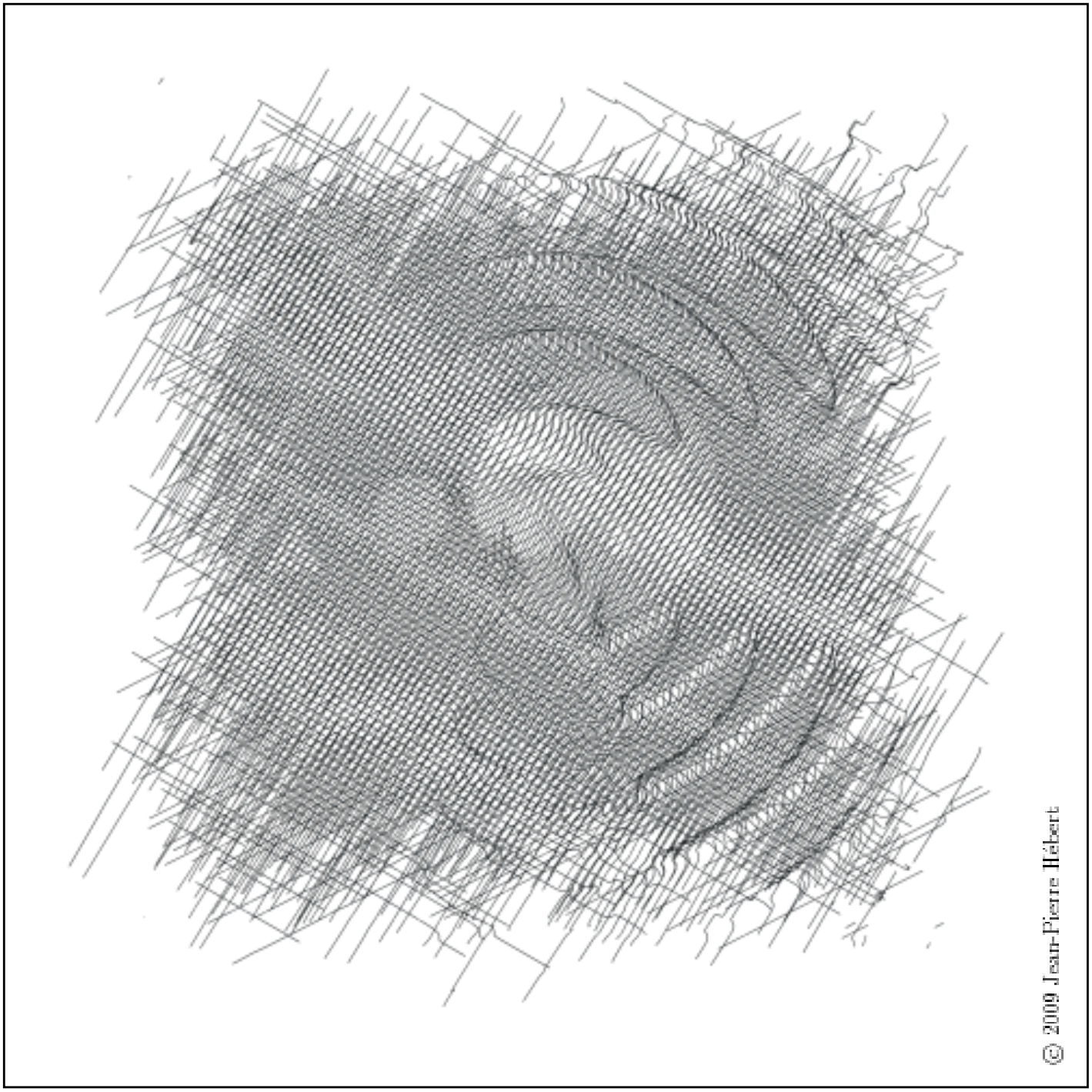

第 8 章。句法扩展

语法扩展或宏用于简化和规范程序中的重复模式，引入具有新评估规则的语法形式，以及执行有助于提高程序效率的转换。

句法扩展通常采用形式（关键字子形式...），其中关键字是命名句法扩展的标识符。每个子形式的语法因句法扩展而异。语法扩展也可以采取不正确的列表甚至单例标识符的形式。

新的语法扩展通过将关键字与转换过程或转换器相关联来定义。句法扩展是使用定义语法形式或使用 let 语法或 letrec 语法定义的。可以使用语法规则创建转换器，这允许执行基于模式的简单转换。它们也可能是接受一个参数并执行任意计算的普通过程。在这种情况下，语法大小写通常用于破坏输入，语法通常用于构造输出。标识符-语法形式和 make-变量-转换器过程允许创建与单例标识符匹配的转换器，并将赋值分配给这些标识符，前者仅限于语法规则等简单模式，后者允许执行任意计算。

语法扩展在评估开始时（在编译或解释之前）由语法扩展器扩展为核心形式。如果扩展程序遇到语法扩展，它将调用关联的转换器来扩展语法扩展，然后对转换器返回的形式重复扩展过程。如果扩展程序遇到核心语法形式，它将递归处理子窗体（如果有），并从展开的子窗体中重建窗体。有关标识符绑定的信息在扩展期间进行维护，以强制执行变量和关键字的词法范围。

本章中描述的句法扩展机制是“语法大小写”系统的一部分。该系统的可移植实现也支持库和顶级程序，可在 http://www.cs.indiana.edu/syntax-case/。关于系统背后的动机和实现的描述可以在文章“方案中的句法抽象”中找到[12]。Chez Scheme 用户指南 [9] 中介绍了尚未标准化的其他功能，包括模块、本地导入和元定义。

第 8.1 节. 关键字绑定

本节介绍在关键字和转换器之间建立绑定的窗体。关键字绑定可以使用定义语法在顶级程序或库主体中建立，也可以使用定义语法、let 语法或 letrec 语法在任何本地范围内建立。

语法：（定义语法关键字 expr）
库：（rnrs base），（rnrs）

expr 必须评估变压器。

下面的示例将 let* 定义为语法扩展，使用语法规则指定转换器（请参见第 8.2 节）。

```
(define-syntax let*
(syntax-rules ()
[(_ () b1 b2 ...) (让 () b1 b2 ...)]
[(_ ((i1 e1) (i2 e2) ...) b1 b2 ...)
(let ([i1 e1])
(let* ([i2 e2] ...) b1 b2 ...)))))
```

由一组内部定义建立的所有绑定（无论是关键字定义还是变量定义）在紧随的主体内的任何位置都可见，包括在定义本身中。例如，表达式

```
(让 ()
(定义偶数?
(lambda (x)
(or (= x 0) (奇数? (- x 1)))))
(定义语法奇数?
(语法规则 ()
[(_ x) (不是 (甚至? x))]))
(甚至? 10))
```

有效，应进行评估以#t。

扩展器从左到右处理库、lambda 或其他主体中的初始表单。如果遇到变量定义，它将记录定义的标识符是变量的事实，但会将右侧表达式的展开推迟到处理完所有定义之后。如果遇到关键字定义，它将展开并计算右侧表达式，并将关键字绑定到生成的转换器。如果遇到表达式，它将完全展开所有延迟的右侧表达式以及当前和剩余的正文表达式。

从左到右处理顺序的含义是，一个内部定义可以影响后续形式是否也是定义。例如，表达式

```
(let ()
(define-syntax bind-to-zero
(syntax-rules ()
[(_ id) (define id 0)]))
(bind-to-zero x)
x)
```

计算结果为 0，而不考虑在 let 表达式之外可能出现的绑定到零的任何绑定。

语法： (let-syntax ((关键字 expr) ...) form1 form2 ...)
语法： (letrec-syntax ((关键字 expr) ...) form1 form2 ...)
返回： 参见下面的
库： (rnrs base)， (rnrs)

每个expr必须评估变压器。对于 let-syntax 和 letrec-syntax，每个关键字都绑定在 form1 form2 ...对于 letrec 语法，绑定范围还包括每个 expr。

let-syntax 或 letrec-syntax 形式可以在允许表达式的任何位置扩展为一个或多个表达式，在这种情况下，生

成的表达式被视为包含在开始表达式中。它还可以扩展到零个或多个定义，只要允许定义，在这种情况下，定义将被视为代替let-syntax或letrec-syntax形式出现。

下面的示例突出显示了 let-syntax 和 letrec-syntax 的不同之处。

```
(let ([f (lambda (x) (+ x 1))])
(let-syntax ([f (syntax-rules ()
[(_ x) x])])
[g (syntax-rules ()
[(_ x) (f x)]))
(list (f 1) (g 1))))⇒ (1 2)

(let ([f (lambda (x) (+ x 1))])
(letrec-syntax ([f (syntax-rules ()
[(_ x) x])])
[g (syntax-rules ()
[(_ x) (f x)]))
(list (f 1) (g 1))) ⇒ (1 1)
```

这两个表达式是相同的，只是第一个表达式中的 let 语法形式是第二个表达式中的 letrec 语法形式。在第一个表达式中，g 中出现的 f 引用 let-bound 变量 f，而在第二个表达式中，它引用关键字 f，其绑定由 letrec 语法形式建立。

第 8.2 节. 语法规则转换器

本节中描述的语法规则形式允许以方便的方式指定简单的转换器。这些转换器可以使用第 8.1 节中描述的机制绑定到关键字。虽然它的表现力远不如第8.3节中描述的机制，但它足以定义许多常见的句法扩展。

语法：（语法规则（字面 ...） 子句 ...）
返回：转换器
库：（rnrs base），（rnrs）

每个文本必须是除下划线（_）或省略号（...）以外的标识符。每个子句必须采用以下形式。

（图案模板）

每个模式指定输入表单可能采用的一种可能的语法，相应的模板指定输出的显示方式。

模式由列表结构、向量结构、标识符和常量组成。模式中的每个标识符要么是文本、模式变量、下划线或省略号。标识符 _ 是下划线，标识符 ...是省略号。除 _ 或 ... 以外的任何标识符都是文本，如果它出现在文本列表中（文本 ...）;否则，它是一个模式变量。文本用作辅助关键字，例如 case 中的 else 和 cond 表达式。模式中的列表和向量结构指定输入所需的基本结构，下划线和模式变量指定任意子结构，文本和常量指定必须完全匹配的原子片段。省略号指定重复出现的子模式。

输入形式 F 匹配模式 P 当且仅当

- P 是下划线或模式变量，
- P 是文字标识符，F 是具有与谓词 free-identifier=? 确定的绑定相同的标识符（第 8.3 节），
- P 是形式（P1 ...Pn）F 是与 P1 到 Pn 匹配的 n 个元素的列表，

- P 是形式 （P_1 ... P_n . P_x）F 是 n 个或多个元素的列表或不正确的列表，其前 n 个元素通过 P_n 与 P_1 匹配，并且其第 n 个 cdr 与 P_x 匹配，

- P 是形式 （P_1 ... P_k Pe 省略号 P_{m+1} ... P_n），其中省略号是标识符...，F是n个元素的正确列表，其前k个元素与P_1到P_k匹配，其下一个m - k元素分别与P_e匹配，其剩余的n - m元素与P_{m} +匹配1 到 P_n，

- P 是形式 （P_1 ... P_k Pe 省略号 P_{m+1} ... P_n . P_x），其中省略号是标识符...，F 是 n 个元素的列表或不正确的列表，其前 k 个元素与 P_1 到 P_k 匹配，其下一个 m - k 个元素都与 P_e 匹配，其下一个 n - m 元素与 P_{m}+ 匹配1 到 P_n，其第 n 个也是最后一个 cdr 与 P_x 匹配，

- P 是 #（P_1 ... P_n）F 是 n 个元素的向量，它们通过 P_n 与 P_1 匹配，

- P 是 #（P_1 ... P_k Pe 省略号 P_{m+1} ... P_n），其中省略号是标识符...，F是n个元素的向量，其前k个元素与P_1到P_k匹配，其下一个m-k元素每个匹配P_e，其剩余的n - m元素与 P_{m} + 1匹配通过 P_n，或

- P 是一个模式基准面（任何非列表、非向量、非符号对象），F 等于 P 在相等的意义上？

语法规则模式的最外层结构实际上必须采用上面的列表结构形式之一，尽管该模式的子模式可以是上述任何形式。此外，最外层模式的第一个元素被忽略，因为它总

是被认为是命名句法形式的关键字。（这些语句不适用于语法大小写;请参见第 8.3 节。

如果传递给语法规则转换器的输入表单与给定子句的模式匹配，则接受该子句，并按照关联模板的指定转换表单。发生此转换时，模式中出现的模式变量将绑定到相应的输入子窗体。出现在子模式中后跟一个或多个省略号的模式变量可以绑定到一个或多个输入子窗体的序列。

模板是一个模式变量，一个不是模式变量的标识符，一个模式基准，一个子模板列表（$S1 \ldots Sn$），子模板列表不正确（$S1\ S2 \ldots Sn\ .T$），或子模板 #（$S1 \ldots Sn$）。每个子模板 Si 都是一个模板，后跟零个或多个省略号。不正确的子模板列表的最后一个元素 T 是模板。

模板中出现的模式变量在输出中被绑定到的输入子窗体替换。不是模式变量的模式数据和标识符将直接插入到输出中。模板中的列表和矢量结构仍然是输出中的列表和矢量结构。子模板后跟省略号将扩展为子模板的零个或多个匹配项。子模板必须至少包含一个来自子模式的模式变量，后跟一个省略号。（否则，扩展器无法确定子窗体应在输出中重复多少次。在子模式后跟一个或多个省略号中出现的模式变量可能仅出现在后跟（至少）相同数量的省略号的子模板中。这些模式变量在输出中被它们绑定到的输入子窗体替换，并按指定的方式分布。如果模板中模式变量后跟的省略号多于关联模式中的省略号，则会根据需要复制输入表单。

表单的模板 （... 模板）与模板相同，只是模板中的省略号没有特殊含义。也就是说，模板中包含的任何省略号都被视为普通标识符。特别是，模板 （... ...） 生成一

个省略号，...。这允许语法扩展扩展为包含省略号的形式，包括语法规则或语法大小写模式和模板。

以下定义演示了语法规则的使用。

```
（define-syntax or
（syntax-rules ）
[（_） #f]
[（_ e） e]
[（_ e1 e2 e3 ...）
（let （[t e1]） （if t t （or e2 e3 ...））））））
```

输入模式指定输入必须由关键字和零个或多个子表达式组成。下划线（ _ ）是匹配任何输入的特殊模式符号，通常用于关键字位置，以提醒程序员和任何阅读定义的人关键字位置永远不会包含预期的关键字，并且不需要匹配。（实际上，如上所述，语法规则会忽略关键字位置中显示的内容。如果存在多个子表达式（第三子句），则扩展的代码既测试第一个子表达式的值，又返回值（如果不是 false）。为避免对表达式进行两次求值，转换器会为临时变量 t 引入绑定。

扩展算法通过根据需要重命名本地标识符来自动维护词法范围。因此，变压器引入的t的绑定仅在变压器引入的代码中可见，而在输入的子形式内不可见。同样，对标识符的引用允许并且 if 不受输入上下文中存在的任何绑定的影响。

```
（让 （[如果#f]）
（让 （[t 'ok]）
（或如果 t））） ⇒ 好
```

此表达式在扩展过程中将转换为与以下表达式等效的表达式。

```
 ( (lambda (if1)
 ( (lambda (t1)
 ( (lambda (t2)
 (if t2 t2 t1) )
if1) )
'ok) )
#f) ⇒ 好
```

在此示例扩展中，if1、t1 和 t2 表示原始表达式中的 if 和 t 以及扩展项中的 t 已重命名的标识符。

下面对 cond 的简化版本的定义（简化后，因为它每个子句至少需要一个输出表达式，并且不支持辅助关键字 =>）演示了如何通过包含在文本列表中来识别转换器的输入中的辅助关键字（如 else）。

```
(define-syntax cond
(syntax-rules (else)
[(_ (else e1 e2 ...)) (开始 e1 e2 ...)]
[(_ (e0 e1 e2 ...)) (如果 e0 (开始 e1 e2 ...))
[(_ (e0 e1 e2 ...) c1 c2 ...)
(如果 e0 (开始 e1 e2 ...) (cond c1 c2 ...)))))
```

语法： _
语法： ...
库： (rnrs base)， (rnrs syntax-case)， (rnrs)

这些标识符是语法规则、标识符语法和语法大小写的辅助关键字。第二个 (...) 也是语法和 quasisyntax 的辅助关键字。引用这些标识符是语法冲突，除非在上下文中将它们识别为辅助关键字。

语法： (标识符语法 tmp1)
语法： (标识符语法 (id1 tmp11) ((set! id2 e2) tmp12))

返回： 转换器
库： (rnrs base)， (rnrs)

当关键字绑定到由第一种形式的标识符语法生成的转换器时，绑定范围内对关键字的引用将替换为 tmpl。

```
(let ()
(define-syntax a (identifier-syntax car))
(list (a '(1 2 3)) a)) ⇒ (1 #<程序>)
```

对于标识符语法的第一种形式，将关联的关键字与 set! 明显分配是语法冲突。第二种更通用的标识符语法形式允许转换器指定使用 set! 时发生的情况。

```
(let ([ls (list 0)])
(define-syntax a
(identifier-syntax
[id (car ls)]
[(set! id e) (set-car! ls e)]))
(let ([before a])
(set! a 1)
(list before a ls))) ⇒ (0 1 (1))
```

第 307 页显示了标识符语法在 make 变量转换器方面的定义。

第 8.3 节. 语法-大小写转换器

本节介绍一种更具表现力的机制，用于创建基于语法大小写的转换器，这是语法规则的通用版本。此机制允许指定任意复杂的转换，包括以受控方式“弯曲”词法范围的转换，从而允许定义更广泛的语法扩展类别。任何可以使用语法规则定义的转换器都可以很容易地重写为使用语

法大小写;事实上，语法规则本身可以定义为语法大小写的句法扩展，如下面的语法描述所示。

有了这个机制，变压器是一个参数的过程。参数是表示要处理的表单的语法对象。返回值是表示输出表单的语法对象。语法对象可以是以下任一语法对象。

- 非对、非向量、非符号值，
- 一对语法对象，
- 语法对象的向量，或
- 包装的对象。

包装语法对象上的换行除了包含表单结构外，还包含有关表单的上下文信息。扩展器使用此上下文信息来维护词法范围。包装还可以包含由实现用于关联源码和目标代码的信息，例如，通过扩展和编译过程跟踪文件、行和字符信息。

所有标识符都必须存在上下文信息，这就是为什么上面的语法对象的定义不允许使用符号的原因，除非它们被包装。表示标识符的语法对象本身称为标识符;因此，术语标识符可以指语法实体（符号、变量或关键字），也可以指语法实体作为语法对象的具体表示形式。

转换器通常使用语法大小写来破坏其输入，并使用语法重新生成其输出。仅这两种形式就足以定义许多语法扩展，包括任何可以使用语法规则定义的扩展。下面将介绍它们以及一组提供附加功能的附加窗体和过程。

语法： （语法大小写 expr （字面 ...） 子句 ...）
返回：请参阅下面的
库：（rnrs syntax-case）， （rnrs）

每个文本都必须是一个标识符。每个子句必须采用以下两种形式之一。

（模式输出表达式）
（图案挡泥板输出表达式）

语法大小写模式可以采用第 8.2 节中描述的任何形式。

syntax-case 首先计算 expr，然后尝试将结果值与第一个子句中的模式进行匹配。此值可以是任何方案对象。如果该值与模式匹配，并且不存在挡泥板，则计算输出表达式，并将其值作为语法大小写表达式的值返回。如果该值与模式不匹配，则将该值与下一个子句进行比较，依此类推。如果值与任何模式都不匹配，则会出现语法冲突。

如果存在可选的挡泥板，则它可作为接受子句的附加约束。如果语法大小写 expr 的值与给定子句的模式匹配，则计算相应的挡泥板。如果挡泥板的计算结果为真值，则接受该子句;否则，该子句将被拒绝，就好像输入与模式不匹配一样。从逻辑上讲，挡泥板是匹配过程的一部分，即，它们指定了表达式基本结构之外的其他匹配约束。

子句模式中包含的模式变量绑定到子句的挡泥板（如果存在）和输出表达式中输入值的相应部分。模式变量与程序变量和关键字占用相同的命名空间;由语法大小写创建的模式变量绑定可以隐藏（并被隐藏）程序变量和关键字绑定以及其他模式变量绑定。但是，模式变量只能在语法表达式中引用。

请参阅语法说明后面的示例。

语法： （语法模板）
语法： #’模板
返回： 参见下面的
库： （rnrs syntax-case）， （rnrs）

#’模板等效于（语法模板）。在宏扩展之前，读取程序时，缩写形式将转换为较长的形式。

语法 表达式类似于 引号 表达式，只是 模板 中出现的模式变量的值入到 模板 中，并且与输入和模板关联的上下文信息保留在输出中以支持词法范围。语法模板与语法规则模板相同，并且处理方式类似。

模板中的列表和矢量结构成为真正的列表或矢量（适用于直接应用列表或矢量操作，如map或矢量-ref），以至于必须复制列表或矢量结构以插入模式变量的值，并且从不包装空列表。例如，如果 x、a、b 和 c 是模式变量，则 #’（x ...）、#’（a b c）、#’（） 都是列表。

或以下的定义等同于第 8.2 节中给出的定义，只是它使用语法大小写和语法来代替语法规则。

```
(define-syntax or
(lambda (x)
(syntax-case x ()
[(_) #'#f]
[(_ e) #'e]
[(_ e1 e2 e3 ...)
#'(let ([t e1]) (if t t (or e2 e3 ...)))))))
```

在此版本中，生成转换器的 lambda 表达式是显式的，每个子句的输出部分中的语法形式也是如此。任何语法规则形式都可以通过使 lambda 表达式和语法表达式显式来使

用语法大小写来表示。这一观察结果导致以下语法规则在语法大小写方面的定义。

```
(define-syntax syntax-rules
(lambda (x)
(syntax-case x ()
[(_ (i ...) ((关键字 . 模式) 模板) ...)
#'(lambda (x)
(syntax-case x (i ...)
[(_ . pattern) #'template] ...)))))))
```

使用下划线代替每个关键字，因为每个语法规则模式的第一个位置始终被忽略。

由于 lambda 和语法表达式以语法规则形式隐式，因此使用语法规则表示的定义通常比使用语法大小写表示的等效定义短。当其中任何一个都足够时，选择使用哪个转换器是一个品味问题，但是许多可以使用语法大小写轻松或根本无法编写语法规则的转换器（参见第8.4节）。

过程：（标识符？obj）
返回：如果 obj 是标识符，则#t，#f否则
库：（rnrs 语法大小写）、（rnrs）

标识符？通常用于挡泥板中，以验证输入表单的某些子形式是否为标识符，如下面的未命名 let 定义所示。

```
(define-syntax let
(lambda (x)
(define ids?
(lambda (ls)
(or (null? ls)
(and (identifier? (汽车 ls))
(IDS? (cdr ls)))))))
(语法大小写 x ()
```

```
[ (_ ((i e) ...) b1 b2 ...)
 (ids? #'(i ...))
#'((lambda (i ...) b1 b2 ...) e ...)))))
```

句法扩展通常采用形式（关键字子形式...），但语法大小写系统也允许它们采用单例标识符的形式。例如，下面表达式中的关键字 pcar 既可以用作标识符（在这种情况下，它可以扩展为对 car 的调用）或结构化形式（在这种情况下，它可以扩展为对 set-car! 的调用！

```
(let ([p (cons 0 #f)])
(define-syntax pcar
(lambda (x)
(syntax-case x ()
[_ (identifier? x) #'(car p)]
[(_ e) #'(set-car! p e)]))
(let ([a pcar])
(pcar 1)
(list a pcar))))⇒ (0 1)
```

挡泥板（标识符？x）用于识别单例标识符大小写。

过程： （自由标识符=？标识符 $_1$ 标识符 $_2$）
过程： （绑定标识符=？identifier1 identifier2）
返回：请参阅下面的
库：（rnrs syntax-case）、（rnrs）

符号名称本身不能区分标识符，除非标识符仅用作符号数据。谓词 free-identifier=? 和 bound-identifier=? 用于根据标识符在给定上下文中作为自由引用或绑定标识符的预期用途来比较标识符。

自由标识符=？用于确定两个标识符在转换器的输出中显示为自由标识符时是否等效。因为标识符引用在词法上是

作用域的，所以这意味着 (free-identifier=? id1 id2) 为真，当且仅当标识符 id1 和 id2 引用相同的绑定时。（对于此比较，如果两个名称相似的标识符都未绑定，则假定它们具有相同的绑定。出现在语法大小写模式中的文本标识符（辅助关键字）（例如 case 和 cond 中的 else）与 free-identifier=? 匹配。

同样，绑定标识符 =? 用于确定两个标识符在转换器的输出中显示为绑定标识符时是否等效。换句话说，如果 bound-identifier=? 对两个标识符返回 true，则一个标识符的绑定将在其范围内捕获对另一个标识符的引用。通常，只有当两个标识符都存在于原始程序中，或者两者都由同一个转换器应用程序引入时，才绑定标识符=?（可能隐式地---参见基准>语法）。绑定标识符 =? 可用于检测绑定构造中的重复标识符，或用于需要检测绑定标识符实例的绑定构造的其他预处理。

下面的定义等效于早期对带有语法规则的简化版本的 cond 的定义，除了通过在挡泥板中显式调用 free-identifier? 而不是通过包含在文本列表中来识别其他定义。

```
(define-syntax cond
(lambda (x)
(syntax-case x ()
[(_ (e0 e1 e2 ...))
(和（标识符? #'e0） （自由标识符=? #'e0 #'else））
#'（开始 e1 e2 ...）]
[(_ (e0 e1 e2 ...)) #'（如果 e0 （开始 e1 e2
...））]
[(_ (e0 e1 e2 ...) c1 c2 ...)
#'（如果 e0 （开始 e1 e2 ...）（cond c1 c2 ...））））
```

对于 cond 的任一定义，如果 else 存在封闭的词法绑定，则 else 不会被识别为辅助关键字。例如

```
(let ([else #f])
(cond [else (write “oops”)]))
```

不写“oops”，因为 else 在词法上是绑定的，因此与 cond 定义中出现的 else 不同。

以下未命名 let 的定义使用绑定标识符 =? 来检测重复的标识符。

```
(define-syntax let
(lambda (x)
(define ids?
(lambda (ls)
(or (null? ls)
(and (identifier? (汽车 ls)) (IDS? (cdr
ls)))))
(定义唯一 ID?
(lambda (ls)
(or (null? ls)
(and (not (memp
(lambda (x) (bound-identifier=? x (car ls)))
(cdr ls)))
(unique-ids? (cdr ls)))))
(语法大小写 x ()
[(_ ((i e) ...) b1 b2 ...)
(和 (ids? #'(i ...)) (唯一 ID? #'(i ...)))
#'((lambda (i ...) b1 b2 ...) e ...))))
```

有了上面的让的定义，表达式

```
(让我们 ([a 3] [a 4]) (+ a))
```

是语法违规，而

```
 (let ([a 0])
 (let-syntax ([dolet (lambda (x)
 (syntax-case x ()
[(_ b)
#'(let ([a 3] [b 4]) (+ a b)))
 (dolet a)))
```

计算结果为 7，因为 dolet 引入的标识符 a 和从输入形式中提取的标识符 a 不是 bound-identifier=? 。但是，由于两次出现的 a 如果保留为自由引用，则将引用相同的绑定，因此 free-identifier=? 不会区分它们。

两个标识符是 free-identifier=? 可能不是绑定标识符=? 。转换器引入的标识符可能引用与转换器未引入的标识符相同的封闭绑定，但对一个引入的绑定不会捕获对另一个的引用。另一方面，绑定标识符=? 的标识符是自由标识符=? ，只要标识符在比较它们的上下文中具有有效的绑定。

语法： (with-syntax ((pattern expr) ...) body1 body2 ...)
返回：最终正文表达式
库的值：(rnrs 语法大小写)、 (rnrs)

有时，将变压器的输出构建成单独的部分，然后将这些部分放在一起是有用的。with-syntax 通过允许创建本地模式绑定来促进这一点。

模式在形式上与语法大小写模式相同。每个 expr 的值根据相应的模式进行计算和反构造，并且模式中的模式变量与语法大小写一样绑定到 body1 body2 ... 中值的适当部分，该部分像 lambda 主体一样进行处理和计算。

with-syntax 可以定义为语法大小写的句法扩展。

```
(define-syntax with-syntax
(lambda (x)
(syntax-case x ()
[(_ ((p e) ...) b1 b2 ...)
#'(语法大小写 (列表 e ...) ()
[(p ...) (让 () b1 b2 ...)])))))
```

以下完全 cond 定义演示了如何使用 with 语法来支持在内部使用递归来构造其输出的转换器。

```
(define-syntax cond
(lambda (x)
(syntax-case x ()
[(_ c1 c2 ...)
(设 f ([c1 #'c1] [cmore #'(c2 ...)])
(if (null? cmore)
(syntax-case c1 (else =>)
[(else e1 e2 ...) #'(begin e1 e2 ...)]
[(e0) #'(let ([t e0]) (if t t))]
[(e0 => e1) #'(let ([t e0]) (if t (e1 t)))]
[(e0 e1 e2 ...) #'(if e0 (begin e1 e2 ...))])
(with-syntax ([rest (f (car cmore) (cdr
cmore))])
(syntax-case c1 (=>)
[(e0) #'(let ([t e0]) (if t t rest))]
[(e0 => e1) #'(let ([t e0]) (if t (e1 t)
rest))]
[(e0 e1 e2 ...)
#'(如果 e0 (开始 e1 e2 ...) rest)))))))))
```

语法: (准语法模板 ...)
语法: #'模板
语法: (unsyntax template ...)
语法: #, 模板
语法: (unsyntax-splicing template ...)
语法: #, @template

返回： 见下文
库： （rnrs syntax-case）， （rnrs）

#'template 等效于 （quasisyntax template），而 #，template 等价于 （unsyntax template）和 #，@template to （unsyntax-splicing template）。在宏扩展之前，在读取程序时，缩写形式将转换为较长的形式。

quasisyntax类似于语法，但它允许以类似于准引号的方式评估部分引用文本（第6.1节）。

在准语法模板中，将评估非语法和非语法拼接形式的子窗体，并且其他所有内容都被视为普通模板材料，就像语法一样。每个 unsyntax 子窗体的值将插入到输出中以代替 unsyntax 窗体，而每个 unsyntax 拼接子窗体的值将拼接到周围的列表或向量结构中。unsyntax 和 unsyntax-splicing 仅在 quasisyntax 表达式中有效。

quasisyntax 表达式可以是嵌套的，每个 quasisyntax 都会引入一个新的语法引用级别，每个 unsyntax 或 unsyntax 拼接会占用一个级别的引用。嵌套在 n 个准语法表达式中的表达式必须位于 n 个非语法表达式或非语法拼接表达式中才能进行求值。

在许多情况下，quasisyntax 可以代替 with 语法。例如，以下 case 定义使用 quasisyntax 来构造其输出，使用内部递归，其方式类似于上面 with 语法描述下给出的 cond 定义。

```
(define-syntax case
(lambda (x)
(syntax-case x ()
[(_ e c1 c2 ...)
#'(let ([t e])
```

```
#, (let f ([c1 #'c1] [cmore #'(c2 ...)])
 (if (null? cmore)
 (syntax-case c1 (else)
[(else e1 e2 ...) #'(begin e1 e2 ...)]
[((k ...) e1 e2 ...)
#'(if (memv t '(k ...)) (开始 e1 e2 ...))])
 (语法大小写 c1 ()
[((k ...) e1 e2 ...)
#'(if (memv t '(k ...))
 (开始 e1 e2 ...)
#, (f (car cmore) (cdr cmore)))))))))))
```

包含零个或多个子窗体的非语法和非语法拼接形式仅在拼接（列表或矢量）上下文中有效。（非语法模板...）等效于（unsyntax template） ...，和 （unsyntax-splicing template ...） 等价于 （unsyntax-splicing template） ...。这些形式主要用作准语法扩展器输出中的中间形式。它们支持某些有用的嵌套准引号（quasisyntax）习语[3]，例如#，@#，@，当在双重嵌套和双重计算的quasisyntax表达式中使用时，它具有双重间接拼接的效果，如第6.1节中所示的嵌套准引号示例。

unsyntax 和 unsyntax-splicing 是 quasisyntax 的辅助关键字。引用这些标识符是语法冲突，除非在上下文中将它们识别为辅助关键字。

过程：（make-variable-transformer procedure）
返回：一个变量转换器
库：（rnrs syntax-case）， （rnrs）

如本节的引言中所述，转换器可能只是接受一个参数（表示输入表单的语法对象）并返回表示输出表单的新语法对象的过程。传递给转换器的形式通常表示一个带括号的形式，其第一个子窗体是绑定到转换器的关键字

或只是关键字本身。make-variable-transformer 可用于将过程转换为一种特殊类型的转换器，扩展器也向其传递 set! 表单，其中关键字出现在 set! 关键字之后，就好像它是要分配的变量一样。这允许程序员控制当关键字出现在此类上下文中时发生的情况。参数，程序，应该接受一个参数。

```
(let ([ls (list 0)])
(define-syntax a
(make-variable-transformer
(lambda (x)
(syntax-case x ()
[id (identifier? #'id) #'(car ls)]
[(set! _ e) #'(set-car! ls e)]
[(_ e ...) #'(car ls) e ...)))))
(let ([before a])
(set! a 1)
(list before a ls))) ⇒ (0 1 (1))
```

这种语法抽象可以使用标识符语法更简洁地定义，如第 8.2 节所示，但 make 变量转换器可用于创建执行任意计算的转换器，而标识符语法仅限于简单的术语重写，如语法规则。标识符语法可以根据 make 变量转换器进行定义，如下所示。

```
(define-syntax identifier-syntax
(lambda (x)
(syntax-case x (set!)
[(_ e)
#'(lambda (x)
(syntax-case x ()
[id (identifier? #'id) #'e]
[(_ x (... ...)) #'(e x (... ...)))))]
[(_ (id exp1) ((set! var val) exp2))
(and (identifier? #'id) (identifier? #'var))
#'(make-variable-transformer
```

```
(lambda (x)
(syntax-case x (set!)
[(set! var val) #'exp2]
[(id x (... ...)) #'(exp1 x (... ...))]
[id (identifier? #'id) #'exp1])))))
```

过程： (syntax->datum obj)
返回： obj 剥离了语法信息
库： (rnrs syntax-case)， (rnrs)

过程语法>基准从语法对象中剥离所有语法信息，并返回相应的方案“基准”。以这种方式剥离的标识符被转换为其符号名称，然后可以将其与eq?进行比较。因此，谓词符号标识符 =? 可以定义如下。

```
(定义符号标识符=?
(lambda (x y)
(eq? (语法>基准 x)
(syntax->datum y))))
```

两个 free-identifier=? 的标识符不必是符号标识符=?：引用同一绑定的两个标识符通常具有相同的名称，但库的导入表单（第 345 页）的重命名和前缀子窗体可能会导致两个标识符具有不同的名称但绑定相同。

过程： (datum->syntax template-identifier obj)
返回： a syntax object
libraries: (rnrs syntax-case)， (rnrs)

datum->syntax 从 obj 构造一个语法对象，该对象包含与模板标识符相同的上下文信息，其效果是语法对象的行为就像在引入模板标识符时引入代码一样。模板标识符通常是从表单中提取的输入表单的关键字，对象通常是命名要构造的标识符的符号。

datum->syntax 允许转换器通过创建隐式标识符来“弯曲”词法范围规则，这些标识符的行为就像它们存在于输入表单中一样，从而允许定义语法扩展，这些语法扩展引入了对输入表单中未显式出现的标识符的可见绑定或引用。例如，我们可以定义一个循环表达式，该表达式将变量中断绑定到循环体中的转义过程。

```
(define-syntax loop
(lambda (x)
(syntax-case x ()
[(k e ...)
(with-syntax ([break (datum->syntax #'k 'break)])
#'(call/cc
(lambda (break)
(let f () e ... (f)))))))

(let ([n 3] [ls '()])
(loop
(if (= n 0) (break ls))
(set! ls (cons 'a ls))
(set! n (- n 1)))))  ⇒ (a a)
```

我们是否要将循环定义为

```
(define-syntax loop
(lambda (x)
(syntax-case x ()
[(_ e ...)
#'(call/cc
(lambda (break)
(let f () e ... (f)))))
```

变量中断在 e ...

对于 obj 来说，表示任意 Scheme 形式也很有用，如下面的 include 定义所示。

```
(define-syntax include
(lambda (x)
(define read-file
(lambda (fn k)
(let ([p (open-input-file fn)])
(let f ([x (read p)])
(if (eof-object? x)
(begin (close-port p) '())
(cons (datum->syntax k x) (f (read p)))))))
(syntax-case x ()
[(k filename)
(let ([fn (syntax->datum #'filename)])
(with-syntax ([(expr ...) (read-file fn #'k)])
#'(开始 expr ...)))))))
```

（包括“文件名”）扩展为一个开始表达式，其中包含在“文件名”命名的文件中找到的表单。例如，如果文件 f-def.ss 包含表达式（定义 f （lambda （） x）），则表达式

```
（让（[x“确定”]）
（包括“f-def.ss”）
（f））
```

评估为“正常”。

包含的定义使用基准>语法将从文件中读取的对象转换为正确词法上下文中的语法对象，以便这些表达式中的标识符引用和定义限定在包含窗体出现的位置。

过程：（生成临时列表）
返回：不同生成的标识符
库的列表：（rnrs 语法大小写）、（rnrs）

转换器可以通过命名每个标识符，将固定数量的标识符引入其输出中。但是，在某些情况下，要引入的标识符

的数量取决于输入表达式的某些特征。例如，letrec 的简单定义需要与输入表达式中存在绑定对一样多的临时标识符。生成临时标识符过程用于构造临时标识符列表。

列表可以是任何列表;其内容并不重要。生成的临时数是列表中的元素数。每个临时标识符都保证与所有其他标识符不同。

下面显示了使用生成临时的 letrec 的定义。

```
(define-syntax letrec
(lambda (x)
(syntax-case x ()
[(_ ((e) ...) b1 b2 ...)
(with-syntax ([(t ...) (生成临时工 #'(i ...))])
#'(让 ([i #f] ...)
(让 ([t e] ...)
(设置! i t)
...
(let () b1 b2 ...))))))))
```

可以重写以这种方式使用生成临时的任何转换器以避免使用它，尽管会失去清晰度。诀窍是使用递归定义的中间形式，该形式为每个扩展步骤生成一个临时，并在生成足够的临时后完成扩展。下面是 let 值的定义（第 99 页），它使用此技术来支持多组绑定。

```
(定义语法 let 值
(语法规则 ()
[(_ () f1 f2 ...) (让 () f1 f2 ...)]
[(_ ((fmls1 expr1) (fmls2 expr2) ...) f1 f2
...)
(lvhelp fmls1 () expr1 ((fmls2 expr2) ...) (f1
f2 ...)))))

(define-syntax lvhelp
```

```
 (syntax-rules )
[ (_ (x1 . fmls) (x ...) (t ...) e m b)
 (lvhelp fmls (x ...x1) (t ...tmp) e m b) ]
[ (_ () (x ...) (t ...) e m b)
 (call-with-values
 (lambda () e)
 (lambda (t ...)
 (let-values m (let ([x t] ...) .b) ) ) )
[ (_ xr (x ...) (t ...) e m b)
 (call-with-values
 (lambda () e)
 (lambda (t ... . tmpr)
 (let-values m (let ([x t] ...[xr tmpr]) .b))))]))
```

lvhelp 的实现很复杂，因为在创建任何绑定之前需要评估所有右侧表达式，并且需要支持不正确的形式列表。

第 8.4 节. 例子

本节介绍一系列使用语法规则或语法大小写定义的说明性语法扩展，从一些简单但有用的语法扩展开始，以相当复杂的机制结束，用于定义具有自动生成的构造函数、谓词、字段访问器和字段设置器的结构。

本节中最简单的示例是以下 rec 的定义。rec 是一个语法扩展，它允许以最小的工作量创建内部递归匿名（未外部命名）过程。

```
 (define-syntax rec
 (syntax-rules ()
[ (_ x e) (letrec ([x e]) x) ) ) )

 (映射 (rec sum
 (lambda (x)
 (if (= x 0)
```

```
0
 (+ x (sum (- x 1))))
'(0 1 2 3 4 5)) ⇒ (0 1 3 6 10 15)
```

使用rec，我们可以定义完整的let（未命名和命名），如下所示。

```
(define-syntax let
(syntax-rules ()
[(_ ((x e) ...) b1 b2 ...)
 ((lambda (x ...) b1 b2 ...) e ...)]
[(_ f ((x e) ...) b1 b2 ...)
 ((rec f (lambda (x ...) b1 b2 ...)) e ...))))
```

我们也可以直接用 letrec 来定义 let，尽管定义有点不太清楚。

```
(define-syntax let
(syntax-rules ()
[(_ ((x e) ...) b1 b2 ...)
 ((lambda (x ...) b1 b2 ...) e ...)]
[(_ f ((x e) ...) b1 b2 ...)
 ((letrec ([f (lambda (x ...) b1 b2 ...)]) f) e
...))))
```

这些定义依赖于这样一个事实，即第一个模式不能匹配命名 let，因为命名 let 的第一个子窗体必须是标识符，而不是绑定列表。以下定义使用挡泥板使此检查更加可靠。

```
(define-syntax let
(lambda (x)
(syntax-case x ()
[(_ ((x e) ...) b1 b2 ...)
#'((lambda (x ...) b1 b2 ...) e ...)]
[(_ f ((x e) ...) b1 b2 ...)
```

```
 (identifier? #'f)
#'(rec f (lambda (x ...) b1 b2 ...)) e ...)))
```

使用挡泥板，我们甚至可以将子句放在相反的顺序中。

```
 (define-syntax let
 (lambda (x)
 (syntax-case x ()
[(_ f ((x e) ...) b1 b2 ...)
 (标识符? #'f)
#'(rec f (lambda (x ...) b1 b2 ...)) e ...)]
[(_ ((x e) ...) b1 b2 ...)
#'((lambda (x ...) b1 b2 ...) e ...)))
```

为了完全健壮，在第 8.3 节中未命名 let 的定义中采用的 ids? 和 unique-id? 检查也应该在这里使用。

let的两种变体都很容易用简单的单行模式来描述，但确实需要更多的工作。do 的精确语法不能直接用单个模式表示，因为 do 表达式的绑定列表中的某些绑定可能采用形式 (var val)，而其他绑定则采用 (var val update) 的形式。do 的以下定义在内部使用语法大小写来独立于整体表单分析绑定。

```
 (define-syntax do
 (lambda (x)
 (syntax-case x ()
[(_ (binding ...) (测试分辨率...) expr ...)
 (with-syntax ) [ ((var val update) ...)
 (map (lambda (b)
 (syntax-case b ()
[(var val) #'(var val var)]
[(var val update) #'(var val update)]))
#'(binding ...)])
#'(let doloop ([var val] ...)
 (如果测试
```

```
（如果#f #f）...）
（开始执行 ...（doloop update ...）））））））））
```

在未提供结果表达式的情况下，在结果表达式之前插入外观奇怪的表达式（如果#f #f），因为 begin 至少需要一个子表达式。的值（如果#f #f）是未指定的，这就是我们想要的，因为如果没有提供结果表达式，则 do 的值是未指定的。以牺牲更多的代码为代价，我们可以使用语法大小写来确定是否提供了任何结果表达式，并在适当的情况下生成一个带有一个或两个臂的循环。由此产生的扩展将更清晰，但在语义上是等效的。

如第 8.2 节所述，省略号在表单的模板中失去了特殊含义（...模板）。这一事实允许语法扩展扩展到包含省略号的语法定义。下面的 be-like-begin 定义说明了这种用法。

```
（define-syntax be-like-begin
（syntax-rules （）
[（_ name）
（define-syntax name
（syntax-rules （）
[（_ e0 e1 （... ...））
（开始 e0 e1 （... ...））））））
```

以这种方式定义 be-like-begin（be-like-begin 序列）具有与以下序列定义相同的效果。

```
（定义语法序列
（语法规则 （）
[（_ e0 e1 ...）（开始 e0 e1 ...））））
```

也就是说，序列形式变得等效于开始形式，例如：

```
（序列（显示“说什么？”）（换行符））
```

打印“说什么？”，后跟换行符。

下面的示例演示如何通过根据内置 if 定义本地 if 来限制给定表达式中的 if 需要“else”（替代）子表达式。在下面的 let 语法绑定的正文中，双臂 if 一如既往地工作：

```
(let-syntax ([if (lambda (x)
(syntax-case x ()
[(_ e1 e2 e3)
#'(if e1 e2 e3)]))
(if (< 1 5) 2 3)) ⇒ 阿拉伯数字
```

但是单臂，如果导致语法错误。

```
(let-syntax ([if (lambda (x)
(syntax-case x ()
[(_ e1 e2 e3)
#'(if e1 e2 e3)]))
(if (< 1 5) 2)) ⇒ 语法冲突
```

尽管 if 的这个局部定义看起来很简单，但有一些微妙的方式可以尝试编写它，但可能会出错。如果使用 letrec 语法代替 let 语法，则插入到输出中的标识符将引用本地 if 而不是内置 if，并且扩展将无限期循环。

同样，如果将下划线替换为标识符 if，则扩展将再次无限期循环。出现在模板中的 if（如果 e1 e2 e3）将被视为绑定到相应标识符的模式变量 if 来自输入表单，这表示 if 的本地版本。

将 if 放在文本列表中以尝试修补后一个版本也不起作用。这将导致语法大小写将模式中的文本 if 与输入表达式中的 if 进行比较，后者的作用域在 let 语法表达式之外，后者的作用域在 let 语法内。由于它们不会引用

相同的绑定，因此它们不会是 free-identifier=? ，并且会导致语法冲突。

下划线（_）的常规用法有助于程序员避免出现错误标识符匹配或意外插入的情况。

生成对输入表单上下文中不存在的标识符的引用是一种语法冲突，如果插入到转换器输出中的标识符的“最接近的封闭词法绑定”没有同时包含输入表单，则可能会发生这种情况。例如

```
(let-syntax ([divide (lambda (x)
(let ([/ +])
(syntax-case x ()
[(_ e1 e2) #'(/ e1 e2)]))
(let ([/ *]) (divide 2 1)))
```

应该会导致语法冲突，其消息大意是 / 在无效上下文中被引用，因为在 divide 的输出中出现的 / 是对变量 / 受转换器内 let 表达式约束的引用。

下一个示例定义了一个可定义可积的形式，该窗体类似于为过程定义定义的窗体，只是它会导致在找到对过程的直接调用的位置集成或插入过程的代码。

```
(define-syntax define-integrable
(syntax-rules (lambda)
[(_ name (lambda formals form1 form2 ...))
(begin
(define xname (lambda formals form1 form2 ...))
(define-syntax name
(lambda (x)
(syntax-case x ()
[_ (identifier? x) #'xname]
[(_ arg (... ...))
```

```
#’（（lambda formals form1 form2 ...）
arg
 (... ...）））））））））
```

形式（定义可积名称 lambda-表达式）扩展为一对定义：名称的语法定义和 xname 的变量定义。名称转换器将对 name 的明显调用转换为对 lambda 表达式的直接调用。由于生成的表单只是直接 lambda 应用程序（相当于 let 表达式），因此实际参数在计算过程主体之前根据需要精确计算一次。对名称的所有其他引用都将替换为对 xname 的引用。xname 的定义将其绑定到 lambda 表达式的值。这允许将过程用作一等值。可定义可积转换器在 lambda 表达式内或调用站点上维护词法范围没有任何特殊作用，因为词法范围由扩展程序自动维护。此外，由于 xname 是由转换器引入的，因此 xname 的绑定在任何地方都不可见，除非名称转换器引入了对它的引用。

上述定义可定义可积不适用于递归过程，因为递归调用将导致无限数量的扩展步骤，可能导致在扩展时耗尽内存。对于直接递归过程，此问题的解决方案是使用 let 语法绑定包装 lambda 表达式的每个匹配项，该绑定无条件地将名称扩展到 xname。

```
 (define-syntax define-integrable
 (syntax-rules （lambda）
[（_ name （lambda formals form1 form2 ...））
 (begin
 (define xname
 (let-syntax （[name （identifier-syntax xname）])
 (lambda formals form1 form2 ...））)
 (define-syntax name
 (lambda （x）
 (syntax-case x （）
[_ （identifier? x） #’xname]
[（_ arg （... ...））
```

```
#'(let-syntax ([name (identifier-syntax xname)])
 (lambda formals form1 form2 ...))
arg (... ...)))))))))))
```

对于相互递归过程，可以通过将 let 语法形式替换为非标准的流体 let 语法形式来解决，这在 Chez Scheme 用户指南 [9] 中有所描述。

define-integrable 的两个定义都处理标识符出现在结构化表达式的第一个位置的情况，与它出现在其他地方的情况不同，标识符描述中给出的 pcar 示例也是如此？在其他情况下，必须对这两种情况进行相同的处理。表单标识符语法可以使这样做更方便。

```
(let ([x 0])
(define-syntax x++
(identifier-syntax
(let ([t x])
(set! x (+ t 1)) t)))
(let ([a x++]) (list a x))) ⇒ (0 1)
```

下面的示例使用标识符语法、基准>语法和局部语法定义来定义方法的形式，这是面向对象编程 （OOP） 系统的基本构建块之一。方法表达式类似于 lambda 表达式，不同之处在于除了形式参数和正文之外，方法表达式还包含实例变量列表 （ivar ...）。调用方法时，始终向其传递一个对象（实例），表示为与实例变量对应的字段向量，以及零个或多个附加参数。在方法主体中，对象隐式绑定到标识符 self，其他参数绑定到形式参数。可以通过实例变量引用或赋值在方法主体内访问或更改对象的字段。

```
(define-syntax method
(lambda (x)
(syntax-case x ()
```

```
[ (k (ivar ...) formals b1 b2 ...)
 (with-syntax ([ (index ...)
 (let f ([i 0] [ls #'(ivar ...) ])
 (if (null? ls)
' ()
 (cons i (f (+ i 1) (cdr ls) ) ) ) )
[self (datum->syntax #'k 'self) ]
[set! (基准面>语法 #'k 'set! ) )
#'(lambda (self . formals)
 (let-syntax ([ivar (identifier-syntax
 (vector-ref self index) ) ]
...)
 (let-syntax ([set!
 (语法规则 (ivar ...)
[ (_ ivar e) (vector-set! self index e) ]
...
[ (_ x e) (set! x e) ]) )
b1 b2 ...) ) ) ) ) )
```

ivar ... 和 set! 的局部绑定使对象的字段看起来像是普通变量，引用和赋值转换为对 vector-ref 和 vector-set! 的调用。基准>语法用于使引入的 self 和 set! 绑定在方法主体中可见。需要嵌套的 let 语法表达式，以便正确限定标识符 ivar ... 作为本地版本的 set! 的辅助关键字。

通过使用标识符语法的一般形式来更直接地处理set!表单，我们可以简化方法的定义。

```
 (define-syntax method
 (lambda (x)
 (syntax-case x ()
[ (k (ivar ...) formals b1 b2 ...)
 (with-syntax ([ (index ...)
 (let f ([i 0] [ls #'(ivar ...) ])
 (if (null? ls)
' ()
```

```
(cons i (f (+ i 1) (cdr ls))))))
[self (datum->syntax #'k 'self)])
#'(lambda (self . formals)
(let-syntax ([ivar (identifier-syntax
[_ (vector-ref self index)]
[(set! _ e)
(vector-set! self index e)])]
...)
b1 b2 ...)))))))))
```

下面的示例演示了方法的简单用法。

```
(let ([m (方法 (a) (x) (列表 a x self))])
(m #(1) 2)) ⇒ (1 2 #(1))

(let ([m (方法 (a) (x)
(set! a x)
(set! x (+ a x))
(list a x self))])
(m #(1) 2)) ⇒ (2 4 #(2))
```

在基于方法的完整OOP系统中，实例变量ivar...可能来自类声明，而不是在方法表单中显式列出，尽管相同的技术将用于使实例变量在方法体中显示为普通变量。

本节的最后一个示例定义了一个简单的结构定义工具，该工具将结构表示为具有命名字段的向量。结构是用定义结构定义的，其形式为

```
(定义结构名称字段...)
```

其中 name 命名结构和字段 ...命名其字段。define-structure 扩展为一系列生成的定义：一个构造函数 make-name、一个类型谓词 name?，以及每个字段名称一个访问器 name-field 和 setter set-name-field!。

```
 (define-syntax define-structure
 (lambda (x)
 (define gen-id
 (lambda (template-id . args)
 (datum->syntax template-id
 (string->symbol
 (apply string-append
 (map (lambda (x)
 (if (string? x)
x
 (symbol->string (syntax->datum x)))
)
 (syntax-case x ()
[(_ name field ...)
 (with-syntax ([constructor (gen-id #'name "make-"
#'name)]
[谓词 (gen-id #'name #'name "?")]
[(访问...)
 (map (lambda (x) (gen-id x #'name "-" x))
#'(field ...))]
[(分配 ...)
 (map (lambda (x)
 (gen-id x "set-" #'name "-" x "!"))
#'(字段 ...))]
[结构长度 (+ (长度 #'(字段 ...)) 1)]
[(索引 ...)
 (设 f ([i 1] [ids #'(字段 ...)])
 (if (null? ids)
'()
 (cons i (f (+ i 1) (cdr ids)))))))
#'(begin
 (define constructor
 (lambda (field ...)
 (矢量'名称字段...)))
 (定义谓
词(lambda (x)
 (和(向量? x)
 (=(向量长度x)结构长度)
 (eq? (矢量引用 x 0)'name)))))
```

```
 (定义访问
 (lambda (x)
 (vector-ref x index)))
...
 (定义赋值
 (lambda (x update)
 (vector-set! x index update)))
...)))))))
```

构造函数接受与结构中存在的字段一样多的参数，并创建一个向量，其第一个元素是符号名称，其其余元素是参数值。如果类型谓词的参数是预期长度的向量（其第一个元素是 name），则类型谓词返回 true。

由于定义结构形式扩展为包含定义的开始，因此它本身就是一个定义，可以在任何定义有效的地方使用。

生成的标识符是使用基准>语法创建的，以允许标识符在定义结构窗体出现的位置可见。

下面的示例演示了定义结构的使用。

```
 (定义结构树左右)
 (定义 t
 (make-tree (make-tree
0 1)
 (make-tree 2 3)))

t ⇒ #(tree #(tree 0 1) #(tree 2 3))
 (tree? t) ⇒ #t
 (tree-left t) ⇒ #(tree 0 1)
 (tree-right t) ⇒ #(tree 2 3)
 (set-tree-left! t 0)
t ⇒ #(tree 0 #(tree 2 3))
```

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

http://www.scheme.com