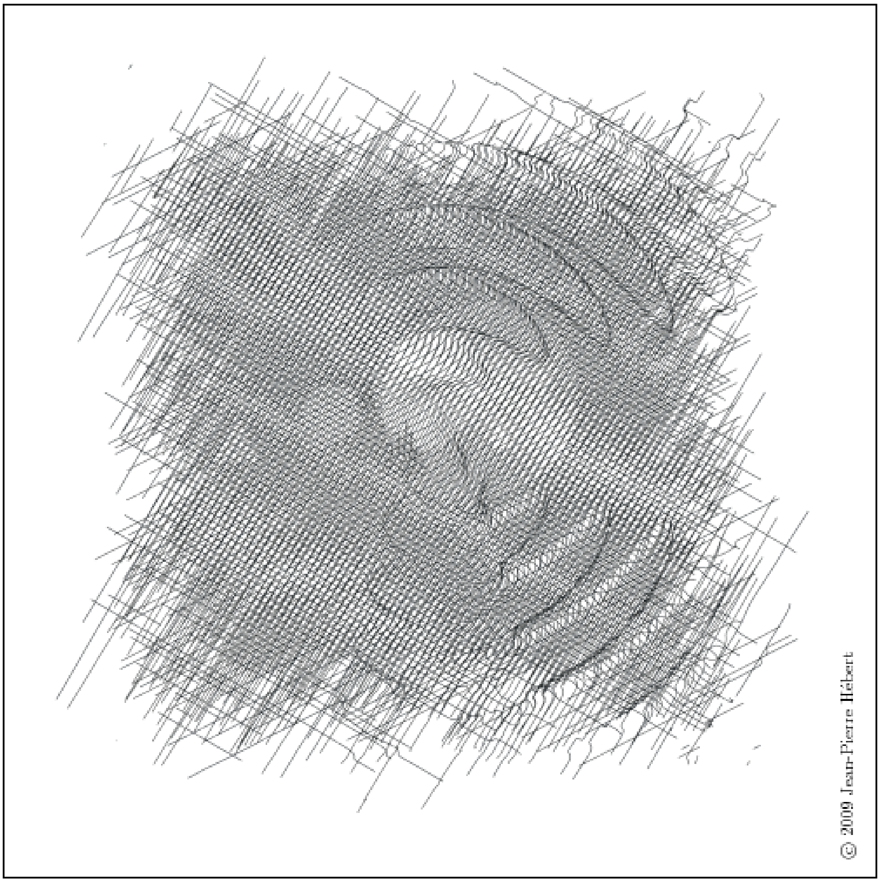

Chapter 8. Syntactic Extension

Syntactic extensions, or *macros*, are used to simplify and regularize repeated patterns in a program, to introduce syntactic forms with new evaluation rules, and to perform transformations that help make programs more efficient.

A syntactic extension most often takes the form `(`*`keyword subform`*` ...)`, where *`keyword`* is the identifier that names the syntactic extension. The syntax of each *`subform`* varies from one syntactic extension to another. Syntactic extensions can also take the form of improper lists or even singleton identifiers.

New syntactic extensions are defined by associating keywords with transformation procedures, or *transformers*. Syntactic extensions are defined using `define-syntax` forms or using `let-syntax` or `letrec-syntax`. Transformers may be created using `syntax-rules`, which allows simple pattern-based transformations to be performed.

They may also be ordinary procedures that accept one argument and perform arbitrary computations. In this case, `syntax-case` is normally used to destructure the input and `syntax` is normally used to construct the output. The `identifier-syntax` form and `make-variable-transformer` procedure allow the creation of transformers that match singleton identifiers and assignments to those identifiers, the former being restricted to simple patterns like `syntax-rules` and the latter allowing arbitrary computations to be performed.

Syntactic extensions are expanded into core forms at the start of evaluation (before compilation or interpretation) by a syntax *expander*. If the expander encounters a syntactic extension, it invokes the associated transformer to expand the syntactic extension, then repeats the expansion process for the form returned by the transformer. If the expander encounters a core syntactic form, it recursively processes the subforms, if any, and reconstructs the form from the expanded subforms. Information about identifier bindings is maintained during expansion to enforce lexical scoping for variables and keywords.

The syntactic extension mechanisms described in this chapter are part of the "syntax-case" system. A portable implementation of the system that also supports libraries and top-level programs is available at http://www.cs.indiana.edu/syntax-case/. A description of the motivations behind and implementation of the system can be found in the article "Syntactic Abstraction in Scheme" [12]. Additional features that have not yet been standardized, including `modules`, local `import`, and meta definitions, are described in the *Chez Scheme User's Guide* [9].

Section 8.1. Keyword Bindings

This section describes forms that establish bindings between keywords and transformers. Keyword bindings may be established within a top-level program or library body using `define-syntax` and in any local scope using `define-syntax`, `let-syntax`, or `letrec-syntax`.

syntax: `(define-syntax` *`keyword expr`*`)`
libraries: `(rnrs base)`, `(rnrs)`

`expr` must evaluate to a transformer.

The following example defines `let*` as a syntactic extension, specifying the transformer with `syntax-rules` (see Section 8.2).

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

All bindings established by a set of internal definitions, whether keyword or variable definitions, are visible everywhere within the immediately enclosing body, including within the definitions themselves. For example, the expression

```
(let ()
  (define even?
    (lambda (x)
      (or (= x 0) (odd? (- x 1)))))
  (define-syntax odd?
    (syntax-rules ()
      [(_ x) (not (even? x))]))
  (even? 10))
```

is valid and should evaluate to `#t`.

The expander processes the initial forms in a `library`, `lambda`, or other body from left to right. If it encounters a variable definition, it records the fact that the defined identifier is a variable but defers expansion of the right-hand-side expression until after all of the definitions have been processed. If it encounters a keyword definition, it expands and evaluates the right-hand-side expression and binds the keyword to the resulting transformer. If it encounters an expression, it fully expands all deferred right-hand-side expressions along with the current and remaining body expressions.

An implication of the left-to-right processing order is that one internal definition can affect whether a subsequent form is also a definition. For example, the expression

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

evaluates to `0`, regardless of any binding for `bind-to-zero` that might appear outside of the `let` expression.

syntax: `(let-syntax ((`*`keyword expr`*`) ...)` *`form`*$_1$ *`form`*$_2$ `...)`
syntax: `(letrec-syntax ((`*`keyword expr`*`) ...)` *`form`*$_1$ *`form`*$_2$ `...)`
returns: see below
libraries: `(rnrs base)`, `(rnrs)`

Each *`expr`* must evaluate to a transformer. For `let-syntax` and `letrec-syntax` both, each *`keyword`* is bound within the forms *`form`*$_1$ *`form`*$_2$ `...`. For `letrec-syntax` the binding scope also includes each *`expr`*.

A let-syntax or letrec-syntax form may expand into one or more expressions anywhere expressions are permitted, in which case the resulting expressions are treated as if enclosed in a begin expression. It may also expand into zero or more definitions anywhere definitions are permitted, in which case the definitions are treated as if they appeared in place of the let-syntax or letrec-syntax form.

The following example highlights how let-syntax and letrec-syntax differ.

```
(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)
```

The two expressions are identical except that the let-syntax form in the first expression is a letrec-syntax form in the second. In the first expression, the f occurring in g refers to the let-bound variable f, whereas in the second it refers to the keyword f whose binding is established by the letrec-syntax form.

Section 8.2. Syntax-Rules Transformers

The syntax-rules form described in this section permits simple transformers to be specified in a convenient manner. These transformers may be bound to keywords using the mechanisms described in Section 8.1. While it is much less expressive than the mechanism described in Section 8.3, it is sufficient for defining many common syntactic extensions.

syntax: (syntax-rules (*literal* ...) *clause* ...)
returns: a transformer
libraries: (rnrs base), (rnrs)

Each *literal* must be an identifier other than an underscore (_) or ellipsis (...). Each clause must take the form below.

```
(pattern template)
```

Each *pattern* specifies one possible syntax that the input form might take, and the corresponding *template* specifies how the output should appear.

Patterns consist of list structure, vector structure, identifiers, and constants. Each identifier within a pattern is either a `literal`, a *pattern variable*, an *underscore*, or an *ellipsis*. The identifier _ is an underscore, and the identifier `...` is an ellipsis. Any identifier other than _ or `...` is a literal if it appears in the list of literals `(literal ...)`; otherwise, it is a pattern variable. Literals serve as auxiliary keywords, such as `else` in `case` and `cond` expressions. List and vector structure within a pattern specifies the basic structure required of the input, the underscore and pattern variables specify arbitrary substructure, and literals and constants specify atomic pieces that must match exactly. Ellipses specify repeated occurrences of the subpatterns they follow.

An input form F matches a pattern P if and only if

- P is an underscore or pattern variable,
- P is a literal identifier and F is an identifier with the same binding as determined by the predicate `free-identifier=?` (Section 8.3),
- P is of the form $(P_1 \ ... \ P_n)$ and F is a list of *n* elements that match P_1 through P_n,
- P is of the form $(P_1 \ ... \ P_n \ . \ P_x)$ and F is a list or improper list of *n* or more elements whose first *n* elements match P_1 through P_n and whose *n*th cdr matches P_x,
- P is of the form $(P_1 \ ... \ P_k \ P_e \ \mathit{ellipsis} \ P_{m+1} \ ... \ P_n)$, where $\mathit{ellipsis}$ is the identifier `...` and F is a proper list of *n* elements whose first *k* elements match P_1 through P_k, whose next *m* - *k* elements each match P_e, and whose remaining *n* - *m* elements match P_{m+1} through P_n,
- P is of the form $(P_1 \ ... \ P_k \ P_e \ \mathit{ellipsis} \ P_{m+1} \ ... \ P_n \ . \ P_x)$, where $\mathit{ellipsis}$ is the identifier `...` and F is a list or improper list of *n* elements whose first *k* elements match P_1 through P_k, whose next *m* - *k* elements each match P_e, whose next *n* - *m* elements match P_{m+1} through P_n, and whose *n*th and final cdr matches P_x,
- P is of the form $\#(P_1 \ ... \ P_n)$ and F is a vector of *n* elements that match P_1 through P_n,
- P is of the form $\#(P_1 \ ... \ P_k \ P_e \ \mathit{ellipsis} \ P_{m+1} \ ... \ P_n)$, where $\mathit{ellipsis}$ is the identifier `...` and F is a vector of *n* elements whose first *k* elements match P_1 through P_k, whose next *m* - *k* elements each match P_e, and whose remaining *n* - *m* elements match P_{m+1} through P_n, or

- `P` is a pattern datum (any nonlist, nonvector, nonsymbol object) and `F` is equal to `P` in the sense of the `equal?` procedure.

The outermost structure of a `syntax-rules` *`pattern`* must actually be in one of the list-structured forms above, although subpatterns of the pattern may be in any of the above forms. Furthermore, the first element of the outermost pattern is ignored, since it is always assumed to be the keyword naming the syntactic form. (These statements do not apply to `syntax-case`; see Section 8.3.)

If an input form passed to a `syntax-rules` transformer matches the pattern for a given clause, the clause is accepted and the form is transformed as specified by the associated template. As this transformation takes place, pattern variables appearing in the pattern are bound to the corresponding input subforms. Pattern variables appearing within a subpattern followed by one or more ellipses may be bound to a sequence or sequences of zero or more input subforms.

A template is a pattern variable, an identifier that is not a pattern variable, a pattern datum, a list of subtemplates $(S_1 \ \ldots \ S_n)$, an improper list of subtemplates $(S_1 \ S_2 \ \ldots \ S_n \ . \ T)$, or a vector of subtemplates $\#(S_1 \ \ldots \ S_n)$. Each subtemplate S_i is a template followed by zero or more ellipses. The final element `T` of an improper subtemplate list is a template.

Pattern variables appearing within a template are replaced in the output by the input subforms to which they are bound. Pattern data and identifiers that are not pattern variables are inserted directly into the output. List and vector structure within the template remains list and vector structure in the output. A subtemplate followed by an ellipsis expands into zero or more occurrences of the subtemplate. The subtemplate must contain at least one pattern variable from a subpattern followed by an ellipsis. (Otherwise, the expander could not determine how many times the subform should be repeated in the output.) Pattern variables that occur in subpatterns followed by one or more ellipses may occur only in subtemplates that are followed by (at least) as many ellipses. These pattern variables are replaced in the output by the input subforms to which they are bound, distributed as specified. If a pattern variable is followed by more ellipses in the template than in the associated pattern, the input form is replicated as necessary.

A template of the form `(...` *`template`*`)` is identical to *`template`*, except that ellipses within the template have no special meaning. That is, any ellipses contained within *`template`* are treated as ordinary identifiers. In particular, the template `(... ...)` produces a single ellipsis, `...`. This allows syntactic extensions to expand into forms containing ellipses, including `syntax-rules` or `syntax-case` patterns and templates.

The definition of `or` below demonstrates the use of `syntax-rules`.

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

The input patterns specify that the input must consist of the keyword and zero or more subexpressions. An underscore (_), which is a special pattern symbol that matches any input, is often used for the keyword position to remind the programmer and anyone reading the definition that the keyword position never fails to contain the expected keyword and need not be matched. (In fact, as mentioned above, `syntax-rules` ignores what appears in the keyword position.) If more than one subexpression is present (third clause), the expanded code both tests the value of the first subexpression and returns the value if it is not false. To avoid evaluating the expression twice, the transformer introduces a binding for the temporary variable `t`.

The expansion algorithm maintains lexical scoping automatically by renaming local identifiers as necessary. Thus, the binding for `t` introduced by the transformer is visible only within code introduced by the transformer and not within subforms of the input. Similarly, the references to the identifiers `let` and `if` are unaffected by any bindings present in the context of the input.

```
(let ([if #f])
  (let ([t 'okay])
    (or if t))) ⇒ okay
```

This expression is transformed during expansion to the equivalent of the expression below.

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

In this sample expansion, `if1`, `t1`, and `t2` represent identifiers to which `if` and `t` in the original expression and `t` in the expansion of `or` have been renamed.

The definition of a simplified version of `cond` below (simplified because it requires at least one output expression per clause and does not support the auxiliary keyword =>) demonstrates how auxiliary keywords such as `else` are recognized in the input to a transformer, via inclusion in the list of literals.

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

syntax: `_`
syntax: `...`
libraries: `(rnrs base)`, `(rnrs syntax-case)`, `(rnrs)`

These identifiers are auxiliary keywords for `syntax-rules`, `identifier-syntax`, and `syntax-case`. The second (`...`) is also an auxiliary keyword for `syntax` and `quasisyntax`. It is a syntax violation to reference these identifiers except in contexts where they are recognized as auxiliary keywords.

syntax: `(identifier-syntax` *`tmpl`*`)`
syntax: `(identifier-syntax (`id_1 $tmpl_1$`) ((set!` id_2 e_2`)` $tmpl_2$`))`
returns: a transformer
libraries: `(rnrs base)`, `(rnrs)`

When a keyword is bound to a transformer produced by the first form of `identifier-syntax`, references to the keyword within the scope of the binding are replaced by *`tmpl`*.

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

With the first form of `identifier-syntax`, an apparent assignment of the associated keyword with `set!` is a syntax violation. The second, more general, form of `identifier-syntax` permits the transformer to specify what happens when `set!` is used.

```
(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))
```

A definition of `identifier-syntax` in terms of `make-variable-transformer` is shown on page 307.

Section 8.3. Syntax-Case Transformers

This section describes a more expressive mechanism for creating transformers, based on `syntax-case`, a generalized version of `syntax-rules`. This mechanism permits arbitrarily complex transformations to be specified, including transformations that "bend" lexical scoping in a controlled manner, allowing a much broader class of syntactic extensions to be defined. Any transformer that may be defined using `syntax-rules` may be rewritten easily to use `syntax-case` instead; in fact, `syntax-rules` itself may be defined as a syntactic extension in terms of `syntax-case`, as demonstrated within the description of `syntax` below.

With this mechanism, transformers are procedures of one argument. The argument is a *syntax object* representing the form to be processed. The return value is a syntax object representing the output form. A syntax object may be any of the following.

- a nonpair, nonvector, nonsymbol value,
- a pair of syntax objects,
- a vector of syntax objects, or
- a wrapped object.

The *wrap* on a wrapped syntax object contains contextual information about a form in addition to its structure. This contextual information is used by the expander to maintain lexical scoping. The wrap may also contain information used by the implementation to correlate source and object code, e.g., track file, line, and character information through the expansion and compilation process.

The contextual information must be present for all identifiers, which is why the definition of syntax object above does not allow symbols unless they are wrapped. A syntax object representing an identifier is itself referred to as an identifier; thus, the term *identifier* may refer either to the syntactic entity (symbol, variable, or keyword) or to the concrete representation of the syntactic entity as a syntax object.

Transformers normally destructure their input with `syntax-case` and rebuild their output with `syntax`. These two forms alone are sufficient for defining many syntactic extensions, including any that can be defined using `syntax-rules`. They are described below along with a set of additional forms and procedures that provide added functionality.

syntax: `(syntax-case` *`expr`* `(`*`literal`* `...)` *`clause`* `...)`
returns: see below
libraries: `(rnrs syntax-case)`, `(rnrs)`

Each *`literal`* must be an identifier. Each *`clause`* must take one of the following two forms.

```
(pattern output-expression)
(pattern fender output-expression)
```

`syntax-case` patterns may be in any of the forms described in Section 8.2.

`syntax-case` first evaluates *`expr`*, then attempts to match the resulting value against the pattern from the first *`clause`*. This value may be any Scheme object. If the value matches the pattern and no *fender* is present, *output-expression* is evaluated and its values returned as the values of the syntax-case expression. If the value does not match the pattern, the value is compared against the next clause, and so on. It is a syntax violation if the value does not match any of the patterns.

If the optional *fender* is present, it serves as an additional constraint on acceptance of a clause. If the value of the syntax-case *expr* matches the pattern for a given clause, the corresponding *fender* is evaluated. If *fender* evaluates to a true value, the clause is accepted; otherwise, the clause is rejected as if the input had failed to match the pattern. Fenders are logically a part of the matching process, i.e., they specify additional matching constraints beyond the basic structure of an expression.

Pattern variables contained within a clause's *pattern* are bound to the corresponding pieces of the input value within the clause's *fender* (if present) and *output-expression*. Pattern variables occupy the same namespace as program variables and keywords; pattern variable bindings created by syntax-case can shadow (and be shadowed by) program variable and keyword bindings as well as other pattern variable bindings. Pattern variables, however, can be referenced only within syntax expressions.

See the examples following the description of syntax.

syntax: (syntax *template*)
syntax: #'*template*
returns: see below
libraries: (rnrs syntax-case), (rnrs)

#'*template* is equivalent to (syntax *template*). The abbreviated form is converted into the longer form when a program is read, prior to macro expansion.

A syntax expression is like a quote expression except that the values of pattern variables appearing within *template* are inserted into *template*, and contextual information associated both with the input and with the template is retained in the output to support lexical scoping. A syntax *template* is identical to a syntax-rules *template* and is treated similarly.

List and vector structures within the template become true lists or vectors (suitable for direct application of list or vector operations, like map or vector-ref) to the extent that the list or vector structures must be copied to insert the values of pattern variables, and empty lists are never wrapped. For example, #'(x ...), #'(a b c), #'() are all lists if x, a, b, and c are pattern variables.

The definition of or below is equivalent to the one given in Section 8.2 except that it employs syntax-case and syntax in place of syntax-rules.

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

In this version, the `lambda` expression that produces the transformer is explicit, as are the `syntax` forms in the output part of each clause. Any `syntax-rules` form can be expressed with `syntax-case` by making the `lambda` expression and `syntax` expressions explicit. This observation leads to the following definition of `syntax-rules` in terms of `syntax-case`.

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      [(_ (i ...) ((keyword . pattern) template) ...)
       #'(lambda (x)
           (syntax-case x (i ...)
             [(_ . pattern) #'template] ...))])))
```

An underscore is used in place of each `keyword` since the first position of each `syntax-rules` pattern is always ignored.

Since the `lambda` and `syntax` expressions are implicit in a `syntax-rules` form, definitions expressed with `syntax-rules` are often shorter than the equivalent definitions expressed with `syntax-case`. The choice of which to use when either suffices is a matter of taste, but many transformers that can be written easily with `syntax-case` cannot be written easily or at all with `syntax-rules` (see Section 8.4).

procedure: `(identifier? *obj*)`
returns: `#t` if *`obj`* is an identifier, `#f` otherwise
libraries: `(rnrs syntax-case)`, `(rnrs)`

`identifier?` is often used within fenders to verify that certain subforms of an input form are identifiers, as in the definition of unnamed `let` below.

```
(define-syntax let
  (lambda (x)
    (define ids?
      (lambda (ls)
        (or (null? ls)
            (and (identifier? (car ls))
                 (ids? (cdr ls))))))
    (syntax-case x ()
      [(_ ((i e) ...) b1 b2 ...)
       (ids? #'(i ...))
       #'((lambda (i ...) b1 b2 ...) e ...)])))
```

Syntactic extensions ordinarily take the form `(`*`keyword subform`*` ...)`, but the `syntax-case` system permits them to take the form of singleton identifiers as well. For example, the keyword `pcar` in the expression below may be used both as an identifier (in which case it expands into a call to `car`) or as a structured form (in which case it expands into a call to `set-car!`).

```
(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)
```

The fender `(identifier? x)` is used to recognize the singleton identifier case.

procedure: `(free-identifier=? `*`identifier`*$_1$ *`identifier`*$_2$`)`
procedure: `(bound-identifier=? `*`identifier`*$_1$ *`identifier`*$_2$`)`
returns: see below
libraries: `(rnrs syntax-case)`, `(rnrs)`

Symbolic names alone do not distinguish identifiers unless the identifiers are to be used only as symbolic data. The predicates `free-identifier=?` and `bound-identifier=?` are used to compare identifiers according to their *intended use* as free references or bound identifiers in a given context.

`free-identifier=?` is used to determine whether two identifiers would be equivalent if they were to appear as free identifiers in the output of a transformer. Because identifier references are lexically scoped, this means `(free-identifier=? `*`id`*$_1$ *`id`*$_2$`)` is true if and only if the identifiers *`id`*$_1$ and *`id`*$_2$ refer to the same binding. (For this comparison, two like-named identifiers are assumed to have the same binding if neither is bound.) Literal identifiers (auxiliary keywords) appearing in `syntax-case` patterns (such as `else` in `case` and `cond`) are matched with `free-identifier=?`.

Similarly, `bound-identifier=?` is used to determine whether two identifiers would be equivalent if they were to appear as bound identifiers in the output of a transformer. In other words, if `bound-identifier=?` returns true for two identifiers, a binding for one will capture references to the other within its scope. In general, two identifiers are `bound-identifier=?` only if both are present in the original program or both are introduced by the same transformer application (perhaps implicitly---see `datum->syntax`). `bound-identifier=?` can be used for detecting duplicate identifiers in a binding construct or for other preprocessing of a binding construct that requires detecting instances of the bound identifiers.

The definition below is equivalent to the earlier definition of a simplified version of `cond` with `syntax-rules`, except that `else` is recognized via an explicit call to `free-identifier?` within a fender rather than via inclusion in the literals list.

```
(define-syntax cond
  (lambda (x)
    (syntax-case x ()
      [(_ (e0 e1 e2 ...))
       (and (identifier? #'e0) (free-identifier=? #'e0 #'else))
       #'(begin e1 e2 ...)]
      [(_ (e0 e1 e2 ...)) #'(if e0 (begin e1 e2 ...))]
      [(_ (e0 e1 e2 ...) c1 c2 ...)
       #'(if e0 (begin e1 e2 ...) (cond c1 c2 ...))])))
```

With either definition of `cond`, `else` is not recognized as an auxiliary keyword if an enclosing lexical binding for `else` exists. For example,

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

does *not* write `"oops"`, since `else` is bound lexically and is therefore not the same `else` that appears in the definition of `cond`.

The following definition of unnamed `let` uses `bound-identifier=?` to detect duplicate identifiers.

```
(define-syntax let
  (lambda (x)
    (define ids?
      (lambda (ls)
        (or (null? ls)
            (and (identifier? (car ls)) (ids? (cdr ls))))))
    (define unique-ids?
      (lambda (ls)
        (or (null? ls)
            (and (not (memp
                        (lambda (x) (bound-identifier=? x (car ls)))
                        (cdr ls)))
                 (unique-ids? (cdr ls))))))
    (syntax-case x ()
      [(_ ((i e) ...) b1 b2 ...)
       (and (ids? #'(i ...)) (unique-ids? #'(i ...)))
       #'((lambda (i ...) b1 b2 ...) e ...)])))
```

With the definition of `let` above, the expression

```
(let ([a 3] [a 4]) (+ a a))
```

is a syntax violation, whereas

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

evaluates to `7` since the identifier `a` introduced by `dolet` and the identifier `a` extracted from the input form are not `bound-identifier=?`. Since both occurrences of `a`, however, if left as free references, would refer to the same binding for `a`, `free-identifier=?` would not distinguish them.

Two identifiers that are `free-identifier=?` may not be `bound-identifier=?`. An identifier introduced by a transformer may refer to the same enclosing binding as an identifier not introduced by the transformer, but an introduced binding for one will not capture references to the other. On the other hand, identifiers that are `bound-identifier=?` are `free-identifier=?`, as long as the identifiers have valid bindings in the context where they are compared.

syntax: `(with-syntax ((`*`pattern expr`*`) ...)` *body*$_1$ *body*$_2$ `...)`
returns: the values of the final body expression
libraries: `(rnrs syntax-case)`, `(rnrs)`

It is sometimes useful to construct a transformer's output in separate pieces, then put the pieces together. `with-syntax` facilitates this by allowing the creation of local pattern bindings.

pattern is identical in form to a `syntax-case` pattern. The value of each *expr* is computed and destructured according to the corresponding *pattern*, and pattern variables within the *pattern* are bound as with `syntax-case` to appropriate portions of the value within the body *body*$_1$ *body*$_2$ `...`, which is processed and evaluated like a `lambda` body.

`with-syntax` may be defined as a syntactic extension in terms of `syntax-case`.

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

The following definition of full `cond` demonstrates the use of `with-syntax` to support transformers that employ recursion internally to construct their output.

```
(define-syntax cond
  (lambda (x)
    (syntax-case x ()
      [(_ c1 c2 ...)
       (let 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 ...)
                  #'(if e0 (begin e1 e2 ...) rest)]))))])))
```

syntax: `(quasisyntax` *`template`* `...)`
syntax: `` #` ``*`template`*
syntax: `(unsyntax` *`template`* `...)`
syntax: `#,`*`template`*
syntax: `(unsyntax-splicing` *`template`* `...)`
syntax: `#,@`*`template`*
returns: see below
libraries: `(rnrs syntax-case)`, `(rnrs)`

`` #` ``*`template`* is equivalent to `(quasisyntax` *`template`*`)`, while `#,`*`template`* is equivalent to `(unsyntax` *`template`*`)`, and `#,@`*`template`* to `(unsyntax-splicing` *`template`*`)`. The abbreviated forms are converted into the longer forms when the program is read, prior to macro expansion.

`quasisyntax` is similar to `syntax`, but it allows parts of the quoted text to be evaluated, in a manner similar to `quasiquote` (Section 6.1).

Within a `quasisyntax` *`template`*, subforms of `unsyntax` and `unsyntax-splicing` forms are evaluated, and everything else is treated as ordinary template material, as with `syntax`. The value of each `unsyntax` subform is inserted into the output in place of the `unsyntax` form, while the value of each `unsyntax-splicing` subform is spliced into the surrounding list or vector structure. `unsyntax` and `unsyntax-splicing` are valid only within `quasisyntax` expressions.

`quasisyntax` expressions may be nested, with each `quasisyntax` introducing a new level of syntax quotation and each `unsyntax` or `unsyntax-splicing` taking away a level of quotation. An expression nested within *n* `quasisyntax` expressions must be within *n* `unsyntax` or `unsyntax-splicing` expressions to be evaluated.

`quasisyntax` can be used in place of `with-syntax` in many cases. For example, the following definition of `case` employs `quasisyntax` to construct its output, using internal recursion in a manner similar to the definition of `cond` given under the description of `with-syntax` above.

```
(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 ...)) (begin e1 e2 ...))])
                   (syntax-case c1 ()
                     [((k ...) e1 e2 ...)
                      #`(if (memv t '(k ...))
                            (begin e1 e2 ...)
                            #,(f (car cmore) (cdr cmore)))]))))])))
```

`unsyntax` and `unsyntax-splicing` forms that contain zero or more than one subform are valid only in splicing (list or vector) contexts. `(unsyntax *template* ...)` is equivalent to `(unsyntax *template*) ...`, and `(unsyntax-splicing *template* ...)` is equivalent to `(unsyntax-splicing *template*) ...`. These forms are primarily useful as intermediate forms in the output of the `quasisyntax` expander. They support certain useful nested quasiquotation (`quasisyntax`) idioms [3], such as `#,@#,@`, which has the effect of a doubly indirect splicing when used within a doubly nested and doubly evaluated `quasisyntax` expression, as with the nested `quasiquote` examples shown in Section 6.1.

`unsyntax` and `unsyntax-splicing` are auxiliary keywords for `quasisyntax`. It is a syntax violation to reference these identifiers except in contexts where they are recognized as auxiliary keywords.

procedure: `(make-variable-transformer *procedure*)`
returns: a variable transformer
libraries: `(rnrs syntax-case)`, `(rnrs)`

As described in the lead-in to this section, transformers may simply be procedures that accept one argument, a syntax object representing the input form, and return a new syntax object representing the output form. The form passed to a transformer usually represents a parenthesized form whose first subform is the keyword bound to the transformer or just the keyword itself. `make-variable-transformer` may be used to convert a procedure into a special kind of transformer to which the expander also passes `set!` forms in which the keyword appears just after the `set!` keyword, as if it were a variable to be assigned. This allows the programmer to control what happens when the keyword appears in such contexts. The argument, *`procedure`*, should accept one argument.

```
(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))
```

This syntactic abstraction can be defined more succinctly using `identifier-syntax`, as shown in Section 8.2, but `make-variable-transformer` can be used to create transformers that perform arbitrary computations, while `identifier-syntax` is limited to simple term rewriting, like `syntax-rules`. `identifier-syntax` can be defined in terms of `make-variable-transformer`, as shown below.

```
(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])))])))
```

procedure: `(syntax->datum obj)`
returns: *`obj`* stripped of syntactic information
libraries: `(rnrs syntax-case)`, `(rnrs)`

The procedure `syntax->datum` strips all syntactic information from a syntax object and returns the corresponding Scheme "datum." Identifiers stripped in this manner are converted to their symbolic names, which can then be compared with `eq?`. Thus, a predicate `symbolic-identifier=?` might be defined as follows.

```
(define symbolic-identifier=?
  (lambda (x y)
    (eq? (syntax->datum x)
         (syntax->datum y))))
```

Two identifiers that are `free-identifier=?` need not be `symbolic-identifier=?`: two identifiers that refer to the same binding usually have the same name, but the `rename` and `prefix` subforms of the library's `import` form (page 345) may result in two identifiers with different names but the same binding.

procedure: `(datum->syntax *template-identifier obj*)`
returns: a syntax object
libraries: `(rnrs syntax-case)`, `(rnrs)`

`datum->syntax` constructs a syntax object from *`obj`* that contains the same contextual information as *`template-identifier`*, with the effect that the syntax object behaves as if it were introduced into the code when *`template-identifier`* was introduced. The template identifier is often the keyword of an input form, extracted from the form, and the object is often a symbol naming an identifier to be constructed.

`datum->syntax` allows a transformer to "bend" lexical scoping rules by creating *implicit identifiers* that behave as if they were present in the input form, thus permitting the definition of syntactic extensions that introduce visible bindings for or references to identifiers that do not appear explicitly in the input form. For example, we can define a `loop` expression that binds the variable `break` to an escape procedure within the loop body.

```
(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 a)
```

Were we to define `loop` as

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

the variable `break` would not be visible in `e ...`.

It is also useful for *`obj`* to represent an arbitrary Scheme form, as demonstrated by the following definition of `include`.

```
(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)])
           #'(begin expr ...)))])))
```

`(include "filename")` expands into a `begin` expression containing the forms found in the file named by `"filename"`. For example, if the file `f-def.ss` contains the expression `(define f (lambda () x))`, the expression

```
(let ([x "okay"])
  (include "f-def.ss")
  (f))
```

evaluates to `"okay"`.

The definition of `include` uses `datum->syntax` to convert the objects read from the file into syntax objects in the proper lexical context, so that identifier references and definitions within those expressions are scoped where the `include` form appears.

procedure: `(generate-temporaries *list*)`
returns: a list of distinct generated identifiers
libraries: `(rnrs syntax-case)`, `(rnrs)`

Transformers can introduce a fixed number of identifiers into their output by naming each identifier. In some cases, however, the number of identifiers to be introduced depends upon some characteristic of the input expression. A straightforward definition of `letrec`, for example, requires as many temporary identifiers as there are binding pairs in the input expression. The procedure `generate-temporaries` is used to construct lists of temporary identifiers.

`list` may be any list; its contents are not important. The number of temporaries generated is the number of elements in *`list`*. Each temporary is guaranteed to be different from all other identifiers.

A definition of `letrec` that uses `generate-temporaries` is shown below.

```
(define-syntax letrec
  (lambda (x)
    (syntax-case x ()
      [(_ ((i e) ...) b1 b2 ...)
       (with-syntax ([(t ...) (generate-temporaries #'(i ...))])
         #'(let ([i #f] ...)
             (let ([t e] ...)
               (set! i t)
               ...
               (let () b1 b2 ...))))]))))
```

Any transformer that uses `generate-temporaries` in this fashion can be rewritten to avoid using it, albeit with a loss of clarity. The trick is to use a recursively defined intermediate form that generates one temporary per expansion step and completes the expansion after enough temporaries have been generated. Here is a definition of `let-values` (page 99) that uses this technique to support multiple sets of bindings.

```
(define-syntax let-values
  (syntax-rules ()
    [(_ () f1 f2 ...) (let () 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))))]))
```

The implementation of `lvhelp` is complicated by the need to evaluate all of the right-hand-side expressions before creating any of the bindings and by the need to support improper formals lists.

Section 8.4. Examples

This section presents a series of illustrative syntactic extensions defined with either `syntax-rules` or `syntax-case`, starting with a few simple but useful syntactic extensions and ending with a fairly complex mechanism for defining structures with automatically generated constructors, predicates, field accessors, and field setters.

The simplest example in this section is the following definition of `rec`. `rec` is a syntactic extension that permits internally recursive anonymous (not externally named) procedures to be created with minimal effort.

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

(map (rec sum
       (lambda (x)
         (if (= x 0)
             0
             (+ x (sum (- x 1))))))
     '(0 1 2 3 4 5)) ⇒ (0 1 3 6 10 15)
```

Using `rec`, we can define the full `let` (both unnamed and named) as follows.

```
(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 ...)]))
```

We can also define `let` directly in terms of `letrec`, although the definition is a bit less clear.

```
(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 ...)]))
```

These definitions rely upon the fact that the first pattern cannot match a named `let`, since the first subform of a named `let` must be an identifier, not a list of bindings. The following definition uses a fender to make this check more robust.

```
(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 ...)])))
```

With the fender, we can even put the clauses in the opposite order.

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

To be completely robust, the `ids?` and `unique-ids?` checks employed in the definition of unnamed `let` in Section 8.3 should be employed here as well.

Both variants of `let` are easily described by simple one-line patterns, but `do` requires a bit more work. The precise syntax of `do` cannot be expressed directly with a single pattern because some of the bindings in a `do` expression's binding list may take the form `(var val)` while others take the form `(var val update)`. The following definition of `do` uses `syntax-case` internally to parse the bindings separately from the overall form.

```
(define-syntax do
  (lambda (x)
    (syntax-case x ()
      [(_ (binding ...) (test res ...) 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] ...)
             (if test
                 (begin (if #f #f) res ...)
                 (begin expr ... (doloop update ...)))))])))
```

The odd-looking expression `(if #f #f)` is inserted before the result expressions `res ...` in case no result expressions are provided, since `begin` requires at least one subexpression. The value of `(if #f #f)` is unspecified, which is what we want since the value of `do` is unspecified if no result expressions are provided. At the expense of a bit more code, we could use `syntax-case` to determine whether any result expressions are provided and to produce a loop with either a one- or two-armed `if` as appropriate. The resulting expansion would be cleaner but semantically equivalent.

As mentioned in Section 8.2, ellipses lose their special meaning within templates of the form `(...` *`template`*`)`. This fact allows syntactic extensions to expand into syntax definitions containing ellipses. This usage is illustrated by the definition below of `be-like-begin`.

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

With `be-like-begin` defined in this manner, `(be-like-begin sequence)` has the same effect as the following definition of `sequence`.

```
(define-syntax sequence
  (syntax-rules ()
    [(_ e0 e1 ...) (begin e0 e1 ...)]))
```

That is, a `sequence` form becomes equivalent to a `begin` form so that, for example:

```
(sequence (display "Say what?") (newline))
```

prints "Say what?" followed by a newline.

The following example shows how one might restrict `if` expressions within a given expression to require the "else" (alternative) subexpression by defining a local `if` in terms of the built-in `if`. Within the body of the `let-syntax` binding below, two-armed `if` works as always:

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

but one-armed if results in a syntax error.

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

Although this local definition of `if` looks simple enough, there are a few subtle ways in which an attempt to write it might go wrong. If `letrec-syntax` were used in place of `let-syntax`, the identifier `if` inserted into the output would refer to the local `if` rather than the built-in `if`, and expansion would loop indefinitely.

Similarly, if the underscore were replaced with the identifier `if`, expansion would again loop indefinitely. The `if` appearing in the template `(if e1 e2 e3)` would be treated as a pattern variable bound to the corresponding identifier `if` from the input form, which denotes the local version of `if`.

Placing `if` in the list of literals in an attempt to patch up the latter version would not work either. This would cause `syntax-case` to compare the literal `if` in the pattern, which would be scoped outside the `let-syntax` expression, with the `if` in the input expression, which would be scoped inside the `let-syntax`. Since they would not refer to the same binding, they would not be `free-identifier=?`, and a syntax violation would result.

The conventional use of underscore (_) helps the programmer avoid situations like these in which the wrong identifier is matched against or inserted by accident.

It is a syntax violation to generate a reference to an identifier that is not present within the context of an input form, which can happen if the "closest enclosing lexical binding" for an identifier inserted into the output of a transformer does not also enclose the input form. For example,

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

should result in a syntax violation with a message to the effect that `/` is referenced in an invalid context, since the occurrence of `/` in the output of `divide` is a reference to the variable `/` bound by the `let` expression within the transformer.

The next example defines a `define-integrable` form that is similar to `define` for procedure definitions except that it causes the code for the procedure to be *integrated*, or inserted, wherever a direct call to the procedure is found.

```
(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
                 (... ...))]))))]))
```

The form (`define-integrable` *`name lambda-expression`*) expands into a pair of definitions: a syntax definition of *`name`* and a variable definition of `xname`. The transformer for *`name`* converts apparent calls to *`name`* into direct calls to *`lambda-expression`*. Since the resulting forms are merely direct `lambda` applications (the equivalent of `let` expressions), the actual parameters are evaluated exactly once and before evaluation of the procedure's body, as required. All other references to *`name`* are replaced with references to `xname`. The definition of `xname` binds it to the value of *`lambda-expression`*. This allows the procedure to be used as a first-class value. The `define-integrable` transformer does nothing special to maintain lexical scoping within the `lambda` expression or at the call site, since lexical scoping is maintained automatically by the expander. Also, because `xname` is introduced by the transformer, the binding for `xname` is not visible anywhere except where references to it are introduced by the the transformer for *`name`*.

The above definition of `define-integrable` does not work for recursive procedures, since a recursive call would cause an indefinite number of expansion steps, likely resulting in exhaustion of memory at expansion time. A solution to this problem for directly recursive procedures is to wrap each occurrence of the `lambda` expression with a `let-syntax` binding that unconditionally expands *name* to `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 (... ...))]))))]))
```

This problem can be solved for mutually recursive procedures by replacing the `let-syntax` forms with the nonstandard `fluid-let-syntax` form, which is described in the *Chez Scheme User's Guide* [9].

Both definitions of `define-integrable` treat the case where an identifier appears in the first position of a structured expression differently from the case where it appears elsewhere, as does the `pcar` example given in the description for `identifier?`. In other situations, both cases must be treated the same. The form `identifier-syntax` can make doing so more convenient.

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

The following example uses `identifier-syntax`, `datum->syntax`, and local syntax definitions to define a form of *method*, one of the basic building blocks of object-oriented programming (OOP) systems. A `method` expression is similar to a `lambda` expression, except that in addition to the formal parameters and body, a `method` expression also contains a list of instance variables `(ivar ...)`. When a method is invoked, it is always passed an *object* (*instance*), represented as a vector of *fields* corresponding to the instance variables, and zero or more additional arguments. Within the method body, the object is bound implicitly to the identifier `self` and the additional arguments are bound to the formal parameters. The fields of the object may be accessed or altered within the method body via instance variable references or assignments.

```
(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! (datum->syntax #'k 'set!)])
         #'(lambda (self . formals)
             (let-syntax ([ivar (identifier-syntax
                                  (vector-ref self index))]
                          ...)
               (let-syntax ([set!
                              (syntax-rules (ivar ...)
                                [(_ ivar e) (vector-
set! self index e)]
                                ...
                                [(_ x e) (set! x e)])])
                 b1 b2 ...))))])))
```

Local bindings for `ivar ...` and for `set!` make the fields of the object appear to be ordinary variables, with references and assignments translated into calls to `vector-ref` and `vector-set!`. `datum->syntax` is used to make the introduced bindings of `self` and `set!` visible in the method body. Nested `let-syntax` expressions are needed so that the identifiers `ivar ...` serving as auxiliary keywords for the local version of `set!` are scoped properly.

By using the general form of `identifier-syntax` to handle `set!` forms more directly, we can simplify the definition of `method`.

```
(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 ...)))])))
```

The examples below demonstrate simple uses of `method`.

```
(let ([m (method (a) (x) (list a x self))])
  (m #(1) 2)) ⇒ (1 2 #(1))

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

In a complete OOP system based on `method`, the instance variables `ivar ...` would likely be drawn from class declarations, not listed explicitly in the `method` forms, although the same techniques would be used to make instance variables appear as ordinary variables within method bodies.

The final example of this section defines a simple structure definition facility that represents structures as vectors with named fields. Structures are defined with `define-structure`, which takes the form

```
(define-structure name field ...)
```

where *`name`* names the structure and *`field`* `...` names its fields. `define-structure` expands into a series of generated definitions: a constructor `make-`*`name`*, a type predicate *`name`*`?`, and one accessor *`name-field`* and setter `set-`*`name-field`*`!` per field name.

```
(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))))
                   args))))))
    (syntax-case x ()
      [(_ name field ...)
       (with-syntax ([constructor (gen-id #'name "make-" #'name)]
                     [predicate (gen-id #'name #'name "?")]
                     [(access ...)
                      (map (lambda (x) (gen-id x #'name "-" x))
                           #'(field ...))]
                     [(assign ...)
                      (map (lambda (x)
                             (gen-id x "set-" #'name "-" x "!"))
                           #'(field ...))]
                     [structure-length (+ (length #'(field ...)) 1)]
                     [(index ...)
                      (let f ([i 1] [ids #'(field ...)])
                        (if (null? ids)
                                   '()
                                   (cons i (f (+ i 1) (cdr ids))))))])
         #'(begin
             (define constructor
               (lambda (field ...)
                 (vector 'name field ...)))
             (define predicate
               (lambda (x)
                 (and (vector? x)
                      (= (vector-length x) structure-length)
                      (eq? (vector-ref x 0) 'name))))
             (define access
               (lambda (x)
                 (vector-ref x index)))
             ...
             (define assign
               (lambda (x update)
                 (vector-set! x index update)))
             ...))])))
```

The constructor accepts as many arguments as there are fields in the structure and creates a vector whose first element is the symbol *`name`* and whose remaining elements are the argument values. The type predicate returns true if its argument is a vector of the expected length whose first element is *`name`*.

Since a `define-structure` form expands into a `begin` containing definitions, it is itself a definition and can be used wherever definitions are valid.

The generated identifiers are created with `datum->syntax` to allow the identifiers to be visible where the `define-structure` form appears.

The examples below demonstrate the use of `define-structure`.

```
(define-structure tree left right)
(define 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
to order this book / about this book

http://www.scheme.com