LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1993
© 1993 Kluwer Academic Publishers – Manufactured in The Netherlands

Syntactic Abstraction in Scheme*

R. KENT DYBVIG (*dyb@cs.indiana.edu*)

ROBERT HIEB†

CARL BRUGGEMAN (*bruggema@cs.indiana.edu*)

Indiana University Computer Science Department
Bloomington, IN 47405

(Received: August, 1992)
(Revised: March, 1993)

Keywords: Syntactic Abstraction, Macros, Program Transformation, Hygienic Macros

Abstract. Naive program transformations can have surprising effects due to the interaction between introduced identifier references and previously existing identifier bindings, or between introduced bindings and previously existing references. These interactions can result in inadvertent binding, or capturing, of identifiers. A further complication is that transformed programs may have little resemblance to original programs, making correlation of source and object code difficult. This article describes an efficient macro system that prevents inadvertent capturing while maintaining the correlation between source and object code. The macro system allows the programmer to define program transformations using an unrestricted, general-purpose language. Previous approaches to the capturing problem have been inadequate, overly restrictive, or inefficient, and the problem of source-object correlation has been largely unaddressed. The macro system is based on a new algorithm for implementing syntactic transformations and a new representation for syntactic expressions.

1. Introduction

A fundamental problem with traditional Lisp macro systems is that they do not respect lexical scoping. When one expression is substituted for another, apparent bindings can be shadowed, resulting in unintended capture of identifier references. The capturing problem, which is the source of many serious and difficult to find bugs, was first addressed by Kohlbecker, Friedman, Felleisen, and Duba [17]. To solve the capturing problem, they proposed a *hygiene condition* for macros and an elegant algorithm for en-

*This material is based on work supported by the National Science Foundation under grant number CCR-8803432.

†Robert Hieb died in an automobile accident in April 1992.

forcing this condition.

A problem of equal practical importance is that Lisp macro systems cannot track source code through the macro-expansion process. Reliable correlation of source code and macro-expanded code is necessary if the compiler, run-time system, and debugger are to communicate with the programmer in terms of the original source program. The effort invested in compilers to correlate source and optimized object code [7, 14, 19] is wasted if the macro expander loses the correlation of source and expanded code before compilation begins. The correlation techniques applied to optimizing compilers do not extend to macro processors because these techniques require that the entire set of possible code transformations be known *a priori*.

This article presents a macro system for Scheme that enforces hygiene automatically while maintaining the correlation between source and object code. The macro system supports macros written in a general purpose programming language (Scheme) as well as macros written in high-level pattern languages such as **extend-syntax** [8, 15, 16] and **syntax-rules** [6]. Local macros are *referentially transparent* in the sense that free identifiers appearing in the output of a local macro are scoped where the macro definition appears [5]. No keywords are reserved; keywords for core forms such as **lambda** as well as macro keywords defined by the program can be rebound, either as new macro keywords or as variables. The system supports a controlled form of identifier capture that allows most common "capturing" macros to be written without violating the spirit of hygiene. The system is based on a new representation for syntactic expressions and a new algorithm for implementing syntactic transformations. The algorithm has at its core a modified version of the Kohlbecker, Friedman, Felleisen, and Duba (KFFD) algorithm. Unlike the KFFD algorithm, the new algorithm adds only constant overhead to the macro expansion process.

Other systems have been proposed that share some of the features listed above; many of these systems are described in the following section. No other system proposed to date, however,

- enforces hygiene with constant overhead for macros written in a full general-purpose language,
- solves the source-object correlation problem for variables and constants as well as structured expressions,
- supplies a hygiene-preserving mechanism for controlled identifier capture,
- maintains referential transparency for all local macros, or
- provides automatic syntax checking, input destructuring, and output restructuring for low-level macros.

The remainder of this article is structured as follows. Section 2 discusses related work. Section 3 describes the interface to our macro system and

examples of its use. Section 4 describes our algorithm. It begins with a variant of the KFFD algorithm applied to abstract syntax, demonstrates how delaying part of the work of this algorithm reduces expansion overhead to a constant, describes how controlled identifier captures are implemented, and shows how the correlation between source and object code is maintained. Section 5 summarizes the article and presents our conclusions.

2. Background

Inadvertent capturing of identifier references occurs in one of two ways. First, an identifier binding introduced by a macro can capture references to identifiers of the same name within the subexpressions of the macro call. For example, consider the expression[1]

```
(let ((t "okay"))
  (or2 #f t))
```

where **or2** is a two-subexpression version of **or** to be expanded as follows

$$(\textbf{or2}\ e1\ e2) \Rightarrow (\textbf{let}\ ((t\ e1))\ (\textbf{if}\ t\ t\ e2))$$

The expression should evaluate to "okay". A naive expansion of the expression, however, produces

```
(let ((t "okay"))
  (let ((t #f))
    (if t t t)))
```

This evaluates to #f, since the reference to *t* in the macro call is captured by the binding for *t* inserted during expansion.

Second, an identifier reference introduced by a macro can be captured by identifier bindings for the same name within the context of the macro call. For example, the expression

```
(let ((if #f))
  (or2 if "okay"))
```

should also evaluate to "okay"[2], but a naive expansion produces

[1] For readability, keywords are shown in boldface, variables in italic, and constants in Roman.

[2] This example presumes that keywords, such as **if**, are not reserved.

```
(let ((if #f))
  (let ((t if))
    (if t t "okay")))
```

This results in an attempt to apply the nonprocedure #f, causing a run-time error.

Various *ad hoc* techniques have been developed to help prevent unintended identifier captures. Capturing problems have been avoided by using generated names, special naming conventions, or the careful use of lexical scoping combined with local procedure definitions. With each of these techniques, the programmer must do something special to avoid capturing, making capturing the *default* even though capturing is rarely desired. What is worse, macros that can cause unintended captures often do not do so immediately but lie dormant, waiting for an unsuspecting programmer to insert just the right (wrong!) identifier name into a macro call or its context.

This insidious problem was first addressed by Kohlbecker, Friedman, Felleisen, and Duba [17], who present an algorithm in which the macro system *automatically* renames bound variables to prevent inadvertent capturing. The fundamental notion underlying the KFFD algorithm is *alpha equivalence*, which equates terms that differ only in the names of bound variables. Alpha equivalence is the basis of Barendregt's *variable convention*, which assumes that the bound variables in terms used in definitions and proofs are always chosen to be different from the free variables [2, page 26]. The KFFD algorithm respects the variable convention and thus is said to be "hygienic." It traverses each expression after it is rewritten in order to give identifiers "time stamps," which are used during alpha conversion to distinguish identifiers that are introduced at different times in the transformation process. Unfortunately, since the algorithm completely traverses each expression after it is rewritten, the time complexity of the macro expansion process increases from linear to quadratic with respect to the number of expressions present in the source code or introduced during macro expansion. This is a serious problem for large programs that make heavy use of macros, *i.e.*, nearly all large Scheme programs.

Clinger and Rees [5] present an algorithm for hygienic macro transformations that does not have the quadratic time complexity of the KFFD algorithm. Their algorithm marks only the new identifiers introduced at each iteration of the macro transformation process, rather than all of the identifiers as in the KFFD algorithm. Their system, however, allows macros to be written only in a restricted high-level specification language in which it is easy to determine where new identifiers will appear in the output of a macro. Since some macros cannot be expressed using this language,

they have developed a low-level interface that requires new identifiers to be marked explicitly [4].

Bawden and Rees [3] approach the capturing problem from a different angle. Rather than providing automatic hygiene, their system forces the programmer to make explicit decisions about the resolution of free identifier references and the scope of identifier bindings. Borrowing from the notion that procedures can be represented by *closures* that encapsulate lexical environments, they allow the programmer to create *syntactic closures* that encapsulate syntactic environments. The result is a system that allows the programmer to avoid unwanted capturing. Unlike traditional closures, however, syntactic closures and their environments must be constructed explicitly. As a result, the mechanism is difficult to use and definitions created using it are hard to understand and verify. Hanson [13] alleviates this problem somewhat by demonstrating that the restricted high-level specification language supported by Clinger and Rees can be built on top of an extended version of syntactic closures.

For a large class of macros, those that cannot be written in this high-level specification language, both the Clinger/Rees and syntactic closures approaches place responsibility for enforcing hygiene on the macro writer rather than on the underlying transformation algorithm. Furthermore, the pattern matching, destructuring, and restructuring facilities provided by the specification language must be completely abandoned for the same class of macros. Both low-level interfaces are completely different in style and usage from the high-level specification language.

Macros defined in the high-level specification language are *referentially transparent* [18] in the sense that a macro-introduced identifier refers to the binding lexically visible where the macro definition appears rather than to the top-level binding or to the binding visible where the macro call appears. This extends hygiene to local macros, which were not supported by the KFFD algorithm. Like automatic hygiene, this transparency is not present in either the syntactic closures or the Clinger/Rees low-level interfaces.

Griffin [12] describes a theory of syntactic definitions in the context of interactive proof development systems. He supports high-level definitions of derived notations in such a way that the definitions have certain formal properties that make them easy to reason about. As a result, however, his system is very restrictive with respect to the sort of macros that can be defined.

Dybvig, Friedman, and Haynes [10, 11] address the source-object correlation problem, demonstrating that their proposed macro expansion protocol, *expansion-passing style*, is capable of maintaining source-object correlation even in the presence of arbitrary user-defined macros. Their mechanism, however, does not enforce hygiene and handles only structured expressions;

in particular, it does not handle variable references.

The Revised[4] Report on Scheme [6] includes an appendix that contains a proposed macro system for Scheme. The high-level system (**syntax-rules**) described therein is a version of Kohlbecker's **extend-syntax** [8, 15, 16] with the same restrictions imposed by Clinger and Rees [5]. The revised-report appendix also describes a low-level system that, although it automatically preserves hygiene and referential transparency, requires manual destructuring of the input and restructuring of the output. The low-level system described in the revised-report appendix was proposed by the authors of this article and is the predecessor of the system described here. The new system provides only a high-level pattern language, similar to the one provided by **syntax-rules**, which is nonetheless powerful enough to provide the functionality of a "low-level" system while maintaining automatic hygiene, referential transparency, and source-object correlation.

3. The language

The macro system supports the set of syntactic forms and procedures shown in Figure 1. Each of these forms and procedures is described in this section.

All extended syntactic forms, or *macro calls*, take the form

(*keyword subform* ...)

where *keyword* is an identifier that names a macro. The syntax of each *subform* is determined by the macro and can vary significantly from macro to macro[3]. When the macro expander encounters a macro call, the macro call expression is passed to the associated transformer to be processed. The expansion process is repeated for the result returned by the transformer until no macro calls remain.

New syntactic forms, or macros, are defined by associating *keywords* with transformation procedures, or *transformers*. Top-level syntactic definitions are created using **define-syntax**.

(**define-syntax** *keyword transformer-expression*)

transformer-expression must be an expression that evaluates to a transformer.

The scope of syntactic definitions can be limited by using the lexical binding forms **let-syntax** and **letrec-syntax**[4]. In both cases *keyword* denotes

[3] Although not shown, macro calls can also take the form of improper lists.

[4] Also, internal **define-syntax** forms may appear wherever internal **define** forms are permitted, in which case the definitions behave as if introduced by **letrec-syntax**.

Macro calls:

(*keyword subform* ...)

Macro definition and binding:

(**define-syntax** *keyword transformer-expression*)
(**let-syntax** ((*keyword transformer-expression*) ...) *body*)
(**letrec-syntax** ((*keyword transformer-expression*) ...) *body*)

Destructuring and restructuring:

(**syntax-case** *input-expression* (*literal* ...)
  (*pattern fender expression*)
  ...)
(**syntax** *template*)
(*syntax-object->datum syntax-object*)
(*datum->syntax-object identifier datum*)

Predicates:

(*identifier? object*)
(*free-identifier=? identifier*$_1$ *identifier*$_2$)
(*bound-identifier=? identifier*$_1$ *identifier*$_2$)

Figure 1: Syntactic forms and procedures provided by the macro system.

new syntax in *body*; for **letrec-syntax** the binding scope also includes each *transformer-expression*.

(**let-syntax** ((*keyword transformer-expression*) ...) *body*)
(**letrec-syntax** ((*keyword transformer-expression*) ...) *body*)

At the language level, the fundamental characteristic of the macro system is the abstract nature of the arguments passed to macro transformers. The argument to a macro transformer is a *syntax object*. A syntax object contains contextual information about an expression in addition to its structure. This contextual information is used by the expander to maintain hygiene and referential transparency. Traditional Lisp macro systems use ordinary list-structured data to represent syntax. Although such list structures are convenient to manipulate, crucial syntactic information cannot be maintained. For example, the ability to distinguish between different identifiers that share the same name is of paramount importance. Information to allow these distinctions to be drawn is contained within each abstract syntax object, so that transformers can compare identifiers according to their *intended use* as free identifiers, bound identifiers, or symbolic data.

Syntax objects may contain other syntactic information that is not of direct interest to the macro writer. In our system, for example, syntax objects can contain source annotations that allow the evaluator to correlate the final object code with the original source code that produced it. Or, as discussed in Section 4, syntax objects may contain information that for efficiency reasons has not yet been fully processed.

Transformers decompose their input using **syntax-case** and rebuild their output using **syntax**. A **syntax-case** expression takes the following form:

(**syntax-case** *input-expression* (*literal* ...) *clause* ...)

Each *clause* takes one of the following two forms:

(*pattern output-expression*)
(*pattern fender output-expression*)

syntax-case first evaluates *input-expression*, then attempts to match the resulting value with the pattern from the first *clause*. This value is usually a syntax object, but it may be any Scheme list structure. If the value matches the pattern, and there is no *fender* present, *output-expression* is evaluated and its value returned as the value of the **syntax-case** expression. If the value does not match the pattern, the value is compared against the next clause, and so on. An error is signaled if the value does not match any of the patterns.

Patterns consist of list structures, identifiers, and constants. Each identifier within a pattern is either a *literal*, a *pattern variable*, or an *ellipsis*. The identifier **...** is an ellipsis. Any identifier other than **...** is a literal if it appears in the list of literals (*literal* ...) in the **syntax-case** expression; otherwise it is a pattern variable. Literals serve as auxiliary keywords, such as **else** in **case** and **cond** expressions. List structure within a pattern specifies the basic structure required of the input, 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 a pattern variable; or
- P is a literal identifier and F is an identifier with the same binding; or
- P is a pattern list $(P_1 \ \ldots \ P_n)$ and F is a list of n forms that match P_1 through P_n, respectively; or
- P is an improper pattern list $(P_1 \ P_2 \ \ldots \ P_n \ . \ P_{n+1})$ and F is a list or improper list of n or more forms that match P_1 through P_n, respectively, and whose nth "cdr" matches P_{n+1}; or

- P is of the form $(P_1 \ldots P_n\ P_{n+1}\ \mathbf{...})$ where F is a proper list of at least n elements, the first n of which match P_1 through P_n, respectively, and each remaining element of F matches P_{n+1}; or
- P is a pattern datum[5] and F is equal to P in the sense of the *equal?* procedure.

If the optional *fender* is present, it serves as an additional constraint on acceptance of a clause. If the value of *input-expression* 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 name space as program variables and keywords; 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. Scheme **syntax** expressions have the following form:

(**syntax** *template*)

A **syntax** form returns a Scheme object in much the same way as **quote** or **quasiquote**, with two important differences: the values of pattern variables appearing within *template* are inserted into *template*, and contextual syntactic information contained within *template* is retained. All list structure within *template* remains ordinary list structure in the output, and all other items (including identifiers that do not represent pattern variables) are inserted without further interpretation. Contextual information associated with the values of inserted pattern variables and any nonlist items from the template is retained in the output.

A *template* is a pattern variable, a literal identifier, a pattern datum, a list of subtemplates $(S_1 \ldots S_n)$, or an improper list $(S_1\ S_2 \ldots S_n\ .\ T)$. Each subtemplate S_i is either a template or a template followed by an ellipsis. The final element T of an improper subtemplate list is a template.

A subtemplate followed by an ellipsis expands into zero or more occurrences of the subtemplate. The subtemplate must contain at least one pattern variable that was in a subpattern followed by an ellipsis in the input. (Otherwise, the expander could not determine how many times the

[5] A *pattern datum* is any nonlist, nonsymbol datum.

subform should be repeated in the output.) This generalizes in a natural way to nested ellipses [8]. There is one exception to this rule: the special template (... ...) expands into ... . This is used by macro-defining macros to introduce ellipses into the defined macros. (See the definition of **be-like-begin** later in this section.)

A pattern variable that occurs in a **syntax** template is replaced by the subform it matched in the **syntax-case** expression that established the pattern variable's binding. 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 subforms they matched in the input, distributed as specified.

The definition for **or** below demonstrates the use of **define-syntax**, **syntax-case**, and **syntax**.

```
(define-syntax or
  (lambda (x)
    (syntax-case x ()
      ((_) (syntax #f))
      ((_ e) (syntax e))
      ((_ e1 e2 e3 ...)
       (syntax (let ((t e1)) (if t t (or e2 e3 ...))))))))
```

The input patterns specify that the input must consist of the keyword[6] and zero or more subexpressions. If there is more than one subexpression (third clause), the expanded code must both test the value of the first subexpression and return the value if it is not false. In order to avoid evaluating the expression twice, the macro introduces a binding for the temporary variable *t*. Because the expansion algorithm maintains hygiene automatically, this binding is visible only within code introduced by the macro and not within subforms of the macro call.

The combined expressive power of **syntax-case**, **syntax**, and pattern variables renders a low-level macro system unnecessary. Unlike restricted rewrite-rule systems such as the Revised[4] Report on Scheme **syntax-rules** system, input patterns are associated with output *expressions* rather than output templates. Arbitrary transformations may be performed since an output expression may be any Scheme expression. The only restriction is

[6] An underscore, which is an ordinary pattern variable, is used by convention for the keyword position to remind the macro writer and anyone reading the macro definition that the keyword position never fails to contain the expected keyword and need not be matched.

that "raw" symbols cannot appear in the output of a transformer; identifiers must be introduced using **syntax** (or *datum->syntax-object*, which is described later). Unlike other low-level proposals, **syntax-case** relieves the programmer of the tedium of pattern matching, destructuring, and restructuring expressions, and it provides a level of syntax checking that macro programmers usually do not provide.

The functionality of **syntax-rules** is subsumed by **syntax-case**. In fact, **syntax-rules** is easily defined as a macro:

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      ((_ (i ...) ((keyword . pattern) template) ...)
       (syntax (lambda (x)
                 (syntax-case x (i ...)
                   ((dummy . pattern) (syntax template))
                   ...)))))))
```

The unreferenced pattern variable *dummy* is used in place of each *keyword* since the Revised[4] Report on Scheme requires that the first position of each **syntax-rules** pattern be ignored. Just as the binding for *t* was not visible in *e2 e3* **...** in **or** above, the pattern variable binding for *dummy* is not visible within *template*.

Although the examples in this article employ **syntax-case** rather than **syntax-rules**, some could use **syntax-rules**, and the choice of which to use in those cases is a matter of taste.

Local macro definitions are referentially transparent in the sense discussed in Section 2. Identifiers within a **syntax** expression, like ordinary identifier references, refer to the closest enclosing lexical binding. For example,

```
(let ((/ +))
  (let-syntax ((divide (lambda (x)
                         (syntax-case x ()
                           ((_ e1 e2) (syntax (/ e1 e2)))))))
    (let ((/ *)) (divide 6 3))))
```

returns 9 since the / inserted by the macro *divide* refers to the outer **let** binding, rather than 2, as would be the case if the reference were made global, or 18, as would be the case if the reference were captured by the inner **let** binding.

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

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

results in an "invalid reference" error, since the occurrence of / in the output of *divide* is a reference to the variable / bound by the **let** expression within the transformer.

Symbolic names alone do not distinguish identifiers unless the identifiers are 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. The predicate *free-identifier=?* is used to determine whether two identifiers would be equivalent if they appeared 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 lexical or top-level binding[7]. Literal identifiers appearing in **syntax-case** patterns (such as **else** in **case** and **cond**) are matched using *free-identifier=?*.

Similarly, the predicate *bound-identifier=?* is used to determine if two identifiers would be equivalent if they appeared as bound identifiers in the output of a transformer. In other words, if *bound-identifier=?* returns true for two identifiers, then 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 macro application (perhaps implicitly; see *datum->syntax-object* below). The predicate *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.

Two identifiers that are *bound-identifier=?* are also *free-identifier=?*, but two identifiers that are *free-identifier=?* may not be *bound-identifier=?*. An identifier introduced by a macro 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.

[7] All variables are assumed to have top-level bindings, whether defined (yet) or not.

The definition for a simplified version of **cond**[8] below demonstrates how an auxiliary keyword, in this case **else**, is handled using the literals list of **syntax-case**:

```
(define-syntax cond
  (lambda (x)
    (syntax-case x (else)
      ((_ (else e1 e2 ...)) (syntax (begin e1 e2 ...)))
      ((_ (e0 e1 e2 ...)) (syntax (if e0 (begin e1 e2 ...))))
      ((_ (e0 e1 e2 ...) c1 c2 ...)
       (syntax (if e0 (begin e1 e2 ...) (cond c1 c2 ...)))))))
```

The definition above is equivalent to the following, which looks for **else** using *free-identifier=?* within a fender:

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

The predicate *identifier?*, used prior to the *free-identifier=?* check in the fender, returns true if and only if its argument is an identifier.

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

```
(let ((else #f))
  (cond (else (display "oops"))))
```

does *not* print "oops", since **else** is bound locally and is therefore not *free-identifier=?* to the identifier **else** appearing in the definition for **cond**.

[8] The simplified version requires at least one output expression per clause and does not support the auxiliary keyword =>.

The following definition for unnamed **let** uses *bound-identifier=?* to detect duplicate identifiers:

```
(define-syntax let
  (lambda (x)
    (define unique-ids?
      (lambda (ls)
        (or (null? ls)
            (and (let notmem? ((x (car ls)) (ls (cdr ls)))
                   (or (null? ls)
                       (and (not (bound-identifier=? x (car ls)))
                            (notmem? x (cdr ls)))))
                 (unique-ids? (cdr ls))))))
    (syntax-case x ()
      ((_ ((i v) ...) e1 e2 ...)
       (if (unique-ids? (syntax (i ...)))
           (syntax ((lambda (i ...) e1 e2 ...) v ...))
           (error (syntax-object->datum x)
                  "duplicate identifier found"))))))
```

For this macro to be completely robust, it should also ensure that the bound variables are indeed identifiers using the predicate *identifier?*. With the definition for **let** above, the expression

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

causes a "duplicate identifier found" error, whereas

```
(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 macro call are not *bound-identifier=?*.

It is also possible to compare identifiers intended to be used as symbolic data. The procedure *syntax-object->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, *symbolic-identifier=?* might be defined as follows:

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

Two identifiers that are *free-identifier=?* are *symbolic-identifier=?*; in order to refer to the same binding, two identifiers must have the same name. The converse is not always true.

It is occasionally useful to define macros that introduce bindings for identifiers that are not supplied explicitly in each macro call. For example, we might wish to define a **loop** macro that binds the variable *exit* to an escape procedure within the loop body. Strict automatic hygiene, however, would prevent an introduced binding for *exit* from capturing references to *exit* within the loop body. Previous hygienic systems have provided mechanisms for *explicit capturing*, typically by allowing a macro to insert a symbol into an expansion as if it were part of the original source program [17]. Unfortunately, this means that macros cannot reliably expand into macros that use explicit capturing.

Our system provides a more consistent way to accommodate such macros. A macro may construct *implicit identifiers* that behave as if they were present in the macro call. Implicit identifiers are created by providing the procedure *datum->syntax-object* with a *template identifier* and a *symbol*. The template identifier is typically the macro keyword itself, extracted from the input, and the symbol is the symbolic name of the identifier to be constructed. The resulting identifier behaves as if it were introduced when the template identifier was introduced. For example, the **loop** macro mentioned above may be defined as follows:

```
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      ((k e1 ...)
       (with-syntax ((exit-id (datum->syntax-object
                               (syntax k)
                               'exit)))
         (syntax (call-with-current-continuation
                  (lambda (exit-id)
                    (let f () e1 ... (f)))))))))))
```

(The **with-syntax** form is like **let** but introduces pattern variable bindings rather than program variable bindings within the scope of its body. Its definition is shown later.)

This same mechanism may be used to create aggregate identifier names typically required when defining structure-definition constructs such as Common Lisp's **defstruct** [1] as macros. The procedure below constructs an implicit identifier using an aggregate name of the form "⟨structure name⟩-⟨field name⟩," from a structure name identifier *s-id* and a field name identifier *f-id*:

```
(define aggregate-identifier
  (lambda (s-id f-id)
    (let ((s-sym (symbol->string (syntax-object->datum s-id)))
          (f-sym (symbol->string (syntax-object->datum f-id))))
      (let ((sym (string->symbol (string-append s-sym "-" f-sym))))
        (datum->syntax-object s-id sym)))))
```

A **defstruct** form would expand into a set of definitions, including accessors for each field whose names are constructed using *aggregate-identifier*.

As its name implies, *datum->syntax-object* can convert an arbitrary datum into a syntax object, *i.e.*, the second argument need not be a symbol. Converting a nonsymbol datum into a syntax object has the effect of treating each symbol contained within the datum as an implicit identifier. The convenience of this feature is illustrated by the following definition for **include**, an expand-time version of *load*. (**include** "filename") expands into a **begin** expression containing the expressions found in the file named by "filename".

```
(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-object k x)
                      (f (read p)))))))))
    (syntax-case x ()
      ((k filename)
       (let ((fn (syntax-object->datum (syntax filename))))
         (with-syntax (((exp ...) (read-file fn (syntax k))))
           (syntax (begin exp ...))))))))
```

The definition for **include** uses *datum->syntax-object* to place the expressions read from "filename" into the proper lexical context so that identifier

references and definitions within those expressions are scoped where the **include** form appears. 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 **with-syntax** expression used in the **loop** and **include** examples above is the most convenient mechanism for establishing pattern variable bindings when no syntax matching is required. **syntax-case** can always be used instead, although doing so results in less readable code as illustrated by the following version of **loop**:

```
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      ((k e1 ...)
       (syntax-case (datum->syntax-object (syntax k) 'exit) ()
         (exit-id (syntax (call-with-current-continuation
                            (lambda (exit-id)
                              (let f () e1 ... (f)))))))))))
```

Given that **syntax-case** can be used in this manner, it is not surprising that **with-syntax** can be defined as a macro in terms of **syntax-case**:

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

A local macro may be written in terms of an existing syntactic form or procedure of the same name using **let-syntax**. The following shows how one might restrict **if** expressions within a given expression to require the "else" (alternative) part:

```
(let-syntax ((if (lambda (x)
                   (syntax-case x ()
                     ((_ e1 e2 e3) (syntax (if e1 e2 e3)))))))
  (if 1 2 3))
```

The expression above evaluates to 2. If the body were (**if** 1 2), however, a syntax error would be reported.

Although the definition above looks simple enough, there are a few subtle ways in which an attempt to write this macro might go wrong. If **letrec-syntax** were used in place of **let-syntax**, the identifier **if** inserted into the local macro's output would refer to the local **if** rather than the top-level **if**, and expansion would loop indefinitely. If the definition were specified as

```
(let-syntax ((if (lambda (x)
                   (syntax-case x ()
                     ((if e1 e2 e3) (syntax (if e1 e2 e3)))))))
  (if 1 2 3))
```

expansion would again loop indefinitely. The identifier *if* appearing at the start of the pattern is treated as a pattern variable, since it is not listed in the literals list of the **syntax-case** expression. Thus, it is bound to the corresponding identifier **if** from the input expression, which denotes the local binding of **if**. Placing **if** in the list of literals in an attempt to patch up the latter version does not work either:

```
(let-syntax ((if (lambda (x)
                   (syntax-case x (if)
                     ((if e1 e2 e3) (syntax (if e1 e2 e3)))))))
  (if 1 2 3))
```

This causes **syntax-case** to compare the literal **if** in the pattern, which is scoped outside of the **let-syntax** expression, with the **if** in the input expression, which is scoped inside the **let-syntax**. Since they do not refer to the same binding, they are not *free-identifier=?*, and a syntax error results.

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

It is sometimes necessary for a macro that generates a macro definition to insert one or more ellipses into the generated macro definition. This is done using the "escape sequence" (**... ...**), as the simple example below demonstrates:

```
(define-syntax be-like-begin
  (lambda (x)
    (syntax-case x ()
      ((_ name)
       (syntax (define-syntax name
                 (lambda (x)
                   (syntax-case x ()
                     ((_ e0 e1 (... ...))
                      (syntax (begin e0 e1 (... ...))))))))))))
```

With **be-like-begin** defined in this manner, (**be-like-begin sequence**) would have the same effect as

```
(define-syntax sequence
  (lambda (x)
    (syntax-case x ()
      ((_ e0 e1 ...)
       (syntax (begin e0 e1 ...))))))
```

That is, a **sequence** expression would be equivalent to a **begin** expression.

Additional **syntax-case** examples appear in [9].

4. The algorithm

4.1. Traditional macro systems

Traditional Lisp macro systems rely on programs and data having the same representation, both textually and as internal structures. This shared representation is exploited not only for macro expansion but also for program evaluation; most Lisp systems provide an evaluation procedure so that programs can construct and execute programs. Consequently, the concrete syntax of Lisp is best seen as consisting of internal data structures rather than text. We assume a concrete syntax of expressions ($e \in \mathit{Exp}$) defined as a data type consisting of an unspecified set of constants ($c \in \mathit{Const}$), symbols ($s \in \mathit{Sym}$), and structures built by pairing. The following signature specifies the abstract data type *Exp*:

$$\begin{array}{rcl} \mathit{Sym} & \subset & \mathit{Exp} \\ \mathit{Const} & \subset & \mathit{Exp} \\ \mathit{cons} & : & \mathit{Exp} \times \mathit{Exp} \rightarrow \mathit{Exp} \\ \mathit{car} & : & \mathit{Exp} \rightarrow \mathit{Exp} \\ \mathit{cdr} & : & \mathit{Exp} \rightarrow \mathit{Exp} \\ \mathit{pair?} & : & \mathit{Exp} \rightarrow \mathit{Bool} \\ \mathit{sym?} & : & \mathit{Exp} \rightarrow \mathit{Bool} \end{array}$$

$$expand : Exp \times Env \rightarrow ExpandedExp$$

$$\begin{array}{l} expand(e, r) = \\ \quad \text{case } parse(e, r) \text{ of:} \\ \qquad constant(c) \rightarrow symbolic\text{-}data(c), \\ \qquad variable(s) \rightarrow variable(s), \\ \qquad application(e_1, e_2) \rightarrow application(expand(e_1, r), expand(e_2, r)), \\ \qquad symbolic\text{-}data(e) \rightarrow symbolic\text{-}data(e), \\ \qquad function(s, e) \rightarrow function(s, expand(e, r[s := Variable])), \\ macro\text{-}application(s, e) \rightarrow expand(t(e), r) \text{ where } t = r(s) \end{array}$$

Figure 2: A traditional macro-expansion algorithm.

The subset *Sym* (symbols) of *Exp* are atomic elements. *Const* includes such traditional Lisp constants as booleans, numbers and the empty list. The variables e, s, and c range over *Exp*, *Sym* and *Const*, respectively. The usual equations for elements of *Exp* hold:

$$\begin{array}{rcl} car(cons(e_1, e_2)) & = & e_1 \\ cdr(cons(e_1, e_2)) & = & e_2 \\ pair?(cons(e_1, e_2)) & = & True \\ sym?(cons(e_1, e_2)) & = & False \\ pair?(s) & = & False \\ sym?(s) & = & True \\ pair?(c) & = & False \\ sym?(c) & = & False \end{array}$$

Figure 2 shows a traditional expansion algorithm for a simplified language. The expander is assumed to be part of a standard evaluation process where the value of a program e is obtained by $eval(expand(e, r_{init}))$. The symbols **quote** and **lambda** are bound to *Special* in the initial expansion environment r_{init}; all other symbols are bound to *Variable*.

$$\begin{array}{rcl} r & \in & Env = Sym \rightarrow Transformer + \{Variable\} + \{Special\} \\ t & \in & Transformer = Exp \rightarrow Exp \end{array}$$

Macro expansion and parsing are inextricably intertwined in Lisp. Although Figure 2 shows the expander driving the parser, the relationship could just as well be reversed. The parser is shown in Figure 3. Pattern matching is used to hide the details of accessing the expression parts. The constructors (such as *symbolic-data*) used to communicate the output of the parser to the expander are not fully specified (their definition is trivial).

$$parse : Exp \times Env \rightarrow ParsedExp$$

$$\begin{aligned}
parse([\![c]\!], r) &= constant(c) \\
parse([\![s]\!], r) &= variable(s) \text{ if } r(s) = Variable \\
parse([\![(e_1\ e_2)]\!], r) &= application(e_1, e_2) \text{ if } e_1 \notin Sym \\
parse([\![(s\ e)]\!], r) &= application(s, e) \text{ if } r(s) = Variable \\
parse([\![(s\ e)]\!], r) &= macro\text{-}application(s, e) \text{ if } r(s) \in Transformer \\
parse([\![(\mathbf{quote}\ e)]\!], r) &= symbolic\text{-}data(e) \text{ if } r([\![\mathbf{quote}]\!]) = Special \\
parse([\![(\mathbf{lambda}\ s\ e)]\!], r) &= function(s, e) \text{ if } r([\![\mathbf{lambda}]\!]) = Special
\end{aligned}$$

Figure 3: A traditional macro-expansion parser.

The constructors of *ExpandedExp* are used to communicate the output of the expander to the evaluator.

This expansion algorithm clearly has serious hygiene problems. It does not prevent free identifiers inserted by a macro application from being captured by program bindings, nor does it prevent bindings introduced by macros from capturing free identifiers in the program.

4.2. A substitution-based macro system

In the λ-calculus, alpha conversion is used to circumvent hygiene problems caused by program transformations. Since the actual name of a bound variable is immaterial, a binding expression can be converted into an equivalent expression in which different names are used for the bound variables. Our algorithm uses alpha conversion to preserve hygiene during macro expansion.

Whether an identifier is being used as symbolic data or as a program variable, pattern variable, or keyword cannot be determined until after macro expansion. Since the name of an identifier used as symbolic data is important, naive alpha conversion is not viable in traditional macro expansion algorithms. Our algorithm makes alpha conversion possible by abandoning the traditional Lisp identification of variables and symbols. Instead, we introduce a new type of object, the *identifier*, which maintains both symbolic names and binding names until an identifier's role in a program is determined. Alpha conversion is accomplished by replacing only the binding names of bound identifiers.

Figure 4 shows the substitution-based macro-expansion algorithm. The parser, shown in Figure 5, has been modified to operate on identifiers rather than symbols and to recognize several new forms: **let-syntax**, **letrec-syntax**, **syntax**, and **plambda**. To simplify the presentation, **let-syntax**

$$expand : Exp \times Env \rightarrow ExpandedExp$$

$$\begin{array}{l}
expand(e, r) = \\
\quad \text{case } parse(e, r) \text{ of:} \\
\qquad variable(i) \rightarrow variable(resolve(i)), \\
\qquad application(e_1, e_2) \rightarrow application(expand(e_1, r), expand(e_2, r)), \\
\qquad symbolic\text{-}data(e) \rightarrow symbolic\text{-}data(strip(e)), \\
\qquad syntax\text{-}data(e) \rightarrow symbolic\text{-}data(e), \\
\qquad function(i, e) \rightarrow function(s, expand(subst(e, i, s), r')) \\
\qquad\qquad \text{where } r' = r[s := Variable] \text{ and } s \text{ is fresh}, \\
\qquad pfunction(i, e) \rightarrow function(s, expand(subst(e, i, s), r')) \\
\qquad\qquad \text{where } r' = r[s := PVariable] \text{ and } s \text{ is fresh}, \\
\qquad macro\text{-}application(i, e) \rightarrow expand(mark(t(mark(e, m)), m), r) \\
\qquad\qquad \text{where } t = r(resolve(i)) \text{ and } m \text{ is fresh}, \\
\qquad syntax\text{-}binding(i, e_1, e_2) \rightarrow expand(subst(e_2, i, s), r[s := t]) \\
\qquad\qquad \text{where } t = eval(expand(e_1, r)) \text{ and } s \text{ is fresh}, \\
\qquad rec\text{-}syntax\text{-}binding(i, e_1, e_2) \rightarrow expand(subst(e_2, i, s), r[s := t]) \\
\qquad\qquad \text{where } t = eval(expand(subst(e_1, i, s), r)) \\
\qquad\qquad\qquad \text{and } s \text{ is fresh}
\end{array}$$

Figure 4: A substitution-based macro-expansion algorithm.

and **letrec-syntax** are each restricted to a single binding. We have also restricted the subform of a **syntax** expression to a single identifier. If the identifier is a pattern variable, the **syntax** form evaluates to the value of the pattern variable; otherwise, the result is the identifier itself. The unrestricted version is a straightforward generalization. We have also chosen to add **plambda**, which binds a single pattern variable within it's body, rather than **syntax-case**, which can be defined in terms of **plambda** and *expose*, which is defined later. Pattern variables are bound to *PVariable* in the expansion environment.

$$Env = Sym \rightarrow Transformer + \{Variable\} + \{PVariable\} + \{Special\}$$

The function *resolve* is used by *expand* to complete alpha substitution and determine the actual binding name of an identifier. The binding name is used in the output for program variables and to look up transformers for syntactic keywords. When expanding a binding expression, *subst* replaces the binding name of the bound identifier with a fresh binding name. To distinguish new identifiers introduced by a transformer, both input to the transformer and output from the transformer are freshly marked. Since identical marks cancel each other, only new syntax retains the mark[9]. The

[9]For the simplified language considered here it would be adequate to mark only the

$$parse : Exp \times Env \rightarrow ParsedExp$$

$$\begin{aligned}
parse([\![c]\!], r) &= symbolic\text{-}data(c) \\
parse([\![i]\!], r) &= variable(i) \text{ if } r(resolve(i)) = Variable \\
parse([\![(e_1\ e_2)]\!], r) &= application(e_1, e_2) \text{ if } e_1 \notin Sym \\
parse([\![(i\ e)]\!], r) &= application(i, e) \\
&\quad \text{if } r(resolve(i)) = Variable \\
parse([\![(i\ e)]\!], r) &= macro\text{-}application(i, e) \\
&\quad \text{if } r(resolve(i)) \in Transformer \\
parse([\![(\mathbf{quote}\ e)]\!], r) &= symbolic\text{-}data(e) \text{ if } r([\![\mathbf{quote}]\!]) = Special \\
parse([\![(\mathbf{lambda}\ i\ e)]\!], r) &= function(i, e) \text{ if } r([\![\mathbf{lambda}]\!]) = Special \\
parse([\![(\mathbf{plambda}\ i\ e)]\!], r) &= pfunction(i, e) \\
&\quad \text{if } r([\![\mathbf{plambda}]\!]) = Special \\
parse([\![(\mathbf{syntax}\ i)]\!], r) &= syntax\text{-}data(i) \\
&\quad \text{if } r(resolve(i)) \neq PVariable \\
parse([\![(\mathbf{syntax}\ i)]\!], r) &= variable(i) \text{ if } r(resolve(i)) = PVariable \\
parse([\![(\mathbf{let\text{-}syntax}\ (i\ e_1)\ e_2)]\!], r) &= syntax\text{-}binding(i, e_1, e_2) \\
&\quad \text{if } r([\![\mathbf{let\text{-}syntax}]\!]) = Special \\
parse([\![(\mathbf{letrec\text{-}syntax}\ (i\ e_1)\ e_2)]\!], r) &= rec\text{-}syntax\text{-}binding(i, e_1, e_2) \\
&\quad \text{if } r([\![\mathbf{letrec\text{-}syntax}]\!]) = Special
\end{aligned}$$

Figure 5: A substitution-based macro-expansion parser.

expander handles two sorts of data, symbolic (introduced by **quote** expressions) and syntactic (introduced by **syntax** expressions). Symbolic data is stripped of identifier substitutions and markings, whereas syntactic data is left intact.

Since *mark* and *subst* both generate elements of *Exp*, they can be treated as constructors in an extended *Exp* algebra.

$$\begin{aligned}
mark &: Exp \times Mark \rightarrow Exp \\
subst &: Exp \times Ident \times Sym \rightarrow Exp
\end{aligned}$$

Marks ($m \in Mark$) can be any countably infinite set. Identifiers ($i \in Ident$) are a subset of the expanded *Exp* domain. An identifier is a symbol that has been subjected to zero or more marking and substitution operations. That is, an identifier is a symbol s, a marked identifier $mark(i, m)$, or a substitution $subst(i_1, i_2, s)$. The intent of $subst(e, i, s)$ is to replace the binding name of the identifier i in the expression e with the symbol s.

input to the transformer. This approach, however, would not work for more complex language constructs in which internal definitions are expanded separately and then recombined into a binding expression. It would also cause complexity problems for the delayed substitution mechanism described in Section 4.4.

Since marking and substitution operations are of interest only insofar as they affect identifiers, it is convenient to think of them as identity operations on constants and as being immediately propagated to the components of a pair. For now, we assume that this is the case, although later we abandon this assumption in order to avoid complexity problems.

$$\begin{aligned}
mark(c, m) &= c && (1)\\
subst(c, i, s) &= c && (2)\\
mark(cons(e_1, e_2), m) &= cons(mark(e_1, m), mark(e_2, m)) && (3)\\
subst(cons(e_1, e_2), i, s) &= cons(subst(e_1, i, s), subst(e_2, i, s)) && (4)
\end{aligned}$$

The function *resolve* is used to determine the binding name of an identifier. It resolves substitutions using the criterion that a substitution should take place if and only if both identifiers have the same marks and the same binding name.

$$\begin{aligned}
resolve &: Ident \rightarrow Sym\\
resolve(s) &= s\\
resolve(mark(i, m)) &= resolve(i)\\
resolve(subst(i_1, i_2, s)) &= \begin{cases} s & \text{if } marksof(i_1) = marksof(i_2) \\ & \text{and } resolve(i_1) = resolve(i_2) \\ resolve(i_1) & \text{otherwise} \end{cases}
\end{aligned}$$

The auxiliary function *marksof* determines an identifier's mark set:

$$\begin{aligned}
marksof &: Ident \rightarrow MarkSet\\
marksof(s) &= \emptyset\\
marksof(mark(i, m)) &= marksof(i) \bowtie \{m\}\\
marksof(subst(i_1, i_2, s)) &= marksof(i_1)
\end{aligned}$$

The operator $\bowtie$ forms an exclusive union, which cancels identical marks.

The function *strip* simply undoes marking and substitution operations:

$$\begin{aligned}
strip &: Exp \rightarrow Exp\\
strip(s) &= s\\
strip(c) &= c\\
strip(cons(e_1, e_2)) &= cons(strip(e_1), strip(e_2))\\
strip(mark(e, m)) &= strip(e)\\
strip(subst(e, i, s)) &= strip(e)
\end{aligned}$$

Two identifiers i_1 and i_2 are *free-identifier=?* if and only if $resolve(i_1) = resolve(i_2)$. Two identifiers i_1 and i_2 are *bound-identifier=?* if and only if $resolve(subst(i_1, i_2, s)) = s$ for a fresh symbol s.

So far the *Exp* algebra has been considered abstractly. A concrete algebra must ensure that the primitive accessors behave as specified. By virtue of equations (1)–(4), constants and pairs can use traditional representations. Identifiers can be represented as distinguished triples of the form $\langle s_1, s_2, \{m \ldots\}\rangle$, where s_1 is the symbolic name, s_2 is the binding name, and $\{m \ldots\}$ is a (possibly empty) set of marks. This representation takes advantage of *strip* requiring only the symbolic name of an identifier and *resolve* requiring only the final binding name of an identifier. Intermediate substitutions and substitutions that cannot succeed can be discarded without affecting the behavior of the accessors. The mark set for an identifier i is just $marksof(i)$. Marks can be represented as integers. Given this representation of identifiers, implementation of the primitive operators is straightforward. $mark(i, m)$ adds its mark to the mark field of i unless it is already present, in which case it removes it. $subst(i_1, i_2, s)$ replaces the binding name field of i_1 with s if the binding names and the marks of i_1 and i_2 are the same, otherwise it leaves the identifier unchanged. $strip(i)$ extracts the symbolic name of an identifier, whereas $resolve(i)$ extracts the binding name of an identifier.

For example, consider the expansion of the expression

(**let** ((*if* #f)) (**or2** *if* *t*))

where **or2** is the two-subexpression version of **or** from Section 2, defined as follows:

```
(define-syntax or2
  (lambda (x)
    (syntax-case x ()
      ((_ e1 e2)
       (syntax (let ((t e1)) (if t t e2)))))))
```

The expression should evaluate to the top-level value of the variable *t*, assuming that *t* is bound at top-level. As the expansion unfolds, observe how substitution and marking prevent the binding for **if** in the source expression from interfering with the macro's use of **if** and the macro's binding for *t* from interfering with the source expression's reference to *t*.

For simplicity, we assume that **let** is handled directly by the expander; with a little more tedium we could first expand it into the corresponding **lambda** application.

As described above, identifiers are represented as ordered triples:

$\langle$*original name*, *binding name*, {*mark* ...}$\rangle$

The original input is thus

```
(⟨let, let, {}⟩ ((⟨if, if, {}⟩ #f))
    (⟨or2, or2, {}⟩ ⟨if, if, {}⟩ ⟨t, t, {}⟩)))
```

On the first step, since **let** has no binding other than its original binding in the top-level environment, the bound variable from the outer **let** expression is replaced with the generated name *G1*, and the occurrence of the identifier within the scope of the **let** expression is replaced with a new identifier that contains both the generated and original names:

```
(let ((G1 #f))
    (⟨or2, or2, {}⟩ ⟨if, G1, {}⟩ ⟨t, t, {}⟩)))
```

Existence of this binding for *G1* is also recorded in the lexical expand-time environment. Next, the transformer for **or2** is invoked, with identifiers in its input marked by mark m_1:

```
(⟨or2, or2, {m1}⟩ ⟨if, G1, {m1}⟩ ⟨t, t, {m1}⟩)
```

The transformer for **or2** produces

```
(⟨let, let, {}⟩ ((⟨t, t, {}⟩ ⟨if, G1, {m1}⟩)))
    (⟨if, if, {}⟩ ⟨t, t, {}⟩ ⟨t, t, {}⟩ ⟨t, t, {m1}⟩)))
```

Next, within the output from the **or2** transformer, identifiers not marked with m_1 are so marked while the m_1 mark is removed from the others (since identical marks cancel).

```
(⟨let, let, {m1}⟩ ((⟨t, t, {m1}⟩ ⟨if, G1, {}⟩)))
    (⟨if, if, {m1}⟩ ⟨t, t, {m1}⟩ ⟨t, t, {m1}⟩ ⟨t, t, {}⟩)))
```

Only the binding name is relevant when an identifier's binding is determined in the expand-time environment, so even though the mark m_1 has been attached to the identifier **let**, it still resolves to the top-level definition for **let**. Therefore, the bound identifier is replaced with a generated name and occurrences of the identifier (with the same binding name and marks) are replaced with a new identifier within the scope of the **let** expression:

```
(let ((G2 ⟨if, G1, {}⟩))
   (⟨if, if, {m1}⟩ ⟨t, G2, {m1}⟩ ⟨t, G2, {m1}⟩ ⟨t, t, {}⟩))
```

Existence of this binding for *G2* is also recorded in the lexical expand-time environment, which still also holds a record of the binding for *G1*.

Since *G1* is recorded as a lexically bound variable in the expand-time environment, the occurrence of ⟨if, G1, {}⟩ expands into a reference to *G1*. Thus, the output expression so far consists of

```
(let ((G1 #f))
   (let ((G2 G1))
      (⟨if, if, {m1}⟩ ⟨t, G2, {m1}⟩ ⟨t, G2, {m1}⟩ ⟨t, t, {}⟩)))
```

with the last line as yet unexpanded. Since the binding name of the identifier ⟨if, if, {m_1}⟩ is **if**, the last line is recognized as an **if** expression:

```
(let ((G1 #f))
   (let ((G2 G1))
      (if ⟨t, G2, {m1}⟩ ⟨t, G2, {m1}⟩ ⟨t, t, {}⟩)))
```

Only the three variable references within this **if** expression remain to be expanded. The binding name, *G2*, for the first and second of these is recorded as a lexical variable in the expand-time environment so both simply expand into *G2*. The binding name for the third is *t*, which has no binding in the lexical expand-time environment; therefore it expands into a top-level reference to *t*. Thus, the final output from the expander is

```
(let ((G1 #f))
   (let ((G2 G1))
      (if G2 G2 t)))
```

4.3. Capturing

The procedure *datum->syntax-object* must construct "implicit identifiers" that behave as if they had appeared in place of the template identifier when the template identifier was first introduced. That is, if *datum->syntax-object* is called with an identifier i_1 and a symbol s_2, where the symbolic name of i_1 is s_1, *datum->syntax-object* should create the identifier i_2 that would have resulted had s_2 appeared in place of s_1 in the original input. The following definition for *imp-id* captures this semantics:

$$\begin{array}{rcl} \textit{imp-id} & : & \textit{Ident} \times \textit{Sym} \rightarrow \textit{Ident} \\ \textit{imp-id}(s_1, s_2) & = & s_2 \\ \textit{imp-id}(\textit{mark}(i, m), s) & = & \textit{mark}(\textit{imp-id}(i, s), m) \\ \textit{imp-id}(\textit{subst}(i_1, i_2, s_1), s_2) & = & \textit{subst}(\textit{imp-id}(i_1, s_2), i_2, s_1) \end{array}$$

Supporting *imp-id* means that the representation of identifiers cannot omit failed substitutions, since the new accessor *imp-id* can observe them. Intermediate substitutions are still unimportant, however, and substitutions that fail because of mismatched marks can still be discarded. Thus the representation of identifiers as triples can be adapted by replacing the binding name in an identifier triple with an environment that maps *Sym* to *Sym*. *resolve* must then apply the environment to the symbolic name to get the binding name. $\textit{imp-id}(i, s)$ simply builds a new identifier triple from s and the environment and marks from i.

4.4. A lazy substitution-based macro system

The substitution-based macro system has the virtue of providing an intuitive, alpha substitution-based solution to the hygiene problem. Unfortunately, its implementation as suggested above is too expensive. The expense arises from the desire to make pairs transparent to hygiene operations. To maintain this transparency, every mark or substitution operation must be propagated immediately to all the identifiers in an expression. Consequently, the overhead incurred by the hygienic algorithm at each expansion step that uses these operations is proportional to the size of the expression, compared to constant overhead in a traditional system. This is precisely the source of the complexity problem for the KFFD algorithm.

We solve this problem by making substitutions and markings "lazy" on structured expressions. Eventually, the work of propagating identifier operations must be done. If *Exp* were being used only as syntax, it would be reasonable to let structure accessors do the work. For instance, we could have

$$\textit{car}(\textit{mark}(\textit{cons}(e_1, e_2), m)) = \textit{mark}(e_1, m).$$

Rather than alter the definitions of *car* and other accessors, however, we provide an accessor that exposes the outermost structure of an expression by pushing identifier information down to its constituent parts:

$$\begin{array}{rcl} \textit{expose} & : & \textit{Exp} \rightarrow \textit{Exp} \\ \textit{expose}(i) & = & i \\ \textit{expose}(\textit{mark}(\textit{cons}(e_1, e_2), m)) & = & \textit{cons}(\textit{mark}(e_1, m), \textit{mark}(e_2, m)) \\ & \textit{etc.} & \end{array}$$

The functionality of *expose* is required only by **syntax-case**, which uses *expose* to destructure the input value as far as necessary to match against the input patterns.

It remains to construct a concrete *Exp* algebra. Expressions with pending substitutions or marks can be represented as distinguished triples (*wrapped expressions*) of the form $\langle e, u, \{m \ldots\}\rangle$, where e is an expression, u is an environment mapping *Ident* to *Sym*, and $\{m \ldots\}$ is a set of zero or more marks. Pushing a substitution onto a wrapped expression involves updating the wrapped expression's environment with the new substitution. Marking a wrapped expression involves adding the mark to the mark set of the wrapped expression or removing it if the mark is already in the set. Marking also requires adding (removing) the mark to (from) the mark sets of each identifier in the wrapped expression's environment. Pairs, symbols, and constants can use traditional representations. To avoid complexity problems, additional constraints must be imposed. In particular, the expression component of a wrapped expression must not be another wrapped expression. This property can be maintained by having *expose* combine mark sets and environments when it pushes a wrapping onto another wrapped expression. Since marks "stick" only to new elements introduced by macro transformers, a wrapped expression will have more than one mark only if it is generated by a macro whose definition was itself generated by a macro. In practice the mark field of a wrapped expression rarely has more than one mark. Consequently, handling marks is cheap and the complexity problems caused by "eager" mark propagation are avoided.

4.5. Source-object correlation

The lazy substitution model can be adapted easily to support source-object correlation. We allow an expression to be annotated with information about its source by extending *Exp* with an additional constructor:

$$source : Exp \times Annotation \rightarrow Exp$$

An annotation ($a \in Annotation$) is an unspecified data structure that provides information about the source of an expression, such as its location in a file.

Source annotations can be passed along by the expander to the evaluator, where they can be used to provide debugging information. Thus we might add to the definition of *expand*:

$$expand(source(e, a), r) = source(expand(e, r), a)$$

The expander can also use source annotations to report errors it detects.

Otherwise, operations ignore or drop source annotations. For instance:

$$\begin{aligned} \mathit{expose}(\mathit{source}(e,a)) &= \mathit{expose}(e) \\ \mathit{mark}(\mathit{source}(e,a),m) &= \mathit{source}(\mathit{mark}(e,m),a) \\ \mathit{id?}(\mathit{source}(e,a)) &= \mathit{id?}(e) \\ \mathit{resolve}(\mathit{source}(i,a)) &= \mathit{resolve}(i) \end{aligned}$$

Since *expose* drops annotations, they are invisible to procedures that need to examine the structure of an expression. Previously constants were the sole class of expressions unaffected by syntactic operations. Since constants can also be annotated, however, they too must be "exposed" before they can be examined. Source annotations can be implemented by adding another field to the wrapped expression structure of Section 4.4.

5. Conclusions

The macro system described in this article, syntactic closures as augmented by Hanson [3, 13], and the Clinger and Rees "explicit renaming" system [4, 5] are all compatible with the "high-level" facility (**syntax-rules**) described in the Revised^4 Report on Scheme [6]. Thus, the three systems differ primarily in the treatment of "low-level" macros. Our system extends automatic hygiene and referential transparency to the low level, whereas the other systems require explicit renaming of identifiers or construction of syntactic closures, which is tedious and error-prone. In addition, we have extended the automatic syntax checking, input destructuring, and output restructuring previously available only to high-level macros to the low level. In fact, our system draws no distinction between high- and low-level macros, so there is never a need to completely rewrite a macro originally written in a high-level style because it needs to perform some low-level operation. We have also provided a mechanism for correlating source and object code and introduced a hygiene-preserving mechanism for controlled identifier capture, both of which are unique to our system.

An important aspect of our work is its thoroughgoing treatment of identifiers. Since identifiers cannot be treated as simple symbolic data in hygienic systems, the macro writer must be given tools that respect their essential properties. We provide tools for introducing new identifiers in a hygienic and referentially-transparent manner, for constructing macros that implicitly bind or reference identifiers, and for comparing identifiers according to their intended use as free identifiers, bound identifiers, or symbolic data.

Although our work is designed to provide a macro system with automatic hygiene for Scheme and other Lisp dialects, it could be adapted to languages in which programs and data do not share the same structure. An abstract syntax object representing each syntactic construct in the language must

be provided, along with appropriate accessors and constructors. Accessors would be responsible for propagating identifier information to the subpieces of an abstract syntax object.

The macro system described in this article has been implemented and the implementation is available via "ftp" from *cs.indiana.edu*. Contact the first author for details.

Acknowledgements: The authors would like to thank Dan Friedman and an anonymous reviewer for their helpful comments on earlier versions of this article.

References

1. Steele Jr., Guy L. *Common Lisp, the Language*. Digital Press, second edition (1990).

2. Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier Science Publishers, revised edition (1984).

3. Bawden, Alan and Rees, Jonathan. Syntactic closures. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming* (July 1988) 86–95.

4. Clinger, William. Hygienic macros through explicit renaming. *LISP Pointers*, 4, 4 (1991).

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

6. Clinger, William, Rees, Jonathan, *et al.* The revised^4 report on the algorithmic language Scheme. *LISP Pointers*, 4, 3 (1991).

7. Coutant, D., Meloy, S., and Ruscetta, M. DOC: A practical approach to source-level debugging of globally optimized code. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (July 1988) 125–134.

8. Dybvig, R. Kent. *The Scheme Programming Language*. Prentice-Hall (1987).

9. Dybvig, R. Kent. *Writing Hygienic Macros in Scheme with Syntax-Case*. Technical Report 356, Indiana Computer Science Department (June 1992).

10. Dybvig, R. Kent, Friedman, Daniel P., and Haynes, Christopher T. Expansion-passing style: Beyond conventional macros. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming* (1986) 143–150.

11. Dybvig, R. Kent, Friedman, Daniel P., and Haynes, Christopher T. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1, 1 (1988) 53–75.

12. Griffin, Timothy G. *Definition and Top-Down Refinement for Interactive Proof Development Systems.* PhD thesis, Cornell University (August 1988).

13. Hanson, Chris. A syntactic closures macro facility. *LISP Pointers*, 4, 4 (1991).

14. Hennessy, J. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4, 3 (July 1982) 323–344.

15. Kohlbecker, Eugene. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, Bloomington (August 1986).

16. Kohlbecker, Eugene and Wand, Mitchell. Macro-by-example: Deriving syntactic transformations from their specifications. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages* (1987) 77–84.

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

18. Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press (1977).

19. Zellweger, P. An interactive high-level debugger for control-flow optimized programs. In *Proceedings of the ACM Software Engineering Symposium on High-Level Debugging* (August 1983) 159–171.