Syntactic Abstraction: The syntax-case expander*

R. Kent Dybvig
Indiana University

June, 2007

When writing computer programs, certain patterns arise over and over again. For example, programs must often loop through the elements of arrays, increment or decrement the values of variables, and perform multi-way conditionals based on numeric or character values. Programming language designers typically acknowledge this fact by including special-purpose syntactic constructs that handle the most common patterns. C, for instance, provides multiple looping constructs, multiple conditional constructs, and multiple constructs for incrementing or otherwise updating the value of a variable [9].

Some patterns are less common but can occur frequently in a certain class of programs or perhaps just within a single program. These patterns might not even be anticipated by a language's designers, who in any case would typically choose not to incorporate syntactic constructs to handle such patterns in the language core. Yet, recognizing that such patterns do arise and that special-purpose syntactic constructs can make programs both simpler and easier to read, language designers sometimes include a mechanism for *syntactic abstraction*, such as C's preprocessor macros or Common Lisp [11] macros. When such facilities are not present or are inadequate for a specific purpose, an external tool, like the m4 [8] macro expander, might be brought to bear.

Syntactic abstraction facilities differ in several significant ways. C's preprocessor macros are essentially token-based, allowing the replacement of a macro call with a sequence of tokens with text from the macro call substituted for the macro's formal parameters, if any. Lisp macros are expression-based, allowing the replacement of a single expression with another expression, computed in Lisp itself and based on the subforms of the macro call, if any.

In both cases, identifiers appearing within a macro-call subform are scoped where they appear in the output, rather than where they appear in the input, possibly leading to unintended *capture* of a variable reference by a variable binding. For example, consider the simple transformation of Scheme's `or` form [7] into `let` and `if` below. (Readers unfamiliar with Scheme might want to read

*This document appeared as a chapter of *Beautiful Code: Leading Programmers Explain How They Think*, edited by Andy Oram and Greg Wilson and published by O'Reilly and Associates in June 2007 the first few chapters of *The Scheme Programming Language, 3rd edition* [4], which is available online at http://www.scheme.com/tspl3/.)

(or e_1 e_2) $\rightarrow$ (let ([t e_1]) (if t t e_2))

An `or` form must return the value of its first subform, if it evaluates to a true (any non-false) value; the `let` expression is used to name this value so that it is not computed twice.

The transformation above works fine in most cases, but it breaks down if the identifier `t` appears *free* in e_2 (i.e., outside of any binding for `t` in e_2), as in the expression below.

```
(let ([t #t]) (or #f t))
```

This should evaluate to the true value `#t`. With the transformation of `or` as specified above, however, the expression expands to

```
(let ([t #t])
  (let ([t #f])
    (if t t t)))
```

which evaluates to the false value `#f`.

Once seen, this problem is easily addressed by using a generated identifier for the introduced binding, i.e.:

(or e_1 e_2) $\rightarrow$ (let ([g e_1]) (if g g e_2))

where g is a generated (fresh) identifier.

As Kohlbecker, Friedman, Felleisen, and Duba observe in their seminal paper on hygienic macro expansion [10], variable capture problems like this are insidious, since a transformation might work correctly for a large body of code only to fail some time later in a way that might be difficult to debug.

While unintended captures caused by introduced identifier *bindings* can always be solved by using generated identifiers, no such simple solution is available for introduced identifier *references*, which might be captured by bindings in the context of the macro call. In the following expression, `if` is lexically bound in the context of an `or` expression.

```
(let ([if (lambda (x y z) "oops")]) (or #f #f))
```

With the second transformation for `or` above, this expression expands into:

```
(let ([if (lambda (x y z) "oops")])
  (let ([g #f])
    (if g g #f)))
```

where g is a fresh identifier. The value of the expression should be `#f` but will actually be `"oops"`, as the locally bound procedure `if` is used in place of the original `if` conditional syntax.

Limiting the language by reserving the names of keywords such as `let` and `if` would solve this problem for keywords, but it would not solve the problem generally; the same situation can arise with the introduced reference to the user-defined variable `add1` in following transformation of `increment`:

(increment x) $\rightarrow$ (set! x (add1 x))

Kohlbecker, et al. invented the concept of *hygienic macro expansion* to solve both kinds of capturing problems, borrowing the term "hygiene" from Barendregt [1]. Barendregt's *hygiene condition* for the λ-calculus holds that the free variables of one expression substituted into another are assumed not to be captured by bindings in the other, unless such capture is explicitly required. Kohlbecker, et al. adapted this into the following *hygiene condition for macro expansion*:

> "Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated at the same transcription step."

In practice, this requirement forces the expander to rename identifiers as necessary to avoid unintended captures. For example, with the original `or` transformation,

(or e_1 e_2) $\rightarrow$ (let ([t e_1]) (if t t e_2))

the expression:

```
(let ([t #t]) (or #f t))
```

expands into the equivalent of:

```
(let ([t0 #t])
  (let ([t1 #f])
    (if t1 t1 t0)))
```

which properly evaluates to `#t`. Similarly, the expression:

```
(let ([if (lambda (x y z) "oops")]) (or #f #f))
```

expands into the equivalent of:

```
(let ([if0 (lambda (x y z) "oops")])
  (let ([t #f])
    (if t t #f)))
```

which properly evaluates to `#f`.

In essence, hygienic macro expansion implements lexical scoping with respect to the source code, whereas unhygienic expansion implements lexical scoping with respect to the code after expansion.

Hygienic expansion can preserve lexical scope only to the extent that the scope is preserved by the transformations it is told to perform. A transformation can still produce code that apparently violates lexical scoping. This can be illustrated with the following (incorrect) transformation of `let`:

(let ((x e)) *body*) $\rightarrow$ (letrec ((x e)) *body*)

The expression e should appear outside the scope of the binding of the variable x, but in the output it appears inside, due to the semantics of `letrec`.

The hygienic macro expansion algorithm (KFFD) described by Kohlbecker, et al. is both clever and elegant. It works by adding a time stamp to each variable introduced by a macro, then uses the timestamps to distinguish like-named variables as it renames lexically bound variables. KFFD has some shortcomings that prevent its direct use in practice, however. The most serious are a lack of support for local macro bindings and quadratic overhead resulting from the complete rewrite of each expression as time stamping and as renaming are performed.

These shortcomings are addressed by the `syntax-rules` system, developed by Clinger, Dybvig, Hieb, and Rees for the Revised^4 Report on Scheme [2]. The simple pattern-based nature of the `syntax-case` system permits it to be implemented easily and efficiently [3]. Unfortunately, it also limits the utility of the mechanism, so that many useful syntactic abstractions are either difficult or impossible to write.

The `syntax-case` system was developed to address the shortcomings of the original algorithm without the limitations of `syntax-rules` [6]. The system supports local macro bindings and operates with constant overhead, yet allows macros to use the full expressive power of the Scheme language. It is upwardly compatible with `syntax-rules`, which can be expressed as a simple macro in terms of `syntax-case`, and it permits the same pattern language to be used even for "low level" macros for which `syntax-rules` cannot be used. It also provides a mechanism for allowing *intended* captures, i.e., allowing hygiene to be "bent" or "broken" in a selective and straightforward manner. In addition, it handles several practical aspects of expansion that must be addressed in a real implementation, such as internal definitions and tracking of source information through macro expansion.

This all comes at a price in terms of the complexity of the expansion algorithm and the size of the code required to implement it. A study of a complete implementation is therefore beyond the scope of this presentation. Instead, we investigate a simplified version of the expander that illustrates the underlying algorithm and the most important aspects of its implementation.

1 Brief introduction to syntax-case

We proceed with a few brief `syntax-case` examples, adapted from the *Chez Scheme Version 7 User's Guide* [5]. Additional examples and a more detailed description of `syntax-case` are given in that document and in *The Scheme Programming Language, 3rd edition* [4].

The definition of `or` below illustrates the form of a `syntax-case` macro definition.

```
(define-syntax or
  (lambda (x)
    (syntax-case x ()
      [(_ e1 e2)
       (syntax (let ([t e1]) (if t t e2)))])))
```

The `define-syntax` form creates a keyword binding, associating a keyword (in this case, `or`), with a transformation procedure, or *transformer*, obtained by evaluating, at expansion time, the `lambda` expression on the right-hand side of the `define-syntax` form. The `syntax-case` form is used to parse the input, and the `syntax` form is used to construct the output via straightforward pattern matching. The *pattern* `(_ e1 e2)` specifies the shape of the input, with the underscore (`_`) used to mark where the keyword `or` appears and the pattern variables `e1` and `e2` bound to the first and second subforms. The *template* `(let ([t e1]) (if t t e2))` specifies the output, with `e1` and `e2` inserted from the input.

The form `(syntax` *template*`)` can be abbreviated `#'`*template*, so the definition above can be rewritten as follows.

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

Macros can also be bound within a single expression via `letrec-syntax`.

```
(letrec-syntax ([or (lambda (x)
                      (syntax-case x ()
                        [(_ e1 e2)
                         #'(let ([t e1]) (if t t e2))]))])
  (or a b))
```

Macros can be recursive, i.e., expand into occurrences of themselves, as illustrated by the following version of `or` that handles an arbitrary number of subforms. Multiple `syntax-case` clauses are required to handle the two base cases and the recursion case.

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

An input or output form followed by an ellipsis in the `syntax-case` pattern language matches or produces zero or more forms.

Hygiene is ensured for the definitions of `or` above so that the introduced binding for `t` and the introduced references to `let`, `if`, and even `or` are scoped properly. If we want to bend or break hygiene, we do so with the procedure `datum->syntax`, which produces a syntax object from an arbitrary s-expression. The identifiers within the s-expression are treated as if they appeared in the original source where the first argument, the *template identifier*, appeared.

We can use this fact to create a simple method syntax that implicitly binds the name `this` to the first (object) argument.

```
(define-syntax method
  (lambda (x)
    (syntax-case x ()
      [(k (x ...) e1 e2 ...)
       (with-syntax ([this (datum->syntax #'k 'this)])
         #'(lambda (this x ...) e1 e2 ...))])))
```

By using the keyword `k`, extracted from the input, as the template variable, the variable `this` is treated as if it were present in the `method` form, so that:

```
(method (a) (f this a))
```

is treated as the equivalent of

```
(lambda (this a) (f this a))
```

with no renaming to prevent the introduced binding of `this` from capturing the source-code reference.

The `with-syntax` form used in the definition of `method` creates local pattern-variable bindings. It is a simple macro written in terms of `syntax-case`.

```
(define-syntax with-syntax
  (lambda (x)
    (syntax-case x ()
      [(_ ((p e0) ...) e1 e2 ...)
       #'(syntax-case (list e0 ...) ()
           [(p ...) (begin e1 e2 ...)])])))
```

The `datum->syntax` procedure can be used for arbitrary s-expressions, as illustrated 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-input-port p) '())
                (cons (datum->syntax k x) (f (read p))))))))
    (syntax-case x ()
      [(k filename)
       (let ([fn (syntax->datum #'filename)])
         (with-syntax ([(e ...) (read-file fn #'k)])
           #'(begin e ...)))])))
```

The form `(include "filename")` has the effect of treating the forms within the named file as if they were present in the source code in place of the `include` form. In addition to using `datum->syntax`, `include` also uses its inverse operator, `syntax->datum`, to convert the filename subform into a string it can pass to `open-input-file`.

2 Algorithm overview

The `syntax-case` expansion algorithm is essentially a lazy variant of the KFFD algorithm that operates on an abstract representation of the input expression rather than on the traditional s-expression representation. The abstract representation encapsulates both a representation of an input form and a *wrap* that enables the algorithm to determine the scope of all identifiers within the form. The wrap consists of *marks* and *substitutions*. Marks are like KFFD timestamps and are added to the portions of a macro's output that are introduced by the macro. Substitutions map identifiers to bindings with the help of a compile-time environment. Substitutions are created whenever a binding form, like `lambda`, is encountered, and they are added to the wraps of the syntax objects representing the forms within the scope of the binding form's bindings. A substitution applies to an identifier only if the identifier has the same name and marks as the substituted identifier.

Expansion operates in a recursive, top-down fashion. As the expander encounters a macro call, it invokes the associated transformer on the form, marking it first with a fresh mark, then marking it again with the same mark. Like marks cancel, so only the introduced portions of the macro's output, i.e., those portions not simply copied from the input to the output, remain marked. When a core form is encountered, a core form in the output language of the expander (in our case, the traditional s-expression representation) is produced, with any subforms recursively expanded as necessary. Variable references are replaced by generated names via the substitution mechanism.

3 Representations

The most important aspect of the `syntax-case` mechanism it its abstract representation of program source code as *syntax objects*. As described above, a syntax object encapsulates not only a representation of the program source code but also a *wrap* that provides sufficient information about the identifiers contained within the code to implement hygiene.

```
(define-record syntax-object (expr wrap))
```

The `define-record` form creates a new type of value with the specified name (in this case, `syntax-object`) and fields (in this case, `expr` and `wrap`), along with a set of procedures to manipulate it, in this case `make-syntax-object`, which returns a new syntax object with the `expr` and `wrap` fields initialized to the values of its arguments, `syntax-object?`, which returns true iff its argument is a syntax object, `syntax-object-expr`, which returns the value of the `expr` field of a syntax-objects, and `syntax-object-wrap`, which return the value of the `wrap` field of a syntax object.

A complete implementation of `syntax-case` might also include, within each syntax object, source information to be tracked through the expansion process.

Each wrap consists of a list of *marks* and *substitutions*. Marks are distinguished by their object identity and do not require any fields.

```
(define-record mark ())
```

A substitution maps a symbolic name and list of marks to a *label*.

```
(define-record subst (sym mark* label))
```

Labels, like marks, are distinguished by their identity require no fields.

```
(define-record label ())
```

The expand-time environment maintained by the expander maps labels to *bindings*. The environment is structured as a traditional *association list*, i.e., a list of pairs, each car of which contains a label and each cdr of which contains a binding. Bindings consist of a type (represented as a symbol) and a value.

```
(define-record binding (type value))
```

The type identifies the nature of the binding, e.g., `macro` for keyword bindings and `lexical` for lexical variable bindings. The value is any additional information required to specify the binding, such as the transformation procedure when the binding is a keyword binding.

4 Producing expander output

The expander's output is a simple s-expression in the core language and is thus constructed for the most part using Scheme's `quasiquote` syntax for creating list structure. For example, a `lambda` expression can be created with formal parameter *var* and body *body* as follows:

```
`(lambda (,var) ,body)
```

The expander does need to create fresh names, however, and does so via the `gen-var` helper, which makes use of the Scheme primitives for converting strings to symbols and visa versa, along with a local sequence counter.

```
(define gen-var
  (let ([n 0])
    (lambda (id)
      (set! n (+ n 1))
      (let ([name (syntax-object-expr id)])
        (string->symbol (format "~s.~s" name n))))))
```

5 Stripping syntax objects

Whenever a `quote` form is encountered in the input, the expander must return a representation of the constant contents appearing within the `quote` form. To do this, it must strip away any embedded syntax objects and wraps, using the strip procedure, which traverses the syntax-object and list structure of its input and recreates an s-expression representation of its input.

```
(define strip
  (lambda (x)
    (cond
      [(syntax-object? x)
       (if (top-marked? (syntax-object-wrap x))
           (syntax-object-expr x)
           (strip (syntax-object-expr x)))]
      [(pair? x)
       (let ([a (strip (car x))] [d (strip (cdr x))])
         (if (and (eq? a (car x)) (eq? d (cdr x)))
             x
             (cons a d)))]
      [else x])))
```

Traversal terminates along any branch of the input expression when something other than a syntax object or pair is found, i.e., when a symbol or immediate value is found. It also terminates when a syntax object is found to be "top marked," i.e., it's wrap contains a unique *top mark*.

```
(define top-mark (make-mark))

(define top-marked?
  (lambda (wrap)
    (and (not (null? wrap))
         (or (eq? (car wrap) top-mark)
             (top-marked? (cdr wrap))))))
```

When the expander creates a syntax object representing the original input, it uses a wrap that contains the top mark at its base, specifically to allow the stripping code detect when it has reached the syntax-object base and need not traverse the object further. This feature prevents the expander from traversing constants unnecessarily so that it can easily preserve shared and cyclic structure and handle quoted syntax objects in the input.

6 Syntax errors

The expander reports syntax errors via syntax-error, which is defined below.

```
(define syntax-error
  (lambda (object message)
    (error #f "~a ~s" message (strip object))))
```

If the implementation attaches source information to syntax objects, this source information can be used to construct an error message that incorporates the source line and character position.

7 Structural predicates

The nonatomic structure of a syntax object is always determined with the patterns of a `syntax-case` form. The predicate `identifier?` determines whether a syntax object represents an identifier.

```
(define identifier?
  (lambda (x)
    (and (syntax-object? x)
         (symbol? (syntax-object-expr x)))))
```

Similarly, the predicate `self-evaluating?` is used, after stripping a syntax object, to determine if it represents a constant.

```
(define self-evaluating?
  (lambda (x)
    (or (boolean? x) (number? x) (string? x) (char? x))))
```

8 Creating wraps

A mark or substitution is added to a syntax object by extending the wrap.

```
(define add-mark
  (lambda (mark x)
    (extend-wrap (list mark) x)))

(define add-subst
  (lambda (id label x)
    (extend-wrap
      (list (make-subst
              (syntax-object-expr id)
              (wrap-marks (syntax-object-wrap id))
              label))
      x)))
```

If the syntax object is only partially wrapped, the wrap is extended simply by creating a syntax object encapsulating the partially wrapped structure. Otherwise, the syntax object is rebuilt with the new wrap joined to the old wrap.

```
(define extend-wrap
  (lambda (wrap x)
    (if (syntax-object? x)
        (make-syntax-object
          (syntax-object-expr x)
          (join-wraps wrap (syntax-object-wrap x)))
        (make-syntax-object x wrap))))
```

Joining two wraps is almost as simple as appending the lists of marks. The only complication is that two like marks must cancel when they meet, to support the anti marking of the input and subsequent marking of the output (Section 11).

```
(define join-wraps
  (lambda (wrap1 wrap2)
    (cond
      [(null? wrap1) wrap2]
      [(null? wrap2) wrap1]
      [else
       (let f ([w (car wrap1)] [w* (cdr wrap1)])
         (if (null? w*)
             (if (and (mark? w) (eq? (car wrap2) w))
                 (cdr wrap2)
                 (cons w wrap2))
             (cons w (f (car w*) (cdr w*)))))])))
```

9 Manipulating environments

Environments map labels to bindings and are represented as association lists. Extending an environment therefore involves adding to the environment a pair associating a label with a binding.

```
(define extend-env
  (lambda (label binding env)
    (cons (cons label binding) env)))
```

10 Identifier resolution

Determining the binding associated with an identifier is a two step process. The first step is to determine the label associated with the identifier in the identifier's wrap, and the second is look the label up in the current environment.

```
(define id-binding
  (lambda (id r)
    (label-binding id (id-label id) r)))
```

The marks and substitutions that appear in an identifier's wrap determine the associated label, if any. Substitutions map names and lists of marks to labels. Any substitution whose name is not the name of the identifier is ignored, as is any whose marks do not match. The names are symbols and are thus compared using the pointer equivalence operator, `eq?`.

The set of marks that are relevant are those that were layered onto the wrap before the substitution. Thus the set of marks to which a substitution's marks are compared changes as the search through the wrap proceeds. The starting set of marks is the entire set that appear in the wrap. Each time a mark is encountered during the search for a matching substitution in the wrap, the first mark in the list is removed.

```
(define id-label
  (lambda (id)
    (let ([sym (syntax-object-expr id)]
          [wrap (syntax-object-wrap id)])
      (let search ([wrap wrap] [mark* (wrap-marks wrap)])
        (if (null? wrap)
            (syntax-error id "undefined identifier")
            (let ([w0 (car wrap)])
              (if (mark? w0)
                  (search (cdr wrap) (cdr mark*))
                  (if (and (eq? (subst-sym w0) sym)
                           (same-marks? (subst-mark* w0) mark*))
                      (subst-label w0)
                      (search (cdr wrap) mark*)))))))))
```

If no matching substitution exists in the wrap, the identifier is undefined and a syntax error is signaled. It would be possible instead to treat all such identifier references as global variable references.

The `id-label` procedure obtains the starting list of marks via `wrap-marks` and uses the `same-marks?` predicate to compare lists of marks.

```
(define wrap-marks
  (lambda (wrap)
    (if (null? wrap)
        '()
        (let ([w0 (car wrap)])
          (if (mark? w0)
              (cons w0 (wrap-marks (cdr wrap)))
              (wrap-marks (cdr wrap)))))))

(define same-marks?
  (lambda (m1* m2*)
    (if (null? m1*)
        (null? m2*)
        (and (not (null? m2*))
             (eq? (car m1*) (car m2*))
             (same-marks? (cdr m1*) (cdr m2*))))))
```

Once a label has been found, `id-binding` is used to find the associated binding, if any, using the `assq` procedure for performing association-list lookups. If an association is found, the binding in the cdr of the association is returned.

```
(define label-binding
  (lambda (id label r)
    (let ([a (assq label r)])
      (if a
          (cdr a)
          (syntax-error id "displaced lexical")))))
```

If no binding is found, the identifier is a "displaced lexical." This occurs when a macro improperly inserts into its output a reference to an identifier that is not visible in the context of the macro output.

11 The expander

With the mechanisms for handling wraps and environments in place, the expander is straightforward. The expression expander, `exp`, handles macro calls, lexical variable references, applications, core forms, and constants. Macro calls come in two forms: singleton macro-keyword references and structured forms with a macro keyword in the first position. The `exp` procedure takes three arguments: a syntax object x, a run-time environment r, and a meta environment mr. The run-time environment is used to process ordinary expressions whose code will appear in the expander's output, while the meta environment is used to process transformer expressions, e.g., on the right-hand sides of `letrec-syntax` bindings, which are evaluated and used at expansion time. The difference between the run-time and meta environments is that the meta environment does not contain lexical variable bindings, since these bindings are not available when the transformer is evaluated and used.

```
(define exp
  (lambda (x r mr)
    (syntax-case x ()
      [id
       (identifier? #'id)
       (let ([b (id-binding #'id r)])
         (case (binding-type b)
           [(macro) (exp (exp-macro (binding-value b) x) r mr)]
           [(lexical) (binding-value b)]
           [else (syntax-error x "invalid syntax")]))]
      [(e0 e1 ...)
       (identifier? #'e0)
       (let ([b (id-binding #'e0 r)])
         (case (binding-type b)
           [(macro) (exp (exp-macro (binding-value b) x) r mr)]
           [(lexical)
            `(,(binding-value b) ,@(exp-exprs #'(e1 ...) r mr))]
           [(core) (exp-core (binding-value b) x r mr)]
           [else (syntax-error x "invalid syntax")]))]
      [(e0 e1 ...)
       `(,(exp #'e0 r mr) ,@(exp-exprs #'(e1 ...) r mr))]
      [_
       (let ([d (strip x)])
         (if (self-evaluating? d)
             d
             (syntax-error x "invalid syntax")))])))
```

Macro calls are handled by `exp-macro` (below), then re-expanded. Lexical variable references are rewritten into the binding value, which is a generated variable name. Applications are rewritten into lists as in the traditional s-expression syntax for Lisp and Scheme, with the subforms expanded recursively. Core forms are handled by `exp-core` (below); any recursion back to the expression expander is performed explicitly by the core transformer. A constant is rewritten into the constant value, stripped of its syntax wrapper.

The expander uses `syntax-case` and `syntax` (in its abbreviated form, i.e., `#'`*template*) to parse and refer to the input or portions thereof. Since the expander is also charged with implementing `syntax-case`, this seems like a paradox of sorts, but in fact is handled by bootstrapping one version of the expander using a previous version. The expander would be much more tedious to write if `syntax-case` and `syntax` were not used.

The `exp-macro` procedure applies the transformation procedure (the value part of the macro binding) and applies it to the entire macro form, which can either be a single macro keyword or a structured expression with the macro keyword at its head. The `exp-macro` procedure first adds a fresh mark to the wrap of the input form, then applies the same mark to the wrap of the output form. The first mark serves as an *anti-mark* that cancels out the second mark, so the net effect is that the mark adheres only to the portions of the output that were introduced by the transformer, thus uniquely identifying the portions of the code introduced at this transcription step.

```
(define exp-macro
  (lambda (p x)
    (let ([m (make-mark)])
      (add-mark m (p (add-mark m x))))))
```

The `exp-core` procedure simply applies the given core transformer (the value part of the core binding) to the input form.

```
(define exp-core
  (lambda (p x r mr)
    (p x r mr)))
```

The `exp-exprs` procedure used to process application subforms simply maps the expander over the forms.

```
(define exp-exprs
  (lambda (x* r mr)
    (map (lambda (x) (exp x r mr)) x*)))
```

12 Core transformers

Transformers for several representative core forms (`quote`, `if`, `lambda`, `let`, and `letrec-syntax`) are described here. Adding transformers for other core forms, like `letrec` and `let-syntax`, is straightforward.

The `exp-quote` procedure produces an s-expression representing a `quote` form, with the data value stripped of its syntax wrap.

```
(define exp-quote
  (lambda (x r mr)
    (syntax-case x ()
      [(_ d) `(quote ,(strip #'d))])))
```

The `exp-if` procedure produces an s-expression representation of an `if` form, with the subforms recursively expanded.

```
(define exp-if
  (lambda (x r mr)
    (syntax-case x ()
      [(_ e1 e2 e3)
       `(if ,(exp #'e1 r mr)
            ,(exp #'e2 r mr)
            ,(exp #'e3 r mr))])))
```

The `exp-lambda` procedure handles `lambda` expressions with only a single formal parameter and only a single body expression. Extending it to handle multiple parameters is straightforward. It is less straightforward to handle arbitrary `lambda` bodies, including internal definitions, but support for internal definitions is beyond the scope of this presentation.

When the s-expression representation of a `lambda` expression is produced, a generated variable name is created for the formal parameter. A substitution mapping the identifier to a fresh label is added to the wrap on the body, and the environment is extended with an association from the label to a `lexical` binding whose value is the generated variable, during the recursive processing of the body.

```
(define exp-lambda
  (lambda (x r mr)
    (syntax-case x ()
      [(_ (var) body)
       (let ([label (make-label)] [new-var (gen-var #'var)])
         `(lambda (,new-var)
            ,(exp (add-subst #'var label #'body)
                  (extend-env label
                    (make-binding 'lexical new-var)
                    r)
                  mr)))])))
```

The meta environment is not extended, since the meta environment should not include lexical variable bindings.

The `exp-let` procedure that transforms single-binding `let` forms is similar to the transformer for `lambda`, but slightly more involved.

```
(define exp-let
  (lambda (x r mr)
```

```
(syntax-case x ()
  [(_ ([var expr]) body)
   (let ([label (make-label)] [new-var (gen-var #'var)])
     `(let ([,new-var ,(exp #'expr r mr)])
        ,(exp (add-subst #'var label #'body)
              (extend-env label
                (make-binding 'lexical new-var)
                r)
              mr)))])))
```

The body is in the scope of the binding created by `let`, so it is expanded with the extended wrap and environment. The right-hand-side expression, `expr`, is not within the scope, so it is expanded with the original wrap and environment.

The `exp-letrec-syntax` procedure handles single-binding `letrec-syntax` forms. As with `lambda` and `let`, a substitution mapping the bound identifier, in this case a keyword rather than a variable, to a fresh label is added to the wrap on the body, and an association from the label to a binding is added to the environment while the body is recursively processed. The binding is a `macro` binding rather than a `lexical` binding, and the binding value is the result of recursively expanding and evaluating the right-hand-side expression of the `letrec-syntax` form. In contrast with `let`, the right-hand-side expression is also wrapped with a substitution from the keyword to the label and expanded with the extended environment; this allows the macro to be recursive. This would not be done if the form were a `let-syntax` form instead of a `letrec-syntax` form. The output produced by expanding a `letrec-syntax` form consists only of the output of the call to the expander on the body of the form.

```
(define exp-letrec-syntax
  (lambda (x r mr)
    (syntax-case x ()
      [(_ ((kwd expr)) body)
       (let ([label (make-label)])
         (let ([b (make-binding 'macro
                    (eval (exp (add-subst #'kwd label #'expr)
                               mr mr)))])
           (exp (add-subst #'kwd label #'body)
                (extend-env label b r)
                (extend-env label b mr))))])))
```

Both the run-time and meta environments are extended in this case, since transformers are available both in run-time and transformer code.

13 Parsing and constructing syntax objects

Macros are written in a pattern-matching style using `syntax-case` to match and take apart the input and `syntax` to reconstruct the output. Implementation of the pattern matching and reconstruction is outside the scope of this presentation, but the following low-level operators can be used as the basis for the implementation. The `syntax-case` form can be built from the following set of three operators that treat syntax objects as abstract s-expressions.

```
(define syntax-pair?
  (lambda (x)
    (pair? (syntax-object-expr x))))

(define syntax-car
  (lambda (x)
    (extend-wrap
      (syntax-object-wrap x)
      (car (syntax-object-expr x)))))

(define syntax-cdr
  (lambda (x)
    (extend-wrap
      (syntax-object-wrap x)
      (cdr (syntax-object-expr x)))))
```

The definitions of `syntax-car` and `syntax-cdr` employ the `extend-wrap` helper defined in Section 8 to push the wrap on the pair onto the car and cdr.

Similarly, `syntax` can be built from the following more basic version of `syntax` that handles constant input but not pattern variables and ellipses.

```
(define exp-syntax
  (lambda (x r mr)
    (syntax-case x ()
      [(_ t) `(quote ,#'t)])))
```

In essence, the simplified version of `syntax` is just like `quote` except that `syntax` does not strip the encapsulated value but rather leaves the syntax wrappers intact.

14 Comparing identifiers

Identifiers are compared based on their intended use. They can be compared as symbols by using the pointer equivalence operator, `eq?`, on the symbolic names of the identifiers. They can also be compared according to their intended use as free or bound identifiers in the output of a macro.

Two identifiers are equivalent by `free-identifier=?` if they would resolve to the same binding if introduced into the output of a macro outside of any binding introduced by the macro. This is accomplished by comparing the labels to which the identifiers resolve.

```
(define free-identifier=?
  (lambda (x y)
    (eq? (id-label x) (id-label y))))
```

The `free-identifier=?` predicate is often used to check for auxiliary keywords, like `else` in `cond` or `case`.

Two identifiers are equivalent by `bound-identifier=?` if a reference to one would be captured by a enclosing binding for another. This is accomplished by comparing the names and marks of the two identifiers.

```
(define bound-identifier=?
  (lambda (x y)
    (and (eq? (syntax-object-expr x) (syntax-object-expr y))
         (same-marks?
           (wrap-marks (syntax-object-wrap x))
           (wrap-marks (syntax-object-wrap y))))))
```

The `bound-identifier=?` predicate is often used to check for duplicate identifier errors in a binding form, such as `lambda` or `let`.

15 Conversions

The conversion from s-expression to syntax object performed by `datum->syntax` requires only that the wrap be transferred from the template identifier to the s-expression.

```
(define datum->syntax
  (lambda (template-id x)
    (make-syntax-object x (syntax-object-wrap template-id))))
```

The opposite conversion involves stripping the wrap away from a syntax object, so `syntax->datum` is just `strip`.

```
(define syntax->datum strip)
```

16 Starting expansion

All of the pieces are now in place to expand Scheme expressions containing macros into expressions in the core language. The main expander merely supplies an initial wrap and environment that include names and bindings for the core forms and primitives.

```
(define expand
  (lambda (x)
    (let-values ([(wrap env) (initial-wrap-and-env)])
      (exp (make-syntax-object x wrap) env env))))
```

The initial wrap consists of a set of substitutions mapping each predefined identifier to a fresh label, and the initial environment associates each of these labels with the corresponding binding.

```
(define initial-wrap-and-env
  (lambda ()
    (define id-binding*
      `((quote . ,(make-binding 'core exp-quote))
        (if . ,(make-binding 'core exp-if))
        (lambda . ,(make-binding 'core exp-lambda))
        (let . ,(make-binding 'core exp-let))
        (letrec-syntax . ,(make-binding 'core exp-letrec-syntax))
        (identifier? . ,(make-binding 'lexical 'identifier?))
        (free-identifier=? .
          ,(make-binding 'lexical 'free-identifier=?))
        (bound-identifier=? .
          ,(make-binding 'lexical 'bound-identifier=?))
        (datum->syntax . ,(make-binding 'lexical 'datum->syntax))
        (syntax->datum . ,(make-binding 'lexical 'syntax->datum))
        (syntax-error . ,(make-binding 'lexical 'syntax-error))
        (syntax-pair? . ,(make-binding 'lexical 'syntax-pair?))
        (syntax-car . ,(make-binding 'lexical 'syntax-car))
        (syntax-cdr . ,(make-binding 'lexical 'syntax-cdr))
        (syntax . ,(make-binding 'core exp-syntax))
        (list . ,(make-binding 'core 'list))))
    (let ([label* (map (lambda (x) (make-label)) id-binding*)])
      (values
        `(,@(map (lambda (sym label)
                   (make-subst sym (list top-mark) label))
                 (map car id-binding*)
                 label*)
          ,top-mark)
        (map cons label* (map cdr id-binding*))))))
```

In addition to the entries listed, the initial environment should also include bindings for the built-in syntactic forms we have not implemented, (e.g., `letrec` and `let-syntax`), as well as for all built-in Scheme procedures. It should also include a full version of `syntax` and, in place of `syntax-pair?`, `syntax-car`, and `syntax-cdr`, it should include `syntax-case`.

17 Example

To illustrate the expansion algorithm, we can trace the expansion of the following example from the overview.

```
(let ([t #t]) (or #f t))
```

We assume that `or` has been defined to do the transformation given in the overview, using the equivalent of the following definition of `or` from Section 1.

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

At the outset, the expander is presented with a syntax object whose expression is (let ([t #t]) (or #f t)) and wrap is empty, except for the contents of the initial wrap, which we suppress for brevity.

```
<(let ((t #t)) (or #f t))>
```

We identify syntax objects by enclosing the expression and wrap entries, if any, in angle brackets.

The expander is also presented with the initial environment, which we assume contains a binding for the macro or as well as for the core forms and built-in procedures. Again, we suppress these environment entries for brevity. We also suppress the meta environment, which plays no role here since we are not expanding any transformer expressions.

The expression above is recognized as a core form, because let is present in the initial wrap and environment. The transformer for let recursively expands the right-hand-side expression, #t, in the input environment, yielding #t. It also recursively expands the body with an extended wrap that maps t to a fresh label l1:

```
<(or #f t) [t × () → l1]>
```

Substitutions are shown with enclosing brackets, the name and list of marks separated by the symbol ×, and the label following a right arrow.

The environment is also extended to map the label to a binding of type lexical with fresh name t.1.

```
l1 → lexical(t.1)
```

The or form is recognized as a macro call, so the transformer for or is invoked, producing a new expression to be evaluated in the same environment. The input to the or transformer is marked with a fresh mark m2, and the same mark is added to the output, yielding:

```
<(<let> ((<t> #f))
   (<if> <t> <t> <t m2 [t × () → l1]>))
 m2>
```

The differences between the syntax objects representing the introduced identifier t and the identifier t extracted from the input are crucial in determining how each is renamed when the expander reaches it as described below.

The #f appearing on the let right-hand side should technically be a syntax object with the same wraps as the occurrence of t extracted from the input, but the wrap is unimportant for constants so we treat it as if it were not wrapped for simplicity.

We have another core let expression. In the process of recognizing and parsing the let expression, the mark m2 is pushed onto the subforms:

```
(<let m2> ((<t m2> #f))
  <(<if> <t> <t> <t m2 [t × () → l1]>)
   m2>)
```

The transformer for `let` recursively expands the right-hand-side expression `#f`, yielding `#f`, then recursively expands the body with an extended wrap mapping the introduced `t` with mark `m2` to a fresh label `l2`:

```
<(<if> <t> <t> <t m2 [t × () → l1]>)
 [t × (m2) → l2]
 m2>
```

The environment is also extended to map the label to a binding of type `lexical` with fresh name `t.2`.

```
l2 → lexical(t.2), l1 → lexical(t.1)
```

The resulting expression is recognized as an `if` core form. In the process of recognizing and parsing it, the expander pushes the outer substitution and marks onto the subforms. The mark `m2` already appearing in the wrap for the last occurrence of `t2` cancels the mark `m2` on the outer wrap, leaving that occurrence of `t2` unmarked.

```
(<if [t × (m2) → l2] m2>
   <t [t × (m2) → l2] m2>
   <t [t × (m2) → l2] m2>
   <t [t × (m2) → l2] [t × () → l1]>)
```

The transformer for `if` recursively processes its subforms in the input environment. The first:

```
<t [t × (m2) → l2] m2>
```

is recognized as an identifier reference, since the expression is a symbol (`t`). The substitution appearing in the wrap applies in this case, since the name (`t`) and marks `(m2)` are the same. So the expander looks for `l2` in the environment and finds that it maps to the lexical variable `t.2`. The second subform is the same and so also maps to `t.2`. The third is different, however:

```
<t [t × (m2) → l2] [t × () → l1]>)
```

This identifier lacks the `m2` mark, so the first substitution does not apply, even though the name is the same. The second does apply, because it has the same name and the same set of marks (none, beyond the top-mark from the suppressed initial wrap). The expander thus looks for `l1` in the environment and finds that it maps to the lexical variable `t.1`.

On the way out, the `if` expression is reconstructed as:

```
(if t.2 t.2 t.1)
```

the inner `let` expression is reconstructed as:

```
(let ([t.2 #f]) (if t.2 t.2 t.1))
```

and the outer `let` expression is reconstructed as:

```
(let ([t.1 #t]) (let ([t.2 #f]) (if t.2 t.2 t.1)))
```

which is exactly what we want, although the particular choice of fresh names is not important as long as they are distinct.

18 Summary

The simplified expander described here illustrates the basic algorithm that underlies a complete implementation of `syntax-case`, without the complexities of the pattern-matching mechanism, handling of internal definitions, and the additional core forms that are usually handled by an expander. The representation of environments is tailored to the single-binding `lambda`, `let`, and `letrec-syntax` forms implemented by the expander; a more efficient representation that handles groups of bindings would typically be used in practice. While these additional features are not trivial to add, they are conceptually independent of the expansion algorithm.

The `syntax-case` expander extends the KFFD hygienic macro-expansion algorithm with support for local syntax bindings and controlled capture, among other things, and also eliminates the quadratic expansion overhead of the KFFD algorithm. The KFFD algorithm is simple and elegant, and an expander based on it could certainly be a beautiful piece of code. The `syntax-case` expander, on the other hand, is of necessity considerably more complex. It is not, however, any less beautiful, for there can still be beauty in complex software as long as it is well structured and does what it is designed to do.

References

[1] H. P. Barendregt. Introduction to the lambda calculus. *Nieuw Archief voor Wisenkunde*, 4(2):337–372, 1984.

[2] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[3] William Clinger and Jonathan Rees. Macros that work. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 155–162, January 1991.

[4] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, third edition, 2003.

[5] R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2005.

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

[7] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The M4 Macro Processor*, 1979.

[9] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

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

[11] Guy L. Steele Jr. *Common Lisp, the Language*. Digital Press, second edition, 1990.