算法语言 Scheme 修订^{5}报告

王咏刚 试译 v 0.9.5

试译稿前言

国内关注函数式编程 (FP) 的朋友越来越多，但相关的中文资料却寥寥无几。其实，和以往借鉴西方思想文化的历程相仿，只有先做足了迻译和推介的功夫，函数式编程的思想才能从少数发烧友走向普通的程序员，然后我们才能期盼着国内也出现 Guy Lewis Steele 或 David Madore 这样的“数学家程序员”。基于这样的想法，在 2004 年 8 月份约一个月的空闲时段里，我不自量力地翻译了 Scheme 语言（事实上）的标准文档 R^{5}RS。

说不自量力，是因为我对 Scheme 的了解非常有限，我的英文和数学积累也少得可怜。对于这样一篇文辞洗练、逻辑缜密的标准文档，与其说我是在“翻译”，还不如说我是在“学习”或“试译”；再加上时间有限，无暇推敲，译文中必然充斥着错误和缺陷。我都不知道该不该把它放在网上供大家阅读了。

算了，还是把它贴出来吧。也许这篇译文质量不高，也许其他人已经或正在做同样的工作，但它应该能起到抛砖引玉的作用。又因为这完全是免费发布，我也用不着为它的质量问题承担过多的责任。

最后，请大家一起来完善这篇译文！我的意思是说，请所有阅读过这篇译文且发现了其中的翻译问题的朋友给我来信，指出问题所在。我会及时修改和更新译文。我的邮件地址是： wangyg@contextfree.net

王咏刚

2004 年 11 月

特别感谢

- 裘宗燕：北京大学教授，著名译者。他翻译了 Scheme 世界里的经典读物《计算机程序的构造和解释》，其功绩不言自明。这一份试译稿完成后，我将它发给裘教授审阅，没想到，裘教授竟从 9 月开始，就试译稿的内容陆陆续续提出了数百条宝贵意见。这些意见中的绝大部分已经被试译稿采纳。在此，谨向裘教授表示衷心的感谢。
- 日文译者： R^{5}RS 的日文译本很早就出现了，而且可以找到两个以上的不同译本。例如， 1999 年 Hisao Suzuki 的译本在这里 (http://www.unixuser.org/~euske/doc/r5rs-ja/index.html)， 2000 年 Dai Inukai 的译本在这里 (http://www.sci.toyama-u.ac.jp/~iwao/Scheme/r5rsj/html/r5rsj_toc.html)。

试译稿版本历史

- 2004.11， v0.9.5，试译稿免费发布。
- 2004.11， v0.9.4，试译稿首次发布前的最后通校。
- 2004.11， v0.9.3，根据裘宗燕教授的校改意见修改了 6.3 节以后的若干内容。
- 2004.10， v0.9.2，根据裘宗燕教授的校改意见修改了 6.3 节以前的若干内容。
- 2004.09， v0.9.1，试译稿初校完成。
- 2004.08， v0.9.0，试译稿翻译。

算法语言 Scheme 修订^{5}报告

王咏刚 试译　v 0.9.5

Richard Kelsey, William Clinger, and Jonathan Rees (编者)

H. Abelson	R. K. Dybvig	C. T. Haynes	G. J. Rozas
N. I. Adams iv	D. P. Friedman	E. Kohlbecker	G. L. Steele Jr.
D. H. Bartley	R. Halstead	D. Oxley	G. J. Sussman
G. Brooks	C. Hanson	K. M. Pitman	M. Wand

谨以此纪念 *Robert Hieb*

1998 年 2 月 20 日

摘要

本报告给出程序设计语言 Scheme 的定义性描述。 Scheme 是 Lisp 程序设计语言的方言，由 Guy Lewis Steele Jr. 和 Gerald Jay Sussman 发明，具有静态作用域和严格尾递归的特点。它的设计目标是拥有异常清晰、简明的语义和较少的表达式异构方式。包括命令式、函数式和消息传递式风格在内的绝大多数程序设计模型都可以用 Scheme 方便地表述。

概述部分简要介绍了 Scheme 语言和本报告的发展历程。

前三章展示了语言的基本概念，说明了用于描述 Scheme 语言和编写 Scheme 程序的记法约定。

第 4 和第 5 章描述了表达式、程序及定义的语法和语义。

第 6 章描述了 Scheme 语言的内置过程，包括语言中数据处理和输入/输出的所有基本操作。

第 7 章用扩展 BNF 给出了 Scheme 语言的形式语法，并为其定义了形式指称语义。形式语法和语义后面有一个使用 Scheme 语言的示例。

本报告的最后是参考文献列表和依字母顺序排列的索引。

目录

概述

程序设计语言的设计不应该是特征的堆砌，而应消除那些需要依赖于多余特征的弱点和局限。 Scheme 语言证明，极少的表达式构造规则和不加限制的表达式复合方式可以构造出实用而高效的程序设计语言，其灵活性足以支持今天的大部分主流编程模型。

Scheme 是最早的像在 lambda 演算里一样提供了第一级过程的程序设计语言之一，并由此在动态类型的语言中提供了有用的静态作用域规则和块结构的特征。在 Lisp 的主要方言中， Scheme 第一次使过程有别于 lambda 表达式和符号，为所有变量使用单一的词法环境，在确定运算符的位置时采用与确定运算对象位置一样的方式。 Scheme 完全依赖过程调用来表示迭代，并以此强调，尾递归过程调用在本质上就是传递参数的 goto。 Scheme 是第一种被广泛使用的，采纳第一级逃逸过程 (Escape procedure) 的程序设计语言。第一级逃逸过程可以合成所有已知的顺序控制结构。 Scheme 的后续版本引入了精确和非精确数的概念，这是 Common Lisp 语言通用算术功能的扩展。最近， Scheme 成为了第一种支持卫生宏 (Hygienic macro) 的程序设计语言，该机制允许我们以一种一致和可靠的方式扩展块结构语言的语法。

背景

Scheme 的第一份描述文档编写于 1975 年[28]。 1978 年发布的修订报告[25] 描述了 MIT 实现中语言的演化情况，该实现带有一个开创性的编译器[26]。 1981 和 1982 年开始的三个独立的项目将 Scheme 的不同变体用于 MIT、 Yale 和 Indiana 大学的课堂教学[21, 17, 10]。一部导论性质的，使用 Scheme 语言的计算机科学教材于 1984 年出版[1]。

由于 Scheme 得到了日益广泛的应用，语言的局部方言开始产生分歧，以至于学生和研究者不时会发现自己很难理解其他地方编写的代码。于是， Scheme 各主要实现版本的十五位代表于 1984 年 10 月聚集在一起，以制定一份更好的、能被更广泛接受的 Scheme 语言标准。他们的报告[4]于 1985 年夏天在 MIT 和 Indiana 大学出版。后续的修订版本于 1986 年春天[23]和 1988 年春天[6]发布。现在这份报告反映了 1992 年 6 月在 Xerox PARC 会议上达成一致的进一步修订意见。

我们希望这份报告属于整个 Scheme 社区，因此我们授权大家免费复制它的全部或部分内容。我们特别鼓励 Scheme 的实现者将这份报告用作手册和其他文档的出发点，并在需要时修改报告的内容。

致谢

我们感谢以下人士的帮助： 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 和 Ozan Yigit。我们感谢 Carol Fessenden、 Daniel Friedman 和 Christopher Haynes，他们允许我们使用 Scheme 311 第 4 版参考手册的内容。我们感谢 Texas Instruments 公司允许我们使用 *TI Scheme* 语言参考手册[30]的内容。我们衷心感谢 MIT Scheme[17] 、 T[22]、 Scheme 84[11]、 Common Lisp[27] 和 Algol 60[18] 的手册对本报告的影响。

我们也感谢 Betty Dexter，她在将本报告设为 TEX 格式的工作中做出了杰出的贡献；感谢 Donald Knuth，他设计的程序给 Betty 添了不少麻烦。

MIT 人工智能实验室， Indiana 大学计算机科学系， Oregon 大学计算机与信息科学系和 NEC 研究院为本报告的编撰提供了支持。国防部先进科研项目部门 (ARPA) 在海军科研办公室的 N00014-80-C-0505 号合同范围内为 MIT 的工作提供了部分支持。国家科学基金会 (NSF) 的 NCS 83-04567 和 NCS 83-03325 基金为 Indiana 大学的工作提供了支持。

SCHEME 语言描述

1. Scheme 概论

1.1. 语义

本节概要介绍了 Scheme 的语义。第 3 章到第 6 章给出了详细的、非正式的语义描述。为便于参考，第 7.2 节提供了 Scheme 的形式语义。

像 Algol 语言一样， Scheme 是一种静态作用域的程序设计语言。对变量的每一次使用都对应于该变量在词法上的一个明显的绑定。

Scheme 中采用的是隐式类型而非显式类型。类型与值（也称对象）相关联，而非与变量相关联（一些作者将隐式类型的语言称为弱类型或动态类型的语言）。其他采用隐式类型的语言还包括 APL、 Snobol 和 Lisp 的其他方言。采用显式类型的语言（有时被称为强类型或静态类型的语言）包括 Algol 60、 Pascal 和 C 语言。

在 Scheme 计算过程中创建的所有对象，包括过程和继续 (Continuation)，都拥有无限的生存期 (Extent)。 Scheme 对象从不被销毁。 Scheme 的实现（通常！）不会耗尽空间的原因是，如果它们能证明某个对象无论如何都不会与未来的任何计算发生关联，它们就可以回收该对象占据的空间。其他允许多数对象拥有无限生存期的语言包括 APL 和其他 Lisp 方言。

Scheme 的实现必须支持严格尾递归。这一机制允许迭代计算在常量空间内执行，即便该迭代计算在语法上是用递归过程描述的。借助严格尾递归的实现，迭代就可以用普通的过程调用机制来表示。这样一来，专用的迭代结构就只剩下语法糖衣的用途了。参见第 3.5 节。

Scheme 过程在本质上都是对象。过程可以动态创建，可以存储于数据结构中，可以作为过程的结果返回，等等。其他拥有这些特性的语言包括 Common Lisp 和 ML。

在大多数其他语言中只在幕后起作用的继续，在 Scheme 中也拥有“第一级”状态，这是 Scheme 的一个独树一帜的特征。继续可用于实现大量不同的高级控制结构，如非局部退出 (Non-local exits)、回溯 (Backtracking) 和协作程序 (Coroutine) 等。参见第 6.4 节。

Scheme 过程的参数总以值的方式传递，即无论过程是否需要实参的值，实参表达式都会在过程获得控制权之前被求值。 ML、 C 和 APL是另外三种总以值的方式传递参数的语言。这与 Haskell 语言懒惰求值 (Lazy-evaluation) 的语义，或 Algol 60 语言按名调用 (Call-by-name) 的语义截然不同。在 Haskell 和 Algol 60 的这两种语义中，直到过程需要实参表达式的值时，才会对它们求值。

Scheme 的算术模型被设计为尽量独立于计算机内数值的特定表示方式。在 Scheme 中，每个整数都是一个有理数，每个有理数都是一个实数，每个实数都是一个复数。因此，对于许多程序设计语言来说非常重要的整数和实数运算的差别，在 Scheme 中并不存在。 Scheme 区分精确算术和非精确算术的概念，前者对应于数学上的理想情况，后者则用于表达近似值。像在 Common Lisp 中一样，精确算术不仅限于整数运算。

1.2. 语法

像大多数 Lisp 方言一样， Scheme 将完全括号化的前缀记法用于程序和（其他）数据； Scheme 的语法构成了用于数据的语言的一种子语言。使用这种简单、统一的表达方式的一个重要结果是， Scheme 程序很容易以统一的方式处理 Scheme 程序和数据。例如， `eval` 过程可对以数据形式表示的 Scheme 程序求值。

`read` 过程对其读入的数据做词法分解和语法分析。 `read` 过程将其输入当作数据（第 7.1.2 节）而非程序来解析。

第 7.1 节描述了 Scheme 的形式语法。

1.3. 记法和术语

1.3.1. 基本特征、库特征和可选特征

每个 Scheme 实现必须支持所有未被标记为“可选 (*optional*)”的特征。 Scheme 实现可以自由省略 Scheme 的可选特征，或在不违背本报告中语言定义的情况下增加扩展特征。特别地， Scheme 实现必须提供一种不违背本报告中词法约定的语法模式，以支持可移植的代码。

为了有助于理解和实现 Scheme，一些特征被标记为“库 *(library)*”。 它们可以简单地用基本特征来实现。从严格意义上说它们是多余的，但它们代表了常见的使用模式，因此，我们将它们作为方便的缩略形式提供出来。

1.3.2. 错误状态和未定义行为

当提到一个错误状态时，本报告使用短语“报告一个错误”来表示 Scheme 实现必须检测并报告该错误。如果在讨论错误时没有出现这样的措辞， Scheme 实现就不一定要检测或报告该错误，但我们鼓励 Scheme 实现检测或报告它。一个 Scheme 实现不一定要检测的错误状态通常被简称为“一个错误”。

例如，向一个过程传递该过程没有显式指明要处理的参数是一个错误，尽管本报告很少提到这样的定义域错误。 Scheme 实现可以扩展过程的定义域，以包含这样的参数。

本报告使用短语“可以报告违反了实现约束”来表示这样的情况： Scheme 实现可以报告说，因为实现本身的某些局限，它无法继续执行一个正确的程序。实现约束当然是令人沮丧的，但我们仍鼓励 Scheme 实现报告违反实现约束的情况。

例如，当 Scheme 实现没有足够的空间运行程序时，它可以报告违反了实现约束。

如果我们说一个表达式的值“未定义 (Unspecified) ”，那么该表达式的值应被设为某个对象，同时不报告错误。未定义的表达式的值由 Scheme 实现规定，本报告并不指明应当返回何值。

1.3.3. 条目格式

第 4 章和第 6章依条目组织。每个条目描述一个语言特征或一组相关特征。一个特征或者是一个语法结构，或者是一个内置过程。一个条目以一个或多个标题行开始。对于必备的基本特征，标题行的格式是：

template *category*

对于库特征或可选特征，标题行的格式是：

template *qualifier category*

其中的 *qualifier* 可以是第 1.3.1 节定义的“library”或“optional”。

如果 *category* 为“syntax”，该条目描述的是一个表达式类型，模板 (*template*) 给出了表达式类型的语法。用尖括号括起的语法变量代表表达式的组成单元，例如，⟨expression⟩、⟨variable⟩。语法变量应被理解为指代了程序文本的片断，例如，⟨expression⟩ 代表任何在语法上可构成有效表达式的字符串。以下记法

⟨thing$_1$⟩ ...

代表零个或更多的 ⟨thing⟩。以下记法

⟨thing$_1$⟩ ⟨thing$_2$⟩ ...

代表一个或多个 ⟨thing⟩。

如果 *category* 为“procedure”，那么该条目描述的是一个过程，标题行给出了调用该过程的模板。模板中的参数名以斜体字表示。因此，标题行

`(vector-ref` *vector k*`)` procedure

表示内置过程 `vector-ref`，该过程有两个参数，一个是向量 *vector*，另一个是精确的非负整数 *k*（见下文）。标题行

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

表示过程 `make-vector` 应被定义为拥有一或两个参数。

执行某操作时使用一个未定义的参数是一个错误。为简明起见，我们遵循这样的约定：如果一个参数的名字也是第 3.2 节中列出的某个类型的名字，那么该参数就应是该类型的参数。例如，上面给出的过程 `vector-ref` 的标题行指明，`vector-ref` 的第一个参数应当是一个向量。下面的命名约定同样意味着类型约束：

obj	任意对象
list, $list_1$, ... $list_j$, ...	表（参见第 6.3.2 节）
z, z_1, ... z_j, ...	复数
x, x_1, ... x_j, ...	实数
y, y_1, ... y_j, ...	实数
q, q_1, ... q_j, ...	有理数
n, n_1, ... n_j, ...	整数
k, k_1, ... k_j, ...	精确的非负整数

1.3.4. 求值示例

程序示例中使用的符号“⟹”应被读作“的值为”。例如，

```
(* 5 8)                    ⟹  40
```

意味着表达式 `(* 5 8)` 的值为对象 `40`。或者，更精确地说：在初始环境中，字符序列“`(* 5 8)`”所代表的表达式的值是一个对象，该对象的外部表示为字符序列“`40`”。参见第 3.3 节有关对象的外部表示的讨论。

1.3.5. 命名约定

习惯上，总是返回布尔值的过程名字通常以“`?`”结尾。这样的过程被称为谓词 (Predicate)。

习惯上，将值存入先前分配的存储位置（参见第 3.4 节）的过程名字通常以“`!`”结尾。这样的过程被称为改变过程 (Mutation procedure)。习惯上，改变过程的返回值是未定义的。

习惯上，以某个类型的对象为参数，返回另一个类型的相应对象的过程名字中间包含“`->`”。例如，过程 `list->vector` 以一个表为参数，返回一个向量，该向量中的元素与表中的元素相同。

2. 词法约定

本节给出了编写 Scheme 程序的一些词法约定的非正式说明。关于 Scheme 的形式语法，参见第 7.1 节。

除了在字符和字符串常量中以外，Scheme 从不区分字母的大小写形式。例如，`Foo` 和 `FOO` 是相同的标识符，`#x1AB` 和 `#X1ab` 是相同的数值。

2.1. 标识符

其他程序设计语言认可的大多数标识符也能被 Scheme 接受。构造标识符的精确规则因不同的 Scheme 实现而不同，但在所有 Scheme 实现中，如果字母、数字和“扩展字符”序列中的第一个字符不是任何数值的起始字符，它就是一个标识符。此外，`+`、`-` 和 `...` 都是标识符。这里有一些标识符的例子：

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

扩展字符可以像字母那样用于标识符内。以下是扩展字符：

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

参见第 7.1.1 节中的标识符形式语法。

在 Scheme 程序中，标识符有两个用处：

- 标识符可被用作一个变量或一个语法关键字 （参见第 3.1 和第 4.3 节）。
- 当标识符作为常量或在常量内部出现时（参见第 4.1.2 节），它表示一个符号（*symbol*，参见第 6.3.3 节）。

2.2. 空白和注释

空白 (*Whitespace*) 字符包括空格和换行（Scheme 实现通常支持附加的空白字符，如制表符和分页符）。空白的作用是提高可读性，分隔不同的记号（Token，指不可分的词法单元，如标识符或数值），但没有实际的意义。空白可以出现在任意两个记号之间，但不能出现在记号内部。空白也可以出现在字符串内，这时，空白是有意义的。

分号 (;) 表示注释的开始。 注释一直延续到出现分号的那一行的结尾。注释对 Scheme 来说是不可见的，但该行的结尾是一个可见的空白符，这可以防止注释出现在标识符或数值的中间。

```
;;; The FACT procedure computes the factorial
;;; of a non-negative integer.
(define fact
  (lambda (n)
    (if (= n 0)
        1        ;Base case: return 1
        (* n (fact (- n 1))))))
```

2.3. 其他记法

关于数值的记法，参见第 6.2 节的描述。

`. + -` 这些符号用于计数，同时也可以出现在标识符中除第一个字符以外的任何位置。单独的加号或减号自身也是标识符。单独的句点（没有出现在数值或标识符内）用于表示点对（Pair，第 6.3.2 节），或在形参列表中表示剩余参数（第 4.1.4 节）。连续三个句点组成的独立串也是一个标识符。

`( )` 括号用于表示组合和描述表（第 6.3.2 节）。

`'` 单引号用于表示常量数据（第 4.1.2 节）。

`` ` `` 反引号用于表示近乎常量的数据（第 4.2.6 节）。

`, ,@` 逗号，以及逗号和 @ 符号组成的序列与反引号一同使用（第 4.2.6 节）。

`"` 双引号用于界定字符串（第 6.3.5 节）。

`\` 反斜线用于字符常量的语法（第 6.3.4 节），或用作字符串常量中的转义符（第 6.3.5 节）。

`[ ] { } |` 左右方括号、左右大括号以及竖线符号被保留给未来可能的语言扩展。

`#` 井号有许多用途，具体取决于其后紧跟的字符是什么。

`#t #f` 这些是布尔常量（第 6.3.1 节）。

`#\` 引入字符常量（第 6.3.4 节）。

`#(` 引入向量常量（第 6.3.6 节）。向量常量由 `)` 终止。

`#e #i #b #o #d #x` 这些符号用于表示数值（第 6.2.4 节）。

3. 基本概念

3.1. 变量、语法关键字和作用域

一个标识符可以命名一个语法类型，或命名一个值的存储位置。命名某语法类型的标识符被称为语法关键字，也就是说它与该语法绑定在一起。命名某存储位置的标识符被称为变量，即它与该存储位置绑定在一起。对程序的某一点有效的所有可见绑定的集合，被称为对该点有效的环境。存储在某变量所绑定的存储位置的值被称为该变量的值。因为术语上的混乱，变量有时也被说成是对值的命名或对值的绑定，这种说法并不十分准确，但很少会引起混淆。

某些表达式类型可用于创建新的语法种类，并将语法关键字绑定到新语法。另一些表达式类型可用于创建新的存储位置，并将变量绑定到那些存储位置。这些表达式类型被称为绑定结构。第 4.3 节列举了绑定语法关键字的绑定结构。最基本的变量绑定结构是 `lambda` 表达式，因为所有其他的变量绑定结构都可以用 `lambda` 表达式来描述。其他变量绑定结构包括 `let`、 `let*`、 `letrec` 和 `do` 表达式（参见第 4.1.4、 4.2.2 和 4.2.4 节）。

Scheme 像 Algol 和 Pascal 一样，是一种支持块结构的静态作用域语言，这与除 Common Lisp 外的其他大多数 Lisp 方言不同。标识符在程序中的每一个绑定位置都对应于一个程序文本的作用域，该绑定仅在该作用域内可见。作用域由创建绑定的特殊绑定结构决定。例如，如果绑定由 `lambda` 表达式创建，它的作用域就是整个 `lambda` 表达式。每个对标识符的引用都指向相应的标识符绑定，该绑定拥有包含该引用的最内层作用域。如果找不到作用域包含该引用的标识符绑定，且最高层环境中存在相应的变量绑定时，该引用就会指向最高层环境中的变量绑定（第 4 章，第 6 章）；如果找不到标识符的绑定，它就被称为未绑定的。

3.2. 类型的互不相交性

没有任何对象可以同时满足下面所列的两个或两个以上的谓词：

```
boolean?          pair?
symbol?           number?
char?             string?
vector?           port?
procedure?
```

这些谓词定义了布尔 *(boolean)*、点对 *(pair)*、符号 *(symbol)*、数值 *(number)*、字符 *(char，或 character)*、字符串 *(string)*、向量 *(vector)*、端口 *(port)* 和过程 *(procedure)* 类型。空表是一个隶属于其自身类型的特殊对象，它不能满足上述任何一个谓词。

尽管有一个独立的布尔类型， Scheme 中的任何值都可以作为布尔值参与条件判断。如第 6.3.1 节所描述的那样，在条件判断中，除了 `#f` 以外，所有值都被视为真。本报告使用“真”来表示除了 `#f` 以外的所有 Scheme 值，使用“假”来表示 `#f`。

3.3. 外部表示

Scheme（和 Lisp）中的一个重要概念是对象的外部表示。外部表示是一个字符序列。例如，整数 28 的外部表示是字符序列“`28`”，包含整数 8 和 13 的表的外部表示是字符序列“`(8 13)`”。

一个对象的外部表示并不一定是惟一的。整数 28 也可以表示为“`#e28.000`”和“`#x1c`”，上一段提到的表也可以表示为“`( 08 13 )`”和“`(8 . (13 . ()))`”（参见第 6.3.2 节）。

许多对象都拥有标准的外部表示，但一些对象，如过程，没有标准的外部表示（尽管特定的 Scheme 实现可为它们定义外部表示）。

外部表示可以被写入程序代码，以便获取相应的对象（参见 `quote`，第 4.1.2 节）。

外部表示还可用于输入和输出。过程 `read` （第 6.6.2 节）解析外部表示，过程 `write`（第 6.6.3 节）生成外部表示，它们共同提供了优雅和强大的输入/输出机制。

注意字符序列“`(+ 2 6)`”不是整数 8 的外部表示，即便它是一个结果为整数 8 的表达式；更准确地说，它是一个拥有三个元素的表的外部表示，该表中的元素是符号 `+` 和整数 2、 6。 Scheme 的语法拥有这样的特性：任何构成表达式的字符序列也同时是某个对象的外部表示。这有可能导致混淆，因为在脱离上下文的情况下，我们不清楚一个给定的字符序列所表示的到底是数据还是程序；但这也是 Scheme 之所以强大的一个原因，因为它可以使我们更容易地编写出将程序作为数据处理（反之亦然）的程序，如解释器或编译器。

第 6 章内的相关小节在描述对象处理的基本特征的同时，也给出了不同种类对象的外部表示语法。

3.4. 存储模型

点对、向量、字符串等变量和对象潜在地代表了存储位置或存储位置的序列。例如，一个字符串表示多个存储位置，其数量与字符串中的字符数量相同（这些存储位置不需要对应于完整的机器字）。可以用 `string-set!` 过程将新的值存入其中的一个存储位置，但该字符串仍对应于原先那一组存储位置。

通过变量引用或诸如 `car`、 `vector-ref`、 `string-ref` 等过程从一个存储位置获取的对象，在 `eqv?`（第 6.1 节）的语义下，与此前最后存入该存储位置的对象相等。

每个存储位置都被加上了是否正被使用的标记。变量或对象永远不会指向未被使用的存储位置。只要本报告提及为某个变量或对象分配存储位置，就意味着，在未被使用的存储位置集合中选出适当数量的存储位置，并在变量或对象指向它们之前，就为这些存储位置加上标记，以表明它们现在处于被使用的状态。

在许多系统中，人们希望在只读内存中存储常量（即常量表达式的值）。为了表示这样的语义，不妨设想每个表示存储位置的对象都与一个标记相关联，该标记表示对象是可变的或不可变的。在这样的系统中，常量和 `symbol->string` 过程返回的字符串是不可变的对象，本报告列举的其他过程创建的所有对象都是可变的对象。试图将一个新值存入不可变对象对应的存储位置是一个错误。

3.5. 严格尾递归

Scheme 实现必须是严格尾递归的。出现在下面定义的语法上下文中的过程调用是“尾调用”。如果 Scheme 实现支持无限数量的活动尾调用，它就是严格尾递归的。如果一个被调用过程仍可以返回，该调用就是活动的。注意，这里所说的返回既包括了通过当前继续从调用中返回的情况，也包括了事先以 `call-with-current-continuation` 过程捕获继续，后来再调用该继续以便从调用中返回的情况。不存在被捕获的继续时，调用最多只能返回一次，活动调用就是那些还没有返回的调用。 [8]中提供了严格尾递归的正式定义。

原理：

直观上，活动尾调用不需要任何存储空间，因为尾调用中使用的继续在语义上与传给包含该调用的过程的继续相同。即便一个错误的实现可能在该调用中使用了一个新的继续，但在返回到新的继续后，它也会立即返回到传入此过程的继续。一个严格尾递归的实现会直接返回到传入此过程的继续。

严格尾递归是 Steele 和 Sussman 最初版本的 Scheme 中的核心概念之一。第一个 Scheme 解释器既实现了函数，也实现了参与者 (Actor)。控制流用参与者来表示。参与者和函数的不同在于，参与者将其结果传给另一个参与者，而非返回给调用者。用本节的术语来说，每个参与者最终都通过尾调用访问另一个参与者。

Steele 和 Sussman 后来注意到，在他们的解释器里，处理参与者的代码与处理函数的代码是一样的，完全没有必要在语言中同时包含这两种特征。

尾调用是发生在尾上下文 *(Tail context)* 的过程调用。尾上下文通过归纳法来定义。注意，对于特定的 lambda 表达式，尾上下文总是确定的。

- lambda 表达式内部的最后一个表达式，即下面记法中的 ⟨tail expression⟩，位于尾上下文中。

  `(lambda` ⟨formals⟩
  ⟨definition⟩* ⟨expression⟩* ⟨tail expression⟩`)`

- 如果以下表达式位于尾上下文中，则其中用⟨tail expression⟩表示的子表达式位于尾上下文中。这些语法来自第 7 章中给出的语法规则，并将其中的某些⟨expression⟩换成了⟨tail expression⟩。这里只显示那些包含尾上下文的语法规则。

  `(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 (`⟨binding spec⟩`*)` ⟨tail body⟩`)`
  `(let` ⟨variable⟩ `(`⟨binding spec⟩`*)` ⟨tail body⟩`)`
  `(let* (`⟨binding spec⟩`*)` ⟨tail body⟩`)`
  `(letrec (`⟨binding spec⟩`*)` ⟨tail body⟩`)`

  `(let-syntax (`⟨syntax spec⟩`*)` ⟨tail body⟩`)`
  `(letrec-syntax (`⟨syntax spec⟩`*)` ⟨tail body⟩`)`

  `(begin` ⟨tail sequence⟩`)`

  `(do (`⟨iteration spec⟩`*)`
  `(`⟨test⟩ ⟨tail sequence⟩`)`
  ⟨expression⟩`*)`

  其中

  ⟨cond clause⟩ ⟶ `(`⟨test⟩ ⟨tail sequence⟩`)`
  ⟨case clause⟩ ⟶ `((`⟨datum⟩`*)` ⟨tail sequence⟩`)`

  ⟨tail body⟩ ⟶ ⟨definition⟩`*` ⟨tail sequence⟩
  ⟨tail sequence⟩ ⟶ ⟨expression⟩`*` ⟨tail expression⟩

- 如果 `cond` 表达式位于尾上下文中，并拥有 `(`⟨expression$_1$⟩ `=>` ⟨expression$_2$⟩`)` 形式的子句，那么，对于⟨expression$_2$⟩的结果过程的（隐含）调用位于尾上下文中。⟨expression$_2$⟩本身不在尾上下文中。

某些内置过程也需要执行尾调用。传递给 `apply` 和 `call-with-current-continuation` 的第一个参数，以及传递给 `call-with-values` 的第二个参数，都必须通过尾调用来调用。同样的， `eval` 在对参数求值时，必须做得就像这个参数位于eval过程内的尾位置一样。

在下面的示例中，对 `f` 的调用是惟一的尾调用。对 `g` 和 `h` 的调用都不是尾调用。对 `x` 的引用位于尾上下文中，但它并不是一个调用，所以它不是尾调用。

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

注： Scheme 实现可以（但不是必须）认为，某些非尾调用（如上述代码中对 `h` 的调用）可被当作尾调用来求值。在上面的示例中， `let` 表达式可被编译为一个对 `h` 的尾调用（我们可以忽略掉 `h` 返回非预期个数的值的可能性，因为在这种情况下，`let` 的结果明显是未定义的或依赖于实现的）。

4. 表达式

表达式类型分为基本类型和派生类型两种。基本表达式类型包括变量和过程调用。派生表达式类型在语义上不是基本类型，但可以被定义为宏。除了宏定义比较复杂的 `quasiquote` 表达式以外，派生表达式均被归类为库特征。第 7.3 节给出了派生表达式的相关定义。

4.1. 基本表达式类型

4.1.1. 变量引用

⟨variable⟩ syntax

由一个变量（第 3.1 节）构成的表达式是变量引用。变量引用的值是存储在变量所绑定的存储位置的值。引用一个未绑定的变量是一个错误。

```
(define x 28)
x                                ⟹  28
```

4.1.2. 常量表达式

`(quote` ⟨datum⟩`)` syntax
`'`⟨datum⟩ syntax
⟨constant⟩ syntax

`(quote` ⟨datum⟩`)` 的值为 ⟨datum⟩。 ⟨datum⟩ 可以是 Scheme 对象的任何外部表示（参见第 3.3 节）。此记法可以将常量嵌入到 Scheme 代码中。

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

`(quote` ⟨datum⟩`)` 可简写为 `'`⟨datum⟩。在任何情况下，两种记法都是等价的。

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

数值常量、字符串常量、字符常量和布尔常量的值是“它们自身”，它们不需要被引用。

```
'"abc"                      ⟹  "abc"
"abc"                       ⟹  "abc"
'145932                     ⟹  145932
145932                      ⟹  145932
'#t                         ⟹  #t
#t                          ⟹  #t
```

如第 3.4 节所述，使用 `set-car!`、`string-set!` 等改变过程改变常量（即常量表达式的值）是一个错误。

4.1.3. 过程调用

(⟨operator⟩ ⟨operand$_1$⟩ ...) syntax

简单地用括号表达式括起被调用过程及传入其中的参数即构成了过程调用。Scheme 实现（以未定义的顺序）求得运算符和运算对象表达式的值，并将结果参数传入结果过程。

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

在初始环境里，许多过程已经作为变量的值而存在。例如，上面例子中的加法和乘法过程是变量 `+` 和 `*` 的值。通过对 `lambda` 表达式求值可以创建新的过程（参见第 4.1.4 节）。

过程调用可以返回任意数量的值（参见第 6.4 节中有关 `values` 的内容）。除了 `values` 以外，初始环境中的过程或者返回一个值，或者像 `apply` 等过程那样，调用自己的一个参数并返回其返回值。

过程调用也被称为组合式 *(Combinations)*。

注： 与其他 Lisp 方言不同，Scheme 过程调用中的求值顺序是未定义的，对运算符表达式和运算对象表达式总采用同样的求值规则。

注： 尽管求值顺序是未定义的，但对运算符和运算对象表达式进行的任何并发求值的结果必须与按某种顺序求值的结果一致。对每个过程调用的求值顺序可以有不同的选择。

注： 在许多 Lisp 方言中，空组合式 `()` 是合法的表达式。在 Scheme 中，组合式中至少要包含一个子表达式，因此 `()` 不是一个语法上有效的表达式。

4.1.4. 过程

`(lambda` ⟨formals⟩ ⟨body⟩`)` syntax

语法：⟨formals⟩ 应为一个本节后面描述的形参列表，⟨body⟩ 应为一个或多个表达式组成的序列。

语义：`lambda` 表达式的值是一个过程。对 `lambda` 表达式求值时起作用的环境也被记为该过程的一部分。随后，当使用实参调用该过程时，`lambda` 表达式求值时使用的环境将被扩展，扩展后的环境包含了形参列表中的变量到新存储位置的绑定，相应的实参值被存入那些新的存储位置。在扩展后的环境中，`lambda` 表达式的 ⟨body⟩ 内的所有表达式被顺序求值。⟨body⟩ 内最后一个表达式的（一个或多个）结果将作为过程调用的（一个或多个）结果返回。

```
(lambda (x) (+ x x))        ⟹  一个过程
((lambda (x) (+ x x)) 4)    ⟹  8

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

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

⟨formals⟩ 应具有下面几种形式之一：

- (⟨variable$_1$⟩ ...): 过程拥有固定数量的参数。当过程被调用时，参数将被存储在相应变量的绑定中。
- ⟨variable⟩: 过程拥有任意数量的参数。当过程被调用时，实参的序列被转换为一个新创建的表，该表存储在 ⟨variable⟩ 的绑定中。
- (⟨variable$_1$⟩ ... ⟨variable$_n$⟩ `.` ⟨variable$_{n+1}$⟩): 如果一个由空格分隔的句点出现在最后一个变量之前，该过程就拥有 n 个或更多个参数，这里的 n 是句点前面形参的个数（至少要有一个）。存储在最后一个参数绑定中的值是一个新创建的表。除了已和其他形参匹配的所有其他实参外，剩余的实参都被存入该表中。

在 ⟨formals⟩ 中，同一个 ⟨variable⟩ 出现两次或多次是一个错误。

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

每个作为 `lambda` 表达式的结果创建出来的过程（在概念上）都对应于一个存储位置，以便使 `eqv?` 和 `eq?` 能够作用于过程（参见第 6.1 节）。

4.1.5. 条件表达式

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

语法：⟨test⟩、⟨consequent⟩ 和 ⟨alternate⟩ 可以是任意的表达式。

语义：`if` 表达式的求值方式如下：首先，求得 ⟨test⟩ 的值。如果得到的值为真（参见第 6.3.1 节），则 ⟨consequent⟩ 被求值，其（一个或多个）结果被返回；否则，⟨alternate⟩ 被求值，其（一个或多个）结果被返回。如果 ⟨test⟩ 的值为假且没有定义 ⟨alternate⟩，表达式的结果就是未定义的。

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

4.1.6. 赋值

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

对 ⟨expression⟩ 求值后的结果被存入 ⟨variable⟩ 所绑定的存储位置。⟨variable⟩ 必须被绑定到某个包含 `set!` 表达式的作用域，或者被绑定到最高层作用域。`set!` 表达式的结果是未定义的。

```
(define x 2)
(+ x 1)                     ⟹  3
(set! x 4)                  ⟹  未定义
(+ x 1)                     ⟹  5
```

4.2. 派生表达式类型

本节中的语法结构是“卫生的”，见第 4.3 节的相关讨论。作为参考，第 7.3 节给出了可以将本节描述的大部分语法结构转换为前面一节定义的基本语法结构的宏定义。

4.2.1. 条件表达式

`(cond` ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...`)` library syntax

语法：每个 ⟨clause⟩ 的格式为：

(⟨test⟩ ⟨expression$_1$⟩ ...)

其中的 ⟨test⟩ 是任意表达式。或者，⟨clause⟩ 的格式也可以是：

(⟨test⟩ `=>` ⟨expression⟩)

最后一个 ⟨clause⟩ 可以是一个“else 子句”，其格式是：

`(else` ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)`.

语义：`cond` 表达式的求值方式是：依次求得每一个 ⟨clause⟩ 中的 ⟨test⟩ 表达式的值，直到其中一个的值是真为止（参见第 6.3.1 节）。当一个 ⟨test⟩ 表达式的值为真时，该 ⟨clause⟩ 内余下的 ⟨expression⟩ 被依次求值，⟨clause⟩ 内最后一个 ⟨expression⟩ 的（一个或多个）结果被作为整个 `cond` 表达式的（一个或多个）结果返回。如果被选中的 ⟨clause⟩ 只包含 ⟨test⟩ 而不包含 ⟨expression⟩，⟨test⟩ 的值就作为结果返回。如果被选中的 ⟨clause⟩ 使用 `=>` 这样的可选格式，⟨expression⟩ 就会被求值。这时，⟨expression⟩ 的值必须是一个只接受一个参数的过程，该过程随即被调用，⟨test⟩ 的值被传入该过程，过程返回的（一个或多个）值作为 `cond` 表达式的值返回。如果所有 ⟨test⟩ 的值都为假，且没有 else 子句，条件表达式的结果就是未定义的；如果有 else 子句，那么该子句中的 ⟨expression⟩ 被求值，最后一个 ⟨expression⟩ 的（一个或多个）结果被返回。

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

`(case` ⟨key⟩ ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...`)` library syntax

语法：⟨key⟩ 可以是任意表达式。每个 ⟨clause⟩ 的格式为

((⟨datum$_1$⟩ ...) ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...),

其中的 ⟨datum⟩ 是某个对象的外部表示。所有的 ⟨datum⟩ 必须互不相同。最后一个 ⟨clause⟩ 可以是“else 子句”，其格式为：

`(else` ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)`.

语义：`case` 表达式的求值方式是：先对 ⟨key⟩ 求值，将其结果与每个 ⟨datum⟩ 进行比较，如果 ⟨key⟩ 的求值结果等于（在 `eqv?` 的语义下；参见第 6.1 节）某个 ⟨datum⟩，相应的 ⟨clause⟩ 中的表达式就被从左至右顺序求值，⟨clause⟩ 中最后一个表达式的（一个或多个）结果作为 `case` 表达式的（一个或多个）结果返回。如果 ⟨key⟩ 的求值结果与每个 ⟨datum⟩ 都不相等，那么，当存在 else 子句时，该子句中的表达式被求值，最后一个表达式的（一个或多个）结果就是 `case` 表达式的结果；否则，`case` 表达式的结果是未定义的。

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

`(and` ⟨test$_1$⟩ ...`)` library syntax

⟨test⟩ 表达式被从左至右求值，第一个结果为假（参见第 6.3.1 节）的表达式的值被返回，剩下的所有表达式均不参与求值。如果所有表达式的值均为真，则返回最后一个表达式的值。如果没有表达式，则返回 `#t`。

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

`(or` ⟨test$_1$⟩ ...`)` library syntax

⟨test⟩ 表达式被从左至右求值，第一个结果为真（参见第 6.3.1 节）的表达式的值被返回，剩下的所有表达式均不参与求值。如果所有表达式的值均为假，则返回最后一个表达式的值。如果没有表达式，则返回 `#f`。

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

4.2.2. 绑定结构

三种绑定结构 `let`、 `let*` 和 `letrec` 为 Scheme 提供了像 Algol 60 那样的块结构。三种结构的语法形式相同，但它们为各自的变量绑定所创建的作用域是不同的。 `let` 表达式在所有变量被绑定前计算初始值； `let*` 表达式顺序完成绑定和求值；在 `letrec` 表达式计算初始值的过程中，所有绑定都会起作用，这使相互递归 (Mutually recursive) 的定义成为了可能。

`(let` ⟨bindings⟩ ⟨body⟩`)` library syntax

语法：⟨bindings⟩ 的格式为：

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...),

其中， ⟨init⟩ 是一个表达式， ⟨body⟩ 是一个或多个表达式的序列。在被绑定的变量列表中，同一个 ⟨variable⟩ 出现两次或多次是一个错误。

语义：在当前环境中，所有 ⟨init⟩ （以某种未定义的顺序）求值，所有 ⟨variable⟩ 被绑定到存储了 ⟨init⟩ 的结果的新存储位置。⟨body⟩ 在扩展后的环境中求值， ⟨body⟩ 中最后一个表达式的（一个或多个）结果被返回。每个 ⟨variable⟩ 的绑定都将 ⟨body⟩ 当作自己的作用域。

```
(let ((x 2) (y 3))
  (* x y))                  ⟹  6

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

另见“命名的 `let`”，第 4.2.4 节。

`(let*` ⟨bindings⟩ ⟨body⟩`)` library syntax

语法：⟨bindings⟩ 的格式为：

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...),

⟨body⟩ 是一个或多个表达式的序列。

语义： `let*` 和 `let` 相似，但绑定是按照从左至右的顺序完成的，(⟨variable⟩ ⟨init⟩) 所指明的绑定的作用域是 `let*` 表达式中该绑定右边的部分。 因此，第二个绑定是在第一个绑定可见的环境中完成的，依此类推。

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

`(letrec` ⟨bindings⟩ ⟨body⟩`)` library syntax

语法：⟨bindings⟩ 的格式为：

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...),

⟨body⟩ 是一个或多个表达式的序列。在被绑定的变量列表中，同一个 ⟨variable⟩ 出现两次或多次是一个错误。

语义：⟨variable⟩ 被绑定到存有未定义值的新存储位置。⟨init⟩ 在结果环境中（以未定义的顺序）被求值，每个 ⟨init⟩ 的求值结果被赋给每个相应的 ⟨variable⟩。 ⟨body⟩ 在结果环境中被求值， ⟨body⟩ 中最后一个表达式的（一个或多个）值被返回。每个 ⟨variable⟩ 的绑定都把整个 `letrec` 表达式当作自己的作用域，这使得定义相互递归的过程成为了可能。

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

有关 `letrec` 的一个限制非常重要：在没有指明或引用任何 ⟨variable⟩ 的值的情况下，必须能求得每个 ⟨init⟩ 的值。违反此限制的情况是一个错误。此限制的必要性在于，Scheme 是按值而不是按名字传递参数的。在 `letrec` 的最常见用法里，所有 ⟨init⟩ 都是 `lambda` 表达式，这自然能满足上述限制条件。

4.2.3. 顺序结构

(begin ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...) library syntax

⟨expression⟩ 被从左至右顺序求值，最后一个 ⟨expression⟩ 的（一个或多个）值被返回。这种表达式类型用于顺序执行有“副作用 (Side effect)”的行为，如输入和输出。

```
(define x 0)

(begin (set! x 5)
       (+ x 1))              ⟹  6

(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))    ⟹  未定义
           并打印输出  4 plus 1 equals 5
```

4.2.4. 迭代

(do ((⟨variable$_1$⟩ ⟨init$_1$⟩ ⟨step$_1$⟩) library syntax
...)
(⟨test⟩ ⟨expression⟩ ...)
⟨command⟩ ...)

do 是一种迭代结构。它绑定了一组变量，指明了开始时它们如何被初始化，以及每一次迭代时它们如何被更新。当终止条件满足时，循环在所有 ⟨expression⟩ 被求值后退出。

do 表达式的求值方式为：⟨init⟩ 表达式（以某种未定义的顺序）被求值，⟨variable⟩ 被绑定到新分配的存储空间，⟨init⟩ 表达式的结果被存入 ⟨variable⟩ 的绑定中，然后开始迭代过程。

每个迭代过程都先求得 ⟨test⟩ 的值。如果结果为假（参见第 6.3.1 节），则 ⟨command⟩ 表达式被顺序求值，⟨step⟩ 表达式以某种未定义的顺序被求值，⟨variable⟩ 被绑定到新分配的存储位置，⟨step⟩ 的结果被存入 ⟨variable⟩ 的绑定中，然后开始下一次迭代。

如果 ⟨test⟩ 的值为真，则 ⟨expression⟩ 被从左至右求值，最后一个 ⟨expression⟩ 的（一个或多个）值被返回。如果没有给出 ⟨expression⟩，则 do 表达式的值是未定义的。

⟨variable⟩ 的绑定的作用域包括除了 ⟨init⟩ 以外的整个 do 表达式。在 do 的变量列表中，同一个 ⟨variable⟩ 出现两次或多次是一个错误。

⟨step⟩ 可以省略，在这种情况下，表达式的结果就像我们把 (⟨variable⟩ ⟨init⟩) 写成 (⟨variable⟩ ⟨init⟩ ⟨variable⟩) 的时候一样。

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

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

(let ⟨variable⟩ ⟨bindings⟩ ⟨body⟩) library syntax

“命名的 let”是 let 语法的变形，它提供了一种比 do 更有普遍性的循环结构，也可用于表示递归。它和普通的 let 在语法和语义上的惟一不同之处在于，在 ⟨body⟩ 中，⟨variable⟩ 被绑定到了一个过程，该过程的形参是那些被绑定的变量，该过程的过程体是 ⟨body⟩。这样，我们就可以通过对名为 ⟨variable⟩ 的过程的调用而重复执行 ⟨body⟩。

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

4.2.5. 推迟求值

(delay ⟨expression⟩) library syntax

delay 结构与 force 过程一同使用，以实现懒惰求值或按需调用 (Call by need)。(delay ⟨expression⟩) 返回一个叫“承诺 (Promise)”的对象，在未来某个时候，（force 过程）可以要求该对象对 ⟨expression⟩ 求值并提供其结果。⟨expression⟩ 返回多个值的后果是未定义的。

参见 force 的描述（第 6.4 节），以得到有关 delay 的更完整描述。

4.2.6. 准引用

(quasiquote ⟨qq template⟩) syntax
`⟨qq template⟩ syntax

“反引用 (Backquote)”或“准引用 (Quasiquote)”表达式用于在已知大部分（但不是全部）目标结构的情况下创建表或向量结构。如果 ⟨qq template⟩ 中没有出现逗号，`⟨qq template⟩ 的求值结果与 '⟨qq template⟩ 的求值结果相同。但是，如果 ⟨qq template⟩ 中出现了逗号，逗号后的表达式就会被求值（“解除引用，unquoted”），其结果代替逗号和表达式被插入到结构中。如果逗号后面紧跟着一个 @ 符号，后面的表达式就被视为一个表，该表的前后括号随即被“剥离”，表中的元素被插入到 ,@ 及其后的表达式所在的位置。,@ 只能出现在表和向量的 ⟨qq template⟩ 中。

```
`(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)
```

准引用格式可以嵌套。置换操作只作用于那些与最外层反引用有同样嵌套级别的解除引用的元素。每进入一个后续的准引用，嵌套级别就增加一，每进入一个解除引用的语法单元，嵌套级别就减少一。

```
`(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)
```

`⟨qq template⟩ 和 (quasiquote ⟨qq template⟩) 这两种记法在所有情况下都是等价的。,⟨expression⟩ 与 (unquote ⟨expression⟩) 等价。,@⟨expression⟩ 与 (unquote-splicing ⟨expression⟩) 等价。对于包含两个元素，第一个元素是这些符号之一的表，write 生成的外部语法在不同的 Scheme 实现中可能不同。

```
(quasiquote (list (unquote (+ 1 2)) 4))
          ⟹  (list 3 4)
'(quasiquote (list (unquote (+ 1 2)) 4))
          ⟹  `(list ,(+ 1 2) 4)
     即 (quasiquote (list (unquote (+ 1 2)) 4))
```

如果 ⟨qq template⟩ 中的 quasiquote、unquote 或 unquote-splicing 符号出现在不符合上述规定的其他位置，结果是不可预知的。

4.3. 宏

Scheme 程序可以定义和使用新的派生表达式类型，它们被称为宏 *(Macros)*。由程序定义的表达式类型的语法为：

(⟨keyword⟩ ⟨datum⟩ ...)

其中 ⟨keyword⟩ 是能惟一确定该表达式类型的标识符。这个标识符被称为宏的语法关键字，或简称为关键字。⟨datum⟩ 的个数及其语法由该表达式类型确定。

宏的每个应用实例被称为对该宏的一次使用 *(Use)*，有关如何将宏的使用转译为更基本的表达式的一组规则被称为宏的转换器 *(Transformer)*。

宏定义机制包含两个部分：

- 一组表达式，用于设定哪些标识符是宏关键字，将它们与宏转换器相关联，并控制宏定义所在的作用域。
- 一种定义宏转换器的模式语言。

宏的语法关键字可以屏蔽变量绑定，局部变量绑定也可以屏蔽关键字绑定。所有使用模式语言定义的宏都是“卫生的”和“引用透明的”，这保持了 Scheme 的词法作用域的特点 [14, 15, 2, 7, 9]：

- 如果一个宏转换器为某个标识符（变量或关键字）插入了一个绑定，就实际效果而言，该标识符在其整个作用域中将被改名使用，以避免与其他标识符冲突。注意，最高层的 define 可以引入，也可以不引入绑定，参见第 5.2 节。
- 如果一个宏转换器插入了对某标识符的自由引用，那么，无论是否存在包围该宏的使用的局部绑定，该引用都将指向定义转换器时可见的绑定。

4.3.1. 语法关键字的绑定结构

Let-syntax 和 letrec-syntax 与 let 和 letrec 类似，但它们不是将变量绑定到存储值的位置，而是将语法关键字绑定到宏转换器。也可以在最高层绑定语法关键字，参见第 5.3 节。

(let-syntax ⟨bindings⟩ ⟨body⟩) syntax

语法：⟨bindings⟩ 的格式为：

((⟨keyword⟩ ⟨transformer spec⟩) ...)

每个 ⟨keyword⟩ 是一个标识符，每个 ⟨transformer spec⟩ 是一个 syntax-rules 的实例，⟨body⟩ 应为一个或多个表达式的序列。在被绑定的关键字列表中，同一个 ⟨keyword⟩ 出现两次或多次是一个错误。

语义：扩展 let-syntax 表达式的语法环境，使其包含关键字为 ⟨keyword⟩ 且绑定到特定转换器的宏。在扩展后的语法环境中展开 ⟨body⟩。每个 ⟨keyword⟩ 的绑定都将 ⟨body⟩ 当作自己的作用域。

```
(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
```

(letrec-syntax ⟨bindings⟩ ⟨body⟩) syntax

语法：与 let-syntax 相同。

语义：扩展 letrec-syntax 表达式的语法环境，使其包含关键字为 ⟨keyword⟩ 且绑定到特定转换器的宏。在扩展后的语法环境中展开 ⟨body⟩。每个 ⟨keyword⟩ 绑定的作用域都包括 ⟨bindings⟩ 和 ⟨body⟩，这样，转换器就可以将表达式转译为 letrec-syntax 表达式引入的宏的使用。

```
(letrec-syntax
  ((my-or (syntax-rules ()
            ((my-or) #f)
```

```
            ((my-or e) e)
            ((my-or e1 e2 ...)
             (let ((temp e1))
               (if temp
                   temp
                   (my-or e2 ...)))))))
(let ((x #f)
      (y 7)
      (temp 8)
      (let odd?)
      (if even?))
  (my-or x
         (let temp)
         (if y)
         y)))            ⟹  7
```

4.3.2. 模式语言

⟨transformer spec⟩ 的格式为：

`(syntax-rules` ⟨literals⟩ ⟨syntax rule⟩ ...`)`

语法：⟨literals⟩ 是由标识符组成的表，每个 ⟨syntax rule⟩ 的格式为：

(⟨pattern⟩ ⟨template⟩)

⟨syntax rule⟩ 中的模式 ⟨pattern⟩ 是以该宏的关键字开头的表 ⟨pattern⟩ 。

⟨pattern⟩ 或者是一个标识符，或者是一个常量，或者是下面记法之一：

(⟨pattern⟩ ...)
(⟨pattern⟩ ⟨pattern⟩ ... . ⟨pattern⟩)
(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩)
#(⟨pattern⟩ ...)
#(⟨pattern⟩ ... ⟨pattern⟩ ⟨ellipsis⟩)

模板 ⟨template⟩ 或者是一个标识符，或者是一个常量，或者是下面记法之一：

(⟨element⟩ ...)
(⟨element⟩ ⟨element⟩ ... . ⟨template⟩)
#(⟨element⟩ ...)

其中的 ⟨element⟩ 是一个后面跟着可选的 ⟨ellipsis⟩ 的 ⟨template⟩，⟨ellipsis⟩ 是标识符“...”（它不能作为标识符出现在模板或模式中）。

语义：`syntax-rules` 的实例通过定义“卫生的”改写规则序列，生成一个新的宏转换器。关键字与 `syntax-rules` 定义的转换器相关联的宏的使用，将与 ⟨syntax rule⟩ 中的模式相匹配。其匹配方法是，从最左边的 ⟨syntax rule⟩ 开始，当发现匹配时，宏的使用就被“卫生地”依照模板转译。

除了模式开头的关键字、罗列在 ⟨literals⟩ 中的标识符或“...”标识符以外，出现在 ⟨syntax rule⟩ 的模式中的标识符都是模式变量。模式变量与任意输入元素匹配，用于在模板中指代输入元素。在 ⟨pattern⟩ 中，同样的模式变量出现两次或多次是一个错误。

⟨syntax rule⟩ 中，模式开头的关键字不参与匹配，也不能被看作是模式变量或字面标识符 (Literal identifier)。

原理： 关键字的作用域由将其绑定到相关宏转换器的表达式或语法定义决定。假如关键字是一个模式变量或字面标识符，那么，无论关键字是由 `let-syntax` 还是由 `letrec-syntax` 绑定的，模式后面的模板都将出现在它的作用域内。

出现在 ⟨literals⟩ 中的标识符被解释为字面标识符，字面标识符与输入中相应的子形式 (Subform) 匹配。输入中的子形式与一个字面标识符相匹配的充分必要条件是，子形式是一个标识符且满足下面两个条件之一：子形式在宏表达式中出现和在宏定义中出现时拥有相同的词法绑定，或者两个标识符相同且都没有词法绑定。

后面跟有 ... 的子模式 (Subpattern) 可以匹配输入中的零个或更多个元素。在 ⟨literals⟩ 中出现 ... 是一个错误。在模式中，标识符 ... 必须跟在非空子模式序列的最后一个元素后面。

更正式地说，输入形式 F 和模式 P 相匹配的充分必要条件是：

- P 不是字面标识符；或者
- P 是字面标识符，且 F 是拥有同样绑定的标识符；或者
- P 是表 (P_1 ... P_n) 且 F 是包含 n 个形式的表，其中的每个形式分别与 P_1 至 P_n 匹配；或者
- P 是非严格表 (P_1 P_2 ... P_n . P_{n+1}) 且 F 是一个包含 n 个或更多形式的表或非严格表，其中的每个形式分别与 P_1 至 P_n 匹配， F 中第 n 个“cdr”与 P_{n+1} 匹配；或者
- P 的格式为 (P_1 ... P_n P_{n+1} ⟨ellipsis⟩)，其中的 ⟨ellipsis⟩ 是标识符 ...，同时， F 是一个至少有 n 个形式的严格的表， F 的前 n 个形式分别与 P_1 至 P_n 匹配， F 中剩下的每个元素与 P_{n+1} 匹配；或者
- P 是一个格式为 #(P_1 ... P_n) 的向量且 F 是一个拥有 n 个形式的向量，其中的每个形式与 P_1 至 P_n 匹配；或者
- P 的格式为 #(P_1 ... P_n P_{n+1} ⟨ellipsis⟩)，其中的 ⟨ellipsis⟩ 是标识符 ...，同时， F 是一个有 n 个或更多形式的向量， F 的前 n 个形式分别与 P_1 至 P_n 匹配， F 中剩下的每个元素与 P_{n+1} 匹配；或者
- P 是一个已知数 (Datum)，并且，在 `equal?` 过程的语义下， F 与 P 相等。

在宏关键字的绑定的作用域中，将该宏关键字用于一个不能与任何模式匹配的表达式是一个错误。

根据匹配的 ⟨syntax rule⟩ 中的模板转译一个宏的使用时，出现在模板中的模式变量被替换为输入中与之匹配的子形式。后面跟有一个或多个标识符 ... 的子模式中出现的模式变量，只在后面跟有同样多的 ... 的子模板 (Subtemplate) 中起作用。输出时，它们在指定的位置，被输入中

所有与之匹配的子形式替换。不能依定义的方式组建输出的情况是一个错误。

出现在模板中但不是模式变量或标识符 ... 的标识符被当作字面标识符插入到输出中。如果一个字面标识符被当作自由标识符插入，那么它就指向 `syntax-rules` 实例所在的作用域里的该标识符的绑定。如果一个字面标识符被当作已绑定的标识符插入，那么就实际效果而言，它会被改名使用，以避免无意中和自由标识符冲突。

例如，如果按第 7.3 节的描述定义 `let` 和 `cond`，那么它们是“卫生的”（这是必要的），且下面的代码是正确的：

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

`cond` 的宏转换器可以将 `=>` 识别为一个局部变量，并进而将它看成一个表达式，而不会把它看成最高层的标识符 `=>`（宏转换器会把最高层的标识符 `=>` 视为一个语法关键字）。于是，示例代码被扩展为：

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

而不是被扩展为：

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

后者将导致一个非法的过程调用。

5. 程序结构

5.1. 程序

Scheme 程序由一系列表达式、定义和语法定义组成。第 4 章给出了表达式的说明，本章后续部分主要介绍定义和语法定义。

尽管可以有其他的存在形式，但程序通常都存储在文件中，或以交互方式输入到运行的 Scheme 系统中。有关用户界面的问题不在本报告的讨论范围之内。（实际上，即便在缺乏具体实现的情况下， Scheme 仍然是表述计算过程的有用方式。）

出现在程序最高层的定义和语法定义可被解释为声明，它们在最高层环境创建绑定，或者改变已有的最高层绑定的值。出现在程序最高层的表达式被解释为命令，当程序被调用或被加载时，它们按顺序执行，通常会完成某些初始化的工作。

在程序的最高层，`(begin` ⟨form$_1$⟩ ...`)` 等价于构成 `begin` 表达式主体的表达式、定义和语法定义的序列。

5.2. 定义

定义可以出现在允许表达式出现的某些（但不是全部）上下文中。定义只能出现在程序 ⟨program⟩ 的最高层或 ⟨body⟩ 的开头。

定义应符合下面几种格式之一：

- `(define` ⟨variable⟩ ⟨expression⟩`)`
- `(define (`⟨variable⟩ ⟨formals⟩`)` ⟨body⟩`)`

  ⟨formals⟩ 或者是零个或更多个变量的序列，或者是一个或多个后面跟有由空格分隔的句点以及另一个变量的变量序列（就像在 lambda 表达式中那样）。这种格式等价于：

  ```
  (define ⟨variable⟩
    (lambda (⟨formals⟩) ⟨body⟩))
  ```

- `(define (`⟨variable⟩ `.` ⟨formal⟩`)` ⟨body⟩`)`

  ⟨formal⟩ 应为一个单独的变量。这种格式等价于：

  ```
  (define ⟨variable⟩
    (lambda ⟨formal⟩ ⟨body⟩))
  ```

5.2.1. 最高层定义

在程序的最高层，如果变量 ⟨variable⟩ 已绑定，定义

```
(define ⟨variable⟩ ⟨expression⟩)
```

的作用在本质上与下面的赋值表达式相同：

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

但如果 ⟨variable⟩ 未被绑定，定义就会在执行赋值之前将 ⟨variable⟩ 绑定到一个新的存储位置。反之，在一个未绑定的变量上执行 `set!` 是一个错误。

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

在某些 Scheme 实现使用的初始环境中，所有可能的变量都被绑定到存储位置，这些存储位置大多包含未定义的值。在这一类实现中，最高层定义完全等价于赋值操作。

5.2.2. 内部定义

定义可以出现在 ⟨body⟩（即，`lambda`、`let`、`let*`、`letrec`、`let-syntax` 或 `letrec-syntax` 表达式的 ⟨body⟩，或适当格式的定义的 ⟨body⟩）的开头。这些定义被称为内部定义，以区别于上面描述的最高层定义。内部定义所定义的变量是 ⟨body⟩ 的局部变量，即，⟨variable⟩ 是被绑定而非单纯的赋值，绑定的作用域是整个 ⟨body⟩，例如：

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

包含内部定义的 ⟨body⟩ 总可以被转换为完全等价的 `letrec` 表达式。例如，上面例子中的 `let` 表达式等价于：

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

正像等价的 letrec 表达式那样，在 ⟨body⟩ 内，即便没有指明或引用任何定义的 ⟨variable⟩ 的值，每个内部定义的 ⟨expression⟩ 也必须能被求值。

无论内部定义出现在哪里，(begin ⟨definition$_1$⟩ ...) 都等价于构成 begin 表达式主体的定义序列。

5.3. 语法定义

语法定义只能出现在 ⟨program⟩ 的最高层。 它们的格式如下：

(define-syntax ⟨keyword⟩ ⟨transformer spec⟩)

⟨keyword⟩ 是一个标识符， ⟨transformer spec⟩ 是 syntax-rules 的一个实例。最高层语法环境被扩展为包含了 ⟨keyword⟩ 到特定转换器的绑定。

没有与 define-syntax 类似的内部定义。

尽管在上下文允许时，宏可以展开为定义和语法定义，但如果定义或语法定义屏蔽了某个语法关键字，且只有借助该语法关键字的含义才能确定包含屏蔽定义的一组形式中的某个形式在事实上是否是一个定义，或者，对于内部定义来说，只有借助该语法关键字的含义才能确定这组形式与其后的表达式之间的边界，那么，此情况是一个错误。例如，下面的代码是错误的：

```
(define define 3)

(begin (define begin list))

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

6. 标准过程

本章描述 Scheme 的内置过程。初始的（或最高层的）Scheme 环境启动时，有许多变量已经绑定在包含着有用的值的存储位置，这些值中的大多数都是处理数据的基本过程。例如，变量 abs 所绑定的（存储位置中最初存储的）过程拥有一个参数，可计算某个数的绝对值；变量 + 所绑定的过程可完成求和运算。可借助其他内置过程简单实现的内置过程被称为“库过程”。

程序可以使用最高层定义绑定任意变量，随后也可以用赋值操作（参见 4.1.6）改变任何一个这样的绑定。这些操作不会改变 Scheme 内置过程的行为。改变任何没有通过定义引入的最高层绑定，其结果对于内置过程的影响是未定义的。

6.1. 相等谓词

谓词是总返回布尔值（#t 或 #f）的过程。相等谓词在计算上对应于数学中的相等关系（它是对称的、自反的和可传递的）。在本节描述的相等谓词中， eq? 是最精细或最有辨别力的，而 equal? 是最宽松的。 eqv? 比 eq? 的辨别力稍差一些。

(eqv? obj_1 obj_2) procedure

eqv? 过程为对象定义了一种有用的相等关系。简单地说，如果 obj_1 和 obj_2 在通常情况下应该被视为相同的对象，eqv? 过程就返回 #t。这种关系中有少量可以补充解释的地方，但下面所列的 eqv? 的部分定义对所有 Scheme 实现都是适用的。

eqv? 过程在以下情况返回 #t：

- obj_1 和 obj_2 都为 #t 或都为 #f。
- obj_1 和 obj_2 都为符号，且

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

  注： 这里假定 obj_1 和 obj_2 都不是第 6.3.3 节提到的“非置入符号 (Uninterned symbol)”。本报告不指明 eqv? 在依赖于实现的扩展中的行为。

- obj_1 和 obj_2 都是数值，它们的值相等（参见 =，第 6.2 节），且同为精确数或同为非精确数。
- obj_1 和 obj_2 都是字符，且在 char=? 过程（第 6.3.4 节）的语义下是相同的字符。
- obj_1 和 obj_2 都是空表。
- obj_1 和 obj_2 是表示同一存储位置的点对、向量或字符串（第 3.4 节）。
- obj_1 和 obj_2 是位置标记 (Location tag) 相等的过程（第 4.1.4 节）。

eqv? 过程在以下情况返回 #f：

- obj_1 和 obj_2 是不同类型的对象（第 3.2 节）。
- obj_1 和 obj_2 中一个是 #t 而另一个是 #f。
- obj_1 和 obj_2 是符号，但

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

- obj_1 和 obj_2 中一个是精确数而另一个是非精确数。
- obj_1 和 obj_2 都是数值，但 = 过程作用于二者的结果是 #f。
- obj_1 和 obj_2 都是字符，但 char=? 过程作用于二者的结果是 #f。
- obj_1 和 obj_2 中一个是空表，而另一个不是。
- obj_1 和 obj_2 是表示不同存储位置的点对、向量或字符串。
- obj_1 和 obj_2 是过程，但二者对于某些参数的行为不同（返回不同的值或有不同的副作用）。

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

下面的示例展示了上述规则未完全定义的 eqv? 的行为。关于这些情形，可以说明的仅仅是， eqv? 的返回值必须是一个布尔值。

```
(eqv? "" "")                 ⟹  未定义
(eqv? '#() '#())             ⟹  未定义
(eqv? (lambda (x) x)
      (lambda (x) x))        ⟹  未定义
(eqv? (lambda (x) x)
      (lambda (y) y))        ⟹  未定义
```

下面一组示例显示了 eqv? 与包含局部状态的过程共同使用的情况。 gen-counter 每次都必须返回不同的过程，因为每个过程都有它自己的内部计数器。但 gen-loser 每次都返回相等的过程，因为局部状态不会影响到该过程的值或副作用。

```
(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-counter)))
  (eqv? g g))                ⟹  #t
(eqv? (gen-counter) (gen-counter))
                             ⟹  #f
(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-loser)))
  (eqv? g g))                ⟹  #t
(eqv? (gen-loser) (gen-loser))
                             ⟹  未定义

(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
         (g (lambda () (if (eqv? f g) 'both 'g))))
  (eqv? f g))
                             ⟹  未定义

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

因为改变一个（常量表达式返回的）常量对象是一个错误， Scheme 实现可以（但不是必须）在适当的时候在常量间共享结构。因此， eqv? 作用于常量的结果有时是依赖于实现的。

```
(eqv? '(a) '(a))             ⟹  未定义
(eqv? "a" "a")               ⟹  未定义
(eqv? '(b) (cdr '(a b)))     ⟹  未定义
(let ((x '(a)))
  (eqv? x x))                ⟹  #t
```

原理： eqv? 的上述定义允许 Scheme 实现灵活对待过程和常量： Scheme 实现可以自由选择是检测还是放弃检测两个过程或两个常量彼此相等，也可以决定是否使用相同的指针或位模式 (Bit pattern) 表示相等的对象，以合并二者的表示方式。

(eq? obj_1 obj_2) procedure

eq? 与 eqv? 类似，只是在某些情况下，eq? 对于差异的辨别能力比 eqv? 更强一些。

应确保 eq? 与 eqv? 对符号、布尔值、空表、点对、过程和非空的字符串及向量的行为是一致的。对于数值和字符， eq? 的行为是依赖于实现的，但它总会返回真或假，并且只在 eqv? 也会返回真的情况下返回真。对于空向量和空字符串， eq? 的行为也可能与 eqv? 不同。

```
(eq? 'a 'a)                  ⟹  #t
(eq? '(a) '(a))              ⟹  未定义
(eq? (list 'a) (list 'a))    ⟹  #f
(eq? "a" "a")                ⟹  未定义
(eq? "" "")                  ⟹  未定义
(eq? '() '())                ⟹  #t
(eq? 2 2)                    ⟹  未定义
(eq? #\A #\A)                ⟹  未定义
(eq? car car)                ⟹  #t
(let ((n (+ 2 3)))
  (eq? n n))                 ⟹  未定义
(let ((x '(a)))
  (eq? x x))                 ⟹  #t
(let ((x '#()))
  (eq? x x))                 ⟹  #t
(let ((p (lambda (x) x)))
  (eq? p p))                 ⟹  #t
```

原理： 通常，有可能比 `eqv?` 更高效地实现 `eq?`，如将 `eq?` 实现为简单的指针比较，而非更复杂的操作。一个原因是，我们可能无法在常数时间内用 `eqv?` 完成两个数值的比较，但利用指针比较实现的 `eq?` 却总可以在常数时间内完成操作。因为 `eq?` 与 `eqv?` 服从同样的约束，在利用过程来实现有状态对象的应用里， `eq?` 也可以像 `eqv?` 那样使用。

(`equal?` obj_1 obj_2) library procedure

`equal?` 以递归方式比较点对、向量和字符串的内容。对于数值、符号等其他对象， `equal?` 使用 `eqv?` 进行比较。一个经验原则是，如果对象的打印输出相同，它们通常就是 `equal?` 的。当 `equal?` 的参数是循环的数据结构时，它可能无法终止。

```
(equal? 'a 'a)                ⟹  #t
(equal? '(a) '(a))            ⟹  #t
(equal? '(a (b) c)
        '(a (b) c))           ⟹  #t
(equal? "abc" "abc")          ⟹  #t
(equal? 2 2)                  ⟹  #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))   ⟹  #t
(equal? (lambda (x) x)
        (lambda (y) y))       ⟹  未定义
```

6.2. 数值

传统上，数值运算曾被 Lisp 社区忽视。直到 Common Lisp 出现以前，除了 MacLisp 系统[20] 在更高效地执行数值代码方面做了少量贡献以外，人们并没有认真设计过组织数值计算的方法。本报告赞赏 Common Lisp 委员会的杰出工作，并接受了他们的许多建议。在某些方面，本报告对他们的建议进行了简化和归纳，以使其与 Scheme 的设计意图保持一致。

对数学上的数值、试图为数学数值建模的 Scheme 数值、实现 Scheme 数值的机器表示以及书写时的数值记法加以区分是相当重要的。本报告使用 *number*、 *complex*、 *real*、 *rational* 和 *integer* 等类型同时指代数学上的数值和 Scheme 数值，使用 *fixnum* 和 *flonum* 等名称指代定点数、浮点数等机器表示。

6.2.1. 数值类型

在数学上，数值可以被组织为一个子类型的塔结构，其中的每一层都是上面一层的子集。

数 (number)
复数 (complex)
实数 (real)
有理数 (rational)
整数 (integer)

例如， 3 是整数，因此 3 也是有理数，实数和复数。这对于为 3 建模的 Scheme 数值来说也是成立的。对于 Scheme 数值，这些类型由 `number?`、 `complex?`、 `real?`、 `rational?` 和 `integer?` 等谓词定义。

数值的类型和它在计算机中的表示方法之间并不存在简单的关联。尽管大多数 Scheme 实现都会提供 3 的至少两种表示方法，但这些不同的表示都指代相同的整数。

Scheme 的数值计算把数值看成抽象数据，尽可能与它们的表示无关。尽管 Scheme 实现可使用定点数、浮点数抑或其他方式来表示数值，但这些对于偶尔编些简单程序的程序员来说不应该是明显可见的。

但我们必须区分被精确表示的数值和可能无法被精确表示的数值。例如，数据结构的索引应当被精确地识别，符号代数系统中的多项式系数也应如此。另一方面，测量的结果在本质上就是不精确的，无理数可以被近似地表示为有理数，这也是不精确的近似。为了在需要精确数的地方使用非精确数， Scheme 显式区分精确数和非精确数。这种区分和类型的划分是正交的关系。

6.2.2. 精确性

Scheme 数值或者是精确的，或者是非精确的。如果一个数值被写作精确的常量，或是完全通过精确运算从精确数得出，它就是精确的。如果一个数值被写作非精确的常量，或者是由非精确的成分得出的，或者是由非精确的运算得出的，它就是非精确的。也就是说，非精确性是数值的一种可传染的特性。

如果两个实现对一个不包含非精确的中间结果的计算都产生了精确的结果，那么这两个最终的结果就将是数学上相等的结果。对于包含非精确数的计算来说，这通常是不成立的，因为其中可能会使用浮点算术等近似的方法，但每个实现仍有责任保证该结果最大程度地接近数学上的理想结果。

使用精确的参数时，诸如 `+` 这样的有理数运算应当总生成精确的结果。如果该运算不能生成精确的结果，它可以报告违反了实现约束，也可以隐式将结果强制转换为非精确的值。参见第 6.2.3 节。

使用任意非精确的参数时，除了 `inexact->exact` 以外，本节描述的运算通常都应返回非精确的结果。但如果一个运算能够证明结果的值不会受到其参数的非精确性的影响，它也可以返回精确的结果。例如，将精确的零与任意数值相乘时，即便其他参数是非精确的，结果也可以是精确的零。

6.2.3. 实现约束

Scheme 实现不需要实现第 6.2.1 节给出的整个子类型塔，但必须实现一个与实现本身的设计意图及 Scheme 语言的精神相一致的连贯的子集。例如，所有数值都是实数的实现也非常有用。

Scheme 实现也可以在满足本节需求的情况下，只支持任意类型的一个有限值域。Scheme 实现支持的任意类型的精确数的值域可以和该类型的非精确数的值域不同。例如，一个使用浮点数表示所有非精确的实数的实现可以在

限制非精确的实数的值域（以及非精确的整数和有理数的值域）为浮点数格式的动态值域的同时，支持事实上没有值域限制的精确的整数和有理数。此外，在采用这种值域限制的实现中，可表示的非精确的整数和有理数之间的差异可能非常大。

对于精确的整数， Scheme 实现必须支持足以表示表、向量和字符串的索引，或足以用来计算表、向量和字符串的长度的值域范围。 `length`、 `vector-length` 和 `string-length` 过程必须返回精确的整数，不使用精确的整数表示索引的情况是一个错误。此外，如果索引值域内的任意整数常量是使用精确整数的语法表示的，它就应当被视作一个精确的整数，而不论是否存在可应用于此值域之外的实现约束。最后，如果所有参数都是精确的整数，且数学上预期的结果可以在实现中表示为精确的整数，下面列出的过程就总会返回精确的整数结果：

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

我们鼓励（但不是强制） Scheme 实现支持事实上拥有无限大小和精度的精确整数和精确有理数，并将上述过程和 `/` 过程实现为，当给出精确的参数时，这些过程总返回精确的结果。当给出精确的参数时，如果这些过程中的某一个不能获得精确的结果，它可以报告违反了实现约束，也可以隐式把结果强制转换成非精确数。这样的强制转换随后有可能导致一个错误。

Scheme 实现可以为非精确数使用浮点和其他近似的表示方法。本报告建议（但不是强制）使用浮点表示法的实现遵循 IEEE 32 位和 64 位浮点标准，使用其他表示方法的实现应当在精度上达到或超过这些浮点标准[12]。

特别地，使用浮点数表示法的 Scheme 实现必须遵循以下规则：浮点数结果的表示精度必须达到或超过参与该运算的任意非精确参数的表示精度。当使用精确的参数时，我们希望（但不是强制） `sqrt` 等潜在的非精确运算在可能的情况下能返回精确的结果（例如精确的 4 的平方根应该是精确的 2）。但是，如果参数是精确数的运算会产生非精确的结果（如通过 `sqrt`），且该结果被表示为浮点数，那么就必须使用当前可用的精度最高的浮点数格式；不过，如果该结果是用其他形式表示的，那么该表示方法在精度上就必须达到或超过当前可用的精度最高的浮点数格式。

尽管 Scheme 允许多种数值的记法，特定的 Scheme 实现可以仅支持其中的几种。例如，一个所有数都是实数的实现不需要支持复数的平面直角坐标和极坐标记法。如果一个实现遇到了自己不能将其表示为精确数的精确数值常量，它可以报告违反了实现约束，也可以隐式把常量表示为非精确数。

6.2.4. 数值常量的语法

第 7.1.1 节描述了数值的书面表示的形式语法。注意那并不特定于数值常量。

可以通过基数前缀，把数值写成二进制、八进制、十进制或十六进制。基数前缀是 `#b` （二进制）、 `#o` （八进制）、 `#d` （十进制）和 `#x` （十六进制）。没有基数前缀时，就假定数值是用十进制表示的。

可以用一个前缀把数值常量定义为精确的或非精确的。前缀 `#e` 用于精确数， `#i` 用于非精确数。精确性前缀可以出现在已有的基数前缀的前面或后面。如果一个数值的书写形式中没有精确性前缀，该常量可能是精确的，也可能是非精确的：如果其中包含小数点、指数或“`#`”字符，该常量就是非精确的，反之，该常量就是精确的。

在拥有不同精度的非精确数的系统中，指明一个常量的精度可能是有益的。为此，数值常量的记法中可以出现一个指数标记，以指明非精确表示的期望精度。字母 `s`、 `f`、 `d` 和 `l` 分别指明要使用 *short*、 *single*、 *double* 和 *long* 精度（当已有的内部非精确表示少于四种时，这四个精度定义被映射到当前可用的定义。例如，只有两种内部表示的实现可以将 short 和 single 映射为一种精度，将 long 和 double 映射为一种）。另外，指数标记 `e` 指明了 Scheme 实现的缺省精度。缺省精度应达到或超过 *double* 的精度，但 Scheme 实现也许会希望用户可设置此缺省精度。

```
3.14159265358979F0
        舍入到 single — 3.141593
0.6L0
        扩展到 long — .600000000000000
```

6.2.5. 数值运算

读者可以在第 1.3.3 节参考用于指明数值程序参数类型约束的命名约定。本节使用的示例假定任何使用精确的记法书写的数值常量都会被表示为精确数。一些示例还假定某些使用非精确的记法书写的数值常量可以在不损失精度的情况下被表示出来；非精确的常量是经过筛选的，以便使那些利用浮点数表示非精确数的实现可以满足这一假定。

`(number?` *obj*`)` procedure
`(complex?` *obj*`)` procedure
`(real?` *obj*`)` procedure
`(rational?` *obj*`)` procedure
`(integer?` *obj*`)` procedure

这些数值类型的谓词可接受任意种类的参数，包括非数值参数。如果对象是谓词名称所指的类型，它们就返回 `#t`，否则就返回 `#f`。通常，如果一个类型谓词对一个数值返回真，那么，所有更高层类型谓词也会对该数值返回真。由此，如果一个类型谓词对一个数值返回假，那么，所有更低层类型谓词也会对该数值返回假。

如果 z 是一个非精确的复数，那么当且仅当 `(zero? (imag-part` z`))` 为真时， `(real?` z`)` 为真。如果 x 是一个非精确的实数，那么当且仅当 `(=` x `(round` x`))` 时， `(integer?` x`)` 为真。

```
(complex? 3+4i)   ⟹  #t
(complex? 3)      ⟹  #t
```

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

注： 对于非精确数，这些类型谓词的行为是不可靠的，因为任何不精确之处都会影响其结果。

注： 在许多实现中， `rational?` 过程与 `real?` 相同， `complex?` 过程与 `number?` 相同。但特殊的实现也可能精确地表示某些无理数，或将数值系统扩展到支持某种非复数。

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

这些数值谓词可以测试一个数量的精确性。对任意的 Scheme 数值，这些谓词中恰有一个为真。

`(=` z_1 z_2 z_3 `...)` procedure
`(<` x_1 x_2 x_3 `...)` procedure
`(>` x_1 x_2 x_3 `...)` procedure
`(<=` x_1 x_2 x_3 `...)` procedure
`(>=` x_1 x_2 x_3 `...)` procedure

如果这些过程的参数（分别）是相等的、单调递增的、单调递减的、单调非减的或单调非增的，它们就返回 `#t`。

这些谓词应当具有可传递性。

注： 在类 Lisp 语言里，这些谓词的传统实现并不具有可传递性。

注： 尽管使用这些谓词比较非精确数并不是一个错误，但其结果可能是不可靠的，因为细微的不精确之处就可能影响到结果；特别是对 `=` 和 `zero?` 来说更是如此。如果有疑问，请咨询数值分析专家。

`(zero?` z`)` library procedure
`(positive?` x`)` library procedure
`(negative?` x`)` library procedure
`(odd?` n`)` library procedure
`(even?` n`)` library procedure

这些数值谓词测试一个数的特定属性，返回 `#t` 或 `#f`。参见上面的注释。

`(max` x_1 x_2 `...)` library procedure
`(min` x_1 x_2 `...)` library procedure

这些过程返回其参数中的最大或最小值。

```
(max 3 4)                    ⟹  4    ; 精确的
(max 3.9 4)                  ⟹  4.0  ; 非精确的
```

注： 如果任何一个参数是非精确的，其结果也将是非精确的（除非该过程可以证明不精确性不会大到影响结果。只有在特殊的实现中才会出现这种情况）。如果用 `min` 或 `max` 比较混合了不同精确度的数值，且不能在不损失精度的情况下把结果数值表示为一个非精确数，那么该过程可以报告违反了实现约束。

`(+` z_1 `...)` procedure
`(*` z_1 `...)` procedure

这些过程返回其参数的和或积。

```
(+ 3 4)                      ⟹  7
(+ 3)                        ⟹  3
(+)                          ⟹  0
(* 4)                        ⟹  4
(*)                          ⟹  1
```

`(-` z_1 z_2`)` procedure
`(-` z`)` procedure
`(-` z_1 z_2 `...)` optional procedure
`(/` z_1 z_2`)` procedure
`(/` z`)` procedure
`(/` z_1 z_2 `...)` optional procedure

有两个或多个参数时，这些过程用左结合的方式返回其参数的差或商。但只有一个参数时，它们返回其参数的负值或倒数。

```
(- 3 4)                      ⟹  -1
(- 3 4 5)                    ⟹  -6
(- 3)                        ⟹  -3
(/ 3 4 5)                    ⟹  3/20
(/ 3)                        ⟹  1/3
```

`(abs` x`)` library procedure

`abs` 返回其参数的绝对值。

```
(abs -7)                     ⟹  7
```

`(quotient` n_1 n_2`)` procedure
`(remainder` n_1 n_2`)` procedure
`(modulo` n_1 n_2`)` procedure

这些过程实现了数论（整数）除法。 n_2 不能为零。这三个过程都返回整数。如果 n_1/n_2 的结果是整数：

```
(quotient n1 n2)             ⟹  n1/n2
(remainder n1 n2)            ⟹  0
(modulo n1 n2)               ⟹  0
```

如果 n_1/n_2 的结果不是整数：

```
(quotient n1 n2)             ⟹  nq
(remainder n1 n2)            ⟹  nr
(modulo n1 n2)               ⟹  nm
```

其中 n_q 是 n_1/n_2 舍去小数部分后的结果， $0 < |n_r| < |n_2|$， $0 < |n_m| < |n_2|$， n_r 或 n_m 与 n_1 的差是 n_2 的整数倍。 n_r 与 n_1 的符号相同， n_m 与 n_2 的符号相同。

由此可知，假设所有参与运算的都是精确数，当 n_1 和 n_2 都为整数，且 n_2 不等于 0 时：

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

(modulo 13 4)                 ⟹  1
(remainder 13 4)              ⟹  1

(modulo -13 4)                ⟹  3
(remainder -13 4)             ⟹  -1

(modulo 13 -4)                ⟹  -3
(remainder 13 -4)             ⟹  1

(modulo -13 -4)               ⟹  -1
(remainder -13 -4)            ⟹  -1

(remainder -13 -4.0)          ⟹  -1.0  ; 非精确数
```

`(gcd` n_1 `...)` library procedure
`(lcm` n_1 `...)` library procedure

这些过程返回其参数的最大公约数或最小公倍数，其结果总是非负数。

```
(gcd 32 -36)                  ⟹  4
(gcd)                         ⟹  0
(lcm 32 -36)                  ⟹  288
(lcm 32.0 -36)                ⟹  288.0  ; 非精确数
(lcm)                         ⟹  1
```

`(numerator` q`)` procedure
`(denominator` q`)` procedure

这些过程返回其参数的分子或分母，计算时它们会将参数约分为最简分数。分母总为正数。 0 的分母被定义为 1 。

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

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

这些过程返回整数。 `floor` 返回不大于 x 的最大整数。 `ceiling` 返回不小于 x 的最小整数。 `truncate` 返回最接近 x 但绝对值不大于 x 的绝对值的整数。 `round` 返回最接近 x 的整数，当 x 位于两个整数中间时，其结果将舍入到偶数。

原理： `round` 舍入到偶数，这是为了与 IEEE 浮点标准定义的缺省舍入模型保持一致。

注： 如果这些过程的参数是非精确的，其结果也将是非精确的。如果需要精确值，应把结果传入 `inexact->exact` 过程。

```
(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  ; 非精确的

(round 7/2)                   ⟹  4    ; 精确的
(round 7)                     ⟹  7
```

`(rationalize` x y`)` library procedure

`rationalize` 返回一个与 x 的差值不大于 y 的最简*(Simplest)*有理数。如果有理数 $r_1 = p_1/q_1$ 和有理数 $r_2 = p_2/q_2$（都以最简分数表示）满足 $|p_1| \le |p_2|$ 且 $|q_1| \le |q_2|$，则称有理数 r_1 比有理数 r_2 更简单。 由此，3/5 比 4/7 更简单。尽管不是所有有理数都可以进行这样的比较（如 2/7 和 3/5 就无法做这样比较），但在任意区间内，总有一个有理数比该区间内其他所有有理数更简单（较简单的 2/5 位于 2/7 到 3/5 的区间内）。注意 $0 = 0/1$ 是所有有理数中最简单的一个。

```
(rationalize
  (inexact->exact .3) 1/10)   ⟹  1/3    ; 精确的
(rationalize .3 1/10)         ⟹  #i1/3  ; 非精确的
```

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

这些过程是每个支持一般实数的实现的组成部分，它们计算普通的超越函数的结果。 `log` 计算 z 的自然对数（不是以 10 为底的对数）。 `asin` 、`acos` 和 `atan` 分别计算反正弦($\sin^{-1}$)、反余弦($\cos^{-1}$)和反正切($\tan^{-1}$)函数。有两个参数的 `atan` 变体计算 `(angle (make-rectangular` x y`))`（见下），即便在那些不支持一般复数的实现中，此函数也可使用。

通常，对数、反正弦、反余弦和反正切等数学函数是用复合方式定义的。 $\log z$ 的值被定义为虚部介于 $-\pi$ （不含）至 π （含）之间的一个数。 $\log 0$ 是未定义的。在 log 的上述定义基础上， $\sin^{-1} z$、 $\cos^{-1} z$ 和 $\tan^{-1} z$ 的值可用以下公式计算：

$$\sin^{-1} z = -i\log(iz + \sqrt{1-z^2})$$
$$\cos^{-1} z = \pi/2 - \sin^{-1} z$$
$$\tan^{-1} z = (\log(1+iz) - \log(1-iz))/(2i)$$

上述定义来自[27]，后者进一步引用了[19]；关于分支切割、边界条件及这些函数实现的更多细节，可参考这些文献。参数为实数时，如果可能，这些函数就会返回实数的结果。

`(sqrt` z`)` procedure

返回 z 的主平方根。其结果或者有正的实部，或者实部为零、虚部非负。

`(expt` z_1 z_2`)` procedure

返回 z_1 的 z_2 次方。对 $z_1 \neq 0$ 有：

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

0^z 在 $z = 0$ 时为 1，否则为 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

这些过程是每个支持一般复数的实现的组成部分。假定 x_1、x_2、x_3 和 x_4 是实数，且 z 是这样的复数：

$$z = x_1 + x_2 i = x_3 \cdot e^{ix_4}$$

则有

`(make-rectangular` x_1 x_2`)` $\Longrightarrow$ z
`(make-polar` x_3 x_4`)` $\Longrightarrow$ z
`(real-part` z`)` $\Longrightarrow$ x_1
`(imag-part` z`)` $\Longrightarrow$ x_2
`(magnitude` z`)` $\Longrightarrow$ $|x_3|$
`(angle` z`)` $\Longrightarrow$ x_{angle}

其中 $-\pi < x_{angle} \leq \pi$，对某些整数 n 满足 $x_{angle} = x_4 + 2\pi n$。

原理： 对于实数参数， `magnitude` 的作用与 `abs` 相同。但所有实现都必须支持 `abs`，而 `magnitude` 只需要出现在支持一般复数的实现中。

`(exact->inexact` z`)` procedure
`(inexact->exact` z`)` procedure

`exact->inexact` 返回一个表示 z 的非精确数。返回的值是最接近参数值的那个非精确数。如果一个精确的参数没有最接近的非精确数，就可以报告违反了实现约束。

`inexact->exact` 返回一个表示 z 的精确数。返回的值是最接近参数值的那个精确数。如果一个非精确的参数没有最接近的精确数，就可以报告违反了实现约束。

在依赖于实现的值域中，这些过程在所有精确的和非精确的整数之间建立了基本的一对一关系。参见第 6.2.3 节。

6.2.6. 数值输入和输出

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

$radix$ 应为精确整数，且是 2、8、10 或 16 中的一个。如果省略 $radix$，缺省值是 10。过程 `number->string` 的参数是一个数值和一个基数，它以字符串的方式返回给定数值在给定基数情况下的外部表示，以使得

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

的值为真。如果不存在使上述表达式的值为真的结果，该情况就是一个错误。

如果 z 是非精确的，基数为 10，且一个包含小数点的结果可满足上述表达式，则该结果包含小数点，且以满足上述表达式为真所需的最少位数的数字（指数和补位的零除外）来表示[3, 5]；否则，结果的格式是未定义的。

`number->string` 返回的结果从不显式地包含基数前缀。

注： 错误的情况只发生在 z 不是一个复数，或者 z 是一个实部或虚部为非有理数的复数的情况下。

原理： 如果 z 是用浮点数表示的非精确数，且基数为 10，则一个包含小数点的结果通常可以满足上述表达式。对于无穷大、NaN 和非浮点表示，过程的结果可以是未定义的。

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

返回给定的 $string$ 所表示的最大精度的数值。 $radix$ 应为精确的整数，且是 2、8、10 或 16 中的一个。如果给出 $radix$， $radix$ 就是缺省基数，但它可以被 $string$ 中显式出现的基数前缀（如 `"#o177"`）覆盖。如果未给出 $radix$，则缺省基数为 10。如果 $string$ 不是一个语法上有效的数值记法，则 `string->number` 返回 `#f`。

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

注： Scheme 实现可以按如下方式限制 `string->number` 的定义域：当 $string$ 显式包含基数前缀时， `string->number` 可返回 `#f`；如果实现支持的所有数值都是实数，则当 $string$ 使用极坐标或平面直角坐标记法表示复数时， `string->number` 可返回 `#f`；如果所有数值都是整数，则当 $string$ 使用分数记法时， `string->number` 可返回 `#f`；如果所有数值都是精确数，则当 $string$ 使用指数标记或显式的精确性前缀，或者 `#` 出现在数字中时， `string->number` 可返回 `#f`；如果所有非精确数都是整数，则当 $string$ 使用小数点时， `string->number` 可返回 `#f`。

6.3. 其他数据类型

本节描述 Scheme 的一些非数值数据类型的操作。这些类型包括：布尔型、点对、表、符号、字符、字符串和向量。

6.3.1. 布尔型

表示真和假的标准布尔对象被记为 `#t` 和 `#f`。但真正重要的是，Scheme 条件表达式 (`if`、`cond`、`and`、`or`、`do`) 把何种对象视为真或假。“真值”（有时简记作“真”）表示任何被条件表达式视为真的对象，“假值”（或“假”）表示任何被条件表达式视为假的对象。

在所有标准的 Scheme 值中，只有 `#f` 在条件表达式中被视为假。除了 `#f` 以外，所有标准的 Scheme 值，包括 `#t`、点对、空表、符号、数值、字符串、向量和过程都被视为真。

注： 习惯了其他 Lisp 方言的程序员应注意，Scheme 中的 `#f` 和空表都与符号 `nil` 不同。

布尔常量的值是它们自身，所以，在程序中它们无需被引用。

```
#t                      ⟹  #t
#f                      ⟹  #f
'#f                     ⟹  #f
```

(`not` *obj*) library procedure

如果 *obj* 为假，`not` 就返回 `#t`，否则返回 `#f`。

```
(not #t)                ⟹  #f
(not 3)                 ⟹  #f
(not (list 3))          ⟹  #f
(not #f)                ⟹  #t
(not '())               ⟹  #f
(not (list))            ⟹  #f
(not 'nil)              ⟹  #f
```

(`boolean?` *obj*) library procedure

如果 *obj* 是 `#t` 或 `#f`，`boolean?` 就返回 `#t`，否则返回 `#f`。

```
(boolean? #f)           ⟹  #t
(boolean? 0)            ⟹  #f
(boolean? '())          ⟹  #f
```

6.3.2. 点对和表

点对（*Pair*，有时也写作 *Dotted pair*）是一个包含两个域的记录结构，两个域分别称为 car 和 cdr（因为历史原因）。点对由 `cons` 过程创建。car 和 cdr 域由 `car` 和 `cdr` 过程访问。car 和 cdr 域由 `set-car!` 和 `set-cdr!` 过程赋值。

点对主要用于表示表。表可以递归定义为：或者是一个空表，或者是一个 cdr 域为表的点对。更准确地说，表的集合被定义为符合以下条件的最小集合 X：

- 空表在 X 中。
- 如果表 *list* 在 X 中，则任何 cdr 域包含着表 *list* 的点对也在 X 中。

表中连续各点对的 car 域内的对象是表的元素。例如，一个拥有两个元素的表是一个点对，该点对的 car 域包含表的第一个元素，其 cdr 域又是一个点对，这个点对的 car 域包含表的第二个元素，cdr 域是空表。表的长度是元素的数量，也等于其点对的数量。

空表是一个隶属于其自身类型的特殊对象（它不是点对），它不包含任何元素，长度为零。

注： 上述定义意味着所有表都有有限的长度，都以空表结尾。

最普遍的 Scheme 点对记法（外部表示）是“点”记法 (c_1 . c_2)，其中 c_1 是 car 域的值，c_2 是 cdr 域的值。例如，`(4 . 5)` 是一个 car 为 4 且 cdr 为 5 的点对。注意 `(4 . 5)` 是点对的外部表示，而不是结果为点对的表达式。

对于表，可以使用一种更流畅的记法：把表中的元素简单地写在括号内，用空格分隔。空表记作 `()`。例如：

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

和

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

是同一个由符号组成的表的等价记法。

不是以空表结尾的点对链称为非严格表。注意，非严格表不是表。可以使用表和点记法的组合表示非严格表：

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

等价于

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

给定的点对是否是一个表，这取决于 cdr 域中存储的是什么。使用 `set-cdr!` 过程时，一个对象可以在这个时刻是表，而在下个时刻不是表：

```
(define x (list 'a 'b 'c))
(define y x)
y                       ⟹  (a b c)
(list? y)               ⟹  #t
(set-cdr! x 4)          ⟹  未定义
x                       ⟹  (a . 4)
(eqv? x y)              ⟹  #t
y                       ⟹  (a . 4)
(list? y)               ⟹  #f
(set-cdr! x x)          ⟹  未定义
(list? x)               ⟹  #f
```

在常量表达式和 `read` 过程读取的对象表示中，'⟨datum⟩、`⟨datum⟩、,⟨datum⟩ 和 ,@⟨datum⟩ 等记法表示拥有两个元素的表，该表的第一个元素分别为符号 `quote`、`quasiquote`、`unquote` 或 `unquote-splicing`，第二个元素总为 ⟨datum⟩。支持这一约定的目的是，任何 Scheme 程序都可以被表示为表。也就是说，依据 Scheme 的语法，每个 ⟨expression⟩ 也同时是一个 ⟨datum⟩（参见

第 7.1.2 节）。除了其他优点外，这一机制还允许我们用 `read` 过程解析 Scheme 程序。参见第 3.3 节。

`(pair?` *obj*`)` procedure

如果 *obj* 是点对，则 `pair?` 返回 `#t`，否则返回 `#f`。

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

`(cons` *obj*$_1$ *obj*$_2$`)` procedure

返回一个新分配的点对，该点对的 car 是 *obj*$_1$， cdr 是 *obj*$_2$。该点对保证与任何已存在的对象都不相同（在 `eqv?` 的语义下）。

```
(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

返回 *pair* 中 car 域的内容。注意，读取空表的 car 是一个错误。

```
(car '(a b c))            ⟹  a
(car '((a) b c d))        ⟹  (a)
(car '(1 . 2))            ⟹  1
(car '())                 ⟹  错误
```

`(cdr` *pair*`)` procedure

返回 *pair* 中 cdr 域的内容。注意，读取空表的 cdr 是一个错误。

```
(cdr '((a) b c d))        ⟹  (b c d)
(cdr '(1 . 2))            ⟹  2
(cdr '())                 ⟹  错误
```

`(set-car!` *pair obj*`)` procedure

把 *obj* 存入 *pair* 的 car 域。 `set-car!` 的返回值是未定义的。

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)          ⟹  未定义
(set-car! (g) 3)          ⟹  错误
```

`(set-cdr!` *pair obj*`)` procedure

把 *obj* 存入 *pair* 的 cdr 域。 `set-cdr!` 的返回值是未定义的。

`(caar` *pair*`)` library procedure
`(cadr` *pair*`)` library procedure
⋮ ⋮
`(cdddar` *pair*`)` library procedure
`(cddddr` *pair*`)` library procedure

这些过程是 `car` 和 `cdr` 的组合。例如， `caddr` 可定义为：

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

Scheme 提供最高可达四层的任意组合。总共有二十八个这样的过程。

`(null?` *obj*`)` library procedure

如果 *obj* 是空表，则返回 `#t`，否则返回 `#f`。

`(list?` *obj*`)` library procedure

如果 *obj* 是表，则返回 `#t`，否则返回 `#f`。根据定义，所有表都有有限的长度，都以空表结尾。

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

`(list` *obj* ...`)` library procedure

返回新分配的表，表中的元素是此过程的参数。

```
(list 'a (+ 3 4) 'c)      ⟹  (a 7 c)
(list)                    ⟹  ()
```

`(length` *list*`)` library procedure

返回 *list* 的长度。

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

`(append` *list* ...`)` library procedure

返回一个表，该表中的元素首先是第一个 *list* 的元素，随后是其他 *list* 的元素。

```
(append '(x) '(y))        ⟹  (x y)
(append '(a) '(b c d))    ⟹  (a b c d)
(append '(a (b)) '((c)))  ⟹  (a (b) (c))
```

除了与最后一个 *list* 参数共享存储结构以外，结果表的其他部分总是新分配的。最后一个参数实际上可以是任何对象；如果最后一个参数不是严格的表，此过程的结果是一个非严格表。

```
(append '(a b) '(c . d))  ⟹  (a b c . d)
(append '() 'a)           ⟹  a
```

(reverse *list*) library procedure

返回一个新分配的表，该表中以倒序包含了 *list* 中的元素。

```
(reverse '(a b c))          ⟹  (c b a)
(reverse '(a (b c) d (e (f))))
          ⟹  ((e (f)) d (b c) a)
```

(list-tail *list k*) library procedure

返回 *list* 略去了前 *k* 个元素后的子表。*list* 的元素少于 *k* 个的情况是一个错误。list-tail 可以定义为：

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

(list-ref *list k*) library procedure

返回 *list* 中的第 *k* 个元素（与 (list-tail *list k*) 的 car 域相同）。*list* 的元素少于 *k* 个的情况是一个错误。

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

(memq *obj list*) library procedure
(memv *obj list*) library procedure
(member *obj list*) library procedure

这些过程返回 *list* 中 car 为 *obj* 的第一个子表。这里所说的 *list* 的子表是指，当 *k* 小于 *list* 的长度时，(list-tail *list k*) 返回的所有非空表。如果 *list* 中没有 *obj*，则返回 #f （不是空表）。比较 *obj* 和 *list* 的元素时，memq 使用 eq?，而 memv 使用 eqv?，member 使用 equal?。

```
(memq 'a '(a b c))          ⟹  (a b c)
(memq 'b '(a b c))          ⟹  (b c)
(memq 'a '(b c d))          ⟹  #f
(memq (list 'a) '(b (a) c)) ⟹  #f
(member (list 'a)
        '(b (a) c))         ⟹  ((a) c)
(memq 101 '(100 101 102))   ⟹  未定义
(memv 101 '(100 101 102))   ⟹  (101 102)
```

(assq *obj alist*) library procedure
(assv *obj alist*) library procedure
(assoc *obj alist*) library procedure

alist（即“关联表”，Association list）必须是元素为点对的表。这些过程在 *alist* 中寻找 car 域为 *obj* 的第一个点对，并返回该点对。如果 *alist* 中没有 car 域为 *obj* 的点对，则返回 #f （不是空表）。比较 *obj* 和 *list* 中点对的 car 域时，assq 使用 eq?，而 assv 使用 eqv?，assoc 使用 equal?。

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)                 ⟹  (a 1)
(assq 'b e)                 ⟹  (b 2)
(assq 'd e)                 ⟹  #f
(assq (list 'a) '(((a)) ((b)) ((c))))
                            ⟹  #f
(assoc (list 'a) '(((a)) ((b)) ((c))))
                            ⟹  ((a))
(assq 5 '((2 3) (5 7) (11 13)))
                            ⟹  未定义
(assv 5 '((2 3) (5 7) (11 13)))
                            ⟹  (5 7)
```

原理： 尽管人们常把 memq、memv、member、assq、assv 和 assoc 当作谓词使用，这些过程的名字中却没有问号，因为它们返回的是有用的值，而不仅仅是 #t 或 #f。

6.3.3. 符号

符号是一类对象，其用途依赖于以下事实：当且仅当两个符号的名字拼写相同时它们就是相同的符号（在 eqv? 的语义下）。在程序中表示标识符时，这是一种必备的特性。因此，大多数 Scheme 实现都在内部使用符号来表示标识符。符号还有许多其他用途，例如，我们可以像在 Pascal 中使用枚举值那样使用符号。

书写符号的规则和书写标识符的规则完全一样，参见第 2.1 和第 7.1.1 节。

Scheme 保证，任何作为常量表达式的一部分返回或使用 read 过程输入，并在随后使用 write 过程输出的符号，再重新读入时还能得到同一个符号（在 eqv? 的语义下）。但 string->symbol 过程却可能在违反这种读/写不变性的情况下创建符号，因为符号的名字可能包含了特殊字符或非标准大小写形式的字母。

注： 某些 Scheme 实现有一个叫“砍削 (Slashification)”的特征，可以确保所有符号的读/写不变性。但从历史上看，这一特征最重要的用途在于弥补缺少字符串数据类型的缺陷。

一些 Scheme 实现也提供“非置入符号 (Uninterned symbols)”的特征，该特征可使读/写不变性失效，即便在拥有“砍削”特征的实现中也是如此。这也为“当且仅当两个符号的名字拼写相同时它们才是相同的符号”的规则引入了例外情况。

(symbol? *obj*) procedure

如果 *obj* 是符号，则返回 #t，否则返回 #f。

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

(symbol->string *symbol*) procedure

以字符串方式返回 *symbol* 的名字。如果该符号是常量表达式（参见第 4.1.2 节）的值或调用 read 过程返回的对象

的一部分，且它的名字包含字母，那么，返回的字符串所包含的是以 Scheme 实现首选的标准大小写形式表示的字符：一些实现首选大写形式，另一些首选小写形式。如果该符号是 `string->symbol` 返回的，此过程返回的字符串中字符的大小写形式将与传入 `string->symbol` 的字符串保持一致。对此过程返回的字符串应用 `string-set!` 之类的改变过程是一个错误。

下面的示例假定 Scheme 实现的标准大小写形式是小写：

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

`(string->symbol` *string*`)` procedure

返回名字为 *string* 的符号。此过程可以创建名字里包含了特殊字符或非标准大小写形式的字母的符号，但创建这样的符号通常不是一个好主意，因为在一些 Scheme 实现中，它们无法被原样输入。参见 `symbol->string`。

下面的示例假定 Scheme 实现的标准大小写形式是小写：

```
(eq? 'mISSISSIppi 'mississippi)
          ⟹  #t
(string->symbol "mISSISSIppi")
          ⟹  名字为 "mISSISSIppi" 的符号
(eq? 'bitBlt (string->symbol "bitBlt"))
          ⟹  #f
(eq? 'JollyWog
     (string->symbol
       (symbol->string 'JollyWog)))
          ⟹  #t
(string=? "K. Harper, M.D."
          (symbol->string
            (string->symbol "K. Harper, M.D.")))
          ⟹  #t
```

6.3.4. 字符

字符是表示字母、数字等打印字符的对象。书写字符时使用 `#\`⟨character⟩ 或 `#\`⟨character name⟩ 的记法。例如：

```
#\a         ; 小写字母
#\A         ; 大写字母
#\(         ; 左括号
#\          ; 空格
#\space     ; 更好的表示空格的记法
#\newline   ; 换行符
```

在 `#\`⟨character⟩ 中，大小写是敏感的，但在 `#\`⟨character name⟩ 中，大小写是不敏感的。如果 `#\`⟨character⟩ 中的 ⟨character⟩ 是字母，那么 ⟨character⟩ 后面的字符必须是一个分隔符，如空格或括号。此规则可防止以下歧义：例如，字符序列“`#\space`”既可能被视作空格字符的表示，也可能被视作字符“`#\s`”后面跟着符号“`pace`”。

使用 `#\` 记法书写的字符的值就是它自身，即，它们在程序中无需被引用。

一些操作字符的过程忽略大写和小写的不同。忽略大小写的过程的名字中都含有“`-ci`”（“Case insensitive”的缩写）。

`(char?` *obj*`)` procedure

如果 *obj* 是字符，则返回 `#t`，否则返回 `#f`。

`(char=?` $char_1$ $char_2$`)` procedure
`(char<?` $char_1$ $char_2$`)` procedure
`(char>?` $char_1$ $char_2$`)` procedure
`(char<=?` $char_1$ $char_2$`)` procedure
`(char>=?` $char_1$ $char_2$`)` procedure

这些过程为整个字符集引入了全序，该顺序应保证：

- 大写字符是有序的。例如，`(char<? #\A #\B)` 返回 `#t`。
- 小写字符是有序的。例如，`(char<? #\a #\b)` 返回 `#t`。
- 数字是有序的。例如，`(char<? #\0 #\9)` 返回 `#t`。
- 或者所有数字排在所有大写字符之前，或者相反。
- 或者所有数字排在所有小写字符之前，或者相反。

某些实现可能会扩展这些过程，使它们支持两个以上的参数，就像相应的数值谓词一样。

`(char-ci=?` $char_1$ $char_2$`)` library procedure
`(char-ci<?` $char_1$ $char_2$`)` library procedure
`(char-ci>?` $char_1$ $char_2$`)` library procedure
`(char-ci<=?` $char_1$ $char_2$`)` library procedure
`(char-ci>=?` $char_1$ $char_2$`)` library procedure

这些过程类似于 `char=?` 等过程，但它们将大写和小写字符视为相同的情况。例如，`(char-ci=? #\A #\a)` 返回 `#t`。某些实现可能会扩展这些过程，使它们支持两个以上的参数，就像相应的数值谓词一样。

`(char-alphabetic?` *char*`)` library procedure
`(char-numeric?` *char*`)` library procedure
`(char-whitespace?` *char*`)` library procedure
`(char-upper-case?` *letter*`)` library procedure
`(char-lower-case?` *letter*`)` library procedure

如果这些过程的参数分别是字母、数字、空白、大写或小写字符，它们就返回 `#t`，否则返回 `#f`。以下特定于 ASCII 字符集的注解仅具有指导意义：字母是 52 个大写和小写字符；数字是 10 个十进制数字；空白可以是空格、制表符、换行符、换页符或回车。

(char->integer *char*) procedure
(integer->char *n*) procedure

给定一个字符， char->integer 返回一个表示该字符的精确整数。给定某字符通过 char->integer 得到的整数，integer->char 返回原字符。这些过程在符合 char<=? 顺序的字符集和符合 <= 顺序的整数子集之间实现了保序同态 (Order-preserving isomorphisms)。也就是说，如果

(char<=? *a b*) ⟹ #t and (<= *x y*) ⟹ #t

且 *x* 和 *y* 位于 integer->char 的定义域内，则

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

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

(char-upcase *char*) library procedure
(char-downcase *char*) library procedure

这些过程返回可满足 (char-ci=? *char* $char_2$) 的字符 $char_2$。此外，如果 *char* 是字母，则 char-upcase 的结果是大写字符而 char-downcase 的结果是小写字符。

6.3.5. 字符串

字符串是字符的序列。字符串记作用双引号 (") 包围的一系列字符。在字符串内，只能用加反斜线 (\) 这样的转义方式表示双引号，如：

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

在字符串内，只能用连续两个反斜线这样的转义方式表示反斜线符号。 Scheme 没有定义字符串中出现的，后面未跟有双引号或反斜线的反斜线符号的作用。

字符串常量可以从一行连通到下一行，但这样的字符串的准确内容是未定义的。

字符串的长度是它包含的字符数量。这个数量是一个精确的、非负的整数。字符串创建后，其长度就是确定的。字符串的有效索引是小于字符串长度的精确的非负整数。 字符串中第一个字符的索引是 0，第二个字符的索引是 1，依此类推。

在类似“*string* 中开始于索引 *start* 而终止于索引 *end* 的所有字符”这样的说法中，索引 *start* 是被包含在内的，而索引 *end* 是被排除在外的。因此，如果 *start* 和 *end* 是相同的索引，上述说法指的就是空字符串；如果 *start* 等于零而 *end* 等于 *string* 的长度，则上述说法指的就是整个字符串。

一些操作字符串的过程忽略大写和小写的不同。忽略大小写的版本的名字中都含有“-ci”（“Case insensitive”的缩写）。

(string? *obj*) procedure

如果 *obj* 是字符串，则返回 #t，否则返回 #f。

(make-string *k*) procedure
(make-string *k char*) procedure

make-string 返回长度为 *k* 的新分配的字符串。如果给出了 *char*，则字符串中的所有元素都初始化为 *char*，否则 *string* 的内容是未定义的。

(string *char ...*) library procedure

返回由此过程的参数组成的新分配的字符串。

(string-length *string*) procedure

返回 *string* 中的字符数量。

(string-ref *string k*) procedure

k 必须是 *string* 的有效索引。 string-ref 使用从零开始的索引返回 *string* 中的第 *k* 号字符。

(string-set! *string k char*) procedure

k 必须是 *string* 的有效索引。 string-set! 将 *char* 存入 *string* 的第 *k* 号元素并返回一个未定义的值。

```
(define (f) (make-string 3 #\*))
(define (g) "***")
(string-set! (f) 0 #\?)     ⟹  未定义
(string-set! (g) 0 #\?)     ⟹  错误
(string-set! (symbol->string 'immutable)
             0
             #\?)           ⟹  错误
```

(string=? $string_1$ $string_2$) library procedure
(string-ci=? $string_1$ $string_2$) library procedure

如果两个字符串长度相同且在同样的位置包含同样的字符，则返回 #t，否则返回 #f。 string-ci=? 把大写和小写字符视为同样的字符，但 string=? 把大写和小写字符视为不同的字符。

(string<? $string_1$ $string_2$) library procedure
(string>? $string_1$ $string_2$) library procedure
(string<=? $string_1$ $string_2$) library procedure
(string>=? $string_1$ $string_2$) library procedure
(string-ci<? $string_1$ $string_2$) library procedure
(string-ci>? $string_1$ $string_2$) library procedure
(string-ci<=? $string_1$ $string_2$) library procedure
(string-ci>=? $string_1$ $string_2$) library procedure

这些过程是相应的字符序在字符串中的词典扩展。例如，string<? 是由 char<? 导出的字符串的词典顺序。如果两个字符串长度不同，但其字符在较短字符串的长度范围内相同，就认为较短字符串在词典顺序上小于较长的字符串。

Scheme 实现可以扩展以上过程及 string=? 和 string-ci=? 过程，使它们支持两个以上的参数，就像相应的数值谓词一样。

(substring *string start end*) library procedure

string 必须是一个字符串， *start* 和 *end* 必须是满足

$$0 \leq start \leq end \leq \texttt{(string-length } string\texttt{)}.$$

的精确整数。 substring 过程返回一个新分配的字符串，该字符串包含 *string* 中开始于索引 *start* （含），终止于索引 *end* （不含）的所有字符。

(string-append *string* ...) library procedure

返回一个新分配的字符串，其中的字符是所有给定字符串的串联。

(string->list *string*) library procedure
(list->string *list*) library procedure

string->list 返回新分配的表，该表中包含构成给定字符串的所有字符。 list->string 返回新分配的字符串，该字符串由表 *list* 中的字符构成，其中的表 *list* 必须是字符的表。在 equal? 的语义下， string->list 和 list->string 是互逆的过程。

(string-copy *string*) library procedure

返回新分配的 *string* 的拷贝。

(string-fill! *string char*) library procedure

将 *char* 存入 *string* 中的每个元素并返回未定义的值。

6.3.6. 向量

向量是一种元素由整数索引的异构结构。向量通常比相同长度的表占据更少的空间，访问随机选定的元素时，向量需要的平均时间通常也比表少。

向量的长度是它包含的元素数量。这个数量是一个精确的、非负的整数。向量创建后，其长度就是确定的。向量的有效索引是小于向量长度的精确的非负整数。 向量中第一个元素的索引是 0，最后一个元素的索引比向量的长度小 1。

向量的记法是 #(*obj* ...)。例如，一个长度为 3，第 0 号元素为数值 0，第 1 号元素为表 (2 2 2 2)，第 2 号元素为字符串 "Anna" 的向量可以记作：

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

注意这是向量的外部表示，而不是一个值为向量的表达式。像表常量一样，向量的常量也必须被引用：

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

(vector? *obj*) procedure

如果 *obj* 是一个向量，则返回 #t，否则返回 #f。

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

返回包含 *k* 个元素的新分配的向量。如果给出了第二个参数，则向量中的每个元素都初始化为 *fill*，否则向量中每个元素的初始内容是未定义的。

(vector *obj* ...) library procedure

返回新分配的向量，其元素包含所有给定的参数。与 list 过程类似。

```
(vector 'a 'b 'c)        ⟹  #(a b c)
```

(vector-length *vector*) procedure

以精确整数返回 *vector* 中的元素数量。

(vector-ref *vector k*) procedure

k 必须是 *vector* 的有效索引。 vector-ref 返回 *vector* 中第 *k* 号元素的内容。

```
(vector-ref '#(1 1 2 3 5 8 13 21)
            5)
          ⟹  8
(vector-ref '#(1 1 2 3 5 8 13 21)
            (let ((i (round (* 2 (acos -1)))))
              (if (inexact? i)
                  (inexact->exact i)
                  i)))
          ⟹ 13
```

(vector-set! *vector k obj*) procedure

k 必须是 *vector* 的有效索引。 vector-set! 将 *obj* 存入 *vector* 的第 *k* 号元素。 vector-set! 的返回值是未定义的。

```
(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")
          ⟹  错误  ; 常数向量
```

(vector->list *vector*) library procedure
(list->vector *list*) library procedure

vector->list 返回新分配的表，该表包含 *vector* 中的所有对象。list->vector 返回新分配的向量，该向量包含表 *list* 中的所有元素。

```
(vector->list '#(dah dah didah))
          ⟹  (dah dah didah)
(list->vector '(dididit dah))
          ⟹  #(dididit dah)
```

(vector-fill! *vector fill*) library procedure

将 *fill* 存入 *vector* 的每个元素。 vector-fill! 的返回值是未定义的。

6.4. 控制特征

本章描述了各种以特定方式控制程序执行顺序的基本过程。 procedure? 谓词也在这里描述。

(procedure? *obj*) procedure

如果 *obj* 是一个过程，则返回 #t，否则返回 #f。

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

(apply *proc* arg_1 ... *args*) procedure

proc 必须是一个过程， *args* 必须是一个表。以表 (append (list arg_1 ...) *args*) 中的元素为实参调用 *proc*。

```
(apply + (list 3 4))        ⟹  7

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

((compose sqrt *) 12 75)    ⟹  30
```

(map *proc* $list_1$ $list_2$ ...) library procedure

list 必须是表， *proc* 必须是一个参数数量和 *list* 数量相同且只返回一个值的过程。如果给出了两个或多个 *list*，则它们的长度必须相同。 map 逐元素地将 *proc* 应用于 *list* 中的元素，并返回一个按顺序包含所有结果的表。 *proc* 作用于 *list* 元素的动态顺序是未定义的。

```
(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) 或 (2 1)
```

(for-each *proc* $list_1$ $list_2$ ...) library procedure

for-each 的参数和 map 的参数类似，但 for-each 调用 *proc* 的目的在于关注其副作用，而非关注其值。不同于 map， for-each 保证 *proc* 作用于 *list* 中的元素时，会遵循从第一个元素到最后一个元素的顺序， for-each 的返回值是未定义的。

```
(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)
```

(force *promise*) library procedure

强制得到承诺 *promise* 的值（参见 delay，第 4.2.5 节）。如果该承诺此前没有被求过值，则计算该承诺的值并返回。该承诺的值被缓冲（或被记忆），因此，当它第二次被强制时，将返回上一次求得的值。

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

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

(head (tail (tail a-stream)))
                            ⟹  2
```

force 和 delay 主要是为编写函数式风格的程序提供的。以下示例不是为了阐明良好的编程风格，但它们反映了无论一个承诺被强制了多少次，都只会求得一个值的特性。

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p                           ⟹  一个承诺
(force p)                   ⟹  6
p                           ⟹  仍是一个承诺
(begin (set! x 10)
       (force p))           ⟹  6
```

这里给出 delay 和 force 的一种可能的实现。承诺在这里被实现为没有参数的过程， force 简单地调用自己的参数：

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

我们将表达式

```
(delay ⟨expression⟩)
```

定义为与以下过程

```
(make-promise (lambda () ⟨expression⟩))
```

的语义相同。定义方法是：

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

其中， make-promise 的定义如下：

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

原理： 正如上面最后一个示例那样，一个承诺可以指代它自己的值。强制这样的承诺可能会造成，在求得该承诺被第一次强制的值以前，该承诺就被第二次强制。这增加了 make-promise 定义的复杂度。

某些实现支持对 delay 和 force 上述语义的不同扩展方式：

- 对一个不是承诺的对象调用 force，可以简单地返回该对象。
- 或许没有可行的办法将一个承诺与其被强制所得的值区分开来。也就是说，根据实现的不同，下面这样的表达式的值可以是 #t，也可以是 #f：

```
(eqv? (delay 1) 1)          ⟹  未定义
(pair? (delay (cons 1 2)))  ⟹  未定义
```

- 一些实现可能实现了“隐式强制 (Implicit forcing)”，即承诺的值可由基本过程，如 cdr 和 + 等强制得到：

```
(+ (delay (* 3 7)) 13)      ⟹  34
```

(call-with-current-continuation *proc*) procedure

proc 必须是拥有一个参数的过程。过程 call-with-current-continuation 将当前继续（参见下面的原理部分）打包为一个“逃逸过程 (Escape procedure)”， 并将该过程作为参数传入 *proc*。逃逸过程是一个 Scheme 过程，如果它在随后被调用，它就会放弃当时起作用的任何继续，而使用逃逸过程创建时起作用的继续。调用逃逸过程可导致 dynamic-wind 安装的 *before* 和 *after* 程序被调用。

逃逸过程的参数数量与最初调用 call-with-current-continuation 时的继续的参数数量相同。除了由 call-with-values 创建的继续以外，所有继续都只有一个参数。为那些并非由 call-with-values 创建的继续传入零个或多于一个的参数，其结果是未定义的。

传入 *proc* 的逃逸过程像其他 Scheme 过程那样拥有无限的生存期。它可以被存入变量或数据结构中，可以被调用任意多次。

以下示例只显示了 call-with-current-continuation 的最常见用法。如果所有真实的应用都像这些示例一样简单，像 call-with-current-continuation 这样强大的过程也就没有存在的必要了。

```
(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
```

原理：

call-with-current-continuation 的一种常见用法是在循环或过程体内实现结构化的非局部退出。但实际上，call-with-current-continuation 对实现大量高级控制结构来说是极为有用的。

对一个 Scheme 表达式求值时，就会有一个希望得到该表达式结果的继续存在。继续表示的是该计算的全部（缺省）未来。例

如，如果在最高层对表达式求值，则其继续会获得表达式的结果，打印输出至屏幕，提示下一个输入，执行输入的表达式，如此永不停歇。大多数时候，继续包含用户代码指定的动作，例如，某个继续会获取结果，把结果与某个局部变量里存储的值相乘，加上七，并将答案反馈给最高层继续以打印输出。通常，这些无处不在的继续是隐藏在幕后的，程序员不会过多地考虑它们。但在少数场合，程序员可能需要显式地处理继续。`call-with-current-continuation` 允许 Scheme 程序员创建一个行为像当前继续一样的过程，以显式地处理继续。

大多数程序设计语言都使用一种或多种有特定用途的逃逸结构，其名称包括 `exit`、`return`，甚至还有 `goto` 等。但在 1965 年，Peter Landin[16] 发明了一种名叫 J-operator 的通用逃逸运算符。John Reynolds[24] 在 1972 年描述了一种更简单但同样强大的结构。在 1975 年的 Scheme 报告中，Sussman 和 Steele 描述的特殊形式 `catch` 和 Reynolds 描述的结构完全一致，但它的名字来自 MacLisp 中一个不那么通用的结构。一些 Scheme 实现者注意到，`catch` 的全部功能可以由一个过程而非一个特殊的语法结构提供。于是，在 1982 年，人们发明了 `call-with-current-continuation` 这个名字。这是一个描述性的名字，但有些人不喜欢这么长的名字，他们使用 `call/cc` 来代替它。

(`values` *obj* ...) procedure

把所有参数传递给自己的继续。除了由 `call-with-values` 创建的继续以外，所有继续都只有一个参数。`values` 可以按如下方式定义：

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

(`call-with-values` *producer consumer*) procedure

调用参数 *producer*，调用时不传入参数，并使用一个特定的继续：当一些值被传入该继续时，该继续就以这些值为参数调用 *consumer* 过程。调用 *consumer* 时的继续就是调用 `call-with-values` 时的继续。

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

(call-with-values * -)      ⟹  -1
```

(`dynamic-wind` *before thunk after*) procedure

以无参数的方式调用 *thunk*，返回该调用的（一个或多个）结果。也以无参数的方式调用 *before* 和 *after*，但必须遵循以下规则（注意，如果没有调用 `call-with-current-continuation` 捕获的继续，则三个参数将按顺序被依次调用）：每当程序的执行进入调用 *thunk* 的动态生存期 (Dynamic extent) 时，*before* 就会被调用；每当程序的执行离开调用 *thunk* 的动态生存期时，*after* 就会被调用。过程调用的动态生存期是介于发起调用和调用返回之间的时间段。在 Scheme 中，因为有 `call-with-current-continuation` 的存在，一个调用的动态生存期可能不是一个单独的、连续的时间段，其定义如下：

- 被调用的过程体开始执行时，即进入其动态生存期。
- 当程序的执行不在动态生存期内，且一个在动态生存期内（被 `call-with-current-continuation`）捕获的继续被调用时，也进入其动态生存期。
- 被调用的过程返回时，离开其动态生存期。
- 当程序的执行位于动态生存期内，且一个在动态生存期以外捕获的继续被调用时，也离开其动态生存期。

如果在调用 *thunk* 的动态生存期内发生了第二次对 `dynamic-wind` 的调用，且在随后执行的继续中，两次调用 `dynamic-wind` 时创建的 *after* 都会被调用，那么，和 `dynamic-wind` 的第二次（内部）调用相关联的 *after* 会首先被调用。

如果在调用 *thunk* 的动态生存期内发生了第二次对 `dynamic-wind` 的调用，且在随后执行的继续中，两次调用 `dynamic-wind` 时创建的 *before* 都会被调用，那么，和 `dynamic-wind` 的第一次（外部）调用相关联的 *before* 会首先被调用。

如果执行某个继续时需要调用一个 `dynamic-wind` 调用创建的 *before*，并调用另一个 `dynamic-wind` 调用创建的 *after*，则 *after* 会被首先调用。

使用被捕获的继续进入或离开 *before* 或 *after* 调用的动态生存期，其结果是未定义的。

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

        ⟹ (connect talk1 disconnect
            connect talk2 disconnect)
```

6.5. 求值

(`eval` *expression environment-specifier*) procedure

在指定的环境中求得并返回 *expression* 的值。*expression* 必须是一个以数据形式表示的有效的 Scheme 表达式，

environment-specifier 必须是下面描述的三个过程之一的返回值。 Scheme 实现可以扩展 `eval`，以使其第一个参数支持非表达式的程序（定义），并允许其他值作为环境传入，但扩展时必须保证， `eval` 不能在与 `null-environment` 或 `scheme-report-environment` 关联的环境中创建新的绑定。

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

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

`(scheme-report-environment` *version*`)` procedure
`(null-environment` *version*`)` procedure

version 必须是对应于本报告（Scheme 修订 5 报告）的修订版本号的精确整数 `5`。 `scheme-report-environment` 返回一个环境定义符，该定义符所指示的环境只包含所有本报告定义为必需，或者本报告定义为可选但为 Scheme 实现所支持的绑定。`null-environment` 返回一个环境定义符，该定义符所指示的环境只包含所有本报告定义为必需，或者本报告定义为可选但为 Scheme 实现所支持的语法关键字的（语法）绑定。

可以使用其他的 *version* 值以指定与本报告的先前修订版本相匹配的环境，但对它们的支持不是必需的。如果 *version* 既不是 `5`，也不是 Scheme 实现支持的其他值，Scheme 实现将报告一个错误。

（通过使用 `eval`）为一个在 `scheme-report-environment` 中绑定的变量（如 `car`）赋值的结果是未定义的。因此，`scheme-report-environment` 指定的环境可以是不可变的。

`(interaction-environment)` optional procedure

此过程返回一个环境定义符，该定义符所指示的环境包含 Scheme 实现定义的绑定，通常是本报告中列出的绑定的超集；其目的是，在此过程返回的环境中， Scheme 实现可以求得用户动态输入的表达式的值。

6.6. 输入和输出

6.6.1. Ports

端口 (Port) 表示输入和输出设备。对于 Scheme 来说，输入端口是一个可以根据命令发送字符的 Scheme 对象，输出端口是一个可以接收字符的 Scheme 对象。

`(call-with-input-file` *string proc*`)` library procedure
`(call-with-output-file` *string proc*`)` library procedure

string 应为命名某文件的字符串， *proc* 应为接收一个参数的过程。调用 `call-with-input-file` 时，文件应当已经存在；调用 `call-with-output-file` 时，如果文件已经存在，其结果是未定义的。这些过程用一个参数调用 *proc* 过程，该参数是以输入或输出方式打开给定名字的文件所得到的端口。如果不能打开该文件，就报告一个错误。如果 *proc* 返回，该端口就会被自动关闭， *proc* 产生的（一个或多个）值被返回。如果 *proc* 没有返回，该端口就不会被自动关闭，除非可以证明该端口再也不会被读写操作使用了。

原理： 因为 Scheme 的逃逸过程拥有无限的生存期，我们可以从当前继续中逃逸并在随后再次逃回。如果允许 Scheme 实现在任意逃离当前继续的地方关闭端口，那么我们就无法写出既使用了 `call-with-current-continuation` 也使用了 `call-with-input-file` 或 `call-with-output-file` 的可移植代码了。

`(input-port?` *obj*`)` procedure
`(output-port?` *obj*`)` procedure

如果 *obj* 分别是输入端口或输出端口，则返回 `#t`，否则返回 `#f`。

`(current-input-port)` procedure
`(current-output-port)` procedure

返回当前缺省的输入或输出端口。

`(with-input-from-file` *string thunk*`)`
optional procedure
`(with-output-to-file` *string thunk*`)`
optional procedure

string 应为命名某文件的字符串， *thunk* 应为没有参数的过程。调用 `with-input-from-file` 时，文件应当已经存在；调用 `with-output-to-file` 时，如果文件已经存在，其结果是未定义的。该文件被以输入或输出方式打开，一个连接到它的输入或输出端口被设置为 `current-input-port` 或 `current-output-port` 的缺省返回值（并被 `(read)`、 `(write` *obj*`)`等过程使用），并以无参数方式调用 *thunk*。当 *thunk* 返回后，该端口被关闭，先前的缺省端口被恢复。`with-input-from-file` 和 `with-output-to-file` 返回 *thunk* 产生的（一个或多个）值。如果使用一个逃逸过程从这些过程的继续中逃逸，它们的行为是依赖于实现的。

`(open-input-file` *filename*`)` procedure

此过程的参数是命名一个已存在文件的字符串，此过程返回一个可以发送该文件中字符的输入端口。如果不能打开文件，就报告一个错误。

`(open-output-file` *filename*`)` procedure

此过程的参数是命名一个待创建的输出文件的字符串，此过程返回一个可以向使用该名字创建的新文件中写入字符的输出端口。如果不能打开文件，就报告一个错误。如果给定名字的文件已经存在，其结果是未定义的。

(close-input-port *port*) procedure
(close-output-port *port*) procedure

关闭与 *port* 关联的文件，令 *port* 不能再发送或接收字符。如果文件已经被关闭，这些过程不会产生任何作用。它们的返回值是未定义的。

6.6.2. 输入

(read) library procedure
(read *port*) library procedure

read 将 Scheme 对象的外部表示转换为对象本身。也就是说，它是一个非终结符 ⟨datum⟩ 的解析器（参见第 7.1.2 和第 6.3.2 节）。read 从给定的输入端口 *port* 中返回下一个可解析的对象，并将 *port* 更新为指向该对象外部表示之后的第一个字符。

在找到任何一个对象的起始字符之前，如果在输入中遇到了文件结尾，就返回一个文件结尾 (End of file) 对象，端口保持打开状态，后续的试图读取对象的操作也将返回一个文件结尾对象。如果在开始输入对象的外部表示后遇到了文件结尾，但外部表示还不完整，无法解析，就报告一个错误。

port 参数可以省略，这时它的缺省值是 current-input-port 的返回值。读取已经关闭的端口是一个错误。

(read-char) procedure
(read-char *port*) procedure

返回输入端口 *port* 中的下一个可用字符，将 *port* 更新为指向后续的字符。如果没有字符可以读取，则返回一个文件结尾对象。 *port* 可以省略，这时它的缺省值是 current-input-port 的返回值。

(peek-char) procedure
(peek-char *port*) procedure

返回输入端口 *port* 中的下一个可用字符，不将 *port* 更新为指向后续的字符。如果没有字符可以读取，则返回一个文件结尾对象。 *port* 可以省略，这时它的缺省值是 current-input-port 的返回值。

注： peek-char 调用的返回值和针对同一个 *port* 的 read-char 调用的返回值相同。惟一的区别是，下一个针对该 *port* 的 read-char 或 peek-char 调用将返回前一个 peek-char 调用的返回值。特别地，只要针对交互式端口的 read-char 会挂起并等待输入，针对同样端口的 peek-char 调用就会挂起并等待输入。

(eof-object? *obj*) procedure

如果 *obj* 是一个文件结尾对象，则返回 #t，否则返回 #f。在不同的 Scheme 实现中，文件结尾对象的精确集合并不相同，但无论如何，没有哪个文件结尾对象是可以被 read 读入的对象。

(char-ready?) procedure
(char-ready? *port*) procedure

如果输入端口 *port* 中的字符已就绪，则返回 #t，否则返回 #f。如果 char-ready 返回 #t，则下一个针对给定 *port* 的 read-char 操作就一定不会挂起。如果 *port* 位于文件结尾，则 char-ready? 返回 #t。 *port* 可以省略，这时它的缺省值是 current-input-port 的返回值。

原理： char-ready? 使程序可以从交互式端口接收字符，而不用一直等待输入。任何与此类端口关联的输入编辑器必须确保，一个已经被 char-ready? 证实存在的字符不会被抹去。如果 char-ready? 在文件结尾处返回 #f，一个位于文件结尾处的端口就不能与一个没有已就绪字符的交互式端口区别开了。

6.6.3. 输出

(write *obj*) library procedure
(write *obj port*) library procedure

向给定端口 *port* 输出 *obj* 的书面表示。出现在书面表示内的字符串被双引号包围，在这些字符串中，反斜线和双引号字符被反斜线转义。字符对象使用 #\ 记法输出。 write 返回未定义的值。 *port* 参数可以省略，这时它的缺省值是 current-output-port 的返回值。

(display *obj*) library procedure
(display *obj port*) library procedure

向给定端口 *port* 输出 *obj* 的表示。出现在书面表示内的字符串不被双引号包围，在这些字符串中，没有任何字符使用转义记法。出现在该表示中的字符对象以 write-char 而非 write 的形式输出。 display 返回未定义的值。 *port* 参数可以省略，这时它的缺省值是 current-output-port 的返回值。

原理： write 用于生成机器可读的输出， display 用于生成人可读的输出。允许符号包含“砍削”特征的实现或许希望由 write 而不是 display 来“砍削”符号中古怪的字符。

(newline) library procedure
(newline *port*) library procedure

向 *port* 输出一个行结束符。此过程的精确行为因操作系统的不同而不同。此过程返回未定义的值。 *port* 参数可以省略，这时它的缺省值是 current-output-port 的返回值。

(write-char *char*) procedure
(write-char *char port*) procedure

向给定端口 *port* 输出字符 *char* （不是字符的外部表示）并返回未定义的值。 *port* 参数可以省略，这时它的缺省值是 current-output-port 的返回值。

6.6.4. 系统接口

有关系统接口的问题超出了本报告的讨论范围。但以下操作非常重要，值得在此描述。

`(load` *filename*`)` optional procedure

filename 是命名一个已存在文件的字符串，该文件中包含 Scheme 源代码。`load` 过程从该文件中读入表达式和定义并依次对它们求值。表达式的结果是否要打印输出是未定义的。`load` 过程不影响 `current-input-port` 和 `current-output-port` 的返回值。`load` 返回未定义的值。

原理： 从可移植性考虑，`load` 的操作对象必须是源文件。它对其他文件类型进行的操作一定是因实现的不同而不同的。

`(transcript-on` *filename*`)` optional procedure
`(transcript-off)` optional procedure

filename 必须是命名一个待创建的输出文件的字符串。`transcript-on` 的作用是以输出方式打开命名的文件，并使得用户和 Scheme 系统间后续交互的副本被写入该文件。`transcript-off` 调用可结束副本输出并关闭副本文件。任何时间都只能有一个副本处于输出状态，但一些实现也可以放宽这一限制。这些过程的返回值是未定义的。

7. 形式语法和语义

本报告的前几章中已经非正式地描述了 Scheme 的语法和语义，本章提供了它们的形式化描述。

7.1. 形式语法

本节用扩展的 BNF 描述了 Scheme 的形式语法。

语法中的所有空格都是为了便于阅读而存在的。语法对大小写是不敏感的，例如，`#x1A` 和 `#X1a` 是等价的。⟨empty⟩ 代表空字符串。

以下对 BNF 的扩展可以使描述更为简洁：⟨thing⟩* 代表零个或更多的 ⟨thing⟩；⟨thing⟩$^+$ 代表至少一个 ⟨thing⟩。

7.1.1. 词法结构

本节描述了字符序列如何构成单独的记号（Token，如标识符、数值等）。下面几节描述了记号序列如何构成表达式和程序。

⟨intertoken space⟩ 可以出现在任意记号的两侧，但不能出现在记号内部。

需要隐式终结的记号（如标识符、数值、字符和句点等）可以被任意 ⟨delimiter⟩ 终结，但不应被其他任何语法元素终结。

以下五个字符保留给将来的语言扩展使用：`[ ] { } |`

⟨token⟩ ⟶ ⟨identifier⟩ | ⟨boolean⟩ | ⟨number⟩
    | ⟨character⟩ | ⟨string⟩
    | `(` | `)` | `#(` | `'` | `` ` `` | `,` | `,@` | `.`
⟨delimiter⟩ ⟶ ⟨whitespace⟩ | `(` | `)` | `"` | `;`
⟨whitespace⟩ ⟶ ⟨space or newline⟩
⟨comment⟩ ⟶ `;` ⟨all subsequent characters up to a line break⟩
⟨atmosphere⟩ ⟶ ⟨whitespace⟩ | ⟨comment⟩
⟨intertoken space⟩ ⟶ ⟨atmosphere⟩*

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

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

| `let` | `let*` | `letrec` | `do` | `delay`
| `quasiquote`

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

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

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

⟨number⟩ ⟶ ⟨num 2⟩| ⟨num 8⟩
| ⟨num 10⟩| ⟨num 16⟩

以下的 ⟨num R⟩、⟨complex R⟩、⟨real R⟩、⟨ureal R⟩、⟨uinteger R⟩ 和 ⟨prefix R⟩ 规则适用于 R = 2, 8, 10, 16 这四种情况。没有 ⟨decimal 2⟩、⟨decimal 8⟩ 和 ⟨decimal 16⟩ 规则，也就是说，包含小数点或指数的数值都必须是十进制的。

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

⟨suffix⟩ ⟶ ⟨empty⟩
| ⟨exponent marker⟩ ⟨sign⟩ ⟨digit 10⟩$^+$
⟨exponent marker⟩ ⟶ `e` | `s` | `f` | `d` | `l`
⟨sign⟩ ⟶ ⟨empty⟩ | `+` | `-`
⟨exactness⟩ ⟶ ⟨empty⟩ | `#i` | `#e`
⟨radix 2⟩ ⟶ `#b`
⟨radix 8⟩ ⟶ `#o`
⟨radix 10⟩ ⟶ ⟨empty⟩ | `#d`
⟨radix 16⟩ ⟶ `#x`
⟨digit 2⟩ ⟶ `0` | `1`
⟨digit 8⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`
⟨digit 10⟩ ⟶ ⟨digit⟩
⟨digit 16⟩ ⟶ ⟨digit 10⟩ | `a` | `b` | `c` | `d` | `e` | `f`

7.1.2. 外部表示

⟨datum⟩ 是 `read` 过程（第 6.6.2 节）成功解析的内容。注意，任何可以被解析为 ⟨expression⟩ 的字符串也都可以被解析为 ⟨datum⟩。

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

7.1.3. 表达式

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

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

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

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

⟨assignment⟩ ⟶ `(set!` ⟨variable⟩ ⟨expression⟩`)`

⟨derived expression⟩ ⟶
`(cond` ⟨cond clause⟩$^+$`)`
| `(cond` ⟨cond clause⟩* `(else` ⟨sequence⟩`))`
| `(case` ⟨expression⟩
⟨case clause⟩$^+$`)`

```
    | (case ⟨expression⟩
        ⟨case clause⟩*
        (else ⟨sequence⟩))
    | (and ⟨test⟩*)
    | (or ⟨test⟩*)
    | (let (⟨binding spec⟩*) ⟨body⟩)
    | (let ⟨variable⟩ (⟨binding spec⟩*) ⟨body⟩)
    | (let* (⟨binding spec⟩*) ⟨body⟩)
    | (letrec (⟨binding spec⟩*) ⟨body⟩)
    | (begin ⟨sequence⟩)
    | (do (⟨iteration spec⟩*)
          (⟨test⟩ ⟨do result⟩)
        ⟨command⟩*)
    | (delay ⟨expression⟩)
    | ⟨quasiquotation⟩

⟨cond clause⟩ ⟶ (⟨test⟩ ⟨sequence⟩)
    | (⟨test⟩)
    | (⟨test⟩ => ⟨recipient⟩)
⟨recipient⟩ ⟶ ⟨expression⟩
⟨case clause⟩ ⟶ ((⟨datum⟩*) ⟨sequence⟩)
⟨binding spec⟩ ⟶ (⟨variable⟩ ⟨expression⟩)
⟨iteration spec⟩ ⟶ (⟨variable⟩ ⟨init⟩ ⟨step⟩)
    | (⟨variable⟩ ⟨init⟩)
⟨init⟩ ⟶ ⟨expression⟩
⟨step⟩ ⟶ ⟨expression⟩
⟨do result⟩ ⟶ ⟨sequence⟩ | ⟨empty⟩

⟨macro use⟩ ⟶ (⟨keyword⟩ ⟨datum⟩*)
⟨keyword⟩ ⟶ ⟨identifier⟩

⟨macro block⟩ ⟶
    (let-syntax (⟨syntax spec⟩*) ⟨body⟩)
    | (letrec-syntax (⟨syntax spec⟩*) ⟨body⟩)
⟨syntax spec⟩ ⟶ (⟨keyword⟩ ⟨transformer spec⟩)
```

7.1.4. 准引用

以下准引用表达式的语法不是上下文无关的。这种表示方法是一种生成无穷多个产生式规则的技巧。试想以下规则在 $D = 1, 2, 3, \ldots$ 时的拷贝。D 记录了嵌套的深度。

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

在 ⟨quasiquotation⟩ 中，一个 ⟨list qq template D⟩ 有时既可以被解释为 ⟨unquotation D⟩，也可以被解释为 ⟨splicing unquotation D⟩，这会引起混淆。这时，它会被优先解释为 ⟨unquotation⟩ 或 ⟨splicing unquotation D⟩。

7.1.5. 转换器

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

7.1.6. 程序和定义

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

`(define-syntax` ⟨keyword⟩ ⟨transformer spec⟩`)`

7.2. 形式语义

本节提供了 Scheme 的基本表达式和部分内置过程的形式指称语义。[29] 中描述了这里使用的概念和记法。该记法的概要如下：

$\langle \ldots \rangle$	序列
$s \downarrow k$	序列 s 的第 k 个元素（从1开始的索引）
$\#s$	序列 s 的长度
$s \S t$	序列 s 和 t 的串联
$s \dagger k$	除去序列 s 的前 k 个元素
$t \to a, b$	McCarthy 条件“如果 t 则 a 否则 b”
$\rho[x/i]$	置换“在 ρ 中用 x 代替 i”
x in D	x 注入域 D
$x \mid \mathtt{D}$	x 到域 D 的投影

表达式继续 (Expression continuation) 之所以采用值的序列，而非单个值的原因是，这可以简化过程调用和多个返回值的形式表示。

与点对、向量和字符串相关联的布尔标记对于可变对象为真，对于不可变对象为假。

一个调用中的求值顺序是未定义的。在这里，我们在求值前后，对调用中的参数使用互逆的强制排列操作 *permute* 和 *unpermute* 来模拟这一语义。这种做法并不十分正确，因为它错误地暗示（对于给定数量的参数）求值顺序在整个程序中总是不变的，但与从左至右的求值顺序相比，这种做法与期望中的语义更接近一些。

存储分配符 *new* 是依赖于实现的，但它必须遵循以下公理：如果 $new\,\sigma \in \mathtt{L}$，则 $\sigma\,(new\,\sigma \mid \mathtt{L}) \downarrow 2 = false$。

省略了 $\mathcal{K}$ 的定义。因为 $\mathcal{K}$ 的精确定义可能使语义复杂化，也不会有太大的价值。

如果 P 是一个程序，在该程序中，所有变量都在被引用或被赋值前定义，则 P 的语义是：

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

其中的 I* 是 P 中定义的变量序列， P′ 是使用赋值替换 P 中每个定义后产生的表达式序列， ⟨undefined⟩ 是值为 *undefined* 的表达式， $\mathcal{E}$ 是为表达式赋予语义的语义函数。

7.2.1. 抽象语法

$\mathrm{K} \in \mathrm{Con}$	常量，包括引用
$\mathrm{I} \in \mathrm{Ide}$	标识符（变量）
$\mathrm{E} \in \mathrm{Exp}$	表达式
$\Gamma \in \mathrm{Com} = \mathrm{Exp}$	命令

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

7.2.2. 论域方程

$\alpha \in \mathtt{L}$		位置
$\nu \in \mathtt{N}$		自然数
$\mathtt{T}$	$= \{false, true\}$	布尔值
$\mathtt{Q}$		符号
$\mathtt{H}$		字符
$\mathtt{R}$		数值
$\mathtt{E}_\mathrm{p}$	$= \mathtt{L} \times \mathtt{L} \times \mathtt{T}$	点对
$\mathtt{E}_\mathrm{v}$	$= \mathtt{L}^* \times \mathtt{T}$	向量
$\mathtt{E}_\mathrm{s}$	$= \mathtt{L}^* \times \mathtt{T}$	字符串
$\mathtt{M}$	$= \{false, true, null, undefined, unspecified\}$	杂项
$\phi \in \mathtt{F}$	$= \mathtt{L} \times (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$	过程值
$\epsilon \in \mathtt{E}$	$= \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E}_\mathrm{p} + \mathtt{E}_\mathrm{v} + \mathtt{E}_\mathrm{s} + \mathtt{M} + \mathtt{F}$	表达值
$\sigma \in \mathtt{S}$	$= \mathtt{L} \to (\mathtt{E} \times \mathtt{T})$	存储
$\rho \in \mathtt{U}$	$= \mathrm{Ide} \to \mathtt{L}$	环境
$\theta \in \mathtt{C}$	$= \mathtt{S} \to \mathtt{A}$	命令继续
$\kappa \in \mathtt{K}$	$= \mathtt{E}^* \to \mathtt{C}$	表达式继续
$\mathtt{A}$		响应
$\mathtt{X}$		错误

7.2.3. 语义函数

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

有意省略了 $\mathcal{K}$ 的定义。

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

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

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

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(lambda (}\mathrm{I}^*\texttt{)}\ \Gamma^*\ \mathrm{E}_0\texttt{)}]\!] = \\
&\quad \lambda\rho\kappa\,.\,\lambda\sigma\,. \\
&\qquad new\,\sigma \in \mathtt{L} \to \\
&\qquad\quad send\,(\langle new\,\sigma \mid \mathtt{L}, \\
&\qquad\qquad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^* \to \\
&\qquad\qquad\quad tievals(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa')) \\
&\qquad\qquad\qquad (extends\,\rho\,\mathrm{I}^*\,\alpha^*)) \\
&\qquad\qquad\qquad\quad \epsilon^*,
\end{aligned}$$

$\quad\quad\quad\quad \mathit{wrong}$ “wrong number of arguments”$\rangle$
$\quad\quad\quad \text{in } \texttt{E})$
$\quad\quad \kappa$
$\quad\quad (\mathit{update}\,(\mathit{new}\,\sigma \mid \texttt{L})\ \mathit{unspecified}\,\sigma),$
$\quad \mathit{wrong}$ “out of memory” σ

$\mathcal{E}[\![\texttt{(lambda (}\mathrm{I}^*\ \texttt{.}\ \mathrm{I}\texttt{)}\ \Gamma^*\ \mathrm{E}_0\texttt{)}]\!] =$
$\quad \lambda\rho\kappa\,.\,\lambda\sigma\,.$
$\quad\quad \mathit{new}\,\sigma \in \texttt{L} \rightarrow$
$\quad\quad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma \mid \texttt{L},$
$\quad\quad\quad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^* \rightarrow$
$\quad\quad\quad\quad\quad \mathit{tievalsrest}$
$\quad\quad\quad\quad\quad\quad (\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa'))$
$\quad\quad\quad\quad\quad\quad\quad (\mathit{extends}\,\rho\ (\mathrm{I}^*\ \S\ \langle\mathrm{I}\rangle)\ \alpha^*))$
$\quad\quad\quad\quad\quad\quad \epsilon^*$
$\quad\quad\quad\quad\quad\quad (\#\mathrm{I}^*),$
$\quad\quad\quad\quad\quad \mathit{wrong}$ “too few arguments”$\rangle$ in $\texttt{E})$
$\quad\quad\quad \kappa$
$\quad\quad\quad (\mathit{update}\,(\mathit{new}\,\sigma \mid \texttt{L})\ \mathit{unspecified}\,\sigma),$
$\quad\quad \mathit{wrong}$ “out of memory” σ

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

$\mathcal{E}[\![\texttt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\texttt{)}]\!] =$
$\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\ \rho\ (\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \rightarrow \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa,$
$\quad\quad\quad\quad\quad\quad\quad\quad \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa))$

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

这里和其他地方，除 *undefined* 以外的任何值都可用于替代 *unspecified*。

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

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

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

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

$\mathcal{C}[\![\Gamma_0\ \Gamma^*]\!] = \lambda\rho\theta\,.\,\mathcal{E}[\![\Gamma_0]\!]\ \rho\ (\lambda\epsilon^*\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho\theta)$

7.2.4. 辅助函数

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

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

$\mathit{wrong} : \texttt{X} \rightarrow \texttt{C}$ [implementation-dependent]

$\mathit{send} : \texttt{E} \rightarrow \texttt{K} \rightarrow \texttt{C}$
$\mathit{send} = \lambda\epsilon\kappa\,.\,\kappa\langle\epsilon\rangle$

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

$\mathit{new} : \texttt{S} \rightarrow (\texttt{L} + \{\mathit{error}\})$ [implementation-dependent]

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

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

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

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

$\mathit{tievalsrest} : (\texttt{L}^* \rightarrow \texttt{C}) \rightarrow \texttt{E}^* \rightarrow \texttt{N} \rightarrow \texttt{C}$
$\mathit{tievalsrest} =$
$\quad \lambda\psi\epsilon^*\nu\,.\,\mathit{list}\,(\mathit{dropfirst}\,\epsilon^*\nu)$
$\quad\quad\quad\quad (\mathit{single}(\lambda\epsilon\,.\,\mathit{tievals}\,\psi\ ((\mathit{takefirst}\,\epsilon^*\nu)\ \S\ \langle\epsilon\rangle)))$

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

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

$\mathit{truish} : \texttt{E} \rightarrow \texttt{T}$
$\mathit{truish} = \lambda\epsilon\,.\,\epsilon = \mathit{false} \rightarrow \mathit{false}, \mathit{true}$

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

$\mathit{unpermute} : \texttt{E}^* \rightarrow \texttt{E}^*$ [inverse of *permute*]

$\mathit{applicate} : \texttt{E} \rightarrow \texttt{E}^* \rightarrow \texttt{K} \rightarrow \texttt{C}$
$\mathit{applicate} =$
$\quad \lambda\epsilon\epsilon^*\kappa\,.\,\epsilon \in \texttt{F} \rightarrow (\epsilon \mid \texttt{F} \downarrow 2)\epsilon^*\kappa, \mathit{wrong}$ “bad procedure”

$\mathit{onearg} : (\texttt{E} \rightarrow \texttt{K} \rightarrow \texttt{C}) \rightarrow (\texttt{E}^* \rightarrow \texttt{K} \rightarrow \texttt{C})$
$\mathit{onearg} =$
$\quad \lambda\zeta\epsilon^*\kappa\,.\,\#\epsilon^* = 1 \rightarrow \zeta(\epsilon^* \downarrow 1)\kappa,$
$\quad\quad\quad \mathit{wrong}$ “wrong number of arguments”

$\mathit{twoarg} : (\texttt{E} \rightarrow \texttt{E} \rightarrow \texttt{K} \rightarrow \texttt{C}) \rightarrow (\texttt{E}^* \rightarrow \texttt{K} \rightarrow \texttt{C})$
$\mathit{twoarg} =$
$\quad \lambda\zeta\epsilon^*\kappa\,.\,\#\epsilon^* = 2 \rightarrow \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\kappa,$
$\quad\quad\quad \mathit{wrong}$ “wrong number of arguments”

$\mathit{list} : \texttt{E}^* \rightarrow \texttt{K} \rightarrow \texttt{C}$
$\mathit{list} =$
$\quad \lambda\epsilon^*\kappa\,.\,\#\epsilon^* = 0 \rightarrow \mathit{send}\ \mathit{null}\,\kappa,$
$\quad\quad\quad \mathit{list}\,(\epsilon^* \dagger 1)(\mathit{single}(\lambda\epsilon\,.\,\mathit{cons}\langle\epsilon^* \downarrow 1, \epsilon\rangle\kappa))$

$\mathit{cons} : \texttt{E}^* \rightarrow \texttt{K} \rightarrow \texttt{C}$
$\mathit{cons} =$
$\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\sigma\,.\,\mathit{new}\,\sigma \in \texttt{L} \rightarrow$
$\quad\quad\quad\quad\quad (\lambda\sigma'\,.\,\mathit{new}\,\sigma' \in \texttt{L} \rightarrow$
$\quad\quad\quad\quad\quad\quad \mathit{send}\,(\langle\mathit{new}\,\sigma \mid \texttt{L}, \mathit{new}\,\sigma' \mid \texttt{L}, \mathit{true}\rangle$
$\quad\quad\quad\quad\quad\quad\quad \text{in } \texttt{E})$
$\quad\quad\quad\quad\quad\quad\quad \kappa$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma' \mid \mathtt{L})\epsilon_2\sigma'),\\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“out of memory”}\sigma')\\
\qquad\qquad\qquad\qquad\quad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L})\epsilon_1\sigma),\\
\qquad\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“out of memory”}\sigma)
\end{array}
$$

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

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

$$
\begin{array}{l}
\mathit{car} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{car} =\\
\quad \mathit{onearg}\,(\lambda\epsilon\kappa\,.\,\epsilon \in \mathtt{E_p} \to \mathit{hold}\,(\epsilon \mid \mathtt{E_p} \downarrow 1)\kappa,\\
\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“non-pair argument to }\texttt{car}\text{”})
\end{array}
$$

$$
\mathit{cdr} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \qquad [\text{similar to } \mathit{car}]
$$

$$
\begin{array}{l}
\mathit{setcar} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{setcar} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,\epsilon_1 \in \mathtt{E_p} \to\\
\qquad\qquad\qquad (\epsilon_1 \mid \mathtt{E_p} \downarrow 3) \to \mathit{assign}\,(\epsilon_1 \mid \mathtt{E_p} \downarrow 1)\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \epsilon_2\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{send}\,\mathit{unspecified}\,\kappa),\\
\qquad\qquad\qquad \mathit{wrong}\ \text{“immutable argument to }\texttt{set-car!}\text{”},\\
\qquad\qquad\qquad \mathit{wrong}\ \text{“non-pair argument to }\texttt{set-car!}\text{”})
\end{array}
$$

$$
\begin{array}{l}
\mathit{eqv} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{eqv} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,(\epsilon_1 \in \mathtt{M} \wedge \epsilon_2 \in \mathtt{M}) \to\\
\qquad\qquad\qquad\quad \mathit{send}\,(\epsilon_1 \mid \mathtt{M} = \epsilon_2 \mid \mathtt{M} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{Q} \wedge \epsilon_2 \in \mathtt{Q}) \to\\
\qquad\qquad\qquad\quad \mathit{send}\,(\epsilon_1 \mid \mathtt{Q} = \epsilon_2 \mid \mathtt{Q} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{H} \wedge \epsilon_2 \in \mathtt{H}) \to\\
\qquad\qquad\qquad\quad \mathit{send}\,(\epsilon_1 \mid \mathtt{H} = \epsilon_2 \mid \mathtt{H} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to\\
\qquad\qquad\qquad\quad \mathit{send}\,(\epsilon_1 \mid \mathtt{R} = \epsilon_2 \mid \mathtt{R} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{E_p} \wedge \epsilon_2 \in \mathtt{E_p}) \to\\
\qquad\qquad\qquad\quad \mathit{send}\,((\lambda p_1 p_2\,.\,((p_1 \downarrow 1) = (p_2 \downarrow 1) \wedge\\
\qquad\qquad\qquad\qquad\qquad\qquad (p_1 \downarrow 2) = (p_2 \downarrow 2)) \to \mathit{true},\\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{false})\\
\qquad\qquad\qquad\qquad\quad (\epsilon_1 \mid \mathtt{E_p})\\
\qquad\qquad\qquad\qquad\quad (\epsilon_2 \mid \mathtt{E_p}))\\
\qquad\qquad\qquad\qquad\quad \kappa,\\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{E_v} \wedge \epsilon_2 \in \mathtt{E_v}) \to \ldots,\\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{E_s} \wedge \epsilon_2 \in \mathtt{E_s}) \to \ldots,\\
\qquad\qquad\qquad (\epsilon_1 \in \mathtt{F} \wedge \epsilon_2 \in \mathtt{F}) \to\\
\qquad\qquad\qquad\quad \mathit{send}\,((\epsilon_1 \mid \mathtt{F} \downarrow 1) = (\epsilon_2 \mid \mathtt{F} \downarrow 1) \to \mathit{true}, \mathit{false})\\
\qquad\qquad\qquad\qquad\quad \kappa,\\
\qquad\qquad\qquad\quad \mathit{send}\ \mathit{false}\,\kappa)
\end{array}
$$

$$
\begin{array}{l}
\mathit{apply} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{apply} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,\epsilon_1 \in \mathtt{F} \to \mathit{valueslist}\,\langle\epsilon_2\rangle(\lambda\epsilon^*\,.\,\mathit{applicate}\,\epsilon_1\epsilon^*\kappa),\\
\qquad\qquad\qquad \mathit{wrong}\ \text{“bad procedure argument to }\texttt{apply}\text{”})
\end{array}
$$

$$
\begin{array}{l}
\mathit{valueslist} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{valueslist} =\\
\quad \mathit{onearg}\,(\lambda\epsilon\kappa\,.\,\epsilon \in \mathtt{E_p} \to\\
\qquad\qquad\qquad\quad \mathit{cdr}\langle\epsilon\rangle\\
\qquad\qquad\qquad\qquad (\lambda\epsilon^*\,.\,\mathit{valueslist}\\
\qquad\qquad\qquad\qquad\qquad\quad \epsilon^*\\
\qquad\qquad\qquad\qquad\qquad\quad (\lambda\epsilon^*\,.\,\mathit{car}\langle\epsilon\rangle(\mathit{single}(\lambda\epsilon\,.\,\kappa(\langle\epsilon\rangle \S \epsilon^*))))),\\
\qquad\qquad\qquad \epsilon = \mathit{null} \to \kappa\langle\,\rangle,\\
\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“non-list argument to }\texttt{values-list}\text{”})
\end{array}
$$

$$
\begin{array}{l}
\mathit{cwcc} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \qquad [\texttt{call-with-current-continuation}]\\
\mathit{cwcc} =\\
\quad \mathit{onearg}\,(\lambda\epsilon\kappa\,.\,\epsilon \in \mathtt{F} \to\\
\qquad\qquad\qquad\quad (\lambda\sigma\,.\,\mathit{new}\,\sigma \in \mathtt{L} \to\\
\qquad\qquad\qquad\qquad\quad \mathit{applicate}\,\epsilon\\
\qquad\qquad\qquad\qquad\qquad\qquad \langle\langle \mathit{new}\,\sigma \mid \mathtt{L}, \lambda\epsilon^*\kappa'\,.\,\kappa\epsilon^*\rangle\ \text{in}\ \mathtt{E}\rangle\\
\qquad\qquad\qquad\qquad\qquad\qquad \kappa\\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L})\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unspecified}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \sigma),\\
\qquad\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“out of memory”}\,\sigma),\\
\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“bad procedure argument”})
\end{array}
$$

$$
\begin{array}{l}
\mathit{values} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}\\
\mathit{values} = \lambda\epsilon^*\kappa\,.\,\kappa\epsilon^*
\end{array}
$$

$$
\begin{array}{l}
\mathit{cwv} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \qquad [\texttt{call-with-values}]\\
\mathit{cwv} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,\mathit{applicate}\,\epsilon_1\langle\,\rangle(\lambda\epsilon^*\,.\,\mathit{applicate}\,\epsilon_2\,\epsilon^*))
\end{array}
$$

7.3. 派生表达式类型

本节使用基本表达式类型（常量、变量、调用、 `lambda`、`if` 和 `set!` 表达式）描述了派生表达式类型的宏定义。有关 `delay` 的一种可能的定义方法，参见第 6.4 节。

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

```
        (cond clause1 clause2 ...)))))

(define-syntax case
  (syntax-rules (else)
    ((case (key ...)
       clauses ...)
     (let ((atom-key (key ...)))
       (case atom-key clauses ...)))
    ((case key
       (else result1 result2 ...))
     (begin result1 result2 ...))
    ((case key
       ((atoms ...) result1 result2 ...))
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)))
    ((case key
       ((atoms ...) result1 result2 ...)
       clause clauses ...)
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)
         (case key clause clauses ...)))))

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

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

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

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

以下的 letrec 宏使用符号 <undefined> 替代一个表达式，该表达式返回某些值，但当这些值被存入一个存储位置后，试图从该位置获取这些值的操作是一个错误（Scheme 中没有定义这种表达式）。这里使用了一个生成临时名字的技巧，以避免指定求值顺序。这也可以借助一个辅助宏来实现。

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


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

下面这种 begin 的替代展开方式没有使用在一个 lambda 表达式里引入两个或多个表达式的机制。无论如何，应注意这些规则只适用于 begin 的主体内不含定义的情况。

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

以下 do 的定义使用了一种展开变量子句的技巧。像上面的 letrec 一样，这也可以用一个辅助的宏来实现。表达式 (if #f #f) 用于获得一个未定义的值。

```
(define-syntax do
```

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

注释

语言的变化

本节列举了自从“修订 4 报告”[6] 发布以来， Scheme 语言的变化情况。

- 本报告现在是 IEEE Scheme 标准[13] 的超集：符合本报告的 Scheme 实现也符合 IEEE 标准。二者的不同在于：
  - 空表现在必须被计算为 true。
  - 有关基本特征和非基本特征的分类已经被舍弃。现在，内置过程被分为三类：基本过程、库过程和可选过程。可选过程是 `load`、`with-input-from-file`、 `with-output-to-file`、 `transcript-on`、 `transcript-off` 和 `interaction-environment`，以及拥有两个以上参数的 `-` 和 `/`。 IEEE 标准不包括这些过程。
  - 允许程序重定义内置过程。这样做不会改变其他内置过程的行为。
- 将 *port* 加入了不相交类型的列表。
- 宏的附录被删除。高层的宏现在是本报告主体的一部分。派生表达式的重写规则被宏定义替代。本报告中没有保留的标识符。
- `syntax-rules` 现在支持向量模式。
- 增加了多值返回、 `eval` 和 `dynamic-wind`。
- 明确定义了必须以严格尾递归形式实现的调用。
- ‘`@`’ 可在标识符内部使用。‘`|`’ 被保留给将来可能的扩展。

附加资源

Internet 上的 Scheme 资源库位于：

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

其中包含一个收录很广的 Scheme 参考书目，还包含论文、程序、实现和其他 Scheme 相关资源。

示例程序

integrate-system 使用 Runge-Kutta 法对微分方程组

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

求积分。

参数 system-derivative 是一个函数，其参数是组状态（一个由状态变量 y_1, ..., y_n 的值组成的向量），其结果是组导数（值 y_1', ..., y_n'）。参数 initial-state 提供初始组状态，参数 h 是积分步长的初始猜测值。

integrate-system 的返回值是一个组状态的无穷流。

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

runge-kutta-4 的参数 f 是一个函数，它根据组状态产生组导数。 runge-kutta-4 返回一个函数，该函数的参数为组状态、结果为新的组状态。

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


(define elementwise
  (lambda (f)
    (lambda vectors
      (generate-vector
        (vector-length (car vectors))
        (lambda (i)
          (apply f
                 (map (lambda (v) (vector-ref  v i))
                      vectors)))))))

(define generate-vector
  (lambda (size proc)
    (let ((ans (make-vector size)))
      (letrec ((loop
                (lambda (i)
                  (cond ((= i size) ans)
                        (else
                         (vector-set! ans i (proc i))
                         (loop (+ i 1)))))))
        (loop 0)))))

(define add-vectors (elementwise +))

(define scale-vector
  (lambda (s)
    (elementwise (lambda (x) (* x s)))))
```

map-streams 类似于 map：它把第一个参数（一个过程）应用到第二个参数（一个流）中的所有元素。

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

无穷流以点对的方式实现，其 car 域包含流的第一个元素，其 cdr 域包含一个获取流的剩余元素的承诺。

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

以下代码演示了使用 integrate-system 对模拟阻尼振子的方程组

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

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

求积分的过程：

```
(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))
```

参考文献

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[22] Jonathan A. Rees, Norman I. Adams IV, and James R. Meehan. The T manual, fourth edition. Yale University Computer Science Department, January 1984.

[23] Jonathan Rees and William Clinger, editors. The revised^3 report on the algorithmic language Scheme. In *ACM SIGPLAN Notices* 21(12), pages 37–79, December 1986.

[24] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[25] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, January 1978.

[26] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[27] Guy Lewis Steele Jr. *Common Lisp: The Language, second edition.* Digital Press, Burlington MA, 1990.

[28] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349, December 1975.

[29] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, 1977.

[30] Texas Instruments, Inc. TI Scheme Language Reference Manual. Preliminary version 1.0, November 1985.

按字母序排列的概念定义、关键字和过程的索引

每个术语、过程或关键字的主要索引页码列在前面，并用分号与其他页码分隔开。