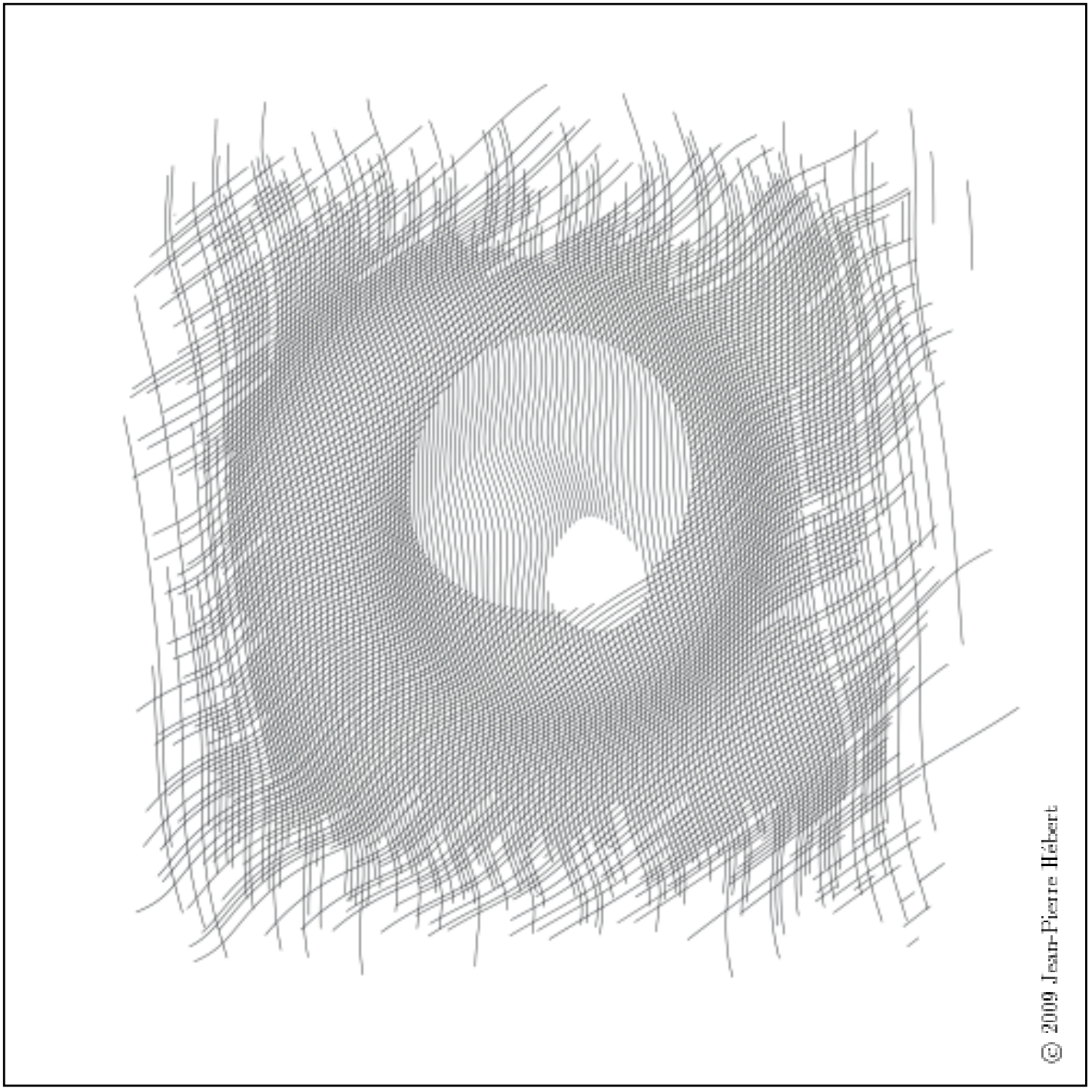

第 1 章。介绍

Scheme是一种通用的计算机编程语言。它是一种高级语言，支持对结构化数据（如字符串、列表和向量）的操作，以及对数字和字符等更传统数据的操作。虽然Scheme通常被认为是符号应用程序，但其丰富的数据类型集和灵活的控制结构使其成为一种真正通用的语言。

Scheme已被用于编写文本编辑器，优化编译器，操作系统，图形包，专家系统，数值应用程序，财务分析包，虚拟现实系统以及几乎所有其他类型的应用程序。Scheme是一种相当简单的学习语言，因为它基于一些语法形式和语义概念，并且大多数实现的交互性质鼓励实验。然而，方案是一种具有挑战性的语言，需要完全理解。培养充分发挥其潜力的能力需要仔细研究和实践。

方案程序在不同机器上同一方案实现的版本之间具有高度可移植性，因为计算机依赖关系几乎完全隐藏在程序员面前。它们也可以跨不同的实现进行移植，因为一组Scheme语言设计人员的努力，他们发布了一系列报告，即关于Scheme的“修订报告”。最近的“修订^{6}报告”[24]通过一组标准库和用于定义新的可移植库和顶级程序的标准机制强调了可移植性。

尽管一些早期的Scheme系统效率低下且速度慢，但许多基于编译器的较新的实现速度很快，程序与用较低级别语言编写的等效程序运行相同。有时仍然存在的相对低效率是由于运行时检查造成的，这些检查支持通用算术并帮助程序员检测和纠正各种常见的编程错误。在许多实现中，可能会禁用这些检查。

Scheme 支持多种类型的数据值或对象，包括字符、字符串、符号、对象的列表或向量，以及一整套数值数据类型，包括复数、实数和任意精度有理数。

保存对象内容所需的存储会根据需要动态分配并保留，直到不再需要，然后自动解除分配，通常由垃圾回收器定期恢复无法访问的对象使用的存储。简单的原子值（如小整数、字符、布尔值和空列表）通常表示为即时值，因此不会产生分配或解除分配开销。

无论表示形式如何，所有对象都是一等数据值;因为它们是无限期保留的，所以它们可以作为参数自由传递给过程，作为过程的值返回，并组合成新的对象。这与许多其他语言形成鲜明对比，在许多其他语言中，复合数据值（如数组）要么是静态分配的，从不解除分配，要么在进入代码块时分配，在退出块时无条件地解除分配，要么由程序员显式分配和解除分配。

Scheme 是一种按值调用的语言，但对于至少可变对象（可以修改的对象），这些值是指向实际存储的指针。但是，这些指针仍保留在幕后，程序员不需要意识到它们，除非了解当对象传递到过程或从过程返回时，不会复制对象的存储。

Scheme语言的核心是句法形式的一个小核心，所有其他形式都是从中构建的。这些核心形式，从它们派生的一组扩展句法形式，以及一组原始过程构成了完整的Scheme语言。Scheme的解释器或编译器可能非常小，并且可能快速且高度可靠。扩展的句法形式和许多原始过程可以在Scheme本身中定义，从而简化了实现并提高可靠性。

方案程序与方案数据结构共享一个通用的打印表示形式。因此，任何 Scheme 程序都具有作为 Scheme 对象的自然而明显的内部表示形式。例如，变量和句法关键字对应于符号，而结构化句法形式对应于列表。这种表示是Scheme提供的句法扩展设施的基础，用于根据现有句法形式和程序定义新的句法形式。它还有助于直接在 Scheme 中实现解释器、编译器和其他程序转换工具，以及 Scheme 中其他语言的程序转换工具。

方案变量和关键字在词法上是作用域的，方案程序是块结构的。标识符可以导入到程序或库中，也可以本地绑定在给定的代码块（如库、程序或过程正文）中。本地绑定仅在词法上可见，即在构成特定代码块的程序文本中。在此块之外出现同名标识符是指不同的绑定;如果块外部不存在标识符的绑定，则引用无效。块可以是嵌套的，并且一个块中的绑定可能会在周围的块中为同名标识符的绑定隐藏。绑定的作用域是绑定标识符可见的块，减去隐藏标识符所在的块的任何部分。块结构和词法范围界定有助于创建模块化、易于阅读、易于维护且可靠的程序。用于词法范围的有效代码是可能的，因为编译器可以在程序评估之前确定所有绑定的作用域以及每个标识符引用解析到的绑定。当然，这并不意味着编译器可以确定所有变量的值，因为在大多数情况下，直到程序执行，才会计算实际值。

在大多数语言中，过程定义只是名称与代码块的关联。块局部的某些变量是过程的参数。在某些语言中，只要仅在执行封闭块期间调用过程，过程定义就可能出现在另一个块或过程中。在其他情况下，只能在顶层定义过程。在 Scheme 中，一个过程定义可以出现在另一个块或过程中，并且该过程可以在此后的任何时间被调用，即使封闭的块已经完成了其执行。为了支持词法范围，过程携带词法上下文（环境）及其代码。

此外，计划程序并不总是被命名。相反，过程是字符串或数字等一等数据对象，变量以与绑定到其他对象相同的方式绑定到过程。

与大多数其他语言中的过程一样，Scheme 过程可能是递归的。也就是说，任何程序都可以直接或间接地调用自身。许多算法都是以递归方式最优雅或最有效地指定

的。递归的一种特殊情况，称为尾递归，用于表示迭代或循环。当一个过程直接返回调用另一个过程的结果时，就会发生尾部调用;当过程以递归方式直接或间接地对自身进行尾部调用时，就会发生尾部递归。需要方案实现才能将尾部调用实现为跳转（gotos），因此避免了通常与递归相关的存储开销。因此，Scheme 程序员只需要主掌握简单的过程调用和递归，而不需要承担通常的各种循环构造的负担。

方案支持定义具有延续的任意控制结构。延续是在程序中的给定点体现程序的其余部分的过程。在程序执行期间，可以随时获得延续。与其他过程一样，延续是一个一等对象，可以在创建后随时调用。每当调用它时，程序都会立即从获得延续的位置继续。延续允许实现复杂的控制机制，包括显式回溯、多线程和协程。

Scheme还允许程序员通过编写转换过程来定义新的句法形式或句法扩展，这些过程确定每个新的句法形式如何映射到现有的句法形式。这些转换过程本身在 Scheme 中借助方便的高级模式语言来表示，该语言可自动执行语法检查、输入解构和输出重建。默认情况下，通过转换过程维护词法范围，但程序员可以控制转换器输出中出现的所有标识符的作用域。句法扩展对于定义新的语言构造，模拟其他语言中的语言构造，实现内联代码扩展的效果，甚至对于在Scheme中模拟整个语言都很有用。大多数大型 Scheme 程序都是从语法扩展和过程定义的组合构建而来的。

Scheme是从Lisp语言演变而来的，被认为是Lisp的一种方言。从 Lisp 继承的方案，将值作为一等对象进行处理，几种重要的数据类型（包括符号和列表）以及将程序表示为对象等。词法范围和块结构是取自Algol 60的

特征[21]。Scheme 是第一个采用词法范围和块结构、一流程序、将尾部调用视为跳跃、延续和词法范围的句法扩展的 Lisp 方言。

Common Lisp [27] 和 Scheme 都是当代 Lisp 语言，两者的发展都受到了另一种语言的影响。与 Scheme 一样，但与早期的 Lisp 语言不同，Common Lisp 采用了词法范围界定和一流程序，尽管 Common Lisp 的语法扩展工具不尊重词法范围。然而，Common Lisp 的过程评估规则与其他对象的评估规则不同，它为过程变量维护了一个单独的命名空间，从而禁止将过程用作一等对象。此外，Common Lisp 不支持延续或需要对尾部调用进行适当的处理，但它确实支持几个在 Scheme 中找不到的不太通用的控制结构。虽然这两种语言是相似的，但Common Lisp包含更专业的构造，而Scheme包含更多通用的构建块，可以从中构建这样的构造（和其他构造）。

本章的其余部分描述了 Scheme 的语法和命名约定，以及本书中使用的排版约定。

第 1.1 节. 方案语法

方案程序由关键字，变量，结构化形式，常量数据（数字，字符，字符串，带引号的向量，带引号的列表，带引号的符号等），空格和注释组成。

关键字、变量和符号统称为标识符。标识符可以由字母、数字和某些特殊字符组成，包括 ？、！、.、+、-、*、/、<、=、>、：、$、%、^、&、_、~ 和 @，以及一组额外的 Unicode 字符。标识符不能以 at 符号 （ @ ）开头，通常不能以任何可以以数字开头的字符开头，即

数字、加号 （ + ）、减号 （ - ） 或小数点 （.）。例外情况是 +、-和 ...，它们是有效的标识符，任何以 -> 开头的标识符。例如，hi、Hello、n、x、x3、x+2 和 ? $&*!!!都是标识符。标识符由空格、注释、圆括号、方括号、字符串（双引号） （ “ ） 和哈希标记 （ # ） 分隔。分隔符或任何其他 Unicode 字符可以作为 \xsv; 形式的转义包含在标识符名称中的任何位置，其中 sv 是字符的标量值，采用十六进制表示法。

方案标识符的长度没有固有的限制;程序员可以根据需要使用任意数量的字符。但是，长标识符不能代替注释，频繁使用长标识符会使程序难以格式化，从而难以阅读。一个好的规则是在标识符的作用域较小时使用短标识符，在作用域较大时使用较长的标识符。

标识符可以写成大写和小写字母的任何组合，并且大小写是显著的，即两个标识符是不同的，即使它们只是在大小写上不同。例如，abcde、Abcde、AbCdE 和 ABCDE 都引用不同的标识符。这是对以前版本的经修订的报告所作的更改。

结构化形式和列表常量括在括号中，例如（a b c）或（*（- x 2） y）。将写入空列表 （）。匹配的括号集（ [ ]）可以代替括号，并且通常用于衬托某些标准句法形式的子表达式以提高可读性，如本书中的示例所示。向量的编写方式与列表类似，只是它们前面有 #（ 并以 ） 结尾，例如 #（这是符号的向量））。字节向量被写成无符号字节值序列（0 到 255 范围内的确切整数），并用 #vu8（ 和 ） 括起来，例如，#vu8（3 250 45 73）。

字符串括在双引号中，例如“我是字符串”。字符前面有 #\，例如 #\a。大小写在字符和字符串常量中很重要，

就像在标识符中一样。数字可以写成整数，例如-123，作为比率，例如1/2，在浮点或科学记数法中，例如，1.3或1e23，或作为矩形或极性表示法的复数，例如，1.3-2.7i或-1.2@73。大小写在数字的语法中并不重要。表示 true 和 false 的布尔值#t和#f写入。方案条件表达式实际上将#f视为 false，将所有其他对象视为 true，因此 3、0、（）、“false”和 nil 都算作 true。

每种类型的常量数据的语法细节在第6章的各个章节中给出，并从第455页开始在Scheme的正式语法中给出。

方案表达式可以跨越多行，并且不需要显式终止符。由于表达式之间的空格字符（空格和换行符）的数量不大，因此应缩进 Scheme 程序，以使代码尽可能可读的方式显示代码的结构。注释可以出现在 Scheme 程序的任何行上，介于分号 （ ; ） 和行尾之间。解释特定 Scheme 表达式的注释通常放置在表达式前面的行上，与表达式位于同一缩进级别。解释一个过程或一组过程的注释通常放在过程之前，不带缩进。多个注释字符通常用于衬托后一种类型的注释，例如 ;;;以下过程....

支持另外两种形式的注释：块注释和基准注释。块注释由 #| 和 |# 对分隔，并且可以嵌套。基准注释由 #; 前缀和它后面的基准（打印的数据值）组成。基准注释通常用于注释掉单个定义或表达式。例如，（三个 #;(不是四个）元素列表）就是它所说的。基准注释也可以嵌套，尽管 #;#;(a) (b) 具有注释掉 (a) 和 (b) 的不明显效果。

某些 Scheme 值（如过程和端口）没有标准的打印表示形式，因此在程序的打印语法中永远不会显示为常量。

本书在显示返回此类值的操作的输出时使用符号 #<description> 例如 #<procedure> 或 #<port>。

第 1.2 节. 方案命名约定

Scheme 的命名约定旨在提供高度的规律性。以下是这些命名约定的列表：

- 谓词名称以问号 （ ？ ） 结尾。谓词是返回真或假答案的过程，例如 eq？、zero？ 和 string=？。常见的数字比较器 =、<、>、<= 和 >= 是此命名约定的例外。
- 类型谓词（如 pair？）是根据类型的名称（在本例中为对）和问号创建的。
- 大多数字符、字符串和矢量过程的名称都以前缀 char-、string- 和 vector-开头，例如 string-append。（某些列表过程的名称以 list- 开头，但大多数过程不以列表开头。
- 将一种类型的对象转换为另一种类型的对象的过程的名称写为 type1->type2，例如 vector->list。
- 引起副作用的过程和句法形式的名称以感叹号 （ ！ ） 结尾。这些包括 set！ 和 vector-set！。从技术上讲，执行输入或输出的过程会导致副作用，但它们的名称是此规则的例外。

程序员应尽可能在自己的代码中使用这些相同的约定。

第 1.3 节. 排版和符号约定

标准程序或句法形式，其唯一目的是执行某些副作用，据说返回未指定。这意味着实现可以自由地返回任意数量的值，每个值都可以是任何 Scheme 对象，作为过程或语法形式的值。不要指望这些值在实现之间是相同的，在同一实现的版本之间是相同的，甚至在过程的两种用法或语法形式的使用中都是相同的。某些 Scheme 系统通常使用特殊对象来表示未指定的值。交互式 Scheme 系统通常会禁止打印此对象，因此不会打印返回未指定值的表达式的值。

虽然大多数标准过程返回单个值，但该语言支持通过第 5.8 节中描述的机制返回零、一、多个甚至可变数量的值的过程。如果某些标准表达式的一个子表达式的计算结果为多个值，则某些标准表达式的计算结果可能为多个值，例如，通过调用返回多个值的过程。当这种情况可能发生时，表达式被称为返回其子表达式的“值”，而不仅仅是“值”。同样，返回从调用过程参数而产生的值的标准过程称为返回过程参数返回的值。

本书使用“必须”和“应该”这两个词来描述程序要求，例如在调用 vector-ref 时提供小于矢量长度的索引的要求。如果使用“必须”一词，则意味着该要求由实现强制执行，即引发异常，通常具有条件类型和断言。如果使用单词“should”，则可能会或可能不会引发异常，如果没有，则程序的行为未定义。

短语“语法冲突”用于描述程序格式不正确的情况。在程序执行之前检测到语法冲突。当检测到语法冲突时，将引发类型 &语法的异常，并且不会执行程序。

本书中使用的排版惯例很简单。所有 Scheme 对象都以打字机字体打印，就像在键盘上键入一样。这包括语法关

键字、变量、常量对象、方案表达式和示例程序。斜体字体用于在句法形式的描述中衬托语法变量，在过程描述中设置参数。斜体也用于在第一次出现时衬托技术术语。通常，句法形式和过程的名称从不大写，即使在句子的开头也是如此。对于用斜体书写的语法变量也是如此。

在句法形式或过程的描述中，一个或多个原型模式显示句法形式或形式，或者用于应用该过程的正确数量或数量的参数。关键字或过程名称以打字机字体表示，括号也是如此。语法或参数的其余部分以斜体显示，使用的名称暗示句法形式或过程所期望的表达式或参数类型。省略号用于指定子表达式或参数的零个或多个匹配项。例如，（或 expr ...） 描述 或 语法形式，它具有零个或多个子表达式，并且 （成员 obj 列表） 描述成员过程，它需要两个参数，一个对象和一个列表。

如果句法形式的结构与其原型不匹配，则会发生语法冲突。同样，如果传递给标准过程的参数数与指定接收的参数数不匹配，则会引发条件类型 &断言的异常。如果标准过程收到的参数类型不是其名称所暗示的类型或不符合过程描述中给出的其他条件，则还会引发具有条件类型 &断言的异常。例如，vector-set！ 的原型是

（矢量集！vector n obj）

并且描述说n必须是严格小于向量长度的精确非负整数。因此，vector-set！ 必须接收三个参数，其中第一个参数必须是向量，第二个参数必须是小于向量长度的精确非负整数，其中第三个参数可以是任何 Scheme 值。否则，将引发条件类型 &断言的异常。

在大多数情况下，所需的参数类型是显而易见的，例如向量、obj 或二进制输入端口。在其他情况下，主要在数值例程的描述中使用缩写，例如int表示整数，exint表示精确整数，fx表示fixnum。这些缩写在包含受影响条目的部分的开头进行了说明。

ISBN 978-0-262-51298-5 / LOC QA76.73.S34D93
订购本书 / 关于这本书

http://www.scheme.com