsibility. It is the implementation’s responsibility to check that these restrictions are satisfied, as long as the macro transformers involved in expanding the form terminate. If the implementation detects that a component does not meet the restriction, an exception with condition type `&syntax` is raised.

6.2. Procedure entries

If *category* is “procedure”, then the entry describes a procedure, and the header line gives a template for a call to the procedure. Parameter names in the template are *italicized*. Thus the header line

`(vector-ref vector k)` procedure

indicates that the built-in procedure `vector-ref` takes two arguments, a vector *vector* and an exact non-negative integer object *k* (see below). The header lines

`(make-vector k)` procedure
`(make-vector k fill)` procedure

indicate that the `make-vector` procedure takes either one or two arguments. The parameter names are case-insensitive: *Vector* is the same as *vector*.

As with syntax templates, an ellipsis ... at the end of a header line, as in

`(= z1 z2 z3 ...)` procedure

indicates that the procedure takes arbitrarily many arguments of the same type as specified for the last parameter name. In this case, `=` accepts two or more arguments that must all be complex number objects.

A procedure that detects an argument that it is not specified to handle must raise an exception with condition type `&assertion`. Also, the argument specifications are exhaustive: if the number of arguments provided in a procedure call does not match any number of arguments accepted by the procedure, an exception with condition type `&assertion` must be raised.

For succinctness, the report follows the convention that if a parameter name is also the name of a type, then the corresponding argument must be of the named type. For example, the header line for `vector-ref` given above dictates that the first argument to `vector-ref` must be a vector. The following naming conventions imply type restrictions:

<i>obj</i>	any object
<i>z</i>	complex number object
<i>x</i>	real number object
<i>y</i>	real number object
<i>q</i>	rational number object
<i>n</i>	integer object
<i>k</i>	exact non-negative integer object
<i>bool</i>	boolean (<code>#f</code> or <code>#t</code>)
<i>octet</i>	exact integer object in $\{0, \dots, 255\}$
<i>byte</i>	exact integer object in $\{-128, \dots, 127\}$
<i>char</i>	character (see section 11.11)
<i>pair</i>	pair (see section 11.9)
<i>vector</i>	vector (see section 11.13)
<i>string</i>	string (see section 11.12)
<i>condition</i>	condition (see library section ??)
<i>bytevector</i>	bytevector (see library chapter ??)
<i>proc</i>	procedure (see section 1.6)

Other type restrictions are expressed through parameter-naming conventions that are described in specific chapters. For example, library chapter ?? uses a number of special parameter variables for the various subsets of the numbers.

With the listed type restrictions, it is the programmer’s responsibility to ensure that the corresponding argument is of the specified type. It is the implementation’s responsibility to check for that type.

A parameter called *list* means that it is the programmer’s responsibility to pass an argument that is a list (see section 11.9). It is the implementation’s responsibility to check that the argument

is appropriately structured for the operation to perform its function, to the extent that this is possible and reasonable. The implementation must at least check that the argument is either an empty list or a pair.

Descriptions of procedures may express other restrictions on the arguments of a procedure. Typically, such a restriction is formulated as a phrase of the form “*x* must be a ...” (or otherwise using the word “must”).

6.3. Implementation responsibilities

In addition to the restrictions implied by naming conventions, an entry may list additional explicit restrictions. These explicit restrictions usually describe both the programmer’s responsibilities, who must ensure that the subforms of a form are appropriate, or that an appropriate argument is passed, and the implementation’s responsibilities, which must check that subform adheres to the specified restrictions (if macro expansion terminates), or if the argument is appropriate. A description may explicitly list the implementation’s responsibilities for some arguments or subforms in a paragraph labeled “*Implementation responsibilities*”. In this case, the responsibilities specified for these subforms or arguments in the rest of the description are only for the programmer. A paragraph describing implementation responsibility does not affect the implementation’s responsibilities for checking subforms or arguments not mentioned in the paragraph.

6.4. Other kinds of entries

If *category* is something other than “syntax” and “procedure”, then the entry describes a non-procedural value, and the *category* describes the type of that value. The header line

```
&who condition type
```

indicates that `&who` is a condition type. The header line

```
unquote auxiliary syntax
```

indicates that `unquote` is a syntax binding that may occur only as part of specific surrounding expressions. Any use as an independent syntactic construct or identifier is a syntax violation. As with “syntax” entries, some “auxiliary syntax” entries carry a suffix (`expand`), specifying that the syntactic keyword of the construct is exported with level 1.

6.5. Equivalent entries

The description of an entry occasionally states that it is *the same* as another entry. This means that both entries are equivalent. Specifically, it means that if both entries have the same name and are thus exported from different libraries, the entries from both libraries can be imported under the same name without conflict.

6.6. Evaluation examples

The symbol “ \Rightarrow ” used in program examples can be read “evaluates to”. For example,

```
(* 5 8)  $\Rightarrow$  40
```

means that the expression `(* 5 8)` evaluates to the object 40. Or, more precisely: the expression given by the sequence of characters “`(* 5 8)`” evaluates, in an environment that imports the relevant library, to an object that may be represented externally by the sequence of characters “40”. See section 4.3 for a discussion of external representations of objects.

The “ \Rightarrow ” symbol is also used when the evaluation of an expression causes a violation. For example,

```
(integer->char #xD800)  $\Rightarrow$  &assertion exception
```

means that the evaluation of the expression `(integer->char #xD800)` must raise an exception with condition type `&assertion`.

Moreover, the “ \Rightarrow ” symbol is also used to explicitly say that the value of an expression is unspecified. For example:

```
(eqv? "" "")  $\Rightarrow$  unspecified
```

Mostly, examples merely illustrate the behavior specified in the entry. In some cases, however, they disambiguate otherwise ambiguous specifications and are thus normative. Note that, in some cases, specifically in the case of inexact number objects, the return value is only specified conditionally or approximately. For example:

```
(atan -inf.0)  $\Rightarrow$  -1.5707963267948965 ; approximately
```

6.7. Naming conventions

By convention, the names of procedures that store values into previously allocated locations (see section 5.10) usually end in “!”.

By convention, “->” appears within the names of procedures that take an object of one type and return an analogous object of another type. For example, `list->vector` takes a list and returns a vector whose elements are the same as those of the list.

By convention, the names of predicates—procedures that always return a boolean value—end in “?” when the name contains any letters; otherwise, the predicate’s name does not end with a question mark.

By convention, the components of compound names are separated by “-”. In particular, prefixes that are actual words or can be pronounced as though they were actual words are followed by a hyphen, except when the first character following the hyphen would be something other than a letter, in which case the hyphen is omitted. Short, unpronounceable prefixes (“`fx`” and “`fl`”) are not followed by a hyphen.

By convention, the names of condition types start with “&”.

7. Библиотеки

Библиотеки являются частями программы, которые могут поставляться независимо. Система библиотек поддерживает макроопределения внутри библиотек, макроэкспорт, а также различает фазы, в которых необходимы определения и импорт. В данной главе описана нотация библиотек и семантика расширения и реализации библиотек.

7.1. Библиотечная форма

Определение библиотеки имеет следующую форму:

```
(library <library name>
  (export <export spec> ...)
  (import <import spec> ...)
  <library body>)
```

Объявление библиотеки содержит следующие элементы:

- `<library name>` указывает имя библиотеки (возможно с версией).
- Подформа **export** указывает список экспорта, именуемый подмножество привязок, определённых внутри или импортированных в библиотеку.
- Подформа **import** указывает импортированные привязки в виде списка зависимостей импорта, где каждая зависимость указывает:
 - имя импортируемой библиотеки и, при необходимости, ограничения её версии,
 - релевантные уровни, например, время разворачивания или выполнения (см секцию 7.2, и

- подмножество библиотечных экспортов, чтобы сделать доступный внутри библиотеки импортирования, и локальных имён, для использования внутри библиотеки импортирования для каждого из экспортов библиотеки.

- `<library body>` - тело библиотеки, состоящее из последовательности определений и следующей за ней последовательности выражений. Определения могут быть как для локальных (неэкспортируемых), так и для экспортируемых связываний, а выражения являются инициализирующими выражениями, которые будут вычисляться для их эффектов.

Идентификатор может импортироваться с тем же локальным именем из двух или более библиотек, или для двух уровней из той же библиотеки только в случае, если привязки, экспортируемые каждой библиотекой, тождественны (то есть, связывание определено в одной библиотеке, и оно поступает только через импорт, экспортируя и реэкспортируя). В противном случае идентификатор не может быть импортирован несколько раз, определён несколько раз, или определён и импортирован. Идентификаторы не видимы внутри библиотеки, за исключением явно импортированных в библиотеку или определённых внутри библиотеки.

`<library name>` однозначно идентифицирует библиотеку внутри реализации и является глобально видимым в разделах **import** (см. ниже) всех других библиотек внутри реализации. `<library name>` имеет следующую форму:

```
((identifier1) (identifier2) ... (version))
```

где `<version>` является пустым или принимает следующую форму:

```
((sub-version) ...)
```

Каждый `<sub-version>` должен представлять точный неотрицательный целый объект. Пустой `<version>` эквивалентен `()`.

`<export spec>` именуется совокупность импортированных и локально определённых привязок для экспортирования, возможно с другими внешними именами. `<export spec>` должен иметь одну из следующих форм:

```
(identifier)
(rename ((identifier1) (identifier2)) ...)
```

В `<export spec>`, `<identifier>` именуется одиночную привязку, определённую внутри или импортированную в библиотеку, причём внешнее имя экспорта совпадает с именем привязки внутри библиотеки. Спецификация **rename** экспортирует привязку с именем `<identifier1>` в каждой паре `((identifier1) (identifier2))`, используя `<identifier2>` в качестве внешнего имени.

Каждый `<import spec>` указывает множество привязок, импортируемых в библиотеку, уровни, на которых они должны быть доступны и локальные имена, которыми они должны быть названы. `<import spec>` должен быть одним из следующего:

```
(import set)
(for (import set) (import level) ...)
```

`<import level>` является одним из следующего:

```
run
expand
(meta (level))
```

Где `<level>` представляет точный целый объект.

Как `<import level>`, **run** является сокращением для **(meta 0)**, а **expand** - сокращением для **(meta 1)**. Уровни и фазы описаны в секции 7.2.

`<import set>` именуется множество привязок из другой библиотеки и, возможно, определяет локальные имена для импортированных привязок. Он должен быть одним из следующего:

```
(library reference)
(library (library reference))
(only (import set) (identifier) ...)
(exception (import set) (identifier) ...)
(prefix (import set) (identifier))
(rename (import set) ((identifier1) (identifier2)) ...)
```

`<library reference>` идентифицирует библиотеку её именем и, произвольно, её версией. Он имеет одну из следующих форм:

```
((identifier1) (identifier2) ...)
((identifier1) (identifier2) ... (version reference))
```

`<library reference>`, чьим первым `<identifier>` является **for**, **library**, **only**, **exception**, **prefix** или **rename**, разрешается только внутри **library (import set)**. `<import set>` **(library (library reference))** в противном случае эквивалентен `<library reference>`.

`<library reference>` без `<version reference>` (первая форма выше) эквивалентен `<library reference>` с `<version reference>` `()`.

`<version reference>` указывает множество `<version>`, которым он соответствует. `<library reference>` идентифицирует все библиотеки того же самого имени и чья версия соответствует `<version reference>`. `<version reference>` имеет следующую форму:

```
((sub-version reference1) ... (sub-version referencen))
(and (version reference) ...)
(or (version reference) ...)
(not (version reference))
```

`<version reference>` первой формы соответствует `<version>` с по крайней мере n элементы, `<sub-version reference>` которых соответствует соответств. `<sub-version>`. **and** `<version reference>` соответствует версии, если все `<version references>` после **and** соответствуют этому. Соответственно, **or** `<version reference>` соответствует версии, если один из `<version references>` после **or** соответствуют этому, и **not** `<version reference>` соответствует версии, если `<version references>` после этого не соответствуют этому.

`<sub-version reference>` имеет одну из следующих форм:

```
(sub-version)
(>= (sub-version))
(<= (sub-version))
(and (sub-version reference) ...)
(or (sub-version reference) ...)
(not (sub-version reference))
```

$\langle \text{sub-version reference} \rangle$ первой формы соответствует $\langle \text{sub-version} \rangle$, если они равны. \geq $\langle \text{sub-version reference} \rangle$ первой формы соответствует sub-version , если оно больше или равно $\langle \text{sub-version} \rangle$ после него; аналогично для \leq . **and** $\langle \text{sub-version reference} \rangle$ соответствует sub-version , если все последующие $\langle \text{sub-version reference} \rangle$ соответствуют ему. Соответственно, **or** $\langle \text{sub-version reference} \rangle$ соответствует sub-version , если один из последующих $\langle \text{sub-version reference} \rangle$ соответствует ему, и **not** $\langle \text{sub-version reference} \rangle$ соответствует sub-version , если последующий $\langle \text{sub-version reference} \rangle$ не соответствует ему.

Примеры:

version reference	version	match?
()	(1)	yes
(1)	(1)	yes
(1)	(2)	no
(2 3)	(2)	no
(2 3)	(2 3)	yes
(2 3)	(2 3 5)	yes
(or (1 (\geq 1)) (2))	(2)	yes
(or (1 (\geq 1)) (2))	(1 1)	yes
(or (1 (\geq 1)) (2))	(1 0)	no
((or 1 2 3))	(1)	yes
((or 1 2 3))	(2)	yes
((or 1 2 3))	(3)	yes
((or 1 2 3))	(4)	no

Если при обращении к библиотеке идентифицировано более одной библиотеки, выбор библиотеки определяется неким зависимым от реализации методом.

Во избежании проблем, таких, как несовместимость типов и дублирование состояний, реализация должна запретить сосуществование в одной программе двух библиотек, библиотечные имена которых состоят из одинаковой последовательности идентификаторов, но версии которых являются несоответствующими.

По умолчанию все экспортируемые привязки из импортируемой библиотеки предполагаются видимыми внутри импортированной библиотеки с именами, присвоенными привязкам в импортируемой библиотеке. Точный набор импортируемых привязок и имён этих привязок может быть установлен с помощью форм **only**, **except**, **prefix** и **rename** как описано ниже.

- Форма **only** порождает подмножество привязок из иного $\langle \text{import set} \rangle$, содержащее только перечисленные $\langle \text{identifier} \rangle$. Включенные $\langle \text{identifier} \rangle$ должны существовать в исходном $\langle \text{import set} \rangle$.

- Форма **except** порождает подмножество привязок из другого $\langle \text{import set} \rangle$, содержащее все $\langle \text{identifier} \rangle$, кроме перечисленных. Включенные $\langle \text{identifier} \rangle$ должны существовать в исходном $\langle \text{import set} \rangle$.
- Форма **prefix** форма добавляет приставку $\langle \text{identifier} \rangle$ к каждому имени из другого $\langle \text{import set} \rangle$.
- Форма **rename**, (**rename** ($\langle \text{identifier}_1 \rangle$ $\langle \text{identifier}_2 \rangle$) ...), удаляет привязки для $\langle \text{identifier}_1 \rangle$... для формирования промежуточного $\langle \text{import set} \rangle$, затем добавляет привязки назад для соответствующего $\langle \text{identifier}_2 \rangle$... для формирования конечного $\langle \text{import set} \rangle$. Каждый $\langle \text{identifier}_1 \rangle$ должен быть в исходном $\langle \text{import set} \rangle$, каждый $\langle \text{identifier}_2 \rangle$ не должен быть в промежуточном $\langle \text{import set} \rangle$, а все $\langle \text{identifier}_2 \rangle$ должны быть разными.

Невыполнение приведённого выше ограничения является синтаксическим нарушением.

$\langle \text{library body} \rangle$ формы **library** состоит из форм, классифицируемых как *определения* или *выражения*. Принадлежность конкретной формы конкретному классу зависит от импортированных библиотек и результата расширения—см. главу 10. В общем случае формы, не являющиеся определениями (о доступных посредством базовой библиотеки определениях см. секцию ??), являются выражениями.

$\langle \text{library body} \rangle$ похоже на $\langle \text{body} \rangle$ (см. секцию 11.3) за исключением того, что в $\langle \text{library body} \rangle$ не обязаны содержаться выражения. Оно должно иметь следующую форму:

$\langle \text{definition} \rangle$... $\langle \text{expression} \rangle$...

Если формы **begin**, **let-syntax** или **letrec-syntax** находятся в теле верхнего уровня до первого выражения, они встраиваются в тело; см. секцию 11.4.7. Всё тело или его часть, включая части, обёрнутые в формы **begin**, **let-syntax** или **letrec-syntax**, может быть определено синтаксической абстракцией (см. секцию 9.2).

Преобразующие выражения и привязки вычисляются и создаются слева направо, как описано в главе 10. Выражения определений переменных вычисляются слева направо, как будто в неявном **letrec***, и выражения тела также вычисляются слева направо после выражений определений переменных. Для каждой экспортируемой переменной создаётся новая ячейка и инициализируется значение её локальной копии. Эффект возвращения дважды к продолжению последнего выражения тела не определён.

Note: Имена **library**, **export**, **import**, **for**, **run**, **expand**, **meta**, **import**, **export**, **only**, **except**, **prefix**, **rename**, **and**, **or**, **not**, \geq и \leq , входящие в библиотечный синтаксис, являются частью синтаксиса и не зарезервированы, то есть, те же имена могут использоваться в других целях внутри библиотеки или даже экспортироваться из библиотеки или импортироваться в библиотеку с другим смысловым содержанием, не затрагивая их использование в форме **library**.

Определённые с библиотекой привязки невидимы в коде вне библиотеки, если они явно не экспортированы из библиотеки. Однако экспортируемый макрос может *неявно экспортировать* в иных случаях неэкспортируемый идентификатор,

определённый в библиотеке или импортированный в неё. Иными словами, он может вставить обращение к этому идентификатору в генерируемый им выходной код.

Все явно экспортированные переменные являются неизменяемыми и в экспортируемых, и в импортируемых библиотеках. Таким образом, наличие явно экспортированной переменной в левой части выражения **set!** является нарушением синтаксиса как в экспортируемых, так и в импортируемых библиотеках.

Все неявно экспортированные переменные также являются неизменяемыми и в экспортируемых, и в импортируемых библиотеках. Таким образом, наличие переменной в левой части выражения **set!** в любом коде, сгенерированном экспортируемым макросом вне библиотеки, в которой определена переменная, является нарушением синтаксиса. Также является нарушением синтаксиса наличие обращения к присваиваемой переменной в любом коде, сгенерированном экспортируемым макросом вне библиотеки, в которой определена переменная, где присваиваемой переменной является находящаяся в левой части выражения **set!** переменная в экспортируемой библиотеке.

Все остальные определённые в библиотеке переменные являются изменяемыми.

7.2. Уровни экспорта и импорта

При разворачивании библиотеки может потребоваться информация этапа выполнения из другой библиотеки. Например, при вызове макропреобразователем процедуры из библиотеки *A*, библиотека *A* должна быть инстанцирована перед разворачиванием любого применения макроса в библиотеке *B*. Библиотека *A* в итоге может и не потребоваться при выполнении библиотеки *B* как части программы, или же она может потребоваться библиотеке *B* также и для этапа выполнения. Механизм библиотек характеризует эти этапы фазами, описанными в данной секции.

Каждая библиотека может характеризоваться информацией этапа разворачивания (как минимум, её импортированные библиотеки, список экспортируемых ключевых слов, список экспортируемых переменных и код вычисления преобразовательных выражений) и информацией этапа выполнения (как минимум, код вычисления выражений правой части определений переменных и код вычисления выражений тела). Информация этапа разворачивания должна быть доступна для раскрытия обращений к любым экспортируемым привязкам, а информация этапа выполнения — для вычисления обращений к любым экспортированным привязкам переменных.

Фаза — это этап, в течение которого вычисляются выражения в библиотеке. Внутри тела библиотеки выражения верхнего уровня и правые части форм **define** вычисляются на этапе выполнения, то есть, в фазе 0, а правые части форм **define-syntax** — на этапе разворачивания, то есть, в фазе 1. Если формы **define-syntax**, **let-syntax** или **letrec-syntax** находятся внутри кода, вычисляемого в фазе *n*, правые части вычисляются в фазе *n* + 1.

Эти фазы связаны с фазой, в которой используется сама библиотека. *Экземпляр* библиотеки соответствует вычислению её определений переменных и выражений в конкретной фазе относительно другой библиотеки — процессу, называемому *инстанцированием*. Например, если выражение верхнего

уровня библиотеки *B* обращается к переменной, экспортируемой из другой библиотеки *A*, то оно обращается к экспорту из экземпляра *A* в фазе 0 (относительно фазы *B*). Но если выражение фазы 1 внутри *B* обращается к тому же связыванию из *A*, то оно обращается к экспорту из экземпляра *A* в фазе 1 (относительно фазы *B*).

Посещение библиотеки соответствует вычислению её определений синтаксиса в конкретной фазе относительно другой библиотеки — процессу, называемому *посещением*. Например, если выражение верхнего уровня библиотеки *B* обращается к макро-экспорту из другой библиотеки *A*, то оно обращается к экспорту из посещения *A* в фазе 0 (относительно фазы *B*), которая соответствует вычислению выражения макропреобразователя в фазе 1.

Уровень — это лексическое свойство идентификатора, определяющее, в каких фазах допускаются обращения к нему. Уровнем каждого идентификатора, привязанного определением внутри библиотеки, является 0; то есть, к идентификатору допускается обращение только в фазе 0 внутри библиотеки. Уровень каждой импортированной привязки определяется не только ограничивающей формой **for** формы **import** в импортируемой библиотеке, но и уровнями идентификаторов в экспортируемой библиотеке. Уровни импорта и экспорта объединяются попарным суммированием всех комбинаций уровней. Например, обращения к импортируемому идентификатору, экспортируемому для уровней p_a и p_b и импортированному для уровней q_a , q_b , и q_c , применимы на уровнях $p_a + q_a$, $p_a + q_b$, $p_a + q_c$, $p_b + q_a$, $p_b + q_b$ и $p_b + q_c$. `<import set>` без ограничивающего **for** эквивалентен `(for <import set> run)`, что тождественно `(for <import set> (meta 0))`.

Уровнем экспорта экспортируемой привязки является 0 для всех привязок, определённых внутри экспортируемой библиотеки. Уровни экспорта реэкспортированной привязки, то есть, экспорта, импортированного из другой библиотеки, являются такими же, как и эффективные уровни импорта этой привязки внутри реэкспортируемой библиотеки.

Для библиотек, определённых в библиотечном документе, экспортным уровнем практически всех привязок является 0. Исключениями являются **syntax-rules**, **identifier-syntax**, ... и **_** из библиотеки `(rnrs base (6))`, экспортируемые с уровнем 1, **set!** из библиотеки `(rnrs base (6))`, экспортируемая с уровнями 0 и 1, и все привязки из композитной библиотеки `(rnrs (6))` (см. библиотечную главу ??), экспортируемые с уровнями 0 и 1.

Раскрытие макроса в библиотеке может привести к обращению к идентификатору, явно не импортированному в библиотеку. В этом случае фаза обращения должна соответствовать уровню идентификатора, сдвинутому на разность фаз исходной библиотеки (то есть, библиотеки, предоставляющей лексический контекст идентификатора) и библиотеки, содержащей обращение. Например, предположим, что при разворачивании библиотеки активизируется макропреобразователь, и при вычислении макропреобразователя происходит обращение к идентификатору, экспортируемому из другой библиотеки (таким образом, используется экземпляр библиотеки фазы 1); предположим также, что значением привязки является синтаксический объект, представляющий идентифи-

катор с привязкой только уровня n ; тогда, в разворачиваемой библиотеке идентификатор должен использоваться только в фазе $n + 1$. Поэтому при такой комбинации уровней и фаз могут применяться отрицательные уровни идентификаторов несмотря на то, что библиотеки существуют только в неотрицательных фазах.

При обращении из развёрнутой формы программы к какому-либо библиотечному определению в фазе 0, перед вычислением определений и выражений программы создаётся экземпляр адресуемой библиотеки для фазы 0. Это правило применяется транзитивно: при обращении из развёрнутой формы некоторой библиотеки в фазе 0 к идентификатору из другой библиотеки, перед инстанцированием библиотеки обращения в фазе n , должна инстанцироваться адресуемая библиотека в фазе n . При обращении к идентификатору в некоторой фазе n , большей чем 0, напротив, библиотека определения инстанцируется в фазе n в произвольное неспецифицированное время до вычисления обращения. Аналогично, при обращении к макро-ключевому слову в фазе n в течение разворачивания библиотеки, библиотека определения посещается в фазе n в произвольное неспецифицированное время до вычисления обращения.

Реализация может разделять экземпляры/посещения библиотек для различных фаз или использовать экземпляр/посещение в любой фазе как экземпляр/посещение в любой другой фазе. Реализация может также раскрывать каждую форму **library** с неодинаковыми посещениями библиотек в любой фазе и/или экземпляры библиотек в фазах выше 0. Реализация может создавать экземпляры/посещения большего количества библиотек в большем количестве фаз, чем необходимо для удовлетворения обращений. Когда идентификатор находится как выражение в фазе, противоречащей уровню идентификатора, реализация может поднять исключение или на этапе разворачивания, или на этапе выполнения, или она может позволить обращение. Таким образом, библиотека, содержание которой зависит от того, различаются ли экземпляры библиотек, или же разделяется при разворачивания фаз или **library**, может быть непортiruемой.

7.3. Примеры

Examples for various `<import spec>s` and `<export spec>s`:

```
(library (stack)
  (export make push! pop! empty!)
  (import (rnrs))

  (define (make) (list ' ()))
  (define (push! s v) (set-car! s (cons v (car s))))
  (define (pop! s) (let ([v (caar s)])
    (set-car! s (cдар s))
    v))
  (define (empty! s) (set-car! s ' ())))

(library (balloons)
  (export make push pop)
  (import (rnrs)))
```

```
(define (make w h) (cons w h))
(define (push b amt)
  (cons (- (car b) amt) (+ (cdr b) amt)))
(define (pop b) (display "Boom! ")
  (display (* (car b) (cdr b)))
  (newline)))
```

```
(library (party)
  ;; Total exports:
  ;; make, push, push!, make-party, pop!
  (export (rename (balloon:make make)
    (balloon:push push))
    push!
    make-party
    (rename (party-pop! pop!)))
  (import (rnrs)
    (only (stack) make push! pop!) ; not empty!
    (prefix (balloons) balloon:)))
```

```
;; Creates a party as a stack of balloons,
;; starting with two balloons
(define (make-party)
  (let ([s (make)]) ; from stack
    (push! s (balloon:make 10 10))
    (push! s (balloon:make 12 9))
    s))
(define (party-pop! p)
  (balloon:pop (pop! p)))
```

```
(library (main)
  (export)
  (import (rnrs) (party))

  (define p (make-party))
  (pop! p) ; displays "Boom! 108"
  (push! p (push (make 5 5) 1))
  (pop! p) ; displays "Boom! 24"
```

Examples for macros and phases:

```
(library (my-helpers id-stuff)
  (export find-dup)
  (import (rnrs)))

(define (find-dup l)
  (and (pair? l)
    (let loop ((rest (cdr l)))
      (cond
        [(null? rest) (find-dup (cdr l))]
        [(bound-identifier=? (car l) (car rest))
         (car rest)]
        [else (loop (cdr rest))])))))

(library (my-helpers values-stuff)
  (export mvlet)
  (import (rnrs) (for (my-helpers id-stuff) expand)))

(define-syntax mvlet
  (lambda (stx)
    (syntax-case stx ()
      [(_ [(id ...) expr] body0 body ...)
       (not (find-dup (syntax (id ...))))]
      (syntax
        (call-with-values
```

```

        (lambda () expr)
        (lambda (id ...) body0 body ...)))))))))

(library (let-div)
  (export let-div)
  (import (rnrs)
    (my-helpers values-stuff)
    (rnrs r5rs))

  (define (quotient+remainder n d)
    (let ([q (quotient n d)]
          (values q (- n (* q d)))))
    (define-syntax let-div
      (syntax-rules ()
        [(_ n d (q r) body0 body ...)
         (mvlet [(q r) (quotient+remainder n d)]
           body0 body ...)])))

```

8. Top-level programs

A *top-level program* specifies an entry point for defining and running a Scheme program. A top-level program specifies a set of libraries to import and code to run. Through the imported libraries, whether directly or through the transitive closure of importing, a top-level program defines a complete Scheme program.

8.1. Top-level program syntax

A top-level program is a delimited piece of text, typically a file, that has the following form:

```
(import form) <top-level body>
```

An *<import form>* has the following form:

```
(import <import spec> ...)
```

A *<top-level body>* has the following form:

```
<top-level body form> ...
```

A *<top-level body form>* is either a *<definition>* or an *<expression>*.

The *<import form>* is identical to the *import* clause in libraries (see section 7.1), and specifies a set of libraries to import. A *<top-level body>* is like a *<library body>* (see section 7.1), except that definitions and expressions may occur in any order. Thus, the syntax specified by *<top-level body form>* refers to the result of macro expansion.

When uses of *begin*, *let-syntax*, or *letrec-syntax* from the *(rnrs base (6))* library occur in a top-level body prior to the first expression, they are spliced into the body; see section 11.4.7. Some or all of the body, including portions wrapped in *begin*, *let-syntax*, or *letrec-syntax* forms, may be specified by a syntactic abstraction (see section 9.2).

8.2. Top-level program semantics

A top-level program is executed by treating the program similarly to a library, and evaluating its definitions and expressions. The

semantics of a top-level body may be roughly explained by a simple translation into a library body: Each *<expression>* that appears before a definition in the top-level body is converted into a dummy definition

```
(define <variable> (begin <expression> <unspecified>))
```

where *<variable>* is a fresh identifier and *<unspecified>* is a side-effect-free expression returning an unspecified value. (It is generally impossible to determine which forms are definitions and expressions without concurrently expanding the body, so the actual translation is somewhat more complicated; see chapter 10.)

On platforms that support it, a top-level program may access its command line by calling the *command-line* procedure (see library section ??).

9. Primitive syntax

After the *import* form within a *library* form or a top-level program, the forms that constitute the body of the library or the top-level program depend on the libraries that are imported. In particular, imported syntactic keywords determine the available syntactic abstractions and whether each form is a definition or expression. A few form types are always available independent of imported libraries, however, including constant literals, variable references, procedure calls, and macro uses.

9.1. Primitive expression types

The entries in this section all describe expressions, which may occur in the place of *<expression>* syntactic variables. See also section 11.4.

Constant literals

<i><number></i>	syntax
<i><boolean></i>	syntax
<i><character></i>	syntax
<i><string></i>	syntax
<i><bytevector></i>	syntax

An expression consisting of a representation of a number object, a boolean, a character, a string, or a bytevector, evaluates “to itself”.

```

145932          ⇒ 145932
#t              ⇒ #t
"abc"           ⇒ "abc"
#vu8(2 24 123) ⇒ #vu8(2 24 123)

```

As noted in section 5.10, the value of a literal expression is immutable.

Variable references

<i><variable></i>	syntax
-------------------------	--------

An expression consisting of a variable (section 5.2) is a variable reference if it is not a macro use (see below). The value of the variable reference is the value stored in the location to which the

variable is bound. It is a syntax violation to reference an unbound variable.

The following example assumes the base library has been imported:

```
(define x 28)
x           ⇒ 28
```

Procedure calls

```
(⟨operator⟩ ⟨operand1⟩ ...)          syntax
```

A procedure call consists of expressions for the procedure to be called and the arguments to be passed to it, with enclosing parentheses. A form in an expression context is a procedure call if ⟨operator⟩ is not an identifier bound as a syntactic keyword (see section 9.2 below).

When a procedure call is evaluated, the operator and operand expressions are evaluated (in an unspecified order) and the resulting procedure is passed the resulting arguments.

The following examples assume the `(rnrns base (6))` library has been imported:

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

If the value of ⟨operator⟩ is not a procedure, an exception with condition type `&assertion` is raised. Also, if ⟨operator⟩ does not accept as many arguments as there are ⟨operand⟩s, an exception with condition type `&assertion` is raised.

Note: In contrast to other dialects of Lisp, the order of evaluation is unspecified, and the operator expression and the operand expressions are always evaluated with the same evaluation rules.

Although the order of evaluation is otherwise unspecified, the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with some sequential order of evaluation. The order of evaluation may be chosen differently for each procedure call.

Note: In many dialects of Lisp, the form `()` is a legitimate expression. In Scheme, expressions written as list/pair forms must have at least one subexpression, so `()` is not a syntactically valid expression.

9.2. Macros

Libraries and top-level programs can define and use new kinds of derived expressions and definitions called *syntactic abstractions* or *macros*. A syntactic abstraction is created by binding a keyword to a *macro transformer* or, simply, *transformer*. The transformer determines how a use of the macro (called a *macro use*) is transcribed into a more primitive form.

Most macro uses have the form:

```
(⟨keyword⟩ ⟨datum⟩ ...)
```

where ⟨keyword⟩ is an identifier that uniquely determines the kind of form. This identifier is called the *syntactic keyword*, or simply *keyword*, of the macro. The number of ⟨datum⟩s and the syntax of each depends on the syntactic abstraction.

Macro uses can also take the form of improper lists, singleton identifiers, or `set!` forms, where the second subform of the `set!` is the keyword (see section 11.19) library section ??):

```
(⟨keyword⟩ ⟨datum⟩ ... . ⟨datum⟩)
⟨keyword⟩
(set! ⟨keyword⟩ ⟨datum⟩)
```

The `define-syntax`, `let-syntax` and `letrec-syntax` forms, described in sections 11.2.2 and 11.18, create bindings for keywords, associate them with macro transformers, and control the scope within which they are visible.

The `syntax-rules` and `identifier-syntax` forms, described in section 11.19, create transformers via a pattern language. Moreover, the `syntax-case` form, described in library chapter ??, allows creating transformers via arbitrary Scheme code.

Keywords occupy the same name space as variables. That is, within the same scope, an identifier can be bound as a variable or keyword, or neither, but not both, and local bindings of either kind may shadow other bindings of either kind.

Macros defined using `syntax-rules` and `identifier-syntax` are “hygienic” and “referentially transparent” and thus preserve Scheme’s lexical scoping [?, ?, ?, ?]:

- If a macro transformer inserts a binding for an identifier (variable or keyword) not appearing in the macro use, the identifier is in effect renamed throughout its scope to avoid conflicts with other identifiers.
- If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.

Macros defined using the `syntax-case` facility are also hygienic unless `datum->syntax` (see library section ??) is used.

10. Expansion process

Macro uses (see section 9.2) are expanded into *core forms* at the start of evaluation (before compilation or interpretation) by a *syntax expander*. The set of core forms is implementation-dependent, as is the representation of these forms in the expander’s output. If the expander encounters a syntactic abstraction, it invokes the associated transformer to expand the syntactic abstraction, then repeats the expansion process for the form returned by the transformer. If the expander encounters a core form, it recursively processes its subforms that are in expression or definition context, if any, and reconstructs the form from the expanded subforms. Information about identifier bindings is maintained during expansion to enforce lexical scoping for variables and keywords.

To handle definitions, the expander processes the initial forms in a ⟨body⟩ (see section 11.3) or ⟨library body⟩ (see section 7.1) from left to right. How the expander processes each form encountered depends upon the kind of form.

macro use The expander invokes the associated transformer to transform the macro use, then recursively performs whichever of these actions are appropriate for the resulting form.

define-syntax form The expander expands and evaluates the right-hand-side expression and binds the keyword to the resulting transformer.

define form The expander records the fact that the defined identifier is a variable but defers expansion of the right-hand-side expression until after all of the definitions have been processed.

begin form The expander splices the subforms into the list of body forms it is processing. (See section 11.4.7.)

let-syntax or letrec-syntax form The expander splices the inner body forms into the list of (outer) body forms it is processing, arranging for the keywords bound by the `let-syntax` and `letrec-syntax` to be visible only in the inner body forms.

expression, i.e., nondefinition The expander completes the expansion of the deferred right-hand-side expressions and the current and remaining expressions in the body, and then creates the equivalent of a `letrec*` form from the defined variables, expanded right-hand-side expressions, and expanded body expressions.

For the right-hand side of the definition of a variable, expansion is deferred until after all of the definitions have been seen. Consequently, each keyword and variable reference within the right-hand side resolves to the local binding, if any.

A definition in the sequence of forms must not define any identifier whose binding is used to determine the meaning of the undeferred portions of the definition or any definition that precedes it in the sequence of forms. For example, the bodies of the following expressions violate this restriction.

```
(let ()
  (define define 17)
  (list define))

(let-syntax ([def0 (syntax-rules ()
                      [(_ x) (define x 0)])])
  (let ([z 3])
    (def0 z)
    (define def0 list)
    (list z)))

(let ()
  (define-syntax foo
    (lambda (e)
      (+ 1 2)))
  (define + 2)
  (foo))
```

The following do not violate the restriction.

```
(let ([x 5])
  (define lambda list)
  (lambda x x)) ⇒ (5 5)

(let-syntax ([def0 (syntax-rules ()
                      [(_ x) (define x 0)])])
  (let ([z 3])
    (define def0 list)
```

```
(def0 z)
(list z))) ⇒ (3)
```

```
(let ()
  (define-syntax foo
    (lambda (e)
      (let ([+ -]) (+ 1 2))))
  (define + 2)
  (foo)) ⇒ -1
```

The implementation should treat a violation of the restriction as a syntax violation.

Note that this algorithm does not directly reprocess any form. It requires a single left-to-right pass over the definitions followed by a single pass (in any order) over the body expressions and deferred right-hand sides.

Example:

```
(lambda (x)
  (define-syntax defun
    (syntax-rules ()
      [(_ x a e) (define x (lambda a e))]))
  (defun even? (n) (or (= n 0) (odd? (- n 1))))
  (define-syntax odd?
    (syntax-rules () [(_ n) (not (even? n))]))
  (odd? (if (odd? x) (* x x) x)))
```

In the example, the definition of `defun` is encountered first, and the keyword `defun` is associated with the transformer resulting from the expansion and evaluation of the corresponding right-hand side. A use of `defun` is encountered next and expands into a `define` form. Expansion of the right-hand side of this `define` form is deferred. The definition of `odd?` is next and results in the association of the keyword `odd?` with the transformer resulting from expanding and evaluating the corresponding right-hand side. A use of `odd?` appears next and is expanded; the resulting call to `not` is recognized as an expression because `not` is bound as a variable. At this point, the expander completes the expansion of the current expression (the call to `not`) and the deferred right-hand side of the `even?` definition; the uses of `odd?` appearing in these expressions are expanded using the transformer associated with the keyword `odd?`. The final output is the equivalent of

```
(lambda (x)
  (letrec* ([even?
            (lambda (n)
              (or (= n 0)
                  (not (even? (- n 1))))))]
    (not (even? (if (not (even? x)) (* x x) x)))))
```

although the structure of the output is implementation-dependent.

Because definitions and expressions can be interleaved in a `<top-level body>` (see chapter 8), the expander's processing of a `<top-level body>` is somewhat more complicated. It behaves as described above for a `<body>` or `<library body>` with the following exceptions: When the expander finds a nondefinition, it defers its expansion and continues scanning for definitions. Once it reaches the end of the set of forms, it processes the deferred right-hand-side and body expressions, then generates the equivalent of a `letrec*` form from the defined variables, expanded right-hand-side expressions, and expanded body expressions. For each body

expression $\langle \text{expression} \rangle$ that appears before a variable definition in the body, a dummy binding is created at the corresponding place within the set of `letrec*` bindings, with a fresh temporary variable on the left-hand side and the equivalent of `(begin $\langle \text{expression} \rangle$ $\langle \text{unspecified} \rangle$)`, where $\langle \text{unspecified} \rangle$ is a side-effect-free expression returning an unspecified value, on the right-hand side, so that left-to-right evaluation order is preserved. The `begin` wrapper allows $\langle \text{expression} \rangle$ to evaluate to an arbitrary number of values.

11. Base library

This chapter describes Scheme's `(rnrns base (6))` library, which exports many of the procedure and syntax bindings that are traditionally associated with Scheme.

Section 11.20 defines the rules that identify tail calls and tail contexts in constructs from the `(rnrns base (6))` library.

11.1. Base types

No object satisfies more than one of the following predicates:

<code>boolean?</code>	<code>pair?</code>
<code>symbol?</code>	<code>number?</code>
<code>char?</code>	<code>string?</code>
<code>vector?</code>	<code>procedure?</code>
<code>null?</code>	

These predicates define the base types *boolean*, *pair*, *symbol*, *number*, *char* (or *character*), *string*, *vector*, and *procedure*. Moreover, the empty list is a special object of its own type.

Note that, although there is a separate boolean type, any Scheme value can be used as a boolean value for the purpose of a conditional test; see section 5.7.

11.2. Definitions

Definitions may appear within a $\langle \text{top-level body} \rangle$ (section 8.1), at the top of a $\langle \text{library body} \rangle$ (section 7.1), or at the top of a $\langle \text{body} \rangle$ (section 11.3).

A $\langle \text{definition} \rangle$ may be a variable definition (section 11.2.1) or keyword definition (section 11.2.1). Macro uses that expand into definitions or groups of definitions (packaged in a `begin`, `let-syntax`, or `letrec-syntax` form; see section 11.4.7) may also appear wherever other definitions may appear.

11.2.1. Variable definitions

The `define` form described in this section is a $\langle \text{definition} \rangle$ used to create variable bindings and may appear anywhere other definitions may appear.

<code>(define $\langle \text{variable} \rangle$ $\langle \text{expression} \rangle$)</code>	syntax
<code>(define $\langle \text{variable} \rangle$)</code>	syntax
<code>(define ($\langle \text{variable} \rangle$ $\langle \text{formals} \rangle$) $\langle \text{body} \rangle$)</code>	syntax
<code>(define ($\langle \text{variable} \rangle$. $\langle \text{formal} \rangle$) $\langle \text{body} \rangle$)</code>	syntax

The first form of `define` binds $\langle \text{variable} \rangle$ to a new location before assigning the value of $\langle \text{expression} \rangle$ to it.

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3)            $\Rightarrow$  6
(define first car)
(first '(1 2))      $\Rightarrow$  1
```

The continuation of $\langle \text{expression} \rangle$ should not be invoked more than once.

Implementation responsibilities: Implementations should detect that the continuation of $\langle \text{expression} \rangle$ is invoked more than once. If the implementation detects this, it must raise an exception with condition type `&assertion`.

The second form of `define` is equivalent to

```
(define  $\langle \text{variable} \rangle$   $\langle \text{unspecified} \rangle$ )
```

where $\langle \text{unspecified} \rangle$ is a side-effect-free expression returning an unspecified value.

In the third form of `define`, $\langle \text{formals} \rangle$ must be either a sequence of zero or more variables, or a sequence of one or more variables followed by a dot `.` and another variable (as in a `lambda` expression, see section 11.4.2). This form is equivalent to

```
(define  $\langle \text{variable} \rangle$ 
  (lambda ( $\langle \text{formals} \rangle$ )  $\langle \text{body} \rangle$ )).
```

In the fourth form of `define`, $\langle \text{formal} \rangle$ must be a single variable. This form is equivalent to

```
(define  $\langle \text{variable} \rangle$ 
  (lambda ( $\langle \text{formal} \rangle$ )  $\langle \text{body} \rangle$ )).
```

11.2.2. Syntax definitions

The `define-syntax` form described in this section is a $\langle \text{definition} \rangle$ used to create keyword bindings and may appear anywhere other definitions may appear.

```
(define-syntax  $\langle \text{keyword} \rangle$   $\langle \text{expression} \rangle$ )          syntax
```

Binds $\langle \text{keyword} \rangle$ to the value of $\langle \text{expression} \rangle$, which must evaluate, at macro-expansion time, to a transformer. Macro transformers can be created using the `syntax-rules` and `identifier-syntax` forms described in section 11.19. See library section ?? for a more complete description of transformers.

Keyword bindings established by `define-syntax` are visible throughout the body in which they appear, except where shadowed by other bindings, and nowhere else, just like variable bindings established by `define`. All bindings established by a set of definitions, whether keyword or variable definitions, are visible within the definitions themselves.

Implementation responsibilities: The implementation should detect if the value of $\langle \text{expression} \rangle$ cannot possibly be a transformer.

Example:

```
(let ()
  (define even?
    (lambda (x)
      (or (= x 0) (odd? (- x 1)))))
  (define-syntax odd?
    (syntax-rules ()
      ((odd? x) (not (even? x)))))
  (even? 10))            $\Rightarrow$  #t
```


An implication of the left-to-right processing order (section 10) is that one definition can affect whether a subsequent form is also a definition.

Example:

```
(let ()
  (define-syntax bind-to-zero
    (syntax-rules ()
      ((bind-to-zero id) (define id 0))))
  (bind-to-zero x)
  x) ⇒ 0
```

The behavior is unaffected by any binding for `bind-to-zero` that might appear outside of the `let` expression.

11.3. Bodies

The `<body>` of a `lambda`, `let`, `let*`, `let-values`, `let*-values`, `letrec`, or `letrec*` expression, or that of a definition with a body consists of zero or more definitions followed by one or more expressions.

`<definition> ... <expression1> <expression2> ...`

Each identifier defined by a definition is local to the `<body>`. That is, the identifier is bound, and the region of the binding is the entire `<body>` (see section 5.2).

Example:

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

When `begin`, `let-syntax`, or `letrec-syntax` forms occur in a body prior to the first expression, they are spliced into the body; see section 11.4.7. Some or all of the body, including portions wrapped in `begin`, `let-syntax`, or `letrec-syntax` forms, may be specified by a macro use (see section 9.2).

An expanded `<body>` (see chapter 10) containing variable definitions can always be converted into an equivalent `letrec*` expression. For example, the `let` expression in the above example is equivalent to

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

11.4. Expressions

The entries in this section describe the expressions of the `(rnrs base (6))` library, which may occur in the position of the `<expression>` syntactic variable in addition to the primitive expression types as described in section 9.1.

11.4.1. Quotation

`(quote <datum>)` syntax

Syntax: `<Datum>` should be a syntactic datum.

Semantics: `(quote <datum>)` evaluates to the datum value represented by `<datum>` (see section 4.3). This notation is used to include constants.

```
(quote a) ⇒ a
(quote #(a b c)) ⇒ #(a b c)
(quote (+ 1 2)) ⇒ (+ 1 2)
```

As noted in section 4.3.5, `(quote <datum>)` may be abbreviated as `'<datum>`:

```
'"abc" ⇒ "abc"
'145932 ⇒ 145932
'a ⇒ a
'#(a b c) ⇒ #(a b c)
'() ⇒ ()
'(+ 1 2) ⇒ (+ 1 2)
'(quote a) ⇒ (quote a)
'a ⇒ (quote a)
```

As noted in section 5.10, constants are immutable.

Note: Different constants that are the value of a `quote` expression may share the same locations.

11.4.2. Procedures

`(lambda <formals> <body>)` syntax

Syntax: `<Formals>` must be a formal parameter list as described below, and `<body>` must be as described in section 11.3.

Semantics: A `lambda` expression evaluates to a procedure. The environment in effect when the `lambda` expression is evaluated is remembered as part of the procedure. When the procedure is later called with some arguments, the environment in which the `lambda` expression was evaluated is extended by binding the variables in the parameter list to fresh locations, and the resulting argument values are stored in those locations. Then, the expressions in the body of the `lambda` expression (which may contain definitions and thus represent a `letrec*` form, see section 11.3) are evaluated sequentially in the extended environment. The results of the last expression in the body are returned as the results of the procedure call.

```
(lambda (x) (+ x x)) ⇒ a procedure
((lambda (x) (+ x x)) 4) ⇒ 8
```

```
((lambda (x)
  (define (p y)
    (+ y 1))
  (+ (p x) x))
5) ⇒ 11
```

```
(define reverse-subtract
  (lambda (x y) (- y x)))
(reverse-subtract 7 10) ⇒ 3
```

```
(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6) ⇒ 10
```

⟨Forms⟩ must have one of the following forms:

- (⟨variable₁⟩ ...): The procedure takes a fixed number of arguments; when the procedure is called, the arguments are stored in the bindings of the corresponding variables.
- ⟨variable⟩: The procedure takes any number of arguments; when the procedure is called, the sequence of arguments is converted into a newly allocated list, and the list is stored in the binding of the ⟨variable⟩.
- (⟨variable₁⟩ ... ⟨variable_n⟩ . ⟨variable_{n+1}⟩): If a period . precedes the last variable, then the procedure takes *n* or more arguments, where *n* is the number of parameters before the period (there must be at least one). The value stored in the binding of the last variable is a newly allocated list of the arguments left over after all the other arguments have been matched up against the other parameters.

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

Any ⟨variable⟩ must not appear more than once in ⟨forms⟩.

11.4.3. Conditionals

```
(if ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩)  syntax
(if ⟨test⟩ ⟨consequent⟩)              syntax
```

Syntax: ⟨Test⟩, ⟨consequent⟩, and ⟨alternate⟩ must be expressions.

Semantics: An `if` expression is evaluated as follows: first, ⟨test⟩ is evaluated. If it yields a true value (see section 5.7), then ⟨consequent⟩ is evaluated and its values are returned. Otherwise ⟨alternate⟩ is evaluated and its values are returned. If ⟨test⟩ yields #f and no ⟨alternate⟩ is specified, then the result of the expression is unspecified.

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

The ⟨consequent⟩ and ⟨alternate⟩ expressions are in tail context if the `if` expression itself is; see section 11.20.

11.4.4. Assignments

```
(set! ⟨variable⟩ ⟨expression⟩)  syntax
```

⟨Expression⟩ is evaluated, and the resulting value is stored in the location to which ⟨variable⟩ is bound. ⟨Variable⟩ must be bound either in some region enclosing the `set!` expression or at the top level. The result of the `set!` expression is unspecified.

```
(let ((x 2))
  (+ x 1)
  (set! x 4)
  (+ x 1))           =>  5
```

It is a syntax violation if ⟨variable⟩ refers to an immutable binding.

Note: The identifier `set!` is exported with level 1 as well. See section 11.19.

11.4.5. Derived conditionals

```
(cond ⟨cond clause1⟩ ⟨cond clause2⟩ ...)  syntax
=>                                           auxiliary syntax
else                                       auxiliary syntax
```

Syntax: Each ⟨cond clause⟩ must be of the form

```
(⟨test⟩ ⟨expression1⟩ ...)
```

where ⟨test⟩ is an expression. Alternatively, a ⟨cond clause⟩ may be of the form

```
(⟨test⟩ => ⟨expression⟩)
```

The last ⟨cond clause⟩ may be an “else clause”, which has the form

```
(else ⟨expression1⟩ ⟨expression2⟩ ...).
```

Semantics: A `cond` expression is evaluated by evaluating the ⟨test⟩ expressions of successive ⟨cond clause⟩s in order until one of them evaluates to a true value (see section 5.7). When a ⟨test⟩ evaluates to a true value, then the remaining ⟨expression⟩s in its ⟨cond clause⟩ are evaluated in order, and the results of the last ⟨expression⟩ in the ⟨cond clause⟩ are returned as the results of the entire `cond` expression. If the selected ⟨cond clause⟩ contains only the ⟨test⟩ and no ⟨expression⟩s, then the value of the ⟨test⟩ is returned as the result. If the selected ⟨cond clause⟩ uses the `=>` alternate form, then the ⟨expression⟩ is evaluated. Its value must be a procedure. This procedure should accept one argument; it is called on the value of the ⟨test⟩ and the values returned by this procedure are returned by the `cond` expression. If all ⟨test⟩s evaluate to #f, and there is no else clause, then the conditional expression returns unspecified values; if there is an else clause, then its ⟨expression⟩s are evaluated, and the values of the last one are returned.

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

For a ⟨cond clause⟩ of one of the following forms

```
(⟨test⟩ ⟨expression1⟩ ...)
(else ⟨expression1⟩ ⟨expression2⟩ ...)
```

the last ⟨expression⟩ is in tail context if the `cond` form itself is.

For a ⟨cond clause⟩ of the form

```
(⟨test⟩ => ⟨expression⟩)
```

the (implied) call to the procedure that results from the evaluation of ⟨expression⟩ is in a tail context if the `cond` form itself is. See section 11.20.

A sample definition of `cond` in terms of simpler forms is in appendix B.

```
(case ⟨key⟩ ⟨case clause1⟩ ⟨case clause2⟩ ...)  syntax
```

Syntax: ⟨Key⟩ must be an expression. Each ⟨case clause⟩ must have one of the following forms:

```
((⟨datum1⟩ ...) ⟨expression1⟩ ⟨expression2⟩ ...)
(else ⟨expression1⟩ ⟨expression2⟩ ...)
```

The second form, which specifies an “else clause”, may only appear as the last <case clause>. Each <datum> is an external representation of some object. The data represented by the <datum>s need not be distinct.

Semantics: A case expression is evaluated as follows. <Key> is evaluated and its result is compared using `eqv?` (see section 11.5) against the data represented by the <datum>s of each <case clause> in turn, proceeding in order from left to right through the set of clauses. If the result of evaluating <key> is equivalent to a datum of a <case clause>, the corresponding (expression)s are evaluated from left to right and the results of the last expression in the <case clause> are returned as the results of the case expression. Otherwise, the comparison process continues. If the result of evaluating <key> is different from every datum in each set, then if there is an else clause its expressions are evaluated and the results of the last are the results of the case expression; otherwise the case expression returns unspecified values.

```
(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
```

The last <expression> of a <case clause> is in tail context if the case expression itself is; see section 11.20.

```
(and <test1> ...) syntax
```

Syntax: The <test>s must be expressions.

Semantics: If there are no <test>s, #t is returned. Otherwise, the <test> expressions are evaluated from left to right until a <test> returns #f or the last <test> is reached. In the former case, the and expression returns #f without evaluating the remaining expressions. In the latter case, the last expression is evaluated and its values are returned.

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

The and keyword could be defined in terms of if using syntax-rules (see section 11.19) as follows:

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

The last <test> expression is in tail context if the and expression itself is; see section 11.20.

```
(or <test1> ...) syntax
```

Syntax: The <test>s must be expressions.

Semantics: If there are no <test>s, #f is returned. Otherwise, the <test> expressions are evaluated from left to right until a <test> returns a true value *val* (see section 5.7) or the last <test> is reached. In the former case, the or expression returns *val* without evaluating the remaining expressions. In the latter case, the last expression is evaluated and its values are returned.

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

The or keyword could be defined in terms of if using syntax-rules (see section 11.19) as follows:

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

The last <test> expression is in tail context if the or expression itself is; see section 11.20.

11.4.6. Binding constructs

The binding constructs described in this section create local bindings for variables that are visible only in a delimited region. The syntax of the constructs `let`, `let*`, `letrec`, and `letrec*` is identical, but they differ in the regions (see section 5.2) they establish for their variable bindings and in the order in which the values for the bindings are computed. In a `let` expression, the initial values are computed before any of the variables become bound; in a `let*` expression, the bindings and evaluations are performed sequentially. In a `letrec` or `letrec*` expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions. In a `letrec` expression, the initial values are computed before being assigned to the variables; in a `letrec*`, the evaluations and assignments are performed sequentially.

In addition, the binding constructs `let-values` and `let*-values` generalize `let` and `let*` to allow multiple variables to be bound to the results of expressions that evaluate to multiple values. They are analogous to `let` and `let*` in the way they establish regions: in a `let-values` expression, the initial values are computed before any of the variables become bound; in a `let*-values` expression, the bindings are performed sequentially.

Sample definitions of all the binding forms of this section in terms of simpler forms are in appendix B.

```
(let <bindings> <body>) syntax
```

Syntax: <Bindings> must have the form

```
((<variable1> <init1>) ...),
```

where each $\langle \text{init} \rangle$ is an expression, and $\langle \text{body} \rangle$ is as described in section 11.3. Any variable must not appear more than once in the $\langle \text{variable} \rangle$ s.

Semantics: The $\langle \text{init} \rangle$ s are evaluated in the current environment (in some unspecified order), the $\langle \text{variable} \rangle$ s are bound to fresh locations holding the results, the $\langle \text{body} \rangle$ is evaluated in the extended environment, and the values of the last expression of $\langle \text{body} \rangle$ are returned. Each binding of a $\langle \text{variable} \rangle$ has $\langle \text{body} \rangle$ as its region.

```
(let ((x 2) (y 3))
  (* x y))            $\Rightarrow$  6
```

```
(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))         $\Rightarrow$  35
```

See also named `let`, section 11.16.

(let* $\langle \text{bindings} \rangle$ $\langle \text{body} \rangle$) syntax

Syntax: $\langle \text{Bindings} \rangle$ must have the form

```
(( $\langle \text{variable}_1 \rangle$   $\langle \text{init}_1 \rangle$ ) ...),
```

where each $\langle \text{init} \rangle$ is an expression, and $\langle \text{body} \rangle$ is as described in section 11.3.

Semantics: The `let*` form is similar to `let`, but the $\langle \text{init} \rangle$ s are evaluated and bindings created sequentially from left to right, with the region of each binding including the bindings to its right as well as $\langle \text{body} \rangle$. Thus the second $\langle \text{init} \rangle$ is evaluated in an environment in which the first binding is visible and initialized, and so on.

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))         $\Rightarrow$  70
```

Note: While the variables bound by a `let` expression must be distinct, the variables bound by a `let*` expression need not be distinct.

(letrec $\langle \text{bindings} \rangle$ $\langle \text{body} \rangle$) syntax

Syntax: $\langle \text{Bindings} \rangle$ must have the form

```
(( $\langle \text{variable}_1 \rangle$   $\langle \text{init}_1 \rangle$ ) ...),
```

where each $\langle \text{init} \rangle$ is an expression, and $\langle \text{body} \rangle$ is as described in section 11.3. Any variable must not appear more than once in the $\langle \text{variable} \rangle$ s.

Semantics: The $\langle \text{variable} \rangle$ s are bound to fresh locations, the $\langle \text{init} \rangle$ s are evaluated in the resulting environment (in some unspecified order), each $\langle \text{variable} \rangle$ is assigned to the result of the corresponding $\langle \text{init} \rangle$, the $\langle \text{body} \rangle$ is evaluated in the resulting environment, and the values of the last expression in $\langle \text{body} \rangle$ are returned. Each binding of a $\langle \text{variable} \rangle$ has the entire `letrec` expression as its region, making it possible to define mutually recursive procedures.

```
(letrec ((even?
  (lambda (n)
    (if (zero? n)
        #t
        (odd? (- n 1)))))
  (odd?
  (lambda (n)
    (if (zero? n)
        #f
        (even? (- n 1)))))
  (even? 88))
 $\Rightarrow$  #t
```

It should be possible to evaluate each $\langle \text{init} \rangle$ without assigning or referring to the value of any $\langle \text{variable} \rangle$. In the most common uses of `letrec`, all the $\langle \text{init} \rangle$ s are `lambda` expressions and the restriction is satisfied automatically. Another restriction is that the continuation of each $\langle \text{init} \rangle$ should not be invoked more than once.

Implementation responsibilities: Implementations must detect references to a $\langle \text{variable} \rangle$ during the evaluation of the $\langle \text{init} \rangle$ expressions (using one particular evaluation order and order of evaluating the $\langle \text{init} \rangle$ expressions). If an implementation detects such a violation of the restriction, it must raise an exception with condition type `&assertion`. Implementations may or may not detect that the continuation of each $\langle \text{init} \rangle$ is invoked more than once. However, if the implementation detects this, it must raise an exception with condition type `&assertion`.

(letrec* $\langle \text{bindings} \rangle$ $\langle \text{body} \rangle$) syntax

Syntax: $\langle \text{Bindings} \rangle$ must have the form

```
(( $\langle \text{variable}_1 \rangle$   $\langle \text{init}_1 \rangle$ ) ...),
```

where each $\langle \text{init} \rangle$ is an expression, and $\langle \text{body} \rangle$ is as described in section 11.3. Any variable must not appear more than once in the $\langle \text{variable} \rangle$ s.

Semantics: The $\langle \text{variable} \rangle$ s are bound to fresh locations, each $\langle \text{variable} \rangle$ is assigned in left-to-right order to the result of evaluating the corresponding $\langle \text{init} \rangle$, the $\langle \text{body} \rangle$ is evaluated in the resulting environment, and the values of the last expression in $\langle \text{body} \rangle$ are returned. Despite the left-to-right evaluation and assignment order, each binding of a $\langle \text{variable} \rangle$ has the entire `letrec*` expression as its region, making it possible to define mutually recursive procedures.

```
(letrec* ((p
  (lambda (x)
    (+ 1 (q (- x 1)))))
  (q
  (lambda (y)
    (if (zero? y)
        0
        (+ 1 (p (- y 1)))))
  (x (p 5))
  (y x))
  y)
 $\Rightarrow$  5
```

It must be possible to evaluate each $\langle \text{init} \rangle$ without assigning or referring to the value of the corresponding $\langle \text{variable} \rangle$ or the $\langle \text{variable} \rangle$ of any of the bindings that follow it in $\langle \text{bindings} \rangle$.

Another restriction is that the continuation of each $\langle \text{init} \rangle$ should not be invoked more than once.

Implementation responsibilities: Implementations must, during the evaluation of an $\langle \text{init} \rangle$ expression, detect references to the value of the corresponding $\langle \text{variable} \rangle$ or the $\langle \text{variable} \rangle$ of any of the bindings that follow it in $\langle \text{bindings} \rangle$. If an implementation detects such a violation of the restriction, it must raise an exception with condition type `&assertion`. Implementations may or may not detect that the continuation of each $\langle \text{init} \rangle$ is invoked more than once. However, if the implementation detects this, it must raise an exception with condition type `&assertion`.

`(let-values $\langle \text{mv-bindings} \rangle$ $\langle \text{body} \rangle$)` syntax

Syntax: $\langle \text{Mv-bindings} \rangle$ must have the form

`(($\langle \text{formals}_1 \rangle$ $\langle \text{init}_1 \rangle$) ...),`

where each $\langle \text{init} \rangle$ is an expression, and $\langle \text{body} \rangle$ is as described in section 11.3. Any variable must not appear more than once in the set of $\langle \text{formals} \rangle$.

Semantics: The $\langle \text{init} \rangle$ s are evaluated in the current environment (in some unspecified order), and the variables occurring in the $\langle \text{formals} \rangle$ are bound to fresh locations containing the values returned by the $\langle \text{init} \rangle$ s, where the $\langle \text{formals} \rangle$ are matched to the return values in the same way that the $\langle \text{formals} \rangle$ in a `lambda` expression are matched to the arguments in a procedure call. Then, the $\langle \text{body} \rangle$ is evaluated in the extended environment, and the values of the last expression of $\langle \text{body} \rangle$ are returned. Each binding of a variable has $\langle \text{body} \rangle$ as its region. If the $\langle \text{formals} \rangle$ do not match, an exception with condition type `&assertion` is raised.

```
(let-values (((a b) (values 1 2))
              ((c d) (values 3 4)))
  (list a b c d))     $\Rightarrow$  (1 2 3 4)
```

```
(let-values (((a b . c) (values 1 2 3 4)))
  (list a b c))       $\Rightarrow$  (1 2 (3 4))
```

```
(let ((a 'a) (b 'b) (x 'x) (y 'y))
      (let-values (((a b) (values x y))
                    ((x y) (values a b))))
  (list a b x y))     $\Rightarrow$  (x y a b)
```

`(let*-values $\langle \text{mv-bindings} \rangle$ $\langle \text{body} \rangle$)` syntax

Syntax: $\langle \text{Mv-bindings} \rangle$ must have the form

`(($\langle \text{formals}_1 \rangle$ $\langle \text{init}_1 \rangle$) ...),`

where each $\langle \text{init} \rangle$ is an expression, and $\langle \text{body} \rangle$ is as described in section 11.3. In each $\langle \text{formals} \rangle$, any variable must not appear more than once.

Semantics: The `let*-values` form is similar to `let-values`, but the $\langle \text{init} \rangle$ s are evaluated and bindings created sequentially from left to right, with the region of the bindings of each $\langle \text{formals} \rangle$ including the bindings to its right as well as $\langle \text{body} \rangle$. Thus the second $\langle \text{init} \rangle$ is evaluated in an environment in which the bindings of the first $\langle \text{formals} \rangle$ is visible and initialized, and so on.

```
(let ((a 'a) (b 'b) (x 'x) (y 'y))
      (let*-values (((a b) (values x y))
                    ((x y) (values a b))))
  (list a b x y))     $\Rightarrow$  (x y x y)
```

Note: While all of the variables bound by a `let-values` expression must be distinct, the variables bound by different $\langle \text{formals} \rangle$ of a `let*-values` expression need not be distinct.

11.4.7. Sequencing

`(begin $\langle \text{form} \rangle$...)` syntax

`(begin $\langle \text{expression} \rangle$ $\langle \text{expression} \rangle$...)` syntax

The $\langle \text{begin} \rangle$ keyword has two different roles, depending on its context:

- It may appear as a form in a $\langle \text{body} \rangle$ (see section 11.3), $\langle \text{library body} \rangle$ (see section 7.1), or $\langle \text{top-level body} \rangle$ (see chapter 8), or directly nested in a `begin` form that appears in a body. In this case, the `begin` form must have the shape specified in the first header line. This use of `begin` acts as a *splicing* form—the forms inside the $\langle \text{body} \rangle$ are spliced into the surrounding body, as if the `begin` wrapper were not actually present.

A `begin` form in a $\langle \text{body} \rangle$ or $\langle \text{library body} \rangle$ must be non-empty if it appears after the first $\langle \text{expression} \rangle$ within the body.

- It may appear as an ordinary expression and must have the shape specified in the second header line. In this case, the $\langle \text{expression} \rangle$ s are evaluated sequentially from left to right, and the values of the last $\langle \text{expression} \rangle$ are returned. This expression type is used to sequence side effects such as assignments or input and output.

```
(define x 0)
```

```
(begin (set! x 5)
       (+ x 1))     $\Rightarrow$  6
```

```
(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))  $\Rightarrow$  unspecified
and prints 4 plus 1 equals 5
```

11.5. Equivalence predicates

A *predicate* is a procedure that always returns a boolean value (`#t` or `#f`). An *equivalence predicate* is the computational analogue of a mathematical equivalence relation (it is symmetric, reflexive, and transitive). Of the equivalence predicates described in this section, `eq?` is the finest or most discriminating, and `equal?` is the coarsest. The `eqv?` predicate is slightly less discriminating than `eq?`.

`(eqv? obj_1 obj_2)` procedure

The `eqv?` procedure defines a useful equivalence relation on objects. Briefly, it returns `#t` if obj_1 and obj_2 should normally

be regarded as the same object and `#f` otherwise. This relation is left slightly open to interpretation, but the following partial specification of `eqv?` must hold for all implementations.

The `eqv?` procedure returns `#t` if one of the following holds:

- *Obj₁* and *obj₂* are both booleans and are the same according to the `boolean=?` procedure (section 11.8).
- *Obj₁* and *obj₂* are both symbols and are the same according to the `symbol=?` procedure (section 11.10).
- *Obj₁* and *obj₂* are both exact number objects and are numerically equal (see `=`, section 11.7).
- *Obj₁* and *obj₂* are both inexact number objects, are numerically equal (see `=`, section 11.7), and yield the same results (in the sense of `eqv?`) when passed as arguments to any other procedure that can be defined as a finite composition of Scheme's standard arithmetic procedures.
- *Obj₁* and *obj₂* are both characters and are the same character according to the `char=?` procedure (section 11.11).
- Both *obj₁* and *obj₂* are the empty list.
- *Obj₁* and *obj₂* are objects such as pairs, vectors, bytevectors (library chapter ??), strings, hashtables, records (library chapter ??), ports (library section ??), or hashtables (library chapter ??) that refer to the same locations in the store (section 5.10).
- *Obj₁* and *obj₂* are record-type descriptors that are specified to be `eqv?` in library section ??.

The `eqv?` procedure returns `#f` if one of the following holds:

- *Obj₁* and *obj₂* are of different types (section 11.1).
- *Obj₁* and *obj₂* are booleans for which the `boolean=?` procedure returns `#f`.
- *Obj₁* and *obj₂* are symbols for which the `symbol=?` procedure returns `#f`.
- One of *obj₁* and *obj₂* is an exact number object but the other is an inexact number object.
- *Obj₁* and *obj₂* are rational number objects for which the `=` procedure returns `#f`.
- *Obj₁* and *obj₂* yield different results (in the sense of `eqv?`) when passed as arguments to any other procedure that can be defined as a finite composition of Scheme's standard arithmetic procedures.
- *Obj₁* and *obj₂* are characters for which the `char=?` procedure returns `#f`.
- One of *obj₁* and *obj₂* is the empty list, but the other is not.
- *Obj₁* and *obj₂* are objects such as pairs, vectors, bytevectors (library chapter ??), strings, records (library chapter ??), ports (library section ??), or hashtables (library chapter ??) that refer to distinct locations.

- *Obj₁* and *obj₂* are pairs, vectors, strings, or records, or hashtables, where the applying the same accessor (i.e. `car`, `cdr`, `vector-ref`, `string-ref`, or record accessors) to both yields results for which `eqv?` returns `#f`.
- *Obj₁* and *obj₂* are procedures that would behave differently (return different values or have different side effects) for some arguments.

Note: The `eqv?` procedure returning `#t` when *obj₁* and *obj₂* are number objects does not imply that `=` would also return `#t` when called with *obj₁* and *obj₂* as arguments.

```
(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
```

The following examples illustrate cases in which the above rules do not fully specify the behavior of `eqv?`. All that can be said about such cases is that the value returned by `eqv?` must be a boolean.

```
(let ((p (lambda (x) x)))
  (eqv? p p))           => unspecified
(eqv? "" "")           => unspecified
(eqv? '#() '#())       => unspecified
(eqv? (lambda (x) x)
      (lambda (x) x))   => unspecified
(eqv? (lambda (x) x)
      (lambda (y) y))   => unspecified
(eqv? +nan.0 +nan.0)    => unspecified
```

The next set of examples shows the use of `eqv?` with procedures that have local state. Calls to `gen-counter` must return a distinct procedure every time, since each procedure has its own internal counter. Calls to `gen-loser` return procedures that behave equivalently when called. However, `eqv?` may not detect this equivalence.

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

Implementations may share structure between constants where appropriate. Furthermore, a constant may be copied at any time by the implementation so as to exist simultaneously in different sets of locations, as noted in section 11.4.1. Thus the value of `eqv?` on constants is sometimes implementation-dependent.

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

(`eq?` *obj*₁ *obj*₂) procedure

The `eq?` predicate is similar to `eqv?` except that in some cases it is capable of discerning distinctions finer than those detectable by `eqv?`.

The `eq?` and `eqv?` predicates are guaranteed to have the same behavior on symbols, booleans, the empty list, pairs, procedures, non-empty strings, bytevectors, and vectors, and records. The behavior of `eq?` on number objects and characters is implementation-dependent, but it always returns either `#t` or `#f`, and returns `#t` only when `eqv?` would also return `#t`. The `eq?` predicate may also behave differently from `eqv?` on empty vectors, empty bytevectors, and empty strings.

```
(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))           ⇒ unspecified
(let ((p (lambda (x) x)))
  (eq? p p))           ⇒ unspecified
```

(`equal?` *obj*₁ *obj*₂) procedure

The `equal?` predicate returns `#t` if and only if the (possibly infinite) unfoldings of its arguments into regular trees are equal as ordered trees.

The `equal?` predicate treats pairs and vectors as nodes with outgoing edges, uses `string=?` to compare strings, uses `bytevector=?` to compare bytevectors (see library chapter ??), and uses `eqv?` to compare other nodes.

```
(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? '#vu8(1 2 3 4 5)
       (u8-list->bytevector
        '(1 2 3 4 5))) ⇒ #t
(equal? (lambda (x) x)
       (lambda (y) y)) ⇒ unspecified

(let* ((x (list 'a))
       (y (list 'a))
       (z (list x y)))
  (list (equal? z (list y x))
        (equal? z (list x x))))
⇒ (#t #t)
```

Note: The `equal?` procedure must always terminate, even if its arguments contain cycles.

11.6. Procedure predicate

(`procedure?` *obj*) procedure

Returns `#t` if *obj* is a procedure, otherwise returns `#f`.

```
(procedure? car)       ⇒ #t
(procedure? 'car)      ⇒ #f
(procedure? (lambda (x) (* x x))) ⇒ #t
(procedure? ' (lambda (x) (* x x))) ⇒ #f
```

11.7. Arithmetic

The procedures described here implement arithmetic that is generic over the numerical tower described in chapter 3. The generic procedures described in this section accept both exact and inexact number objects as arguments, performing coercions and selecting the appropriate operations as determined by the numeric subtypes of their arguments.

Library chapter ?? describes libraries that define other numerical procedures.

11.7.1. Propagation of exactness and inexactness

The procedures listed below must return the mathematically correct exact result provided all their arguments are exact:

+	-	*
max	min	abs
numerator	denominator	gcd
lcm	floor	ceiling
truncate	round	rationalize
real-part	imag-part	make-rectangular

The procedures listed below must return the correct exact result provided all their arguments are exact, and no divisors are zero:

/		
div	mod	div-and-mod
div0	mod0	div0-and-mod0

Moreover, the procedure `expt` must return the correct exact result provided its first argument is an exact real number object and its second argument is an exact integer object.

The general rule is that the generic operations return the correct exact result when all of their arguments are exact and the result is mathematically well-defined, but return an inexact result when any argument is inexact. Exceptions to this rule include `sqrt`, `exp`, `log`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `expt`, `make-polar`, `magnitude`, and `angle`, which may (but are not required to) return inexact results even when given exact arguments, as indicated in the specification of these procedures.

One general exception to the rule above is that an implementation may return an exact result despite inexact arguments if that exact result would be the correct result for all possible substitutions of exact arguments for the inexact ones. An example is `(* 1.0 0)` which may return either 0 (exact) or 0.0 (inexact).

11.7.2. Representability of infinities and NaNs

The specification of the numerical operations is written as though infinities and NaNs are representable, and specifies many operations with respect to these number objects in ways that are consistent with the IEEE-754 standard for binary floating-point arithmetic. An implementation of Scheme may or may not represent infinities and NaNs; however, an implementation must raise a continuable exception with condition type `&no-infinities` or `&no-nans` (respectively; see library section ??) whenever it is unable to represent an infinity or NaN as specified. In this case, the continuation of the exception handler is the continuation that otherwise would have received the infinity or NaN value. This requirement also applies to conversions between number objects and external representations, including the reading of program source code.

11.7.3. Semantics of common operations

Some operations are the semantic basis for several arithmetic procedures. The behavior of these operations is described in this section for later reference.

Integer division

Scheme's operations for performing integer division rely on mathematical operations `div`, `mod`, `div0`, and `mod0`, that are defined as follows:

`div`, `mod`, `div0`, and `mod0` each accept two real numbers x_1 and x_2 as operands, where x_2 must be nonzero.

`div` returns an integer, and `mod` returns a real. Their results are specified by

$$\begin{aligned} x_1 \text{ div } x_2 &= n_d \\ x_1 \text{ mod } x_2 &= x_m \end{aligned}$$

where

$$\begin{aligned} x_1 &= n_d \cdot x_2 + x_m \\ 0 &\leq x_m < |x_2| \end{aligned}$$

Examples:

$$\begin{aligned} 123 \text{ div } 10 &= 12 \\ 123 \text{ mod } 10 &= 3 \\ 123 \text{ div } -10 &= -12 \\ 123 \text{ mod } -10 &= 3 \\ -123 \text{ div } 10 &= -13 \\ -123 \text{ mod } 10 &= 7 \\ -123 \text{ div } -10 &= 13 \\ -123 \text{ mod } -10 &= 7 \end{aligned}$$

`div0` and `mod0` are like `div` and `mod`, except the result of `mod0` lies within a half-open interval centered on zero. The results are specified by

$$\begin{aligned} x_1 \text{ div}_0 x_2 &= n_d \\ x_1 \text{ mod}_0 x_2 &= x_m \end{aligned}$$

where:

$$\begin{aligned} x_1 &= n_d \cdot x_2 + x_m \\ -|\frac{x_2}{2}| &\leq x_m < |\frac{x_2}{2}| \end{aligned}$$

Examples:

$$\begin{aligned} 123 \text{ div}_0 10 &= 12 \\ 123 \text{ mod}_0 10 &= 3 \\ 123 \text{ div}_0 -10 &= -12 \\ 123 \text{ mod}_0 -10 &= 3 \\ -123 \text{ div}_0 10 &= -12 \\ -123 \text{ mod}_0 10 &= -3 \\ -123 \text{ div}_0 -10 &= 12 \\ -123 \text{ mod}_0 -10 &= -3 \end{aligned}$$

Transcendental functions

In general, the transcendental functions `log`, \sin^{-1} (arcsine), \cos^{-1} (arccosine), and \tan^{-1} are multiply defined. The value of `log` z is defined to be the one whose imaginary part lies in the range from $-\pi$ (inclusive if -0.0 is distinguished, exclusive otherwise) to π (inclusive). `log` 0 is undefined.

The value of `log` z for non-real z is defined in terms of `log` on real numbers as

$$\log z = \log |z| + (\text{angle } z)i$$

where `angle` z is the angle of $z = a \cdot e^{ib}$ specified as:

$$\text{angle } z = b + 2\pi n$$

with $-\pi \leq \text{angle } z \leq \pi$ and `angle` $z = b + 2\pi n$ for some integer n .

With the one-argument version of `log` defined this way, the values of the two-argument-version of `log`, $\sin^{-1} z$, $\cos^{-1} z$, $\tan^{-1} z$,


```

(integer-valued? 3+0i)  => #t
(integer-valued? 3+0.0i) => #t
(integer-valued? 3.0)   => #t
(integer-valued? 3.0+0.0i) => #t
(integer-valued? 8/4)   => #t

```

Note: These procedures test whether a given number object can be coerced to the specified type without loss of numerical accuracy. Specifically, the behavior of these predicates differs from the behavior of `real?`, `rational?`, and `integer?` on complex number objects whose imaginary part is inexact zero.

Note: The behavior of these type predicates on inexact number objects is unreliable, because any inaccuracy may affect the result.

```

(exact? z)           procedure
(inexact? z)         procedure

```

These numerical predicates provide tests for the exactness of a quantity. For any number object, precisely one of these predicates is true.

```

(exact? 5)           => #t
(inexact? +inf.0)    => #t

```

Generic conversions

```

(inexact z)           procedure
(exact z)             procedure

```

The `inexact` procedure returns an inexact representation of `z`. If inexact number objects of the appropriate type have bounded precision, then the value returned is an inexact number object that is nearest to the argument. If an exact argument has no reasonably close inexact equivalent, an exception with condition type `&implementation-violation` may be raised.

Note: For a real number object whose magnitude is finite but so large that it has no reasonable finite approximation as an inexact number, a reasonably close inexact equivalent may be `+inf.0` or `-inf.0`. Similarly, the inexact representation of a complex number object whose components are finite may have infinite components.

The `exact` procedure returns an exact representation of `z`. The value returned is the exact number object that is numerically closest to the argument; in most cases, the result of this procedure should be numerically equal to its argument. If an inexact argument has no reasonably close exact equivalent, an exception with condition type `&implementation-violation` may be raised.

These procedures implement the natural one-to-one correspondence between exact and inexact integer objects throughout an implementation-dependent range.

The `inexact` and `exact` procedures are idempotent.

Arithmetic operations

```

(= z1 z2 z3 ...)      procedure
(< x1 x2 x3 ...)      procedure
(> x1 x2 x3 ...)      procedure
(<= x1 x2 x3 ...)     procedure
(>= x1 x2 x3 ...)     procedure

```

These procedures return `#t` if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing, and `#f` otherwise.

```

(= +inf.0 +inf.0)      => #t
(= -inf.0 +inf.0)      => #f
(= -inf.0 -inf.0)      => #t

```

For any real number object `x` that is neither infinite nor NaN:

```

(< -inf.0 x +inf.0)    => #t
(> +inf.0 x -inf.0)    => #t

```

For any number object `z`:

```

(= +nan.0 z)           => #f

```

For any real number object `x`:

```

(< +nan.0 x)           => #f
(> +nan.0 x)           => #f

```

These predicates must be transitive.

Note: The traditional implementations of these predicates in Lisp-like languages are not transitive.

Note: While it is possible to compare inexact number objects using these predicates, the results may be unreliable because a small inaccuracy may affect the result; this is especially true of `=` and `zero?` (below).

When in doubt, consult a numerical analyst.

```

(zero? z)              procedure
(positive? x)          procedure
(negative? x)          procedure
(odd? n)               procedure
(even? n)              procedure
(finite? x)            procedure
(infinite? x)          procedure
(nan? x)               procedure

```

These numerical predicates test a number object for a particular property, returning `#t` or `#f`. The `zero?` procedure tests if the number object is `=` to zero, `positive?` tests whether it is greater than zero, `negative?` tests whether it is less than zero, `odd?` tests whether it is odd, `even?` tests whether it is even, `finite?` tests whether it is not an infinity and not a NaN, `infinite?` tests whether it is an infinity, `nan?` tests whether it is a NaN.

```

(zero? +0.0)           => #t
(zero? -0.0)           => #t
(zero? +nan.0)         => #f
(positive? +inf.0)     => #t
(negative? -inf.0)     => #t
(positive? +nan.0)     => #f
(negative? +nan.0)     => #f
(finite? +inf.0)       => #f

```

```

(finite? 5)           ⇒ #t
(finite? 5.0)         ⇒ #t
(infinite? 5.0)       ⇒ #f
(infinite? +inf.0)    ⇒ #t

```

Note: As with the predicates above, the results may be unreliable because a small inaccuracy may affect the result.

```

(max x1 x2 ...)      procedure
(min x1 x2 ...)      procedure

```

These procedures return the maximum or minimum of their arguments.

```

(max 3 4)             ⇒ 4
(max 3.9 4)           ⇒ 4.0

```

For any real number object x :

```

(max +inf.0 x)        ⇒ +inf.0
(min -inf.0 x)        ⇒ -inf.0

```

Note: If any argument is inexact, then the result is also inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If `min` or `max` is used to compare number objects of mixed exactness, and the numerical value of the result cannot be represented as an inexact number object without loss of accuracy, then the procedure may raise an exception with condition type `&implementation-restriction`.

```

(+ z1 ...)           procedure
(* z1 ...)           procedure

```

These procedures return the sum or product of their arguments.

```

(+ 3 4)              ⇒ 7
(+ 3)                ⇒ 3
(+)                  ⇒ 0
(+ +inf.0 +inf.0)    ⇒ +inf.0
(+ +inf.0 -inf.0)    ⇒ +nan.0

```

```

(* 4)                ⇒ 4
(*)                  ⇒ 1
(* 5 +inf.0)         ⇒ +inf.0
(* -5 +inf.0)        ⇒ -inf.0
(* +inf.0 +inf.0)    ⇒ +inf.0
(* +inf.0 -inf.0)    ⇒ -inf.0
(* 0 +inf.0)         ⇒ 0 or +nan.0
(* 0 +nan.0)         ⇒ 0 or +nan.0
(* 1.0 0)            ⇒ 0 or 0.0

```

For any real number object x that is neither infinite nor NaN:

```

(+ +inf.0 x)         ⇒ +inf.0
(+ -inf.0 x)         ⇒ -inf.0

```

For any real number object x :

```

(+ +nan.0 x)         ⇒ +nan.0

```

For any real number object x that is not an exact 0:

```

(* +nan.0 x)         ⇒ +nan.0

```

If any of these procedures are applied to mixed non-rational real and non-real complex arguments, they either raise an exception with condition type `&implementation-restriction` or return an unspecified number object.

Implementations that distinguish `-0.0` should adopt behavior consistent with the following examples:

```

(+ 0.0 -0.0)         ⇒ 0.0
(+ -0.0 0.0)         ⇒ 0.0
(+ 0.0 0.0)          ⇒ 0.0
(+ -0.0 -0.0)        ⇒ -0.0

```

```

(- z)                procedure
(- z1 z2 ...)       procedure

```

With two or more arguments, this procedure returns the difference of its arguments, associating to the left. With one argument, however, it returns the additive inverse of its argument.

```

(- 3 4)              ⇒ -1
(- 3 4 5)            ⇒ -6
(- 3)                ⇒ -3
(- +inf.0 +inf.0)    ⇒ +nan.0

```

If this procedure is applied to mixed non-rational real and non-real complex arguments, it either raises an exception with condition type `&implementation-restriction` or returns an unspecified number object.

Implementations that distinguish `-0.0` should adopt behavior consistent with the following examples:

```

(- 0.0)              ⇒ -0.0
(- -0.0)             ⇒ 0.0
(- 0.0 -0.0)         ⇒ 0.0
(- -0.0 0.0)         ⇒ -0.0
(- 0.0 0.0)          ⇒ 0.0
(- -0.0 -0.0)        ⇒ 0.0

```

```

(/ z)                procedure
(/ z1 z2 ...)       procedure

```

If all of the arguments are exact, then the divisors must all be nonzero. With two or more arguments, this procedure returns the quotient of its arguments, associating to the left. With one argument, however, it returns the multiplicative inverse of its argument.

```

(/ 3 4 5)            ⇒ 3/20
(/ 3)                ⇒ 1/3
(/ 0.0)              ⇒ +inf.0
(/ 1.0 0)            ⇒ +inf.0
(/ -1 0.0)           ⇒ -inf.0
(/ +inf.0)           ⇒ 0.0
(/ 0 0)              ⇒ &assertion exception
(/ 3 0)              ⇒ &assertion exception
(/ 0 3.5)            ⇒ 0.0
(/ 0 0.0)            ⇒ +nan.0
(/ 0.0 0)            ⇒ +nan.0
(/ 0.0 0.0)          ⇒ +nan.0

```

If this procedure is applied to mixed non-rational real and non-real complex arguments, it either raises an exception with condition type `&implementation-restriction` or returns an unspecified number object.

(abs *x*) procedure

Returns the absolute value of its argument.

```
(abs -7)           ⇒ 7
(abs -inf.0)       ⇒ +inf.0
```

(div-and-mod *x*₁ *x*₂) procedure
 (div *x*₁ *x*₂) procedure
 (mod *x*₁ *x*₂) procedure
 (div0-and-mod0 *x*₁ *x*₂) procedure
 (div0 *x*₁ *x*₂) procedure
 (mod0 *x*₁ *x*₂) procedure

These procedures implement number-theoretic integer division and return the results of the corresponding mathematical operations specified in section 11.7.3. In each case, *x*₁ must be neither infinite nor a NaN, and *x*₂ must be nonzero; otherwise, an exception with condition type `&assertion` is raised.

```
(div x1 x2)           ⇒ x1 div x2
(mod x1 x2)           ⇒ x1 mod x2
(div-and-mod x1 x2) ⇒ x1 div x2, x1 mod x2
                               ; two return values
(div0 x1 x2)          ⇒ x1 div0 x2
(mod0 x1 x2)          ⇒ x1 mod0 x2
(div0-and-mod0 x1 x2) ⇒ x1 div0 x2, x1 mod0 x2
                               ; two return values
```

(gcd *n*₁ ...) procedure
 (lcm *n*₁ ...) procedure

These procedures return the greatest common divisor or least common multiple of their arguments. The result is always non-negative.

```
(gcd 32 -36)       ⇒ 4
(gcd)              ⇒ 0
(lcm 32 -36)       ⇒ 288
(lcm 32.0 -36)     ⇒ 288.0
(lcm)              ⇒ 1
```

(numerator *q*) procedure
 (denominator *q*) procedure

These procedures return the numerator or denominator of their argument; the result is computed as if the argument was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0 is defined to be 1.

```
(numerator (/ 6 4)) ⇒ 3
(denominator (/ 6 4)) ⇒ 2
(denominator
  (inexact (/ 6 4))) ⇒ 2.0
```

(floor *x*) procedure
 (ceiling *x*) procedure
 (truncate *x*) procedure
 (round *x*) procedure

These procedures return inexact integer objects for inexact arguments that are not infinities or NaNs, and exact integer objects for exact rational arguments. For such arguments, `floor` returns the largest integer object not larger than *x*. The `ceiling` procedure returns the smallest integer object not smaller than *x*. The `truncate` procedure returns the integer object closest to *x* whose absolute value is not larger than the absolute value of *x*. The `round` procedure returns the closest integer object to *x*, rounding to even when *x* represents a number halfway between two integers.

Note: If the argument to one of these procedures is inexact, then the result is also inexact. If an exact value is needed, the result should be passed to the `exact` procedure.

Although infinities and NaNs are not integer objects, these procedures return an infinity when given an infinity as an argument, and a NaN when given a NaN.

```
(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
```

```
(round 7/2)        ⇒ 4
(round 7)           ⇒ 7
```

```
(floor +inf.0)     ⇒ +inf.0
(ceiling -inf.0)   ⇒ -inf.0
(round +nan.0)      ⇒ +nan.0
```

(rationalize *x*₁ *x*₂) procedure

The `rationalize` procedure returns the a number object representing the *simplest* rational number differing from *x*₁ by no more than *x*₂. A rational number *r*₁ is *simpler* than another rational number *r*₂ if *r*₁ = *p*₁/*q*₁ and *r*₂ = *p*₂/*q*₂ (in lowest terms) and $|p_1| \leq |p_2|$ and $|q_1| \leq |q_2|$. Thus 3/5 is simpler than 4/7. Although not all rationals are comparable in this ordering (consider 2/7 and 3/5) any interval contains a rational number that is simpler than every other rational number in that interval (the simpler 2/5 lies between 2/7 and 3/5). Note that 0 = 0/1 is the simplest rational of all.

```
(rationalize (exact .3) 1/10)
  ⇒ 1/3
(rationalize .3 1/10)
  ⇒ #i1/3 ; approximately
```

```
(rationalize +inf.0 3) ⇒ +inf.0
(rationalize +inf.0 +inf.0) ⇒ +nan.0
(rationalize 3 +inf.0) ⇒ 0.0
```

The first two examples hold only in implementations whose inexact real number objects have sufficient precision.

(exp z)	procedure
(log z)	procedure
(log $z_1 z_2$)	procedure
(sin z)	procedure
(cos z)	procedure
(tan z)	procedure
(asin z)	procedure
(acos z)	procedure
(atan z)	procedure
(atan $x_1 x_2$)	procedure

These procedures compute the usual transcendental functions. The `exp` procedure computes the base- e exponential of z . The `log` procedure with a single argument computes the natural logarithm of z (not the base-ten logarithm); (log $z_1 z_2$) computes the base- z_2 logarithm of z_1 . The `asin`, `acos`, and `atan` procedures compute arcsine, arccosine, and arctangent, respectively. The two-argument variant of `atan` computes (angle (make-rectangular $x_2 x_1$)).

See section 11.7.3 for the underlying mathematical operations. These procedures may return inexact results even when given exact arguments.

(exp +inf.0)	\Rightarrow +inf.0
(exp -inf.0)	\Rightarrow 0.0
(log +inf.0)	\Rightarrow +inf.0
(log 0.0)	\Rightarrow -inf.0
(log 0)	\Rightarrow &assertion exception
(log -inf.0)	\Rightarrow +inf.0+3.141592653589793i
	; approximately
(atan -inf.0)	\Rightarrow -1.5707963267948965 ; approximately
(atan +inf.0)	\Rightarrow 1.5707963267948965 ; approximately
(log -1.0+0.0i)	\Rightarrow 0.0+3.141592653589793i ; approximately
(log -1.0-0.0i)	\Rightarrow 0.0-3.141592653589793i ; approximately
	; if -0.0 is distinguished

(sqrt z)	procedure
-------------	-----------

Returns the principal square root of z . For rational z , the result has either positive real part, or zero real part and non-negative imaginary part. With log defined as in section 11.7.3, the value of (sqrt z) could be expressed as $e^{\frac{\log z}{2}}$.

The `sqrt` procedure may return an inexact result even when given an exact argument.

(sqrt -5)	\Rightarrow 0.0+2.23606797749979i ; approximately
(sqrt +inf.0)	\Rightarrow +inf.0
(sqrt -inf.0)	\Rightarrow +inf.0i

(exact-integer-sqrt k)	procedure
---------------------------	-----------

The `exact-integer-sqrt` procedure returns two non-negative exact integer objects s and r where $k = s^2 + r$ and $k < (s+1)^2$.

(exact-integer-sqrt 4)	\Rightarrow 2 0	; two return values
(exact-integer-sqrt 5)	\Rightarrow 2 1	; two return values

(expt $z_1 z_2$)	procedure
-------------------	-----------

Returns z_1 raised to the power z_2 . For nonzero z_1 , this is $e^{z_2 \log z_1}$. 0.0^z is 1.0 if $z = 0.0$, and 0.0 if (real-part z) is positive. For other cases in which the first argument is zero, either an exception is raised with condition type &implementation-restriction, or an unspecified number object is returned.

For an exact real number object z_1 and an exact integer object z_2 , (expt $z_1 z_2$) must return an exact result. For all other values of z_1 and z_2 , (expt $z_1 z_2$) may return an inexact result, even when both z_1 and z_2 are exact.

(expt 5 3)	\Rightarrow 125
(expt 5 -3)	\Rightarrow 1/125
(expt 5 0)	\Rightarrow 1
(expt 0 5)	\Rightarrow 0
(expt 0 5+.0000312i)	\Rightarrow 0
(expt 0 -5)	\Rightarrow unspecified
(expt 0 -5+.0000312i)	\Rightarrow unspecified
(expt 0 0)	\Rightarrow 1
(expt 0.0 0.0)	\Rightarrow 1.0

(make-rectangular $x_1 x_2$)	procedure
(make-polar $x_3 x_4$)	procedure
(real-part z)	procedure
(imag-part z)	procedure
(magnitude z)	procedure
(angle z)	procedure

Suppose a_1, a_2, a_3 , and a_4 are real numbers, and c is a complex number such that the following holds:

$$c = a_1 + a_2 i = a_3 e^{i a_4}$$

Then, if x_1, x_2, x_3 , and x_4 are number objects representing a_1, a_2, a_3 , and a_4 , respectively, (make-rectangular $x_1 x_2$) returns c , and (make-polar $x_3 x_4$) returns c .

(make-rectangular 1.1 2.2)	\Rightarrow 1.1+2.2i ; approximately
(make-polar 1.1 2.2)	\Rightarrow 1.1@2.2 ; approximately

Conversely, if $-\pi \leq a_4 \leq \pi$, and if z is a number object representing c , then (real-part z) returns a_1 (imag-part z) returns a_2 , (magnitude z) returns a_3 , and (angle z) returns a_4 .

(real-part 1.1+2.2i)	\Rightarrow 1.1 ; approximately
(imag-part 1.1+2.2i)	\Rightarrow 2.2i ; approximately
(magnitude 1.1@2.2)	\Rightarrow 1.1 ; approximately
(angle 1.1@2.2)	\Rightarrow 2.2 ; approximately

(angle -1.0)	\Rightarrow 3.141592653589793 ; approximately
(angle -1.0+0.0i)	

```

      ⇒ 3.141592653589793 ; approximately
(angle -1.0-0.0i)
      ⇒ -3.141592653589793 ; approximately
      ; if -0.0 is distinguished
(angle +inf.0)      ⇒ 0.0
(angle -inf.0)
      ⇒ 3.141592653589793 ; approximately

```

Moreover, suppose x_1, x_2 are such that either x_1 or x_2 is an infinity, then

```

(make-rectangular  $x_1$   $x_2$ ) ⇒  $z$ 
(magnitude  $z$ )      ⇒ +inf.0

```

The `make-polar`, `magnitude`, and `angle` procedures may return inexact results even when given exact arguments.

```

(angle -1)
      ⇒ 3.141592653589793 ; approximately

```

Numerical Input and Output

```

(number->string  $z$ )           procedure
(number->string  $z$   $radix$ )      procedure
(number->string  $z$   $radix$   $precision$ ) procedure

```

Radix must be an exact integer object, either 2, 8, 10, or 16. If omitted, *radix* defaults to 10. If a *precision* is specified, then z must be an inexact complex number object, *precision* must be an exact positive integer object, and *radix* must be 10. The `number->string` procedure takes a number object and a radix and returns as a string an external representation of the given number object in the given radix such that

```

(let ((number  $z$ ) (radix  $radix$ ))
  (eqv? (string->number
         (number->string number radix)
         radix)
        number))

```

is true. If no possible result makes this expression true, an exception with condition type `&implementation-restriction` is raised.

Note: The error case can occur only when z is not a complex number object or is a complex number object with a non-rational real or imaginary part.

If a *precision* is specified, then the representations of the inexact real components of the result, unless they are infinite or NaN, specify an explicit `<mantissa width>` p , and p is the least $p \geq \text{precision}$ for which the above expression is true.

If z is inexact, the radix is 10, and the above expression and condition can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent, trailing zeroes, and mantissa width) needed to make the above expression and condition true `[?, ?]`; otherwise the format of the result is unspecified.

The result returned by `number->string` never contains an explicit radix prefix.

```

(string->number  $string$ )           procedure
(string->number  $string$   $radix$ )      procedure

```

Returns a number object with maximally precise representation expressed by the given *string*. *Radix* must be an exact integer object, either 2, 8, 10, or 16. If supplied, *radix* is a default radix that may be overridden by an explicit radix prefix in *string* (e.g., `"#o177"`). If *radix* is not supplied, then the default radix is 10. If *string* is not a syntactically valid notation for a number object or a notation for a rational number object with a zero denominator, then `string->number` returns `#f`.

```

(string->number "100") ⇒ 100
(string->number "100" 16) ⇒ 256
(string->number "1e2") ⇒ 100.0
(string->number "0/0") ⇒ #f
(string->number "+inf.0") ⇒ +inf.0
(string->number "-inf.0") ⇒ -inf.0
(string->number "+nan.0") ⇒ +nan.0

```

Note: The `string->number` procedure always returns a number object or `#f`; it never raises an exception.

11.8. Booleans

The standard boolean objects for true and false have external representations `#t` and `#f`. However, of all objects, only `#f` counts as false in conditional expressions. See section 5.7.

Note: Programmers accustomed to other dialects of Lisp should be aware that Scheme distinguishes both `#f` and the empty list from each other and from the symbol `nil`.

```

(not  $obj$ )           procedure

```

Returns `#t` if obj is `#f`, and returns `#f` otherwise.

```

(not #t)      ⇒ #f
(not 3)       ⇒ #f
(not (list 3)) ⇒ #f
(not #f)      ⇒ #t
(not '())     ⇒ #f
(not (list))  ⇒ #f
(not 'nil)    ⇒ #f

```

```

(boolean?  $obj$ )           procedure

```

Returns `#t` if obj is either `#t` or `#f` and returns `#f` otherwise.

```

(boolean? #f)      ⇒ #t
(boolean? 0)       ⇒ #f
(boolean? '())     ⇒ #f

```

```

(boolean=?  $bool_1$   $bool_2$   $bool_3$  ...) procedure

```

Returns `#t` if the booleans are the same.

11.9. Pairs and lists

A *pair* is a compound structure with two fields called the *car* and *cdr* fields (for historical reasons). Pairs are created by the procedure `cons`. The *car* and *cdr* fields are accessed by the procedures `car` and `cdr`.

Pairs are used primarily to represent lists. A list can be defined recursively as either the empty list or a pair whose *cdr* is a list. More precisely, the set of lists is defined as the smallest set X such that

- The empty list is in X .
- If *list* is in X , then any pair whose *cdr* field contains *list* is also in X .

The objects in the *car* fields of successive pairs of a list are the elements of the list. For example, a two-element list is a pair whose *car* is the first element and whose *cdr* is a pair whose *car* is the second element and whose *cdr* is the empty list. The length of a list is the number of elements, which is the same as the number of pairs.

The empty list is a special object of its own type. It is not a pair. It has no elements and its length is zero.

Note: The above definitions imply that all lists have finite length and are terminated by the empty list.

A chain of pairs not ending in the empty list is called an *improper list*. Note that an improper list is not a list. The list and dotted notations can be combined to represent improper lists:

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

is equivalent to

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

Whether a given pair is a list depends upon what is stored in the *cdr* field.

(`pair? obj`) procedure

Returns #t if *obj* is a pair, and otherwise returns #f.

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

(`cons obj1 obj2`) procedure

Returns a newly allocated pair whose *car* is *obj₁* and whose *cdr* is *obj₂*. The pair is guaranteed to be different (in the sense of `eqv?`) from every existing object.

```
(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)
```

(`car pair`) procedure

Returns the contents of the *car* field of *pair*.

```
(car '(a b c))      => a
(car '((a) b c d))  => (a)
(car '(1 . 2))       => 1
(car '())            => &assertion exception
```

(`cdr pair`) procedure

Returns the contents of the *cdr* field of *pair*.

```
(cdr '((a) b c d))  => (b c d)
(cdr '(1 . 2))       => 2
(cdr '())            => &assertion exception
```

(`caar pair`) procedure

(`cadr pair`) procedure

⋮

(`cdddar pair`) procedure

(`cddddr pair`) procedure

These procedures are compositions of `car` and `cdr`, where for example `caddr` could be defined by

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

Arbitrary compositions, up to four deep, are provided. There are twenty-eight of these procedures in all.

(`null? obj`) procedure

Returns #t if *obj* is the empty list, #f otherwise.

(`list? obj`) procedure

Returns #t if *obj* is a list, #f otherwise. By definition, all lists are chains of pairs that have finite length and are terminated by the empty list.

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

(`list obj ...`) procedure

Returns a newly allocated list of its arguments.

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

(`length list`) procedure

Returns the length of *list*.

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

(`append list ... obj`) procedure

Returns a possibly improper list consisting of the elements of the first *list* followed by the elements of the other *lists*, with *obj* as the *cdr* of the final pair. An improper list results if *obj* is not a list.

```

(append '(x) '(y))      ⇒ (x y)
(append '(a) '(b c d)) ⇒ (a b c d)
(append '(a (b)) '((c))) ⇒ (a (b) (c))
(append '(a b) '(c . d)) ⇒ (a b c . d)
(append '() 'a)         ⇒ a

```

If `append` constructs a nonempty chain of pairs, it is always newly allocated. If no pairs are allocated, *obj* is returned.

(reverse *list*) procedure

Returns a newly allocated list consisting of the elements of *list* in reverse order.

```

(reverse '(a b c))      ⇒ (c b a)
(reverse '(a (b c) d (e (f))))
  ⇒ ((e (f)) d (b c) a)

```

(list-tail *list* *k*) procedure

List should be a list of size at least *k*. The `list-tail` procedure returns the subchain of pairs of *list* obtained by omitting the first *k* elements.

```
(list-tail '(a b c d) 2) ⇒ (c d)
```

Implementation responsibilities: The implementation must check that *list* is a chain of pairs whose length is at least *k*. It should not check that it is a chain of pairs beyond this length.

(list-ref *list* *k*) procedure

List must be a list whose length is at least *k* + 1. The `list-ref` procedure returns the *k*th element of *list*.

```
(list-ref '(a b c d) 2) ⇒ c
```

Implementation responsibilities: The implementation must check that *list* is a chain of pairs whose length is at least *k* + 1. It should not check that it is a list of pairs beyond this length.

(map *proc* *list*₁ *list*₂ ...) procedure

The *lists* should all have the same length. *Proc* should accept as many arguments as there are *lists* and return a single value. *Proc* should not mutate any of the *lists*.

The `map` procedure applies *proc* element-wise to the elements of the *lists* and returns a list of the results, in order. *Proc* is always called in the same dynamic environment as `map` itself. The order in which *proc* is applied to the elements of the *lists* is unspecified. If multiple returns occur from `map`, the values returned by earlier returns are not mutated.

```

(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)

```

Implementation responsibilities: The implementation should check that the *lists* all have the same length. The implementation must check the restrictions on *proc* to the extent performed by applying it as described. An implementation may check whether *proc* is an appropriate argument before applying it.

(for-each *proc* *list*₁ *list*₂ ...) procedure

The *lists* should all have the same length. *Proc* should accept as many arguments as there are *lists*. *Proc* should not mutate any of the *lists*.

The `for-each` procedure applies *proc* element-wise to the elements of the *lists* for its side effects, in order from the first elements to the last. *Proc* is always called in the same dynamic environment as `for-each` itself. The return values of `for-each` are unspecified.

```

(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)

(for-each (lambda (x) x) '(1 2 3 4))
  ⇒ unspecified

```

```
(for-each even? '()) ⇒ unspecified
```

Implementation responsibilities: The implementation should check that the *lists* all have the same length. The implementation must check the restrictions on *proc* to the extent performed by applying it as described. An implementation may check whether *proc* is an appropriate argument before applying it.

Note: Implementations of `for-each` may or may not tail-call *proc* on the last elements.

11.10. Symbols

Symbols are objects whose usefulness rests on the fact that two symbols are identical (in the sense of `eq?`, `eqv?` and `equal?`) if and only if their names are spelled the same way. A symbol literal is formed using `quote`.

(symbol? *obj*) procedure

Returns `#t` if *obj* is a symbol, otherwise returns `#f`.

```

(symbol? 'foo)      ⇒ #t
(symbol? (car '(a b))) ⇒ #t
(symbol? "bar")     ⇒ #f
(symbol? 'nil)      ⇒ #t
(symbol? '())       ⇒ #f
(symbol? #f)        ⇒ #f

```

(symbol->string *symbol*) procedure

Returns the name of *symbol* as an immutable string.


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

```
(symbol=? symbol1 symbol2 symbol3 ...)    procedure
```

Returns #t if the symbols are the same, i.e., if their names are spelled the same.

```
(string->symbol string)    procedure
```

Returns the symbol whose name is *string*.

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

11.11. Characters

Characters are objects that represent Unicode scalar values [?].

Note: Unicode defines a standard mapping between sequences of *Unicode scalar values* (integers in the range 0 to #x10FFFF, excluding the range #xD800 to #xDFFF) in the latest version of the standard and human-readable “characters”. More precisely, Unicode distinguishes between glyphs, which are printed for humans to read, and characters, which are abstract entities that map to glyphs (sometimes in a way that’s sensitive to surrounding characters). Furthermore, different sequences of scalar values sometimes correspond to the same character. The relationships among scalar, characters, and glyphs are subtle and complex.

Despite this complexity, most things that a literate human would call a “character” can be represented by a single Unicode scalar value (although several sequences of Unicode scalar values may represent that same character). For example, Roman letters, Cyrillic letters, Hebrew consonants, and most Chinese characters fall into this category.

Unicode scalar values exclude the range #xD800 to #xDFFF, which are part of the range of Unicode *code points*. However, the Unicode code points in this range, the so-called *surrogates*, are an artifact of the UTF-16 encoding, and can only appear in specific Unicode encodings, and even then only in pairs that encode scalar values. Consequently, all characters represent code points, but the surrogate code points do not have representations as characters.

```
(char? obj)    procedure
```

Returns #t if *obj* is a character, otherwise returns #f.

```
(char->integer char)    procedure
(integer->char sv)    procedure
```

Sv must be a Unicode scalar value, i.e., a non-negative exact integer object in $[0, \text{\#xD7FF}] \cup [\text{\#xE000}, \text{\#x10FFFF}]$.

Given a character, *char->integer* returns its Unicode scalar value as an exact integer object. For a Unicode scalar value *sv*, *integer->char* returns its associated character.

```
(integer->char 32)    ⇒ #\space
(char->integer (integer->char 5000))
    ⇒ 5000
(integer->char #\xD800) ⇒ &assertion exception
```

```
(char=? char1 char2 char3 ...)    procedure
(char<? char1 char2 char3 ...)    procedure
(char>? char1 char2 char3 ...)    procedure
(char<=? char1 char2 char3 ...)    procedure
(char>=? char1 char2 char3 ...)    procedure
```

These procedures impose a total ordering on the set of characters according to their Unicode scalar values.

```
(char<? #\z #\B)    ⇒ #t
(char<? #\z #\Z)    ⇒ #f
```

11.12. Strings

Strings are sequences of characters.

The *length* of a string is the number of characters that it contains. This number is fixed when the string is created. The *valid indices* of a string are the integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

```
(string? obj)    procedure
```

Returns #t if *obj* is a string, otherwise returns #f.

```
(make-string k)    procedure
(make-string k char)    procedure
```

Returns a newly allocated string of length *k*. If *char* is given, then all elements of the string are initialized to *char*, otherwise the contents of the *string* are unspecified.

```
(string char ...)    procedure
```

Returns a newly allocated string composed of the arguments.

```
(string-length string)    procedure
```

Returns the number of characters in the given *string* as an exact integer object.

```
(string-ref string k)    procedure
```

K must be a valid index of *string*. The *string-ref* procedure returns character *k* of *string* using zero-origin indexing.

Note: Implementors should make *string-ref* run in constant time.

(string=? *string*₁ *string*₂ *string*₃ ...) procedure

Returns #t if the strings are the same length and contain the same characters in the same positions. Otherwise, the string=? procedure returns #f.

```
(string=? "Straße" "Strasse")
⇒ #f
```

(string<? *string*₁ *string*₂ *string*₃ ...) procedure
 (string>? *string*₁ *string*₂ *string*₃ ...) procedure
 (string<=? *string*₁ *string*₂ *string*₃ ...) procedure
 (string>=? *string*₁ *string*₂ *string*₃ ...) procedure

These procedures are the lexicographic extensions to strings of the corresponding orderings on characters. For example, string<? is the lexicographic ordering on strings induced by the ordering char<? on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string.

```
(string<? "z" "ß") ⇒ #t
(string<? "z" "zz") ⇒ #t
(string<? "z" "Z") ⇒ #f
```

(substring *string* *start* *end*) procedure

String must be a string, and *start* and *end* must be exact integer objects satisfying

$$0 \leq \text{start} \leq \text{end} \leq (\text{string-length } \textit{string}).$$

The substring procedure returns a newly allocated string formed from the characters of *string* beginning with index *start* (inclusive) and ending with index *end* (exclusive).

(string-append *string* ...) procedure

Returns a newly allocated string whose characters form the concatenation of the given strings.

(string->list *string*) procedure
 (list->string *list*) procedure

List must be a list of characters. The string->list procedure returns a newly allocated list of the characters that make up the given string. The list->string procedure returns a newly allocated string formed from the characters in *list*. The string->list and list->string procedures are inverses so far as equal? is concerned.

(string-for-each *proc* *string*₁ *string*₂ ...) procedure

The *strings* must all have the same length. *Proc* should accept as many arguments as there are *strings*. The string-for-each procedure applies *proc* element-wise to the characters of the *strings* for its side effects, in order from the first characters to the last. *Proc* is always called in the same dynamic environment as string-for-each itself. The return values of string-for-each are unspecified.

Analogous to for-each.

Implementation responsibilities: The implementation must check the restrictions on *proc* to the extent performed by applying it as described. An implementation may check whether *proc* is an appropriate argument before applying it.

(string-copy *string*) procedure

Returns a newly allocated copy of the given *string*.

11.13. Vectors

Vectors are heterogeneous structures whose elements are indexed by integers. A vector typically occupies less space than a list of the same length, and the average time needed to access a randomly chosen element is typically less for the vector than for the list.

The *length* of a vector is the number of elements that it contains. This number is a non-negative integer that is fixed when the vector is created. The *valid indices* of a vector are the exact non-negative integer objects less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

Like list constants, vector constants must be quoted:

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

(vector? *obj*) procedure

Returns #t if *obj* is a vector. Otherwise the procedure returns #f.

(make-vector *k*) procedure
 (make-vector *k* *fill*) procedure

Returns a newly allocated vector of *k* elements. If a second argument is given, then each element is initialized to *fill*. Otherwise the initial contents of each element is unspecified.

(vector *obj* ...) procedure

Returns a newly allocated vector whose elements contain the given arguments. Analogous to list.

```
(vector 'a 'b 'c) ⇒ #(a b c)
```

(vector-length *vector*) procedure

Returns the number of elements in *vector* as an exact integer object.

(vector-ref *vector* *k*) procedure

K must be a valid index of *vector*. The vector-ref procedure returns the contents of element *k* of *vector*.

```
(vector-ref '#(1 1 2 3 5 8 13 21) 5)
⇒ 8
```

(vector-set! *vector* *k* *obj*) procedure

K must be a valid index of *vector*. The `vector-set!` procedure stores *obj* in element *k* of *vector*, and returns unspecified values. Passing an immutable vector to `vector-set!` should cause an exception with condition type `&assertion` to be raised.

```
(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")
⇒ unspecified
; constant vector
; should raise &assertion exception
```

(vector->list *vector*) procedure
(list->vector *list*) procedure

The `vector->list` procedure returns a newly allocated list of the objects contained in the elements of *vector*. The `list->vector` procedure returns a newly created vector initialized to the elements of the list *list*.

```
(vector->list '#(dah dah didah))
⇒ (dah dah didah)
(list->vector '(dididit dah))
⇒ #(dididit dah)
```

(vector-fill! *vector* *fill*) procedure

Stores *fill* in every element of *vector* and returns unspecified values.

(vector-map *proc* *vector*₁ *vector*₂ ...) procedure

The *vectors* must all have the same length. *Proc* should accept as many arguments as there are *vectors* and return a single value.

The `vector-map` procedure applies *proc* element-wise to the elements of the *vectors* and returns a vector of the results, in order. *Proc* is always called in the same dynamic environment as `vector-map` itself. The order in which *proc* is applied to the elements of the *vectors* is unspecified. If multiple returns occur from `vector-map`, the return values returned by earlier returns are not mutated.

Analogous to `map`.

Implementation responsibilities: The implementation must check the restrictions on *proc* to the extent performed by applying it as described. An implementation may check whether *proc* is an appropriate argument before applying it.

(vector-for-each *proc* *vector*₁ *vector*₂ ...) procedure

The *vectors* must all have the same length. *Proc* should accept as many arguments as there are *vectors*. The `vector-for-each` procedure applies *proc* element-wise to the elements of the *vectors* for its side effects, in order from the first elements to the last. *Proc* is always called in the same dynamic environment as `vector-for-each` itself. The return values of `vector-for-each` are unspecified.

Analogous to `for-each`.

Implementation responsibilities: The implementation must check the restrictions on *proc* to the extent performed by applying it as described. An implementation may check whether *proc* is an appropriate argument before applying it.

11.14. Errors and violations

(error *who* *message* *irritant*₁ ...) procedure
(assertion-violation *who* *message* *irritant*₁ ...) procedure

Who must be a string or a symbol or `#f`. *Message* must be a string. The *irritants* are arbitrary objects.

These procedures raise an exception. The `error` procedure should be called when an error has occurred, typically caused by something that has gone wrong in the interaction of the program with the external world or the user. The `assertion-violation` procedure should be called when an invalid call to a procedure was made, either passing an invalid number of arguments, or passing an argument that it is not specified to handle.

The *who* argument should describe the procedure or operation that detected the exception. The *message* argument should describe the exceptional situation. The *irritants* should be the arguments to the operation that detected the operation.

The condition object provided with the exception (see library chapter ??) has the following condition types:

- If *who* is not `#f`, the condition has condition type `&who`, with *who* as the value of its field. In that case, *who* should be the name of the procedure or entity that detected the exception. If it is `#f`, the condition does not have condition type `&who`.
- The condition has condition type `&message`, with *message* as the value of its field.
- The condition has condition type `&irritants`, and its field has as its value a list of the *irritants*.

Moreover, the condition created by `error` has condition type `&error`, and the condition created by `assertion-violation` has condition type `&assertion`.

```
(define (fac n)
  (if (not (integer-valued? n))
      (assertion-violation
        'fac "non-integral argument" n)
      (if (negative? n)
          (assertion-violation
            'fac "negative argument" n)
          (letrec
            ((loop (lambda (n r)
                      (if (zero? n)
                          r
                          (loop (- n 1) (* r n))))))
              (loop n 1))))))
```

```
(fac 5)           ⇒ 120
(fac 4.5)         ⇒ &assertion exception
(fac -3)          ⇒ &assertion exception
```

```
(assert <expression>)          syntax
```

An `assert` form is evaluated by evaluating `<expression>`. If `<expression>` returns a true value, that value is returned from the `assert` expression. If `<expression>` returns `#f`, an exception with condition types `&assertion` and `&message` is raised. The message provided in the condition object is implementation-dependent.

Note: Implementations should exploit the fact that `assert` is syntax to provide as much information as possible about the location of the assertion failure.

11.15. Control features

This chapter describes various primitive procedures which control the flow of program execution in special ways.

```
(apply proc arg1 ... rest-args)          procedure
```

Rest-args must be a list. *Proc* should accept *n* arguments, where *n* is number of *args* plus the length of *rest-args*. The `apply` procedure calls *proc* with the elements of the list `(list arg1 ...) rest-args` as the actual arguments.

If a call to `apply` occurs in a tail context, the call to *proc* is also in a tail context.

```
(apply + (list 3 4)) ⇒ 7
```

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

```
((compose sqrt *) 12 75) ⇒ 30
```

```
(call-with-current-continuation proc) procedure
(call/cc proc)           procedure
```

Proc should accept one argument. The procedure `call-with-current-continuation` (which is the same as the procedure `call/cc`) packages the current continuation as an “escape procedure” and passes it as an argument to *proc*. The escape procedure is a Scheme procedure that, if it is later called, will abandon whatever continuation is in effect at that later time and will instead reinstate the continuation that was in effect when the escape procedure was created. Calling the escape procedure may cause the invocation of *before* and *after* procedures installed using `dynamic-wind`.

The escape procedure accepts the same number of arguments as the continuation of the original call to `call-with-current-continuation`.

The escape procedure that is passed to *proc* has unlimited extent just like any other procedure in Scheme. It may be stored in

variables or data structures and may be called as many times as desired.

If a call to `call-with-current-continuation` occurs in a tail context, the call to *proc* is also in a tail context.

The following examples show only some ways in which `call-with-current-continuation` is used. If all real uses were as simple as these examples, there would be no need for a procedure with the power of `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
(call-with-current-continuation procedure?)
⇒ #t
```

Note: Calling an escape procedure reenters the dynamic extent of the call to `call-with-current-continuation`, and thus restores its dynamic environment; see section 5.12.

```
(values obj ...)          procedure
```

Delivers all of its arguments to its continuation. The values procedure might be defined as follows:

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

The continuations of all non-final expressions within a sequence of expressions, such as in `lambda`, `begin`, `let`, `let*`, `letrec`, `letrec*`, `let-values`, `let*-values`, `case`, and `cond` forms, usually take an arbitrary number of values.

Except for these and the continuations created by `call-with-values`, `let-values`, and `let*-values`, continuations implicitly accepting a single value, such as the continuations of `<operator>` and `<operand>`s of procedure calls or the `<test>` expressions in conditionals, take exactly one value. The effect of passing an inappropriate number of values to such a continuation is undefined.

```
(call-with-values producer consumer) procedure
```

Producer must be a procedure and should accept zero arguments. *Consumer* must be a procedure and should accept as many values

as *producer* returns. The `call-with-values` procedure calls *producer* with no arguments and a continuation that, when passed some values, calls the *consumer* procedure with those values as arguments. The continuation for the call to *consumer* is the continuation of the call to `call-with-values`.

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

```
(call-with-values * -) ⇒ -1
```

If a call to `call-with-values` occurs in a tail context, the call to *consumer* is also in a tail context.

Implementation responsibilities: After *producer* returns, the implementation must check that *consumer* accepts as many values as *consumer* has returned.

(dynamic-wind *before* *thunk* *after*) procedure

Before, *thunk*, and *after* must be procedures, and each should accept zero arguments. These procedures may return any number of values. The `dynamic-wind` procedure calls *thunk* without arguments, returning the results of this call. Moreover, `dynamic-wind` calls *before* without arguments whenever the dynamic extent of the call to *thunk* is entered, and *after* without arguments whenever the dynamic extent of the call to *thunk* is exited. Thus, in the absence of calls to escape procedures created by `call-with-current-continuation`, `dynamic-wind` calls *before*, *thunk*, and *after*, in that order.

While the calls to *before* and *after* are not considered to be within the dynamic extent of the call to *thunk*, calls to the *before* and *after* procedures of any other calls to `dynamic-wind` that occur within the dynamic extent of the call to *thunk* are considered to be within the dynamic extent of the call to *thunk*.

More precisely, an escape procedure transfers control out of the dynamic extent of a set of zero or more active `dynamic-wind` calls *x* ... and transfer control into the dynamic extent of a set of zero or more active `dynamic-wind` calls *y* ... It leaves the dynamic extent of the most recent *x* and calls without arguments the corresponding *after* procedure. If the *after* procedure returns, the escape procedure proceeds to the next most recent *x*, and so on. Once each *x* has been handled in this manner, the escape procedure calls without arguments the *before* procedure corresponding to the least recent *y*. If the *before* procedure returns, the escape procedure reenters the dynamic extent of the least recent *y* and proceeds with the next least recent *y*, and so on. Once each *y* has been handled in this manner, control is transferred to the continuation packaged in the escape procedure.

Implementation responsibilities: The implementation must check the restrictions on *thunk* and *after* only if they are actually called.

```
(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)
```

```
(let ((n 0))
  (call-with-current-continuation
    (lambda (k)
      (dynamic-wind
        (lambda ()
          (set! n (+ n 1))
          (k))
        (lambda ()
          (set! n (+ n 2)))
        (lambda ()
          (set! n (+ n 4))))))
    n)
  ⇒ 1
```

```
(let ((n 0))
  (call-with-current-continuation
    (lambda (k)
      (dynamic-wind
        values
        (lambda ()
          (dynamic-wind
            values
            (lambda ()
              (set! n (+ n 1))
              (k))
            (lambda ()
              (set! n (+ n 2))
              (k))))
        (lambda ()
          (set! n (+ n 4))))))
    n)
  ⇒ 7
```

Note: Entering a dynamic extent restores its dynamic environment; see section 5.12.

11.16. Iteration

(let <variable> <bindings> <body>) syntax

“Named `let`” is a variant on the syntax of `let` that provides a general looping construct and may also be used to express recursion. It has the same syntax and semantics as ordinary `let` except that <variable> is bound within <body> to a procedure whose parameters are the bound variables and whose body is <body>. Thus the execution of <body> may be repeated by invoking the procedure named by <variable>.

```
(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))

```

11.17. Quasiquotation

(quasiquote *<qq template>*) syntax
 unquote auxiliary syntax
 unquote-splicing auxiliary syntax

“Backquote” or “quasiquote” expressions are useful for constructing a list or vector structure when some but not all of the desired structure is known in advance.

Syntax: *<Qq template>* should be as specified by the grammar at the end of this entry.

Semantics: If no unquote or unquote-splicing forms appear within the *<qq template>*, the result of evaluating (quasiquote *<qq template>*) is equivalent to the result of evaluating (quote *<qq template>*).

If an (unquote *<expression>* ...) form appears inside a *<qq template>*, however, the *<expression>*s are evaluated (“unquoted”) and their results are inserted into the structure instead of the unquote form.

If an (unquote-splicing *<expression>* ...) form appears inside a *<qq template>*, then the *<expression>*s must evaluate to lists; the opening and closing parentheses of the lists are then “stripped away” and the elements of the lists are inserted in place of the unquote-splicing form.

Any unquote-splicing or multi-operand unquote form must appear only within a list or vector *<qq template>*.

As noted in section 4.3.5, (quasiquote *<qq template>*) may be abbreviated \square *<qq template>*, (unquote *<expression>*) may be abbreviated ,*<expression>*, and (unquote-splicing *<expression>*) may be abbreviated ,@*<expression>*.

```

( (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)
( (let ((name 'foo))
  ( (unquote name name name) ) ) )
  ⇒ (foo foo foo)
( (let ((name '(foo)))
  ( (unquote-splicing name name name) ) ) )
  ⇒ (foo foo foo)
( (let ((q ' ( (append x y) (sqrt 9) ) ) )
  ( (foo „q) ) ) )

```

```

⇒ ( (foo
  (unquote (append x y) (sqrt 9) ) ) )
( (let ((x '(2 3))
  (y '(4 5)))
  ( (foo (unquote (append x y) (sqrt 9) ) ) ) ) )
  ⇒ (foo (2 3 4 5) 3)

```

Quasiquote forms may be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost quasiquote. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation.

```

( (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) )

```

A quasiquote expression may return either fresh, mutable objects or literal structure for any structure that is constructed at run time during the evaluation of the expression. Portions that do not need to be rebuilt are always literal. Thus,

```

( (let ((a 3)) ( ( (1 2) , a , 4 , 'five 6) ) ) )

```

may be equivalent to either of the following expressions:

```

' ( (1 2) 3 4 five 6 )
( (let ((a 3))
  (cons ' (1 2)
    (cons a (cons 4 (cons 'five ' (6) ) ) ) ) ) ) )

```

However, it is not equivalent to this expression:

```

( (let ((a 3)) (list (list 1 2) a 4 'five 6) ) )

```

It is a syntax violation if any of the identifiers quasiquote, unquote, or unquote-splicing appear in positions within a *<qq template>* otherwise than as described above.

The following grammar for quasiquote expressions is not context-free. It is presented as a recipe for generating an infinite number of production rules. Imagine a copy of the following rules for $D = 1, 2, 3, \dots$. D keeps track of the nesting depth.

```

<qq template> → <qq template 1>
<qq template 0> → <expression>
<quasiquotation D> → (quasiquote <qq template D>)
<qq template D> → <lexeme 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>)
  | <quasiquotation D + 1>
<vector qq template D> → # (<qq template or splice D>*)
<unquotation D> → (unquote <qq template D - 1>)
<qq template or splice D> → <qq template D>
  | <splicing unquotation D>
<splicing unquotation D> →
  (unquote-splicing <qq template D - 1>*)
  | (unquote <qq template D - 1>*)

```

In \langle quasiquote \rangle s, a \langle list qq template D \rangle can sometimes be confused with either an \langle unquote \rangle D or a \langle splicing unquote \rangle D . The interpretation as an \langle unquote \rangle or \langle splicing unquote \rangle D takes precedence.

11.18. Binding constructs for syntactic keywords

The `let-syntax` and `letrec-syntax` forms bind keywords. Like a `begin` form, a `let-syntax` or `letrec-syntax` form may appear in a definition context, in which case it is treated as a definition, and the forms in the body must also be definitions. A `let-syntax` or `letrec-syntax` form may also appear in an expression context, in which case the forms within their bodies must be expressions.

`(let-syntax \langle bindings \rangle \langle form \rangle ...)` syntax

Syntax: \langle Bindings \rangle must have the form

`((\langle keyword \rangle \langle expression \rangle) ...)`

Each \langle keyword \rangle is an identifier, and each \langle expression \rangle is an expression that evaluates, at macro-expansion time, to a *transformer*. Transformers may be created by `syntax-rules` or `identifier-syntax` (see section 11.19) or by one of the other mechanisms described in library chapter ?? . It is a syntax violation for \langle keyword \rangle to appear more than once in the list of keywords being bound.

Semantics: The \langle form \rangle s are expanded in the syntactic environment obtained by extending the syntactic environment of the `let-syntax` form with macros whose keywords are the \langle keyword \rangle s, bound to the specified transformers. Each binding of a \langle keyword \rangle has the \langle form \rangle s as its region.

The \langle form \rangle s of a `let-syntax` form are treated, whether in definition or expression context, as if wrapped in an implicit `begin`; see section 11.4.7. Thus definitions in the result of expanding the \langle form \rangle s have the same region as any definition appearing in place of the `let-syntax` form would have.

Implementation responsibilities: The implementation should detect if the value of \langle expression \rangle cannot possibly be a transformer.

```
(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

(let ()
  (let-syntax
    ((def (syntax-rules ()
      ((def stuff ...) (define stuff ...))))
    (def foo 42)))
```

`foo)` ⇒ 42

```
(let ()
  (let-syntax ()
    5))
⇒ 5
```

`(letrec-syntax \langle bindings \rangle \langle form \rangle ...)` syntax

Syntax: Same as for `let-syntax`.

Semantics: The \langle form \rangle s are expanded in the syntactic environment obtained by extending the syntactic environment of the `letrec-syntax` form with macros whose keywords are the \langle keyword \rangle s, bound to the specified transformers. Each binding of a \langle keyword \rangle has the \langle bindings \rangle as well as the \langle form \rangle s within its region, so the transformers can transcribe forms into uses of the macros introduced by the `letrec-syntax` form.

The \langle form \rangle s of a `letrec-syntax` form are treated, whether in definition or expression context, as if wrapped in an implicit `begin`; see section 11.4.7. Thus definitions in the result of expanding the \langle form \rangle s have the same region as any definition appearing in place of the `letrec-syntax` form would have.

Implementation responsibilities: The implementation should detect if the value of \langle expression \rangle cannot possibly be a transformer.

```
(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
```

The following example highlights how `let-syntax` and `letrec-syntax` differ.

```
(let ((f (lambda (x) (+ x 1))))
  (let-syntax ((f (syntax-rules ()
    ((f x) x)))
    (g (syntax-rules ()
      ((g x) (f x)))))
    (list (f 1) (g 1))))
⇒ (1 2)

(let ((f (lambda (x) (+ x 1))))
  (letrec-syntax ((f (syntax-rules ()
    ((f x) x)))
    (g (syntax-rules ()
      ((g x) (f x)))))
    (list (f 1) (g 1))))
⇒ (1 1)
```

The two expressions are identical except that the `let-syntax` form in the first expression is a `letrec-syntax` form in the second. In the first expression, the `f` occurring in `g` refers to the `let`-bound variable `f`, whereas in the second it refers to the keyword `f` whose binding is established by the `letrec-syntax` form.

11.19. Macro transformers

```
(syntax-rules ((literal) ...) (syntax rule) ...)
                                     syntax (expand)
_                                     auxiliary syntax (expand)
...                                 auxiliary syntax (expand)
```

Syntax: Each `<literal>` must be an identifier. Each `<syntax rule>` must have the following form:

```
((srpattern) <template>)
```

An `<srpattern>` is a restricted form of `<pattern>`, namely, a nonempty `<pattern>` in one of four parenthesized forms below whose first subform is an identifier or an underscore `_`. A `<pattern>` is an identifier, constant, or one of the following.

```
((pattern) ...)
((pattern) (pattern) ... . (pattern))
((pattern) ... (pattern) (ellipsis) (pattern) ...)
((pattern) ... (pattern) (ellipsis) (pattern) ... . (pattern))
# ((pattern) ...)
# ((pattern) ... (pattern) (ellipsis) (pattern) ...)
```

An `<ellipsis>` is the identifier “...” (three periods).

A `<template>` is a pattern variable, an identifier that is not a pattern variable, a pattern datum, or one of the following.

```
((subtemplate) ...)
((subtemplate) ... . (template))
# ((subtemplate) ...)
```

A `<subtemplate>` is a `<template>` followed by zero or more ellipses.

Semantics: An instance of `syntax-rules` evaluates, at macro-expansion time, to a new macro transformer by specifying a sequence of hygienic rewrite rules. A use of a macro whose keyword is associated with a transformer specified by `syntax-rules` is matched against the patterns contained in the `<syntax rule>`s, beginning with the leftmost `<syntax rule>`. When a match is found, the macro use is transcribed hygienically according to the template. It is a syntax violation when no match is found.

An identifier appearing within a `<pattern>` may be an underscore (`_`), a literal identifier listed in the list of literals (`<literal> ...`), or an ellipsis (`...`). All other identifiers appearing within a `<pattern>` are *pattern variables*. It is a syntax violation if an ellipsis or underscore appears in `<literal> ...`.

While the first subform of `<srpattern>` may be an identifier, the identifier is not involved in the matching and is not considered a pattern variable or literal identifier.

Pattern variables match arbitrary input subforms and are used to refer to elements of the input. It is a syntax violation if the same pattern variable appears more than once in a `<pattern>`.

Underscores also match arbitrary input subforms but are not pattern variables and so cannot be used to refer to those elements. Multiple underscores may appear in a `<pattern>`.

A literal identifier matches an input subform if and only if the input subform is an identifier and either both its occurrence in the input expression and its occurrence in the list of literals have the same lexical binding, or the two identifiers have the same name and both have no lexical binding.

A subpattern followed by an ellipsis can match zero or more elements of the input.

More formally, an input form F matches a pattern P if and only if one of the following holds:

- P is an underscore (`_`).
- P is a pattern variable.
- P is a literal identifier and F is an identifier such that both P and F would refer to the same binding if both were to appear in the output of the macro outside of any bindings inserted into the output of the macro. (If neither of two like-named identifiers refers to any binding, i.e., both are undefined, they are considered to refer to the same binding.)
- P is of the form $(P_1 \dots P_n)$ and F is a list of n elements that match P_1 through P_n .
- P is of the form $(P_1 \dots P_n . P_x)$ and F is a list or improper list of n or more elements whose first n elements match P_1 through P_n and whose n th cdr matches P_x .
- P is of the form $(P_1 \dots P_k P_e \langle\text{ellipsis}\rangle P_{m+1} \dots P_n)$, where `<ellipsis>` is the identifier “...” and F is a list of n elements whose first k elements match P_1 through P_k , whose next $m - k$ elements each match P_e , and whose remaining $n - m$ elements match P_{m+1} through P_n .
- P is of the form $(P_1 \dots P_k P_e \langle\text{ellipsis}\rangle P_{m+1} \dots P_n . P_x)$, where `<ellipsis>` is the identifier “...” and F is a list or improper list of n elements whose first k elements match P_1 through P_k , whose next $m - k$ elements each match P_e , whose next $n - m$ elements match P_{m+1} through P_n , and whose n th and final cdr matches P_x .
- P is of the form $\#(P_1 \dots P_n)$ and F is a vector of n elements that match P_1 through P_n .
- P is of the form $\#(P_1 \dots P_k P_e \langle\text{ellipsis}\rangle P_{m+1} \dots P_n)$, where `<ellipsis>` is the identifier “...” and F is a vector of n or more elements whose first k elements match P_1 through P_k , whose next $m - k$ elements each match P_e , and whose remaining $n - m$ elements match P_{m+1} through P_n .
- P is a pattern datum (any nonlist, nonvector, nonsymbol datum) and F is equal to P in the sense of the `equal?` procedure.

When a macro use is transcribed according to the template of the matching `<syntax rule>`, pattern variables that occur in the template are replaced by the subforms they match in the input.

Pattern data and identifiers that are not pattern variables or ellipses are copied into the output. A subtemplate followed by an ellipsis expands into zero or more occurrences of the subtemplate. Pattern variables that occur in subpatterns followed by one or more ellipses may occur only in subtemplates that are followed by (at least) as many ellipses. These pattern variables are replaced in the output by the input subforms to which they are bound, distributed as specified. If a pattern variable is followed by more ellipses in the subtemplate than in the associated subpattern, the input form is replicated as necessary. The subtemplate must contain at least one pattern variable from a subpattern followed by an ellipsis, and for at least one such pattern variable, the subtemplate must be followed by exactly as many ellipses as the subpattern in which the pattern variable appears. (Otherwise, the expander would not be able to determine how many times the subform should be repeated in the output.) It is a syntax violation if the constraints of this paragraph are not met.

A template of the form $\langle\langle\text{ellipsis}\rangle\ \langle\text{template}\rangle\rangle$ is identical to $\langle\text{template}\rangle$, except that ellipses within the template have no special meaning. That is, any ellipses contained within $\langle\text{template}\rangle$ are treated as ordinary identifiers. In particular, the template $(\dots\ \dots)$ produces a single ellipsis, \dots . This allows syntactic abstractions to expand into forms containing ellipses.

```
(define-syntax be-like-begin
  (syntax-rules ()
    ((be-like-begin name)
     (define-syntax name
       (syntax-rules ()
         ((name expr (... ...))
          (begin expr (... ...)))))))

(be-like-begin sequence)
(sequence 1 2 3 4)       $\Rightarrow$  4
```

As an example for hygienic use of auxiliary identifier, if `let` and `cond` are defined as in section 11.4.6 and appendix B then they are hygienic (as required) and the following is not an error.

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

The macro transformer for `cond` recognizes `=>` as a local variable, and hence an expression, and not as the identifier `=>`, which the macro transformer treats as a syntactic keyword. Thus the example expands into

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

instead of

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

which would result in an assertion violation.

```
(identifier-syntax <template>)    syntax (expand)
(identifier-syntax
  (<id1> <template1>))
  ((set! <id2> <pattern>)
   <template2>))
```

`set!`

auxiliary syntax (`expand`)

Syntax: The $\langle\text{id}\rangle$ s must be identifiers. The $\langle\text{template}\rangle$ s must be as for `syntax-rules`.

Semantics: When a keyword is bound to a transformer produced by the first form of `identifier-syntax`, references to the keyword within the scope of the binding are replaced by $\langle\text{template}_1\rangle$. Uses of `set!` with the identifier are replaced by $\langle\text{template}_2\rangle$.

```
(define p (cons 4 5))
(define-syntax p.car (identifier-syntax (car p)))
p.car       $\Rightarrow$  4
(set! p.car 15)       $\Rightarrow$  &syntax exception
```

The second, more general, form of `identifier-syntax` permits the transformer to determine what happens when `set!` is used. In this case, uses of the identifier by itself are replaced by $\langle\text{template}_1\rangle$, and uses of `set!` with the identifier are replaced by $\langle\text{template}_2\rangle$.

```
(define p (cons 4 5))
(define-syntax p.car
  (identifier-syntax
    (_ (car p))
    ((set! _ e) (set-car! p e))))
(set! p.car 15)
p.car       $\Rightarrow$  15
p           $\Rightarrow$  (15 5)
```

11.20. Tail calls and tail contexts

A *tail call* is a procedure call that occurs in a *tail context*. Tail contexts are defined inductively. Note that a tail context is always determined with respect to a particular lambda expression.

- The last expression within the body of a lambda expression, shown as $\langle\text{tail expression}\rangle$ below, occurs in a tail context.

```
(lambda <formals>
  <definition>*
  <expression>* <tail expression>)
```

- If one of the following expressions is in a tail context, then the subexpressions shown as $\langle\text{tail expression}\rangle$ are in a tail context. These were derived from specifications of the syntax of the forms described in this chapter by replacing some occurrences of $\langle\text{expression}\rangle$ with $\langle\text{tail expression}\rangle$. Only those rules that contain tail contexts are shown here.

```
(if <expression> <tail expression> <tail expression>)
(if <expression> <tail expression>)
```

```
(cond <cond clause>+ )
(cond <cond clause>* (else <tail sequence>))
```

```
(case <expression>
  <case clause>+ )
(case <expression>
  <case clause>*
  (else <tail sequence>))
```

```

(and <expression>* <tail expression>)
(or <expression>* <tail expression>)

(let <bindings> <tail body>)
(let <variable> <bindings> <tail body>)
(let* <bindings> <tail body>)
(letrec* <bindings> <tail body>)
(letrec <bindings> <tail body>)
(let-values <mv-bindings> <tail body>)
(let*-values <mv-bindings> <tail body>)

(let-syntax <bindings> <tail body>)
(letrec-syntax <bindings> <tail body>)

(begin <tail sequence>)

```

A *<cond clause>* is

```
(<test> <tail sequence>),
```

a *<case clause>* is

```
((<datum>*) <tail sequence>),
```

a *<tail body>* is

```
<definition>* <tail sequence>,
```

and a *<tail sequence>* is

```
<expression>* <tail expression>.
```

- If a *cond* expression is in a tail context, and has a clause of the form $(\langle \text{expression}_1 \rangle \Rightarrow \langle \text{expression}_2 \rangle)$ then the (implied) call to the procedure that results from the evaluation of $\langle \text{expression}_2 \rangle$ is in a tail context. $\langle \text{Expression}_2 \rangle$ itself is not in a tail context.

Certain built-in procedures must also perform tail calls. The first argument passed to *apply* and to *call-with-current-continuation*, and the second argument passed to *call-with-values*, must be called via a tail call.

In the following example the only tail call is the call to *f*. None of the calls to *g* or *h* are tail calls. The reference to *x* is in a tail context, but it is not a call and thus is not a tail call.

```

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

```

Note: Implementations may recognize that some non-tail calls, such as the call to *h* above, can be evaluated as though they were tail calls. In the example above, the *let* expression could be compiled as a tail call to *h*. (The possibility of *h* returning an unexpected number of values can be ignored, because in that case the effect of the *let* is explicitly unspecified and implementation-dependent.)

APPENDICES

Приложение А. Formal semantics

This appendix presents a non-normative, formal, operational semantics for Scheme, that is based on an earlier semantics [?]. It does not cover the entire language. The notable missing features are the macro system, I/O, and the numerical tower. The precise list of features included is given in section A.2.

The core of the specification is a single-step term rewriting relation that indicates how an (abstract) machine behaves. In general, the report is not a complete specification, giving implementations freedom to behave differently, typically to allow optimizations. This underspecification shows up in two ways in the semantics.

The first is reduction rules that reduce to special “**unknown:** *string*” states (where the string provides a description of the unknown state). The intention is that rules that reduce to such states can be replaced with arbitrary reduction rules. The precise specification of how to replace those rules is given in section A.12.

The other is that the single-step relation relates one program to multiple different programs, each corresponding to a legal transition that an abstract machine might take. Accordingly we use the transitive closure of the single step relation \rightarrow^* to define the semantics, \mathcal{S} , as a function from programs (\mathcal{P}) to sets of observable results (\mathcal{R}):

$$\begin{aligned} \mathcal{S} : \mathcal{P} &\longrightarrow 2^{\mathcal{R}} \\ \mathcal{S}(\mathcal{P}) &= \{\mathcal{O}(\mathcal{A}) \mid \mathcal{P} \rightarrow^* \mathcal{A}\} \end{aligned}$$

where the function \mathcal{O} turns an answer (\mathcal{A}) from the semantics into an observable result. Roughly, \mathcal{O} is the identity function on simple base values, and returns a special tag for more complex values, like procedure and pairs.

So, an implementation conforms to the semantics if, for every program \mathcal{P} , the implementation produces one of the results in $\mathcal{S}(\mathcal{P})$ or, if the implementation loops forever, then there is an infinite reduction sequence starting at \mathcal{P} , assuming that the reduction relation \rightarrow has been adjusted to replace the **unknown:** states.

The precise definitions of \mathcal{P} , \mathcal{A} , \mathcal{R} , and \mathcal{O} are also given in section A.2.

To help understand the semantics and how it behaves, we have implemented it in PLT Redex. The implementation is available at the report’s website: <http://www.r6rs.org/>. All of the reduction rules and the metafunctions shown in the figures in this semantics were generated automatically from the source code.

A.1. Background

We assume the reader has a basic familiarity with context-sensitive reduction semantics. Readers unfamiliar with this system may wish to consult Felleisen and Flatt’s monograph [?] or Wright and Felleisen [?] for a thorough introduction, including the relevant technical background, or an introduction to PLT Redex [?] for a somewhat lighter one.

As a rough guide, we define the operational semantics of a language via a relation on program terms, where the relation corresponds to a single step of an abstract machine. The relation is defined using evaluation contexts, namely terms with a distinguished place in them, called *holes*, where the next step of evaluation occurs. We say that a term e decomposes into an evaluation context E and another term e' if e is the same as E but with the hole replaced by e' . We write $E[e']$ to indicate the term obtained by replacing the hole in E with e' .

For example, assuming that we have defined a grammar containing non-terminals for evaluation contexts (E), expressions (e), variables (x), and values (v), we would write:

$$\begin{aligned} E_1[(\text{lambda } (x_1 \dots) e_1) v_1 \dots] &\rightarrow \\ E_1[\{x_1 \dots \mapsto v_1 \dots\}e_1] &\quad (\#x_1 = \#v_1) \end{aligned}$$

to define the β_v rewriting rule (as a part of the \rightarrow single step relation). We use the names of the non-terminals (possibly with subscripts) in a rewriting rule to restrict the application of the rule, so it applies only when some term produced by that grammar appears in the corresponding position in the term. If the same non-terminal with an identical subscript appears multiple times, the rule only applies when the corresponding terms are structurally identical (nonterminals without subscripts are not constrained to match each other). Thus, the occurrence of E_1 on both the left-hand and right-hand side of the rule above means that the context of the application expression does not change when using this rule. The ellipses are a form of Kleene star, meaning that zero or more occurrences of terms matching the pattern proceeding the ellipsis may appear in place of the ellipsis and the pattern preceding it. We use the notation $\{x_1 \dots \mapsto v_1 \dots\}e_1$ for capture-avoiding substitution; in this case it means that each x_1 is replaced with the corresponding v_1 in e_1 . Finally, we write side-conditions in parentheses beside a rule; the side-condition in the above rule indicates that the number of x_1 s must be the same as the number of v_1 s. Sometimes we use equality in the side-conditions; when we do it merely means simple term equality, i.e., the two terms must have the same syntactic shape.

Making the evaluation context E explicit in the rule allows us to define relations that manipulate their context. As a simple example, we can add another rule that signals a violation when a procedure is applied to the wrong number of arguments by discarding the evaluation context on the right-hand side of a rule:

$$\begin{aligned} E[(\text{lambda } (x_1 \dots) e) v_1 \dots] &\rightarrow \\ \textbf{violation: wrong argument count} &\quad (\#x_1 \neq \#v_1) \end{aligned}$$

Later we take advantage of the explicit evaluation context in more sophisticated ways.

A.2. Grammar

Figure A.2a shows the grammar for the subset of the report this semantics models. Non-terminals are written in *italics* or in a calligraphic font (\mathcal{P} , \mathcal{A} , \mathcal{R} , and \mathcal{R}_v) and literals are written in a monospaced font.

Рис. A.2a: Grammar for programs and observables

Рис. A.2b: Grammar for evaluation contexts

The \mathcal{P} non-terminal represents possible program states. The first alternative is a program with a store and an expression. The second alternative is an uncaught exception, and the third is used to indicate a place where the model does not completely specify the behavior of the primitives it models (see section A.12 for details of those situations). The \mathcal{A} non-terminal represents a final result of a program. It is just like \mathcal{P} except that expression has been reduced to some sequence of values.

The \mathcal{R} and \mathcal{R}_v non-terminals specify the observable results of a program. Each \mathcal{R} is either a sequence of values that correspond to the values produced by the program that terminates normally, or a tag indicating an uncaught exception was raised, or `unknown` if the program encounters a situation the semantics does not cover. The \mathcal{R}_v non-terminal specifies what the observable results are for a particular value: a pair, the empty list, a symbol, a self-quoting value (`#t`, `#f`, and numbers), a condition, or a procedure.

The \mathcal{S} non-terminal generates individual elements of the store. The store holds all of the mutable state of a program. It is explained in more detail along with the rules that manipulate it.

Expressions (\mathcal{E}) include quoted data, `begin` expressions, `begin0` expressions¹, application expressions, `if` expressions, `set!` expressions, variables, non-procedure values (*nonproc*), primitive procedures (*pproc*), lambda expressions, `letrec` and `letrec*` expressions.

The last few expression forms are only generated for intermediate states (`dw` for dynamic-wind, `throw` for continuations, `unspecified` for the result of the assignment operators, handlers for exception handlers, and `!` and `reinit` for `letrec`), and should not appear in an initial program. Their use is described in the relevant sections of this appendix.

The \mathcal{F} non-terminal describes the formals for lambda expressions. (The `dot` is used instead of a period for procedures that accept an arbitrary number of arguments, in order to avoid meta-circular confusion in our PLT Redex model.)

The \mathcal{S} non-terminal covers all datums, which can be either non-empty sequences (*seq*), the empty sequence, self-quoting values (*sqv*), or symbols. Non-empty sequences are either just a sequence of datums, or they are terminated with a dot followed by either a symbol or a self-quoting value. Finally the self-quoting values are numbers and the booleans `#t` and `#f`.

¹ `begin0` is not part of the standard, but we include it to make the rules for dynamic-wind and `letrec` easier to read. Although we model it directly, it can be defined in terms of other forms we model here that do come from the standard:

```
(begin0 e1 e2 ...) = (call-with-values
  (lambda () e1)
  (lambda x
    e2 ...
    (apply values x)))
```

The \mathcal{P} non-terminal represents programs that have no quoted data. Most of the reduction rules rewrite p to p , rather than \mathcal{P} to \mathcal{P} , since quoted data is first rewritten into calls to the list construction functions before ordinary evaluation proceeds. In parallel to \mathcal{E} , \mathcal{E} represents expressions that have no quoted expressions.

The values (\mathcal{V}) are divided into four categories:

- Non-procedures (*nonproc*) include pair pointers (`pp`), the empty list (`null`), symbols, self-quoting values (*sqv*), and conditions. Conditions represent the report's condition values, but here just contain a message and are otherwise inert.
- User procedures (`(lambda f e e ...)`) include multi-arity lambda expressions and lambda expressions with dotted parameter lists,
- Primitive procedures (*pproc*) include
 - arithmetic procedures (*aproc*): `+`, `-`, `/`, and `*`,
 - procedures of one argument (*proc1*): `null?`, `pair?`, `car`, `cdr`, `call/cc`, `procedure?`, `condition?`, `unspecified?`, `raise`, and `raise-continuable`,
 - procedures of two arguments (*proc2*): `cons`, `set-car!`, `set-cdr!`, `eqv?`, and `call-with-values`,
 - as well as `list`, `dynamic-wind`, `apply`, `values`, and `with-exception-handler`.
- Finally, continuations are represented as `throw` expressions whose body consists of the context where the continuation was grabbed.

The next three set of non-terminals in figure A.2a represent pairs (*pp*), which are divided into immutable pairs (*ip*) and mutable pairs (*mp*). The final set of non-terminals in figure A.2a, *sym*, *x*, and *n* represent symbols, variables, and numbers respectively. The non-terminals *ip*, *mp*, and *sym* are all assumed to all be disjoint. Additionally, the variables *x* are assumed not to include any keywords or primitive operations, so any program variables whose names coincide with them must be renamed before the semantics can give the meaning of that program.

The set of non-terminals for evaluation contexts is shown in figure A.2b. The \mathcal{P} non-terminal controls where evaluation happens in a program that does not contain any quoted data. The \mathcal{E} and \mathcal{F} evaluation contexts are for expressions. They are factored in that manner so that the *PG*, *G*, and *H* evaluation contexts can re-use \mathcal{F} and have fine-grained control over the context to support exceptions and dynamic-wind. The starred

Рис. А.3: Quote

Рис. А.4: Multiple values and call-with-values

and circled variants, E^* , E° , F^* , and F° dictate where a single value is promoted to multiple values and where multiple values are demoted to a single value. The U context is used to manage the report's underspecification of the results of `set!`, `set-car!`, and `set-cdr!` (see section A.12 for details). Finally, the S context is where quoted expressions can be simplified. The precise use of the evaluation contexts is explained along with the relevant rules.

To convert the answers (\mathcal{A}) of the semantics into observable results, we use these two functions:

They eliminate the store, and replace complex values with simple tags that indicate only the kind of value that was produced or, if no values were produced, indicates that either an uncaught exception was raised, or that the program reached a state that is not specified by the semantics.

A.3. Quote

The first reduction rules that apply to any program is the rules in figure A.3 that eliminate quoted expressions. The first two rules erase the quote for quoted expressions that do not introduce any pairs. The last two rules lift quoted datums to the top of the expression so they are evaluated only once, and turn the datums into calls to either `cons` or `consi`, via the metafunctions \mathcal{Q}_i and \mathcal{Q}_m .

Note that the left-hand side of the `[6qcons]` and `[6qconsi]` rules are identical, meaning that if one rule applies to a term, so does the other rule. Accordingly, a quoted expression may be lifted out into a sequence of `cons` expressions, which create mutable pairs, or into a sequence of `consi` expressions, which create immutable pairs (see section A.7 for the rules on how that happens).

These rules apply before any other because of the contexts in which they, and all of the other rules, apply. In particular, these rule applies in the S context. Figure A.2b shows that the S context allows this reduction to apply in any subexpression of an e , as long as all of the subexpressions to the left have no quoted expressions in them, although expressions to the right may have quoted expressions. Accordingly, this rule applies once for each quoted expression in the program, moving out to the beginning of the program. The rest of the rules apply in contexts that do not contain any quoted expressions, ensuring that these rules convert all quoted data into lists before those rules apply.

Although the identifier qp does not have a subscript, the semantics of PLT Redex's "fresh" declaration takes special care to ensure that the qp on the right-hand side of the rule is indeed the same as the one in the side-condition.

A.4. Multiple values

The basic strategy for multiple values is to add a rule that demotes (values v) to v and another rule that promotes v to (values v). If we allowed these rules to apply in an arbitrary evaluation context, however, we would get infinite reduction sequences of endless alternation between promotion and demotion. So, the semantics allows demotion only in a context expecting a single value and allows promotion only in a context expecting multiple values. We obtain this behavior with a small extension to the Felleisen-Hieb framework (also present in the operational model for R⁵RS [?]). We extend the notation so that holes have names (written with a subscript), and the context-matching syntax may also demand a hole of a particular name (also written with a subscript, for instance $E[e]_\star$). The extension allows us to give different names to the holes in which multiple values are expected and those in which single values are expected, and structure the grammar of contexts accordingly.

To exploit this extension, we use three kinds of holes in the evaluation context grammar in figure A.2b. The ordinary hole $[]$ appears where the usual kinds of evaluation can occur. The hole $[]_\star$ appears in contexts that allow multiple values and $[]_\circ$ appears in contexts that expect a single value. Accordingly, the rule `[6promote]` only applies in $[]_\star$ contexts, and `[6demote]` only applies in $[]_\circ$ contexts.

To see how the evaluation contexts are organized to ensure that promotion and demotion occur in the right places, consider the F , F^* and F° evaluation contexts. The F^* and F° evaluation contexts are just the same as F , except that they allow promotion to multiple values and demotion to a single value, respectively. So, the F evaluation context, rather than being defined in terms of itself, exploits F^* and F° to dictate where promotion and demotion can occur. For example, F can be $(\text{if } F^\circ ee)$ meaning that demotion from (values v) to v can occur in the test of an `if` expression. Similarly, F can be $(\text{begin } F^* ee \dots)$ meaning that v can be promoted to (values v) in the first subexpression of a `begin`.

In general, the promotion and demotion rules simplify the definitions of the other rules. For instance, the rule for `if` does not need to consider multiple values in its first subexpression. Similarly, the rule for `begin` does not need to consider the case of a single value as its first subexpression.

The other two rules in figure A.4 handle `call-with-values`. The evaluation contexts for `call-with-values` (in the F non-terminal) allow evaluation in the body of a procedure that has been passed as the first argument to `call-with-values`, as long as the second argument has been reduced to a value. Once evaluation inside that procedure completes, it will produce

Рис. А.5: Exceptions

Рис. А.6: Arithmetic and basic forms

multiple values (since it is an F^* position), and the entire `call-with-values` expression reduces to an application of its second argument to those values, via the rule [6cwvd]. Finally, in the case that the first argument to `call-with-values` is a value, but is not of the form `(lambda () e)`, the rule [6cwvw] wraps it in a thunk to trigger evaluation.

A.5. Exceptions

The workhorses for the exception system are

`(handlers proc ... e)`

expressions and the G and PG evaluation contexts (shown in figure A.2b). The `handlers` expression records the active exception handlers (`proc ...`) in some expression (e). The intention is that only the nearest enclosing `handlers` expression is relevant to raised exceptions, and the G and PG evaluation contexts help achieve that goal. They are just like their counterparts E and P , except that `handlers` expressions cannot occur on the path to the hole, and the exception system rules take advantage of that context to find the closest enclosing handler.

To see how the contexts work together with handler expressions, consider the left-hand side of the [6xunee] rule in figure A.5. It matches expressions that have a call to `raise` or `raise-continuable` (the non-terminal `raise*` matches both exception-raising procedures) in a PG evaluation context. Since the PG context does not contain any `handlers` expressions, this exception cannot be caught, so this expression reduces to a final state indicating the uncaught exception. The rule [6xuneh] also signals an uncaught exception, but it covers the case where a `handlers` expression has exhausted all of the handlers available to it. The rule applies to expressions that have a `handlers` expression (with no exception handlers) in an arbitrary evaluation context where a call to one of the exception-raising functions is nested in the `handlers` expression. The use of the G evaluation context ensures that there are no other handler expressions between this one and the `raise`.

The next two rules cover call to the procedure `with-exception-handler`. The [6xwh1] rule applies when there are no handler expressions. It constructs a new one and applies v_2 as a thunk in the handler body. If there already is a handler expression, the [6xwhn] applies. It collects the current handlers and adds the new one into a new `handlers` expression and, as with the previous rule, invokes the second argument to `with-exception-handlers`.

The next two rules cover exceptions that are raised in the context of a `handlers` expression. If a `continuable` exception is raised,

[6xrc] applies. It takes the most recently installed handler from the nearest enclosing `handlers` expression and applies it to the argument to `raise-continuable`, but in a context where the exception handlers do not include that latest handler. The [6xr] rule behaves similarly, except it raises a new exception if the handler returns. The new exception is created with the `make-cond` special form.

The `make-cond` special form is a stand-in for the report's conditions. It does not evaluate its argument (note its absence from the E grammar in figure A.2b). That argument is just a literal string describing the context in which the exception was raised. The only operation on conditions is `condition?`, whose semantics are given by the two rules [6ct] and [6cf].

Finally, the rule [6xdone] drops a `handlers` expression when its body is fully evaluated, and the rule [6weherr] raises an exception when `with-exception-handler` is supplied with incorrect arguments.

A.6. Arithmetic and basic forms

This model does not include the report's arithmetic, but does include an idealized form in order to make experimentation with other features and writing test suites for the model simpler. Figure A.6 shows the reduction rules for the primitive procedures that implement addition, subtraction, multiplication, and division. They defer to their mathematical analogues. In addition, when the subtraction or division operator are applied to no arguments, or when division receives a zero as a divisor, or when any of the arithmetic operations receive a non-number, an exception is raised.

The bottom half of figure A.6 shows the rules for `if`, `begin`, and `begin0`. The relevant evaluation contexts are given by the F non-terminal.

The evaluation contexts for `if` only allow evaluation in its test expression. Once that is a value, the rules reduce an `if` expression to its consequent if the test is not `#f`, and to its alternative if it is `#f`.

The `begin` evaluation contexts allow evaluation in the first subexpression of a `begin`, but only if there are two or more subexpressions. In that case, once the first expression has been fully simplified, the reduction rules drop its value. If there is only a single subexpression, the `begin` itself is dropped.

Like the `begin` evaluation contexts, the `begin0` evaluation contexts allow evaluation of the first subexpression of a `begin0` expression when there are two or more subexpressions. The `begin0` evaluation contexts also allow evaluation in the second subexpression of a `begin0` expression, as long as the first

Рис. А.7: Lists

Рис. А.8: Eqv

Рис. А.9а: Procedures & application

subexpression has been fully simplified. The `[6begin0n]` rule for `begin0` then drops a fully simplified second subexpression. Eventually, there is only a single expression in the `begin0`, at which point the `[6begin01]` rule fires, and removes the `begin0` expression.

A.7. Lists

The rules in figure A.7 handle lists. The first two rules handle `list` by reducing it to a succession of calls to `cons`, followed by `null`.

The next two rules, `[6cons]` and `[6consi]`, allocate new `cons` cells. They both move `(cons v1 v2)` into the store, bound to a fresh pair pointer (see also section A.3 for a description of “fresh”). The `[6cons]` uses a `mp` variable, to indicate the pair is mutable, and the `[6consi]` uses a `ip` variable to indicate the pair is immutable.

The rules `[6car]` and `[6cdr]` extract the components of a pair from the store when presented with a pair pointer (the `pp` can be either `mp` or `ip`, as shown in figure A.2a).

The rules `[6setcar]` and `[6setcdr]` handle assignment of mutable pairs. They replace the contents of the appropriate location in the store with the new value, and reduce to `unspecified`. See section A.12 for an explanation of how `unspecified` reduces.

The next four rules handle the `null?` predicate and the `pair?` predicate, and the final four rules raise exceptions when `car`, `cdr`, `set-car!` or `set-cdr!` receive non pairs.

A.8. Eqv

The rules for `eqv?` are shown in figure A.8. The first two rules cover most of the behavior of `eqv?`. The first says that when the two arguments to `eqv?` are syntactically identical, then `eqv?` produces `#t` and the second says that when the arguments are not syntactically identical, then `eqv?` produces `#f`. The structure of `v` has been carefully designed so that simple term equality corresponds closely to `eqv?`’s behavior. For example, pairs are represented as pointers into the store and `eqv?` only compares those pointers.

The side-conditions on those first two rules ensure that they do not apply when simple term equality does not match the behavior of `eqv?`. There are two situations where it does not match:

comparing two conditions and comparing two procedures. For the first, the report does not specify `eqv?`’s behavior, except to say that it must return a boolean, so the remaining two rules (`[6eqct]`, and `[6eqcf]`) allow such comparisons to return `#t` or `#f`. Comparing two procedures is covered in section A.12.

A.9. Procedures and application

In evaluating a procedure call, the report leaves unspecified the order in which arguments are evaluated. So, our reduction system allows multiple, different reductions to occur, one for each possible order of evaluation.

To capture unspecified evaluation order but allow only evaluation that is consistent with some sequential ordering of the evaluation of an application’s subexpressions, we use non-deterministic choice to first pick a subexpression to reduce only when we have not already committed to reducing some other subexpression. To achieve that effect, we limit the evaluation of application expressions to only those that have a single expression that is not fully reduced, as shown in the non-terminal *F*, in figure A.2b. To evaluate application expressions that have more than two arguments to evaluate, the rule `[6mark]` picks one of the subexpressions of an application that is not fully simplified and lifts it out in its own application, allowing it to be evaluated. Once one of the lifted expressions is evaluated, the `[6appN]` substitutes its value back into the original application.

The `[6appN]` rule also handles other applications whose arguments are finished by substituting the first argument for the first formal parameter in the expression. Its side-condition uses the relation in figure A.9b to ensure that there are no `set!` expressions with the parameter *x*₁ as a target. If there is such an assignment, the `[6appN!]` rule applies (see also section A.3 for a description of “fresh”). Instead of directly substituting the actual parameter for the formal parameter, it creates a new location in the store, initially bound the actual parameter, and substitutes a variable standing for that location in place of the formal parameter. The store, then, handles any eventual assignment to the parameter. Once all of the parameters have been substituted away, the rule `[6app0]` applies and evaluation of the body of the procedure begins.

At first glance, the rule `[6appN]` appears superfluous, since it seems like the rules could just reduce first by `[6appN!]` and then look up the variable when it is evaluated. There are two

Рис. A.9b: Variable-assignment relation

reasons why we keep the `[6appN]`, however. The first is purely conventional: reducing applications via substitution is taught to us at an early age and is commonly used in rewriting systems in the literature. The second reason is more technical: the `[6mark]` rule requires that `[6appN]` be applied once e_i has been reduced to a value. `[6appN!]` would lift the value into the store and put a variable reference into the application, leading to another use of `[6mark]`, and another use of `[6appN!]`, which continues forever.

The rule `[6μapp]` handles a well-formed application of a function with a dotted parameter lists. It such an application into an application of an ordinary procedure by constructing a list of the extra arguments. Similarly, the rule `[6μapp1]` handles an application of a procedure that has a single variable as its parameter list.

The rule `[6var]` handles variable lookup in the store and `[6set]` handles variable assignment.

The next two rules `[6proct]` and `[6procf]` handle applications of `procedure?`, and the remaining rules cover applications of non-procedures and arity violations.

The rules in figure A.9c cover `apply`. The first rule, `[6applyf]`, covers the case where the last argument to `apply` is the empty list, and simply reduces by erasing the empty list and the `apply`. The second rule, `[6applyc]` covers a well-formed application of `apply` where `apply`'s final argument is a pair. It reduces by extracting the components of the pair from the store and putting them into the application of `apply`. Repeated application of this rule thus extracts all of the list elements passed to `apply` out of the store.

The remaining five rules cover the various violations that can occur when using `apply`. The first one covers the case where `apply` is supplied with a cyclic list. The next four cover applying a non-procedure, passing a non-list as the last argument, and supplying too few arguments to `apply`.

A.10. Call/cc and dynamic wind

The specification of `dynamic-wind` uses `(dw x e e e)` expressions to record which dynamic-wind *thunks* are active at each point in the computation. Its first argument is an identifier that is globally unique and serves to identify invocations of `dynamic-wind`, in order to avoid exiting and re-entering the same dynamic context during a continuation switch. The second, third, and fourth arguments are calls to some *before*, *thunk*, and *after* procedures from a call to `dynamic-wind`. Evaluation only occurs in the middle expression; the `dw` expression only serves to record which *before* and *after* procedures need to be run during a continuation switch. Accordingly, the reduction rule for an application of `dynamic-wind` reduces to a call to the *before* procedure, a `dw` expression and a call to the *after* procedure, as shown in rule `[6wind]` in figure A.10. The next two rules cover abuses of the `dynamic-wind` procedure: calling it with non-procedures, and calling it with the wrong number of arguments.

The `[6dwdone]` rule erases a `dw` expression when its second argument has finished evaluating.

The next two rules cover `call/cc`. The rule `[6call/cc]` creates a new continuation. It takes the context of the `call/cc` expression and packages it up into a `throw` expression that represents the continuation. The `throw` expression uses the fresh variable x to record where the application of `call/cc` occurred in the context for use in the `[6throw]` rule when the continuation is applied. That rule takes the arguments of the continuation, wraps them with a call to `values`, and puts them back into the place where the original call to `call/cc` occurred, replacing the current context with the context returned by the \mathcal{T} metafunction.

The \mathcal{T} (for “trim”) metafunction accepts two D contexts and builds a context that matches its second argument, the destination context, except that additional calls to the *before* and *after* procedures from `dw` expressions in the context have been added.

The first clause of the \mathcal{T} metafunction exploits the H context, a context that contains everything except `dw` expressions. It ensures that shared parts of the dynamic-wind context are ignored, recurring deeper into the two expression contexts as long as the first `dw` expression in each have matching identifiers (x_1). The final rule is a catchall; it only applies when all the others fail and thus applies either when there are no `dws` in the context, or when the `dw` expressions do not match. It calls the two other metafunctions defined in figure A.10 and puts their results together into a `begin` expression.

The \mathcal{R} metafunction extracts all of the *before* procedures from its argument and the \mathcal{S} metafunction extracts all of the *after* procedures from its argument. They each construct new contexts and exploit H to work through their arguments, one `dw` at a time. In each case, the metafunctions are careful to keep the right `dw` context around each of the procedures in case a continuation jump occurs during one of their evaluations. Since \mathcal{R} , receives the destination context, it keeps the intermediate parts of the context in its result. In contrast \mathcal{S} discards all of the context except the `dws`, since that was the context where the call to the continuation occurred.

A.11. Letrec

Figure A.11 shows the rules that handle `letrec` and `letrec*` and the supplementary expressions that they produce, `l!` and `reinit`. As a first approximation, both `letrec` and `letrec*` reduce by allocating locations in the store to hold the values of the *init* expressions, initializing those locations to `bh` (for “black hole”), evaluating the *init* expressions, and then using `l!` to update the locations in the store with the value of the *init* expressions. They also use `reinit` to detect when an *init* expression in a `letrec` is reentered via a continuation.

Before considering how `letrec` and `letrec*` use `l!` and `reinit`, first consider how `l!` and `reinit` behave. The first

Рис. А.9с: Apply

Рис. А.10: Call/cc and dynamic wind

Рис. А.11: Letrec and letrec*

two rules in figure A.11 cover `let!`. It behaves very much like `set!`, but it initializes both ordinary variables, and variables that are current bound to the black hole (`bh`).

The next two rules cover ordinary `set!` when applied to a variable that is currently bound to a black hole. This situation can arise when the program assigns to a variable before `letrec` initializes it, eg `(letrec ((x (set! x 5))) x)`. The report specifies that either an implementation should perform the assignment, as reflected in the `[6setdt]` rule or it raise an exception, as reflected in the `[6setdte]` rule.

The `[6dt]` rule covers the case where a variable is referred to before the value of a `init` expression is filled in, which must always raise an exception.

A `reinit` expression is used to detect a program that captures a continuation in an initialization expression and returns to it, as shown in the three rules `[6init]`, `[6reinit]`, and `[6reinite]`. The `reinit` form accepts an identifier that is bound in the store to a boolean as its argument. Those are identifiers are initially `#f`. When `reinit` is evaluated, it checks the value of the variable and, if it is still `#f`, it changes it to `#t`. If it is already `#t`, then `reinit` either just does nothing, or it raises an exception, in keeping with the two legal behaviors of `letrec` and `letrec*`.

The last two rules in figure A.11 put together `let!` and `reinit`. The `[6letrec]` rule reduces a `letrec` expression to an application expression, in order to capture the unspecified order of evaluation of the `init` expressions. Each `init` expression is wrapped in a `begin0` that records the value of the `init` and then uses `reinit` to detect continuations that return to the `init` expression. Once all of the `init` expressions have been evaluated, the procedure on the right-hand side of the rule is invoked, causing the value of the `init` expression to be filled in the store, and evaluation continues with the body of the original `letrec` expression.

The `[6letrec*]` rule behaves similarly, but uses a `begin` expression rather than an application, since the `init` expressions are evaluated from left to right. Moreover, each `init` expression is filled into the store as it is evaluated, so that subsequent `init` expressions can refer to its value.

A.12. Underspecification

The rules in figure A.12 cover aspects of the semantics that are explicitly unspecified. Implementations can replace the rules `[6ueqv]`, `[6uval]` and with different rules that cover the left-hand

sides and, as long as they follow the informal specification, any replacement is valid. Those three situations correspond to the case when `eqv?` applied to two procedures and when multiple values are used in a single-value context.

The remaining rules in figure A.12 cover the results from the assignment operations, `set!`, `set-car!`, and `set-cdr!`. An implementation does not adjust those rules, but instead renders them useless by adjusting the rules that insert `unspecified`: `[6setcar]`, `[6setcdr]`, `[6set]`, and `[6setd]`. Those rules can be adjusted by replacing `unspecified` with any number of values in those rules.

So, the remaining rules just specify the minimal behavior that we know that a value or values must have and otherwise reduce to an **unknown** state. The rule `[6udemand]` drops `unspecified` in the `U` context. See figure A.2b for the precise definition of `U`, but intuitively it is a context that is only a single expression layer deep that contains expressions whose value depends on the value of their subexpressions, like the first subexpression of a `if`. Following that are rules that discard `unspecified` in expressions that discard the results of some of their subexpressions. The `[6ubegin]` shows how `begin` discards its first expression when there are more expressions to evaluate. The next two rules, `[6uhandlers]` and `[6udw]` propagate `unspecified` to their context, since they also return any number of values to their context. Finally, the two `begin0` rules preserve `unspecified` until the rule `[6begin01]` can return it to its context.

Приложение В. Sample definitions for derived forms

This appendix contains sample definitions for some of the keywords described in this report in terms of simpler forms:

cond

The `cond` keyword (section 11.4.5) could be defined in terms of `if`, `let` and `begin` using `syntax-rules` as follows:

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

Рис. A.12: Explicitly unspecified behavior

```

    (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 ...))))

```

case

The `case` keyword (section 11.4.5) could be defined in terms of `let`, `cond`, and `memv` (see library chapter ??) using syntax-rules as follows:

```

(define-syntax case
  (syntax-rules (else)
    ((case expr0
      ((key ...) res1 res2 ...)
      ...
      (else else-res1 else-res2 ...))
     (let ((tmp expr0))
       (cond
        ((memv tmp '(key ...)) res1 res2 ...)
        ...
        (else else-res1 else-res2 ...))))
    ((case expr0
      ((keya ...) res1a res2a ...)
      ((keyb ...) res1b res2b ...)
      ...)
     (let ((tmp expr0))
       (cond
        ((memv tmp '(keya ...)) res1a res2a ...)
        ((memv tmp '(keyb ...)) res1b res2b ...)
        ...))))

```

let*

The `let*` keyword (section 11.4.6) could be defined in terms of `let` using syntax-rules as follows:

```

(define-syntax let*
  (syntax-rules ()
    ((let* () body1 body2 ...)
     (let () body1 body2 ...))
    ((let* ((name1 expr1) (name2 expr2) ...)
     body1 body2 ...))

```

```

    (let ((name1 expr1))
      (let* ((name2 expr2) ...)
        body1 body2 ...))))

```

letrec

The `letrec` keyword (section 11.4.6) could be defined approximately in terms of `let` and `set!` using syntax-rules, using a helper to generate the temporary variables needed to hold the values before the assignments are made, as follows:

```

(define-syntax letrec
  (syntax-rules ()
    ((letrec () body1 body2 ...)
     (let () body1 body2 ...))
    ((letrec ((var init) ...) body1 body2 ...)
     (letrec-helper
      (var ...)
      ()
      ((var init) ...)
      body1 body2 ...))))

```

```

(define-syntax letrec-helper
  (syntax-rules ()
    ((letrec-helper
      ()
      (temp ...)
      ((var init) ...)
      body1 body2 ...)
     (let ((var <undefined>) ...)
       (let ((temp init) ...)
         (set! var temp)
         ...))
     (let () body1 body2 ...)))
    ((letrec-helper
      (x y ...)
      (temp ...)
      ((var init) ...)
      body1 body2 ...)
     (letrec-helper
      (y ...)
      (newtemp temp ...)
      ((var init) ...)
      body1 body2 ...))))

```

The syntax `<undefined>` represents an expression that returns something that, when stored in a location, causes an exception with condition type `&assertion` to be raised if an attempt to read from or write to the location occurs before the assignments generated by the `letrec` transformation take place. (No such expression is defined in Scheme.)

A simpler definition using syntax-case and generate-temporaries is given in library chapter ??.

letrec*

The `letrec*` keyword could be defined approximately in terms of `let` and `set!` using syntax-rules as follows:

```
(define-syntax letrec*
  (syntax-rules ()
    ((letrec* ((var1 init1) ...) body1 body2 ...)
      (let ((var1 <undefined>) ...)
        (set! var1 init1)
        ...
        (let () body1 body2 ...))))))
```

The syntax `<undefined>` is as in the definition of `letrec` above.

let-values

The following definition of `let-values` (section 11.4.6) using `syntax-rules` employs a pair of helpers to create temporary names for the formals.

```
(define-syntax let-values
  (syntax-rules ()
    ((let-values (binding ...) body1 body2 ...)
      (let-values-helper1
       ()
       (binding ...)
       body1 body2 ...))))

(define-syntax let-values-helper1
  ;; map over the bindings
  (syntax-rules ()
    ((let-values
      ((id temp) ...)
      ()
      body1 body2 ...)
      (let ((id temp) ...) body1 body2 ...))
    ((let-values
      assocs
      ((formals1 expr1) (formals2 expr2) ...)
      body1 body2 ...)
      (let-values-helper2
       formals1
       ()
       expr1
       assocs
       ((formals2 expr2) ...)
       body1 body2 ...))))

(define-syntax let-values-helper2
  ;; create temporaries for the formals
  (syntax-rules ()
    ((let-values-helper2
      ()
      temp-formals
      expr1
      assocs
      bindings
      body1 body2 ...)
      (call-with-values
       (lambda () expr1)
       (lambda temp-formals
        (let-values-helper1
         assocs
         bindings
         body1 body2 ...))))
    ((let-values-helper2
      (first . rest)
      ...
      body1 body2 ...))))
```

```
(temp ...)
expr1
(assoc ...)
bindings
body1 body2 ...)
(let-values-helper2
 rest
 (temp ... newtemp)
 expr1
 (assoc ... (first newtemp))
 bindings
 body1 body2 ...))
((let-values-helper2
 rest-formal
 (temp ...)
 expr1
 (assoc ...)
 bindings
 body1 body2 ...)
 (call-with-values
  (lambda () expr1)
  (lambda (temp ... . newtemp)
   (let-values-helper1
    (assoc ... (rest-formal newtemp))
    bindings
    body1 body2 ...))))))
```

let*-values

The following macro defines `let*-values` in terms of `let` and `let-values` using `syntax-rules`:

```
(define-syntax let*-values
  (syntax-rules ()
    ((let*-values () body1 body2 ...)
      (let () body1 body2 ...))
    ((let*-values (binding1 binding2 ...)
      body1 body2 ...)
      (let-values (binding1)
        (let*-values (binding2 ...)
          body1 body2 ...))))))
```

let

The `let` keyword could be defined in terms of `lambda` and `letrec` using `syntax-rules` as follows:

```
(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 ...))))
```

Приложение С. Additional material

This report itself, as well as more material related to this report such as reference implementations of some parts of Scheme and archives of mailing lists discussing this report is at

<http://www.r6rs.org/>

The Schemers web site at

<http://www.schemers.org/>

as well as the Readscheme site at

<http://library.readscheme.org/>

contain extensive Scheme bibliographies, as well as papers, programs, implementations, and other material related to Scheme.

Приложение D. Example

This section describes an example consisting of the (runge-kutta) library, which provides an integrate-system procedure that integrates the system

$$y'_k = f_k(y_1, y_2, \dots, y_n), \quad k = 1, \dots, n$$

of differential equations with the method of Runge-Kutta.

As the (runge-kutta) library makes use of the (rnrs base (6)) library, its skeleton is as follows:

```
#!r6rs
(library (runge-kutta)
  (export integrate-system
            head tail)
  (import (rnrs base))
  (library body))
```

The procedure definitions described below go in the place of (library body).

The parameter system-derivative is a function that takes a system state (a vector of values for the state variables y_1, \dots, y_n) and produces a system derivative (the values y'_1, \dots, y'_n). The parameter initial-state provides an initial system state, and h is an initial guess for the length of the integration step.

The value returned by integrate-system is an infinite stream of system states.

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

The runge-kutta-4 procedure takes a function, f, that produces a system derivative from a system state. The runge-kutta-4 procedure produces a function that takes a system state and produces a new system state.

```
(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)))))
```

The map-streams procedure is analogous to map: it applies its first argument (a procedure) to all the elements of its second argument (a stream).

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

Infinite streams are implemented as pairs whose car holds the first element of the stream and whose cdr holds a procedure that delivers the rest of the stream.

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

The following program illustrates the use of integrate-system in integrating the system

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

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

which models a damped oscillator.

```

#!r6rs
(import (rnrs base)
        (rnrs io simple)
        (runge-kutta))

(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))

(letrec ((loop (lambda (s)
                  (newline)
                  (write (head s))
                  (loop (tail s)))))
  (loop the-states))

```

This prints output like the following:

```

#(1 0)
#(0.99895054 9.994835e-6)
#(0.99780226 1.9978681e-5)
#(0.9965554 2.9950552e-5)
#(0.9952102 3.990946e-5)
#(0.99376684 4.985443e-5)
#(0.99222565 5.9784474e-5)
#(0.9905868 6.969862e-5)
#(0.9888506 7.9595884e-5)
#(0.9870173 8.94753e-5)

```

Приложение E. Language changes

This chapter describes most of the changes that have been made to Scheme since the “Revised⁵ Report” [?] was published:

- Scheme source code now uses the Unicode character set. Specifically, the character set that can be used for identifiers has been greatly expanded.
- Identifiers can now start with the characters `->`.
- Identifiers and symbol literals are now case-sensitive.
- Identifiers and representations of characters, booleans, number objects, and `.` must be explicitly delimited.
- `#` is now a delimiter.
- Bytevector literal syntax has been added.
- Matched square brackets can be used synonymously with parentheses.
- The read-syntax abbreviations `#'` (for `syntax`), `#□` (for `quasisyntax`), `#,` (for `unsyntax`), and `#,@` (for `unsyntax-splicing` have been added; see section 4.3.5.)

- `#` can no longer be used in place of digits in number representations.
- The external representation of number objects can now include a mantissa width.
- Literals for NaNs and infinities were added.
- String and character literals can now use a variety of escape sequences.
- Block and datum comments have been added.
- The `#!r6rs` comment for marking report-compliant lexical syntax has been added.
- Characters are now specified to correspond to Unicode scalar values.
- Many of the procedures and syntactic forms of the language are now part of the `(rnrs base (6))` library. Some procedures and syntactic forms have been moved to other libraries; see figure A.1.
- The base language has the following new procedures and syntactic forms: `letrec*`, `let-values`, `let*-values`, `real-valued?`, `rational-valued?`, `integer-valued?`, `exact`, `inexact`, `finite?`, `infinite?`, `nan?`, `div`, `mod`, `div-and-mod`, `div0`, `mod0`, `div0-and-mod0`, `exact-integer-sqrt`, `boolean=?`, `symbol=?`, `string-for-each`, `vector-map`, `vector-for-each`, `error`, `assertion-violation`, `assert`, `call/cc`, `identifier-syntax`.
- The following procedures have been removed: `char-ready?`, `transcript-on`, `transcript-off`, `load`.
- The case-insensitive string comparisons (`string-ci=?`, `string-ci<?`, `string-ci>?`, `string-ci<=?`, `string-ci>=?`) operate on the case-folded versions of the strings rather than as the simple lexicographic ordering induced by the corresponding character comparison procedures.
- Libraries have been added to the language.
- A number of standard libraries are described in a separate report [?].
- Many situations that “were an error” now have defined or constrained behavior. In particular, many are now specified in terms of the exception system.
- The full numerical tower is now required.
- The semantics for the transcendental functions has been specified more fully.
- The semantics of `expt` for zero bases has been refined.
- In `syntax-rules` forms, a `_` may be used in place of the keyword.

- The `let-syntax` and `letrec-syntax` no longer introduce a new environment for their bodies.
- For implementations that support NaNs or infinities, many arithmetic operations have been specified on these values consistently with IEEE 754.
- For implementations that support a distinct `-0.0`, the semantics of many arithmetic operations with regard to `-0.0` has been specified consistently with IEEE 754.
- Scheme's real number objects now have an exact zero as their imaginary part.
- The specification of `quasiquote` has been extended. Nested quasiquotations work correctly now, and `unquote` and `unquote-splicing` have been extended to several operands.
- Procedures now may or may not refer to locations. Consequently, `eqv?` is now unspecified in a few cases where it was specified before.
- The mutability of the values of `quasiquote` structures has been specified to some degree.
- The dynamic environment of the *before* and *after* procedures of `dynamic-wind` is now specified.
- Various expressions that have only side effects are now allowed to return an arbitrary number of values.
- The order and semantics for macro expansion has been more fully specified.
- Internal definitions are now defined in terms of `letrec*`.
- The old notion of program structure and Scheme's top-level environment has been replaced by top-level programs and libraries.
- The denotational semantics has been replaced by an operational semantics based on an earlier semantics for the language of the "Revised⁵ Report" [?, ?].

identifier	moved to
assoc	(rnrs lists (6))
assv	(rnrs lists (6))
assq	(rnrs lists (6))
call-with-input-file	(rnrs io simple (6))
call-with-output-file	(rnrs io simple (6))
char-upcase	(rnrs unicode (6))
char-downcase	(rnrs unicode (6))
char-ci=?	(rnrs unicode (6))
char-ci<?	(rnrs unicode (6))
char-ci>?	(rnrs unicode (6))
char-ci<=?	(rnrs unicode (6))
char-ci>=?	(rnrs unicode (6))
char-alphabetic?	(rnrs unicode (6))
char-numeric?	(rnrs unicode (6))
char-whitespace?	(rnrs unicode (6))
char-upper-case?	(rnrs unicode (6))
char-lower-case?	(rnrs unicode (6))
close-input-port	(rnrs io simple (6))
close-output-port	(rnrs io simple (6))
current-input-port	(rnrs io simple (6))
current-output-port	(rnrs io simple (6))
display	(rnrs io simple (6))
do	(rnrs control (6))
eof-object?	(rnrs io simple (6))
eval	(rnrs eval (6))
delay	(rnrs r5rs (6))
exact->inexact	(rnrs r5rs (6))
force	(rnrs r5rs (6))
identifier	moved to
inexact->exact	(rnrs r5rs (6))
member	(rnrs lists (6))
memv	(rnrs lists (6))
memq	(rnrs lists (6))
modulo	(rnrs r5rs (6))
newline	(rnrs io simple (6))
null-environment	(rnrs r5rs (6))
open-input-file	(rnrs io simple (6))
open-output-file	(rnrs io simple (6))
peek-char	(rnrs io simple (6))
quotient	(rnrs r5rs (6))
read	(rnrs io simple (6))
read-char	(rnrs io simple (6))
remainder	(rnrs r5rs (6))
scheme-report-environment	(rnrs r5rs (6))
set-car!	(rnrs mutable-pairs (6))
set-cdr!	(rnrs mutable-pairs (6))
string-ci=?	(rnrs unicode (6))
string-ci<?	(rnrs unicode (6))
string-ci>?	(rnrs unicode (6))
string-ci<=?	(rnrs unicode (6))
string-ci>=?	(rnrs unicode (6))
string-set!	(rnrs mutable-strings (6))
string-fill!	(rnrs mutable-strings (6))
with-input-from-file	(rnrs io simple (6))
with-output-to-file	(rnrs io simple (6))
write	(rnrs io simple (6))
write-char	(rnrs io simple (6))

Рис. A.1: Identifiers moved to libraries