Hygienic Macro Technology

WILLIAM D CLINGER, USA
MITCHELL WAND, Northeastern University, USA

Shepherd: Guy L. Steele Jr., Oracle Labs, USA

The fully parenthesized Cambridge Polish syntax of Lisp, originally regarded as a temporary expedient to be replaced by more conventional syntax, possesses a peculiar virtue: A `read` procedure can parse it without knowing the syntax of any expressions, statements, definitions, or declarations it may represent. The result of that parsing is a list structure that establishes a standard representation for uninterpreted abstract syntax trees.

This representation provides a convenient basis for macro processing, which allows the programmer to specify that some simple piece of abstract syntax should be replaced by some other, more complex piece of abstract syntax. As is well-known, this yields an abstraction mechanism that does things that procedural abstraction cannot, such as introducing new binding structures.

The existence of that standard representation for uninterpreted abstract syntax trees soon led Lisp to a greater reliance upon macros than was common in other high-level languages. The importance of those features is suggested by the ten pages devoted to macros in an earlier ACM HOPL paper, "The Evolution of Lisp."

However, naïve macro expansion was a leaky abstraction, because the movement of a piece of syntax from one place to another might lead to the accidental rebinding of a program's identifiers. Although this problem was recognized in the 1960s, it was 20 years before a reliable solution was discovered, and another 10 before a solution was discovered that was reliable, flexible, and efficient.

In this paper, we summarize that early history with greater focus on hygienic macros, and continue the story by describing the further development, adoption, and influence of hygienic and partially hygienic macro technology in Scheme. The interplay between the desire for standardization and the development of new algorithms is a major theme of that story.

We then survey the ways in which hygienic macro technology has been adapted into recent non-parenthetical languages. Finally, we provide a short history of attempts to provide a formal account of macro processing.

CCS Concepts: • **Software and its engineering → Macro languages**.

Additional Key Words and Phrases: macro, hygiene, Lisp, Scheme

ACM Reference Format:
William D Clinger and Mitchell Wand. 2020. Hygienic Macro Technology. *Proc. ACM Program. Lang.* 4, HOPL, Article 80 (June 2020), 110 pages. https://doi.org/10.1145/3386330

Authors' addresses: William D Clinger, USA, will@larcenists.org; Mitchell Wand, Northeastern University, USA, wand@ccs.neu.edu.

2475-1421/2020/6-ART80
https://doi.org/10.1145/3386330

Contents

1 THE CAPTURING PROBLEM

The essence of a macro is the substitution of one expression for another in a piece of code.

As is well-known in logic or lambda calculus, naïve substitution is unsound. Consider, for example, the universal elimination rule of first order logic. That rule is often stated as

$$\frac{\forall x \,.\, \varphi}{\varphi[t/x]}\ \forall\text{E}$$

which says we are allowed to substitute any term t for all occurrences of the universally quantified variable x that appear free within the formula φ. With naïve substitution, however, that formulation of the rule is unsound, preserving neither truth nor validity, as can be seen by taking φ to be the formula $\exists y \,.\, (x = y - 1)$ and t to be the term $y + 5$. Then $\forall x \,.\, \exists y \,.\, (x = y - 1)$ is true in the universe of natural numbers, but $\varphi[t/x]$ is $\exists y \,.\, (y + 5 = y - 1)$, which is emphatically not true. To obtain a sound formulation of the rule, we must add other side conditions such as

provided none of the variables that occur free in t occur bound in φ

That side condition prevents the rule from being applied to our example. Before we can substitute $y + 5$ for x, we must rename the bound variables of φ to obtain an equivalent formula such as $\exists z \,.\, (x = z - 1)$; such renaming of bound variables is called *alpha conversion.* To simplify the side conditions of universal elimination and other inference rules, logicians often define more complicated notions of substitution that automatically alpha-convert as necessary.

Similar problems occur in macro systems used by programmers, where the greater complexity of programming languages magnifies the problems, especially in block-structured, higher-order, and object-oriented programming languages. Capturing problems become even more subtle when local macros define other local macros, or separately compiled modules limit the visibility of some identifiers; such features have no direct analogues in logic or lambda calculus.

Consider, for example, a short-circuiting `or` macro for Lisp that evaluates a sequence of expressions from left to right, returning the first non-false value. We might hope to implement that macro via these three rewrite rules:

```
(or)           =>   #false
(or <test>)    =>   <test>
(or <test1> <test2> ...)        ; start of third rewrite rule
  =>
  (let ((temp <test1>))
    (if temp temp (or <test2> ...)))
```

The `let` in that third rewrite rule avoids evaluating `<test1>` twice, which would be incorrect if that expression has side effects. This illustrates the usefulness of block structure when defining macros. If it were not possible for expressions to bind local variables, this `or` macro would be harder to write.

On the other hand, the third rewrite rule also illustrates the problem of inadvertent capturing. Suppose a programmer uses the `or` macro in a context where `temp` is already being used as a local variable:

```
(let ((temp (read-temperature)))
  (if (or (not (real? temp))
          (< temp 0)
          (> temp UPPER_LIMIT))
      (display "Temperature out of range.\n")
      (process-temperature temp)))
```

With naïve transcription of the rewrite rules, that code would macro-expand into

```
(let ((temp (read-temperature)))
  (if (let ((temp (not (real? temp))))
        (if temp
            temp
            (let ((temp (< temp 0))) ; compares false to zero
              (if temp
                  temp
                  (> temp UPPER_LIMIT)))))
      (display "Temperature out of range.\n")
      (process-temperature temp)))
```

which is incorrect because it passes `#false` as an argument to the < comparison.

A related problem arises in lambda calculus, where its solution (automatic renaming of bound variables) is regarded as a matter of *hygiene* [Barendregt 1985, 1992]:

> For reasons of hygiene it will always be assumed that the bound variables that occur in a certain expression are different from the free ones. This can be fulfilled by renaming bound variables.

That capturing problem is even worse for macros, because identifiers bound at the site of the macro definition may be unbound, or bound to different values, where the macro is used. What if the identifier `if` had some nonstandard meaning in the context of the use?

Hygienic macros avoid the capturing problem by systematically renaming variables as necessary to respect the language's scope rules. In the Scheme programming language as it has been specified since 1998 [Kelsey et al. 1998; Sperber et al. 2009a; Shinn et al. 2013], a perfectly reliable `or` macro can be defined as follows:

```
(define-syntax or
  (syntax-rules ()
    ((or) #false)
    ((or ?e) ?e)
    ((or ?e1 ?e2 ?e3 ...)
     (let ((temp ?e1))
       (if temp temp (or ?e2 ?e3 ...))))))
```

The temperature-validating example shown above will macro-expand into code equivalent to

```
(let ((temp (read-temperature)))
  (if (let ((temp.1 (not (real? temp))))
        (if temp.1
            temp.1
            (let ((temp.2 (< temp 0)))
              (if temp.2
                  temp.2
                  (> temp UPPER_LIMIT)))))
      (display "Temperature out of range.\n")
      (process-temperature temp)))
```

and all is well.

1.1 Other Problems

The macro systems common in the 1980s suffered from several other problems that are solved by hygienic macro systems. The following examples are taken or derived from a paper on "Macros That Work" [Clinger and Rees 1991a].

Macro processors that substitute sequences of characters for sequences of characters can *fail to respect the integrity of lexical tokens:* With the original macro preprocessor for C,

```
#define veggie "salad
printf(veggie bar");
```

would print `"salad bar"`.

Macro processors that respect the integrity of tokens may still *fail to respect the structure of expressions:*

```
#define discriminant(a,b,c) b*b-4*a*c
2*discriminant(x-y,x+y,x-y)*5
```

expands into

```
2*x+y*x+y-4*x-y*x-y*5
```

when

```
2*((x+y)*(x+y)-4*(x-y)*(x-y))*5
```

is more likely to have been what the programmer intended. While it is true that most C programmers know the right hand side of that particular macro definition should surround all macro parameters with parentheses, that should be recognized as a workaround that burdens the programmer with a responsibility for doing something that could have been done automatically and reliably by a better macro processor.

Another problem arises when an expression substituted for the macro parameter `b` has a side effect. Because that expression appears twice in the macro-expanded output, the side effect will happen twice. Hygienic macro technology does not solve that problem (because no automatic technology can reliably guess whether that is the intended behavior), but block-structured expressions make it easy for programmers to arrange to evaluate each of the substituted expressions only once:

```
(define-syntax discriminant
  (syntax-rules ()
   ((discriminant a b c)
    (let ((temp b))
      (- (* temp temp) (* 4 a c))))))
```

On the other hand, naïve expansion of macros that introduce local variables can capture occurrences of those variables that happen to occur free within a use of the macro. As the `or` macro illustrates, that capturing problem is solved reliably and automatically by hygienic macro technology.

1.2 Recognizing the Problems

The problems mentioned above were first recognized in the 1960s.

Cheatham [1966] identified three places at which macro processing might be added to a compiler:

- *Preceding Lexical Analysis* [. . .]
- *During Syntactic Analysis*—allowing the introduction of new syntactic structure and the corresponding semantic structure into the language.
- *Following Syntactic Analysis*—particularly useful for introducing "open coding" for various procedures and functions, such as mapping functions for array elements.

If macro processing precedes lexical analysis, it will fail to respect the integrity of lexical tokens.

If macro processing is performed during syntactic analysis, as may be necessary for some designs in languages that use conventional syntax, it must be done carefully lest it fail to respect the structure of expressions.

To perform macro processing following syntactic analysis, Cheatham identified three requirements:

- syntax for defining and calling macros
- a mechanism for "recording" macro definitions during parsing
- a mechanism for expanding macro calls that are encountered during parsing

Cheatham suggested macro definitions be restricted to the beginning of a program or block. More significantly, Cheatham proposed to support two different kinds of macros. With the first kind of macro, calls to the macro would be marked by a special character. With the second kind of macro, calls would look like ordinary statements or declarations; macros of this second kind would declare their syntactic category (e.g., statement, declaration, type).

At the same time, Leavenworth [1966] identified the capturing problem while proposing "a new formalism called a syntax-macro" that "allows one to extend the syntax and semantics of a given high-level base language":

> There is a possible danger of conflict of identifiers after the macro has been expanded, for example if any of the expressions used in a call on the **sum** function would refer to a free variable named *t*. This conflict is avoided if identifiers and labels are qualified by their block numbers in an internal representation while the macros are being scanned.

Attaching that additional information to identifiers is part of hygienic macro technology. As will be seen, however, obtaining and using that information appropriately is not entirely trivial.

A few years later, MacLaren [1969] described a macro system that allowed programmers to describe transformations on abstract syntax trees, as in Lisp:

> Macro processing takes place after syntactic analysis and concurrent with the processing of declarations and consequent interpretation of expressions. This means that the programmer is freed from concern with the details of syntactic analysis. . . .
>
> Macros ultimately expand not into source language phrases but rather into phrases in an abstract object language. In this language, declarations and macro language elements do not occur, and most identifiers are replaced by abstract entities such as variables and labels, which were introduced by declaration.

MacLaren gave an example of the capturing problem identified by Leavenworth and stated a more detailed solution:

> Of course there is nothing essentially new about this problem. Most macro systems provide some sort of method for generating unique local identifiers; and, I am told, the original Lisp interpreter had in it an atom named something like "VERY LONG NAME NOT LIKELY TO BE USED ANYWHERE ELSE." Here the block structure has just complicated the old problem. Fortunately, the localization of identifier binding introduced by the block structure, is also a basis for the problem's solution. All we need to do is apply the principal [*sic*] that *as soon as applicable when the binding of an identifier is known, the identifier should be replaced by the identifier to which it is bound. . . .*

> Naturally, some language is required to indicate when replacement is to occur, and care must be taken to insure [*sic*] correct results.

Evidently MacLaren expected programmers to indicate which identifiers should be renamed to avoid capturing, and realized that responsibility would require some care.

Leavenworth and MacLaren apparently did not implement their ideas for solving the capturing problem.

2 MACROS IN LISP BEFORE HYGIENE

The early history of macros in Lisp has already been covered by a previous paper in this conference [Steele and Gabriel 1993a]. In this section we mention only a few of the high points, giving special attention to matters connected to the later development of hygienic macro technology.

Lisp's "Cambridge Polish" syntax—fully parenthesized, with prefix notation for all syntaxes, operators, and function calls—made Lisp programs look like sequences of lists, which were the data structure that gave Lisp its name. Because programs looked just like data, programs could be pre-parsed by the same `read` procedure that was used to parse data structures read from files or interactive inputs, resulting in an "S-expression" (short for "symbolic expression") representation of the program's abstract syntax trees.

The Cambridge Polish, S-expression notation for programs was at first regarded as a temporary expedient, to be replaced by a more conventional M-expression syntax [McCarthy et al. 1962; McCarthy 1981]. That replacement never happened, despite multiple attempts [Steele and Gabriel 1993a]. The S-expression notation was just too convenient for interpreters, compilers, and other programs that manipulate programs. As Lisp programmers came to recognize the ability to manipulate programs as one of the strengths of Lisp, the S-expression notation that made that manipulation so easy became Lisp's most recognizable (and widely mocked) feature among the programming community at large.

That S-expression notation had profound influence on the development of macro technology in Lisp. At a time when most macro processors operated on strings of characters or linear sequences of tokens, Lisp macros could transform abstract syntax trees into abstract syntax trees:

$$\text{source code} \xrightarrow{\texttt{read}} \text{AST} \xrightarrow{\text{macro expansion}} \text{AST}$$

The abstract syntax trees that are produced by macro expansion do not have to be represented in exactly the same way as the abstract syntax trees that serve as inputs. In most dialects of Lisp they were represented the same way, but the freedom to design a representation for the output that differed from that of the input would eventually be exploited by hygienic macro technology. For that part of the story, see Section 9.

Macros had been introduced into Lisp by 1963 [Hart 1963], and became popular in all major dialects [Teitelman 1974; Pitman 1983; Weinreb and Moon 1981; Steele 1990]. Many details differed between dialects and over time, eventually converging toward the Common Lisp macro system.

In some of the dialects that preceded Common Lisp, the `push` macro whose use is illustrated by the Scheme code shown in Figure 1 could be defined somewhat like this:

```
(defmacro push (item stk)
  `(setq ,stk (cons ,item ,stk)))))
```

The two arguments to the macro call, `item` and `stk`, are passed separately. The body of the macro definition uses backquote (`` ` ``), known in Scheme as quasiquote, to describe a list whose unvarying parts are written normally and whose varying parts are preceded by a comma (known as unquote) to indicate the expression following the comma is to be evaluated and its result inserted into the list. The macro expander uses that procedural specification of the macro's syntactic transformation to

```
(define stk '())
(push 1 stk)
(push 2 stk)
(push 3 stk)
(push 4 stk)
(push 5 stk)    ; stk is now the list (5 4 3 2 1)
```

Fig. 1. Use of a trivial push macro.

expand any use of the push macro into macro-expanded code that uses cons to construct a longer list and setq to assign that list to the stk variable.

In Common Lisp, however, that definition is not reliable, since Common Lisp already has a push macro, which is far more general than our definition above. If push could be redefined, or even bound locally using Common Lisp's macrolet, code that depended on the original push macro might break, just as the trivial push macro shown above would break if the macro were used within the scope of a labels or flet or macrolet binding of setq or cons.

Common Lisp was created to codify, as best as could be done at that time, the common or nearly common parts of existing implementations whose heritage could be traced to MacLisp. These implementations may have responded to such redefinitions in a variety of different ways. Clinger raised this issue during standardization of Common Lisp; it was written up as LISP-SYMBOL-REDEFINITION [Masinter et al. 1990]. That issue was resolved by saying the "consequences are undefined" whenever a symbol exported from the COMMON-LISP package is defined or bound as a macro, function, variable, or structure [Masinter et al. 1990]. Hence the defmacro above might have consequences that would vary from one implementation to another. In Common Lisp, that resolution effectively assigns responsibility for avoiding the breakage to programmers who might otherwise be tempted to re-bind or re-define any of the 978 external symbols contained within the COMMON-LISP package, without making implementations of Common Lisp responsible for detecting or warning about the re-binding or re-defining.

(As noted in the rationale for LISP-SYMBOL-REDEFINITION, "programs can redefine functions safely by creating new symbols in their own package, possibly shadowing the name" [Masinter et al. 1990]. For the purposes of this paper, we can describe the Common Lisp package system as a way to control the visibility of symbols by manipulating the lexical analyzer's symbol table.)

That capturing problem had already been used as a *post hoc* rationalization for using separate environments for functions and variables, a tradition that had begun with Lisp 1.5 and had continued in almost all subsequent dialects of Lisp, including Common Lisp [Gabriel and Pitman 1988]. With a unified environment, Lisp's list procedure would be shadowed by any local variables named list, which happens to be a popular name for local variables in Lisp. With Common Lisp's non-hygienic macro system, that shadowing would break any uses of macros that call the list procedure. Users of those macros would often not even realize a macro calls list, so that kind of bug is both mysterious and hard to track down. Using separate environments for functions and variables does not really solve the problem, but it does reduce the frequency of bugs caused by the capturing problem to a level that could more easily be ignored.

On the other hand, using different scope rules for functions and variables made the use of higher-order functions unnecessarily awkward. Although the inventor of Lisp was inspired by the functional notation of lambda calculus [McCarthy 1978, 1981], many Lisp programmers regarded higher-order programming as an eccentric or unreliable idiom[1] until Scheme returned to a unified environment with lexical scoping as in the lambda calculus [Sussman and Steele 1975; Steele and Sussman 1978].

These examples show how the capturing problem has affected both the fine print and the fundamental design of a programming language.

Lisp programmers became accustomed to the contortions needed to work around the capturing problem, and wore them proudly. As noted by Weinberg [1985], this phenomenon is not unique to Lisp:

> We do not know whether the language designers are geniuses, or we ordinary programmers are cripples. Generally speaking, we only know how bad our present programming language is when we finally overcome the psychological barriers and learn a new one.

3 MACROS IN SCHEME BEFORE HYGIENE

The Scheme programming language was designed and implemented in 1975 by Guy L Steele Jr and Gerald J Sussman. Originally conceived as a sequential implementation of Carl Hewitt's Actor model [Steele and Gabriel 1993a; Clinger 2008], Scheme differed from most other dialects of Lisp in these ways:

- proper tail recursion (so procedure calls do not push stack unnecessarily)
- first-class continuations (from which all sequential control structures known at that time could be synthesized)
- lexical scoping (as in lambda calculus, Algol, and many other block-structured languages)
- first-class procedures (as in lambda calculus)
- uniform evaluation (as in lambda calculus)
- unified environment (as in lambda calculus)

In consequence of those last four features, Scheme is a *higher order language* in the sense of lambda calculus: it is easy to create new procedures at run time whose behavior depends on the free variables of their code. Most programming languages are first order (e.g., C) in the sense that new procedures cannot be created at run time, or essentially second order (e.g., Java and Common Lisp) because, although new function objects can be created at run time, they are referenced and/or called using syntax that differs from the usual syntax for variable references and method/function calls.

As will be explained below, it was not easy to add naïve macros to Scheme without compromising its status as a higher order language. That conflict helped to motivate the development of hygienic macro technology, which resolved the conflict, and partly explains why hygienic macro technology was first developed for Scheme.

Sussman and Steele published the original report on Scheme in December 1975, and a revised report in January 1978 [Sussman and Steele 1975; Steele and Sussman 1978]. For his master's thesis, Steele wrote a compiler [Steele 1977].

Scheme languished from 1978 to 1981, as Steele focussed on finishing his PhD. By 1982, faculty at three universities had begun to use Scheme as a teaching language: Gerry Sussman and Hal Abelson (MIT), Paul Hudak (Yale), and Mitch Wand, Dan Friedman, and Will Clinger (Indiana University). An unofficial process of standardization began in 1984 when Mitch Wand chaired the

[1]For example [Weizenbaum 1968]: "…The problem remains ill understood. Bobrow, for example, published a faulty solution for it only recently." See also [Bobrow and Murphy 1967; Moses 1970]. As late as 1975, when Clinger reported what he assumed to be a bug in MacLisp, one of that system's implementors explained that the unreliability of higher-order programming in MacLisp (caused by dynamic scoping) was a feature, not a bug.

meeting at Brandeis University that will be described in Section 3.4. That process has produced a series of reports and one IEEE standard:

- The revised revised report on Scheme, or an uncommon Lisp (RRRS) [Clinger, Abelson, Adams, Bartley, Brooks, Friedman, Halstead, Hanson, Haynes, Kohlbecker, Oxley, Pitman, Rees, Rozas, Sussman, and Wand 1985]
- The revised^3 report on the algorithmic language Scheme (R3RS) [Rees and Clinger 1986]
- The revised^4 report on the algorithmic language Scheme (R4RS) [Clinger and Rees 1991b]
- IEEE Standard for the Scheme Programming Language (IEEE Std 1178-1990) [Bartley, Hanson, and Miller 1991]
- The revised^5 report on the algorithmic language Scheme (R5RS) [Kelsey et al. 1998]
- The revised^6 report on the algorithmic language Scheme (R6RS) [Sperber, Dybvig, Flatt, and van Straaten 2009a]
- The revised^7 report on the algorithmic language Scheme (R7RS) [Shinn, Cowan, and Gleckler 2013]

Those reports record a bare outline of the development and adoption of hygienic macro technology within Scheme. In this section, we discuss the history of naïve macro technology in Scheme, from the original report of 1975 up until the 1984 meeting at Brandeis.

3.1 Why Macros Matter

From its inception, the design of Scheme showed how a sufficiently simple and powerful programming language could rely on macros to define most of the syntaxes that are built into more complicated and less powerful languages. That set the stage for greater reliance upon macros than had been traditional, even in Lisp.

That design philosophy permeated the early reports and papers on Scheme written by Steele and Sussman, but got lost in the overly timid and prosaic introduction Clinger wrote for the second revised report [Clinger et al. 1985]. Sussman wanted a more ideological manifesto that would not only describe how different Scheme is from most other programming languages, but also explain why that difference is important. Clinger responded with these sentences, which became the opening paragraph of the third revised report and all subsequent reports [Rees and Clinger 1986]:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

That was not just empty talk. From Steele's Rabbit compiler to the present day, most Scheme compilers rely on macros to translate all source code into a very small core language understood by the compiler. As Steele explained, this approach can produce a notably robust compiler that generates efficient code even for source programs that use syntaxes the compiler writer never anticipated [Steele 1977]:

> Rather than having specialized knowledge about a large variety of control and environment constructs, the compiler handles only a small basis set which reflects the semantics of lambda-calculus. All of the traditional imperative constructs, such as sequencing, assignment, looping, GOTO, as well as many standard LISP constructs such as AND, OR, and COND, are expressed as macros in terms of the applicative basis set. A small number of optimization techniques, coupled

with the treatment of function calls as GOTO statements, serve to produce code as good as that produced by more traditional compilers. The macro approach enables speedy implementation of new constructs as desired without sacrificing efficiency in the generated code.

For that approach to work, macros have to be both powerful and completely reliable. Naïve macros are powerful, but not completely reliable.

3.2 Working Around Limitations of Naïve Macros

The original and first revised reports described Scheme as a language with Lisp's usual naïve macro expansion.

As part of his master's thesis, Guy Steele [1977] showed how Scheme's higher order procedures could be used to alleviate (but not entirely eliminate) the capturing problem illustrated by the `OR` macro of Section 1. Quoting Steele:

A simple-minded approach to `OR` would be:

```
(OR)           =>  'NIL
(OR x . rest)  =>  (IF x x (OR . rest))
```

There is an objection to this, which is that the code for `x` is duplicated. Not only does this consume extra space, but it can execute erroneously if `x` has any side-effects. We must arrange to evaluate `x` only once, and then test its value:

```
(OR)           =>  'NIL
(OR x . rest)  =>  ((LAMBDA (V) (IF V V (OR . rest))) x)
```

This certainly evaluates `x` only once, but admits a possible naming conflict between the variable `V` and any variables used by `rest`. This is avoided by the same technique used for `BLOCK`:

```
(OR)           =>  'NIL
(OR x . rest)  =>  ((LAMBDA (V R) (IF V V (R)))
                     x
                     (LAMBDA () (OR . rest)))
```

Each of the `LAMBDA` expressions evaluates to a *closure*: a procedure that remembers what its free variables meant in the context of the lambda expression, and uses those meanings for the free variables even if the procedure is called in a context where those variables are bound to different values. A zero-argument procedure or closure is called a *thunk*, so Steele's technique for ameliorating the capturing problem is an example of *thunkification*, which is often used for the unrelated purpose of achieving call-by-name semantics in call-by-value languages [Amtoft 1993; Steckler and Wand 1994].

Steele did not consider the possibility that `LAMBDA` or `IF` might have a nonstandard meaning in the context of the macro use, so this was only a partial solution to the capturing problem. Even if those keywords were reserved, thunkification would not avoid unintended captures of procedure and variable names (such as the reference to `cons` in the `push` macro shown earlier). Despite its limitations, thunkification would continue to be the state of the art until Kohlbecker's algorithm debuted in 1986 [Kohlbecker 1986; Kohlbecker et al. 1986].

Alternatively, the author of a macro can arrange for each use of the macro to generate fresh identifiers, known as *gensyms*, for all local variables that need to be bound in macros such as `OR`. This workaround has the same limitations as thunkification, but is useful for languages in which higher-order procedures (closures) are unavailable or unlikely to be optimized reliably. Section 15 shows an example that uses gensyms to define a variation of the `OR` macro in Clojure.

3.3 Development of Common Lisp

Circa 1980, the major implementations of Lisp included Interlisp (at Xerox and BBN), Portable Standard Lisp (University of Utah), and a variety of implementations descended from MIT's MacLisp: Franz Lisp (Berkeley), Lisp Machine Lisp (LMI, Symbolics), NIL (MIT), and Spice Lisp (CMU).

Under pressure from ARPA to consolidate those efforts, users and implementors of those systems met at SRI in April 1981, where Scott Fahlman defended the proliferation of dialects by saying "the MacLisp community is *not* in a state of chaos. It consists of four well-defined groups going in four well-defined directions" [Steele and Gabriel 1993a].

Following that meeting, the post-MacLisp community coöperated to design a common language that became known as Common Lisp [NA Ohlander 1984; Steele et al. 1984; Steele 1990].

The story of Common Lisp will be continued in Section 5.

3.4 Workshop at Brandeis

Although Scheme had some influence on Common Lisp's adoption of lexical scoping as the default, Scheme itself escaped the push to standardize on Common Lisp, mainly because Scheme was considered a toy language, useful only for research and teaching, too unimportant and too different from MacLisp and its successors to warrant consideration when the effort to design a Common Lisp began in 1981.

By 1982, however, variants of Scheme were being used in courses at MIT, Yale, and Indiana University. As dialects in use at those universities diverged, teachers and researchers began to have trouble writing Scheme code for publications and grant proposals that would be understood by other teachers and researchers, even if those readers were familiar with Scheme as it had been defined in 1975 and 1978.

At the 1982 Lisp Conference [Park et al. 1982], the first day ended with four talks on implementation (of Interlisp, Portable Standard Lisp, S-1 Common Lisp, and Scheme) in a session at which Guy Steele [1982] also presented "An Overview of Common Lisp." Some of the reasons Steele gave for defining a Common Lisp would also apply to Scheme. Realizing that the small community of Scheme users could ill afford further fragmentation, a small huddle of Schemers agreed to work toward defining a common core of the language they were implementing and using. No written record was made of this impromptu get-together at the side of the conference hall, but the group included Clinger and Wand, probably included Norman Adams, Dan Friedman, Paul Hudak, Jonathan Rees, and Gerry Sussman, and may have included Dick Gabriel, Kent Pitman, Guy Steele, and others.

3.4.1 Invitations to the Workshop. Two years later, Jonathan Rees set up a mailing list for that purpose. In the first substantive message to that list, on 10 October 1984, the Brandeis workshop was announced by Mitchell Wand [1984b; quoted in Figure 2].

Hanson "(firmly)" requested that Bill Rozas be invited, and it was done [Hanson 1984]. Kent Dybvig, a PhD student at UNC Chapel Hill, was also invited but would not be able to attend [Wand 1984a].

Wand, of Indiana University but on sabbatical at Brandeis, presided as chair during the two days of the meeting. Except for Dybvig, Gabriel, Pitman, and Steele, all of those who were invited attended the Brandeis meeting and were listed as authors of the subsequent report [Clinger et al. 1985]. Although Pitman was unable to attend, he contributed to the agenda and was listed as an author. Dybvig's contributions to the electronic discussion would be recognized by listing him as an author of the third revised report [Rees and Clinger 1986].

```
        WORKSHOP ON THE PROGRAMMING LANGUAGE SCHEME

                October 22-23, 1984
                Brandeis University
                Waltham, MA

General Chairman:        Dan Friedman, Indiana
Agenda Committee:        Will Clinger, Indiana (chair)
                         Chris [Hanson], MIT
                         Gary Brooks, Indiana
                         Jonathan Rees, Yale/Dec/MIT
                         David Bartley, TI
Local Arrangements:      Mitch Wand, Indiana/Brandeis/MIT

The purpose of this workshop is to identify those features of Scheme
which may be made common among various implementations, to discuss
the nature and format of interim standards for Scheme, and to expose
some of the problems which may make other features difficult to
standardize.

[. . . ]

Attendance:  Attendance is by invitation only, as this is a working
meeting.  The following is the current invitation list

        Abelson (MIT)
        Adams (Yale)
        Bartley (TI)
        Brooks (IU/TI)
        Clinger (IU)
        Friedman (IU)
        Gabriel, Dick (Stanford)
        Halstead, Bert (MIT)
        Hanson (MIT)
        Haynes (IU)
        Kohlbecker (IU)
        Oxley (TI)
        Pitman (MIT)
        Rees (MIT/Yale)
        Steele (Tartan Labs)
        Sussman (MIT)
        Wand (IU/Brandeis)

If you are not on this list, please accept the apologies of the
organizers.
```

Fig. 2. Invitation to attend the workshop at Brandeis [Wand 1984b].

That is how the "RRRS authors" group came to be. That self-selected group of authors would grow slowly over the next decade, even as its occasional meetings were opened to the wider community of Scheme users, implementors, teachers, and researchers.

3.4.2 Implementations Represented at Brandeis. In 1984, when the RRRS authors met at Brandeis, Gerry Sussman and Hal Abelson were using Scheme to teach MIT's introductory course in computer science, and were writing a textbook for that course [Abelson et al. 1985]. At that time, the most important implementations of Scheme were

- T (Yale)
- MIT Scheme
- Scheme84 (Indiana University)
- Scheme 312 (Will Clinger)
- PC Scheme (Texas Instruments)
- Chez Scheme (Kent Dybvig)

T, implemented under Professor Paul Hudak's supervision by a team that included Jonathan Rees and PhD students Norman Adams, David Kranz, and Richard Kelsey, was based upon a sophisticated optimizing compiler named Orbit (to rhyme with Steele's Rabbit compiler) [Rees (no date); Shivers (no date); Rees and Adams 1982; Adams et al. 1986; NA Rees et al. 1984; Kranz 1988; Kelsey 1989; Kelsey and Hudak 1989].

MIT Scheme, written by Chris Hanson, Bill Rozas, and others, would be released to the public in 1986. It was implemented as a byte-code interpreter (written in Motorola 68000 assembly language) for the HP 9836 and (later) as a compiler and interpreter for most x86 machines [NA MIT 1981; Hanson 1986-2019].

In 1984, Scheme84 was mostly a renaming of Scheme 311, which was a Scheme interpreter written in Franz Lisp by Will Clinger for the undergraduate programming languages course at Indiana University, using ideas from meta-circular interpreters designed by Dan Friedman [NA Foderaro 1980; Fessenden et al. 1983; Friedman et al. 1984]. By January 1985, however, only three months after the Brandeis meeting, Scheme 84 would support hygienic procedural macros, which were implemented using Kohlbecker's algorithm as described in Section 4 below [NA Friedman et al. 1985].

Scheme 312, written by Will Clinger, was a byte-code interpreter for the Motorola 68000. MIT gave Clinger permission to incorporate MIT Scheme's bignum code for calculations on very large integers. John Ulrich added an editor and user interface for the Apple Macintosh, and their company would release MacScheme in 1985 as a commercial product [NA Lightship Software 1985-1990]; in 1986, Clinger would add a compatible native code compiler.

PC Scheme, written by members of the Computer Science Lab at Texas Instruments, was a byte-code interpreter for the x86 [NA Texas Instruments 1990, 1987]. MIT gave TI permission to incorporate Edwin, an Emacs-like editor written (in Scheme) by Chris Hanson. PC Scheme would be released in 1985 as a commercial product, and would remain the most widely used implementation of Scheme for several years thereafter.

In February of 1985, three months after the Brandeis meeting, Kent Dybvig announced Chez Scheme, a native code compiler for the VAX [Dybvig 1985]. Dybvig would go on to write code generators for the x86, SPARC, and PowerPC. Marketed as a commercial product by Dybvig's Cadence Systems, Chez Scheme immediately became known as one of the fastest implementations of Scheme, and has retained that reputation to the present day.

Dybvig had studied at Indiana University as an undergraduate and master's student, receiving his MS in 1983, and wrote Chez Scheme while he was a PhD student at the University of North Carolina at Chapel Hill. He then joined the faculty of Indiana University, where he served until leaving for Cisco in 2011. In 2006, the ACM recognized Dybvig's development of Chez Scheme by naming him a Distinguished Engineer.

3.4.3 Workshop Agenda. Clinger prepared the agenda for the Brandeis meeting, allocating 30 minutes for discussion of macros and asking "Does anybody have a syntax and semantics for macros that is good enough and stable enough that we should consider committing ourselves to it for a period measured in years?" [Clinger 1984b,a]. Subsequent discussion via the mailing list recorded considerable controversy [Clinger 1984c; Brooks 1984]:

- TI wanted "to distinguish syntactic elements (macros) from ordinary names (variables) while using the same lexical rules to construct instances of each." What that means is not entirely clear, but TI appears to have been assuming separate environments ("namespaces") would be used for macro names and for variable names.
- As Gary Brooks pointed out, "The crux of the problem. . . is the proliferation of environments. For example, if we examine Scheme84 we find that we have a (core) reserved word environment, a global macro environment, a global primitive environment, a global environment for everything else, a lexical environment, and a fluid environment."
- That proliferation of environments would make it necessary to invent disambiguation rules, such as preferring the reserved word and macro name environments in operator positions while preferring the lexical and global variable environments in operand positions.
- As Brooks stated: "The complicated and unintuitive scope rules resulting from position sensitive precedence is intolerable." (Although similarly complex scope rules had already been proposed and adopted in Common Lisp, the simple scope rules that result from using a single environment were essential if Scheme were to retain its status as a higher order language whose core semantics could accurately be described as "call-by-value lambda calculus.")
- In pure lambda calculus, the situation is simpler because *all* names refer to variables, and the syntax of compound expressions is determined entirely by weird characters: parentheses and the Greek letter λ.
- But there aren't enough ASCII characters to assign a separate weird character to every syntax for which a programmer might want to define a macro. In both Lisp and Scheme, therefore, the syntax of a compound expression is determined by the expression that would be in its operator (leftmost) position if the expression were regarded as an application (procedure call). If that expression is an identifier naming a predefined or macro syntax, the syntax and semantics of the entire expression is determined accordingly. Otherwise the expression is an application.
- Scheme is a very dynamic language. What should happen if, say, the predefined `let` syntax were redefined to make it the name of an ordinary procedure? What should happen if `let` were redefined as an altogether different macro?
- While it might seem desirable to outlaw such redefinitions, programmers sometimes have legitimate reasons to redefine syntaxes. For example: Every procedure that is defined or created by a Scheme program is created by the `lambda` syntax. It is therefore easy to add instrumention to every procedure by redefining `lambda`, and this has often been done by debuggers, profilers, and researchers such as ourselves.
- Finally, Brooks cited both of the capturing problems we have explained in Section 1. Brooks gave examples of a macro that refers to the name of a special form being used in a context where that name had been bound as a local variable, and used the `or` macro as an example of a macro that introduces local variable bindings that might inadvertently capture references within a use of the macro.

All of those issues would eventually be resolved by hygienic macros and module systems based on hygienic macro technology, but hygienic macro technology would not be published until 1986. Even then, it would take more than a decade for hygienic macro technology to be refined and accepted, and it would take yet another decade of design before module systems based on that technology would be standardized.

As of the Brandeis meeting, the capturing problem had led the implementers of MIT Scheme to eliminate all official support for macros [NA MIT 1981; Clinger 1984a]. Most implementations supported global macro definitions with naïve macro expansion, relying on the programmer to avoid inadvertent captures via thunkification. T supported local macro definitions in addition to global macros [Rees (no date); Shivers (no date); Rees and Adams 1982; Adams et al. 1986; NA Rees et al. 1984; Kranz 1988; Kelsey 1989; Kelsey and Hudak 1989].

To alleviate the capturing problem, several members of the agenda committee suggested it should be illegal or impossible to shadow macro names or the names of core syntaxes [NA Bartley 1984a; Clinger 1984b; Hanson 1984; Pitman 1984]. In particular, Texas Instruments argued for a combination of reserved words and putting macro names in a lexical environment that would be distinct from the lexical environment used for lambda-bound variables [NA Bartley 1984b]. (That position statement is similar but not identical to the position statement of 16 October 1984 quoted by Clinger [1984c].) The final agenda for the meeting asked the following questions [NA Clinger 1984c], which were answered as shown in italics [NA Clinger 1984d].

- Should the keywords of special forms have global or lexical scope? *Lexical.*
- Should the keywords of special forms be shadowed by lambda-binding? *We don't know enough yet to answer this.*
- Should the keywords of special forms be shadowed by local syntax definitions? *Yes. [This raises a problem with macros that rely on other macros. Several incompatible solutions were discussed.]*

3.5 Outcome of Workshop

The Brandeis workshop led to a Revised Revised Report, which summarized the situation as follows [Clinger 1985; Clinger et al. 1985]:

> *Scheme does not have any standard facility for defining new special forms.*
>
> *Rationale:* The ability to define new special forms creates numerous problems. All current implementations of Scheme have macro facilities that solve those problems to one degree or another, but the solutions are quite different and it isn't clear at this time which solution is best, or indeed whether any of the solutions are truly adequate. Rather than standardize, we are encouraging implementations to experiment with different solutions.
>
> The main problems with traditional macros are: They must be defined to the system before any code using them is loaded; this is a common source of obscure bugs. They are usually global; macros can be made to follow lexical scope rules as in Common Lisp's `macrolet`, but many people find the resulting scope rules confusing. Unless they are written very carefully, macros are vulnerable to inadvertant [*sic*] capture of free variables; to get around this, for example, macros may have to generate code in which procedure values appear as quoted constants. There is a similar problem with keywords if the keywords of special forms are not reserved. If keywords are reserved, then either macros introduce new reserved words, invalidating old code, or else special forms defined by the programmer do not have the same status as special forms defined by the system.

In a revision of that report, R3RS Section 7.3 gave rewrite rules for several derived expression types, including these three rules for or expressions [Rees and Clinger 1986]:

```
(or)            ≡   #f
(or <test>)     ≡   <test>
(or <test1> <test2> ...)
  ≡  (let ((x <test1>)
             (thunk (lambda () (or <test2> ...))))
         (if x x (thunk)))
```

The third of those rules uses Steele's thunkification trick to prevent the binding introduced for x from capturing an unrelated reference to a variable x that might occur free within <test2>. As was noted in Section 3.2, naïve rewriting of the third rule would not be reliable, even with its thunkification, if the let, lambda, and if keywords can be bound locally. The R3RS therefore assumed syntactic keywords were reserved [Rees and Clinger 1986]:

> Some implementations allow arbitrary syntactic keywords to be used as variable names, instead of reserving them, as this report would have it.

This unsatisfactory state of affairs had motivated the development of hygienic macro technology.

That development began in 1983, but was not published until 1986. In the RRRS authors' email archives, the first reference to hygienic macro technology appears in the minutes of a lunchtime meeting at the 1986 ACM Conference on Lisp and Functional Programming [Clinger 1986]:

```
Attendance was not taken.

Jonathan Rees was congratulated on getting a draft of the R^3RS done in
time for the conference.

After lunch we drew up a list of language areas that need work. . . .

MACROS.  Difficult research area.  We're awaiting Eugene Kohlbecker's
thesis.
```

4 KOHLBECKER'S ALGORITHM

We now consider *Kohlbecker's algorithm*, which seems to have been the first fully reliable technology that (1) gave programmers the power to avoid inadvertent capturing and (2) was actually implemented.

By the mid-1980s, over half a dozen members of Indiana University's department of computer science were publishing research on new macro technologies, including the design and use of pattern languages to specify syntactic transformations, and were inventing new idioms and styles for writing sophisticated macros [Kohlbecker, Friedman, Felleisen, and Duba 1986; Kohlbecker and Wand 1987; Dybvig, Friedman, and Haynes 1986, 1988]. Those developments led to greater reliance upon macros at Indiana University, leading to greater awareness of the capturing problem and its importance.

Eugene Kohlbecker, a PhD student in the department, spent four years working on syntactic extensions in Lisp and Scheme, completing his dissertation in 1986 under the direction of Dan Friedman [Kohlbecker et al. 1986; Kohlbecker 1986]. Section 4.8 quotes Friedman's recollections of those years.

4.1 Kohlbecker's Terminology

In a draft "Position Statement on Macros", Kohlbecker defined some of the terms he would use in his dissertation almost two years later [NA Kohlbecker 1984]:

> I term a particular use of a macro a *macro expression*. However, on occasion I shall also refer to this as a macro *call*. Old habits are hard to break. The lexical structure surrounding a macro expression is the *context* of use of the macro. When a macro is *expanded*, that is, processed by compiler, the transform function is "applied" to macro expression [*sic*]. The result of that application is another expression called the *expansion* that replaces the macro expression in its context.

In subsequent papers by Kohlbecker and others, that "transform function" is often called a *syntax transformer*. Syntax transformers that are defined by code written in Scheme or some other general purpose programming language are said to be *procedural* or *low-level* syntax transformers. Syntax transformers that are defined by rules expressed using a specialized pattern language are often called *high-level* syntax transformers, and are said to define high-level macros.

4.2 Goals and Design Decisions

Kohlbecker began his dissertation by identifying and proposing to solve five problems with the macro systems that had been available to Lisp programmers:

(1) Macros are hard to read and write.
(2) Macro definitions give "little explicit information about the form macro calls are to take."
(3) "Syntactic checking of macro calls is usually ignored."
(4) "The notion of a macro binding for an identifier" obscures "what macros really should be."
(5) The power of procedural macros allowed programmers to write macros that are not just syntactic abstractions.

Finally, almost as an afterthought, Kohlbecker mentioned the capturing problem:

> Furthermore, the conventional algorithm used for the expansion of macro calls within Lisp often causes the inadvertent capture of an identifier appearing within the macro call by a macro-generated, binding instance of the same identifier. Lisp programmers have developed a few techniques for avoiding this problem, but they all have depended upon the macro writer taking some sort of special preventative action.

After surveying the history of macro processing, both generally and in Lisp, Kohlbecker listed 12 general design principles, which he italicized and numbered in boldface. Of those twelve, these three seem most important:

> **Principle 2.** *Macro expansion should take place late enough during language processing so that the macro calls replace syntactic entities rather than strings of characters.*
>
> **Principle 6.** *The macro writer should perceive the set of syntactic extensions as existing in tabular form. Each entry in the table is a production. In each production, the left-hand side specifies a macro call, and the right-hand side specifies the call's transcription.*
>
> **Principle 10.** *In order to preserve program semantics, the complete expansion of syntactic extensions should take place prior to program interpretation or code generation.*

Kohlbecker then stated 10 specific design decisions; we quote and comment upon six of them.

> 1. The syntactic extension mechanism is expression oriented. There is no means of declaring syntactic extensions in any other syntactic domain.

That decision was made feasible by the fact that Lisp and Scheme are expression-oriented languages. As it turned out, however, his system allowed extensions in the syntactic domain of definitions as well as expressions, which could be viewed as a violation of his **Principle 3**: *Given a syntactic domain X, syntactic extensions to domain X should only be used in a program context that allows any term from domain X.*

> 5. Macro calls have no fixed delimiter structure. The pattern variables of the syntax table patterns serve as formal parameters.

In stating this decision, Kohlbecker failed to acknowledge the role of Cambridge Polish notation. The fully parenthesized nature of Lisp and Scheme, combined with the use of a reader that parses the character-level representation of a program into an uninterpreted abstract syntax tree known as an S-expression, already imposes a fixed delimiter structure that determines the beginning and end of each macro call, along with the beginning and end of every structured subcomponent of the call that can match a pattern variable. With conventional syntax, delimiting macro uses and arguments can be a real issue. See Section 16 for some approaches that have been taken.

> 6. Macro calls are expanded by-name.

In Kohlbecker's terminology, "by-name" corresponds to performing macro expansion outside-in rather than inside-out.

> 7. The expansion of a syntactic extension is not naïve. However, provision must be made for the capturing of free identifiers.

For our history of hygienic macro technology, this was the key design decision. It was implemented by the algorithm we describe below.

> 8. The set of keywords is disjoint from the set of identifiers.

That rules out local macros, whose definition converts the name of a local macro from an identifier into a keyword. This decision was reversed in later standards for Scheme, but the status of syntactic keywords such as `else` and the field names of records has remained somewhat controversial, partly because it is hard to design a semantics for keywords that is adequately convenient without creating loopholes that can be exploited, as shown in Section 8.1; see also Section 14.

> 10. Macro definitions are made so that the macro writer does not have to compose [a syntactic transform function].

Kohlbecker envisioned a system that would compile pattern-based specifications of macros into procedural syntax transformers. That aspect of his design made it compatible with Common Lisp and other languages that rely on procedural specifications of macros.

4.3 How It Worked

As originally implemented, Kohlbecker's algorithm worked only with macros defined by procedural syntax transformers, as in Common Lisp [Kohlbecker 1986; Kohlbecker et al. 1986]. The "Macro-by-example" paper Kohlbecker wrote with Wand does not mention hygiene, which they regarded as orthogonal to the pattern language described in that paper [Kohlbecker and Wand 1987]. To simplify our exposition, however, we will use Kohlbecker's pattern language (including the ellipsis notation seen in the example below) to specify the examples given in this section, relying on the Kohlbecker/Wand algorithm to translate those macros into the procedural syntax transformers whose output was made hygienic by Kohlbecker's algorithm for hygienic macro expansion.

Barendregt had used the word "hygiene" in connection with the α-conversion necessary to perform correct β-conversion [Barendregt 1985, 1992]. Matthias Felleisen suggested Kohlbecker use that word as well (Section 4.8). Citing Barendregt, Kohlbecker stated the following *Hygiene Condition* [Kohlbecker 1986, page 157; Kohlbecker et al. 1986]:

> Generated identifiers that become binding instances in a fully expanded macro call must bind only identifiers that are generated during the same transcription step.

To illustrate that hygiene condition, consider the following definition of an `or` macro in Kohlbecker's system:

```
(extend-syntax (or) ()
 ((or)
  #false)
 ((or exp)
  exp)
 ((or exp1 exp2 exp3 ...)
  (let ((temp exp1))
    (if temp temp (or exp2 exp3 ...)))))
```

To explain how Kohlbecker's algorithm achieved his hygiene condition, consider the following use of that macro:

```
(let ((temp (read-temperature)))
  (if (or (not (real? temp))
          (< temp 0)
          (> temp UPPER_LIMIT))
      (display "Temperature out of range.\n")
      (process-temperature temp)))
```

Kohlbecker's algorithm began by *time-stamping* every identifier that appears within the expression. How time-stamping is accomplished is not important, so long as

- Time-stamped identifiers are *fresh* in the sense that they do not appear anywhere within the program as it existed before the time-stamped identifiers are introduced.
- The original identifier can be recovered by removing the time stamp.

Freshness can be achieved by arranging for time-stamped identifiers to use a lexical syntax that would be illegal in textual representations of a program but allowed in abstract syntax trees, while incorporating a numerical time-stamp. In this paper, we will write the time-stamped versions of an original identifier `x` as `x.0`, `x.1`, and so on.

In this example, we will mark identifiers that appear within the original expression by appending a suffix `.0`, and will obtain new time stamps every time a macro is expanded by incrementing the numeric part of that suffix. With its initial time stamps, the expression becomes

```
(let.0 ((temp.0 (read-temperature.0)))
  (if.0 (or.0 (not.0 (real?.0 temp.0))
              (<.0 temp.0 0)
              (>.0 temp.0 UPPER_LIMIT.0))
        (display.0 "Temperature out of range.\n")
        (process-temperature.0 temp.0)))
```

(Kohlbecker's algorithm would not have time-stamped the `let`, `if`, and `or` because those are syntactic keywords, which Kohlbecker regarded as disjoint from identifiers. We are taking the liberty of explaining how Kohlbecker's algorithm could be made to work when syntactic keywords are identifiers.)

The macro expander traverses the abstract syntax tree recursively, keeping track of which identifiers have been bound locally. When it encounters the `or` expression, it knows `or.0` has not been bound locally, so this `or` refers to the macro definition of `or` given earlier. We are assuming the macro system had already compiled the declarative definition of the `or` macro into a procedural syntactic transform function. That procedure is called on the abstract syntax tree for the `or` expression, and returns

```
(let ((temp (not.0 (real?.0 temp.0))))
  (if temp
      temp
      (or (<.0 temp.0 0)
          (>.0 temp.0 UPPER_LIMIT.0))))
```

Kohlbecker's algorithm then traverses that output, attaching a new time stamp to all identifiers that do not already have a time stamp. Kohlbecker refers to these as *generated identifiers*, whose meanings should be taken from their meanings at the site of macro definition. (Kohlbecker assumed all macros were defined globally, but our exposition here is setting the stage for generalizations that would come later.) Following that traversal, the expression has become

```
(let.1 ((temp.1 (not.0 (real?.0 temp.0))))
  (if.1 temp.1
        temp.1
        (or.1 (<.0 temp.0 0)
              (>.0 temp.0 UPPER_LIMIT.0))))
```

Continuing the recursive traversal, the macro expander encounters the reference to `or.1`. Because `or.1` has a non-zero time stamp, it must have been inserted by some macro expansion. Its meaning must therefore come from a local binding inserted by some previous macro expansion, or from its meaning in the macro definition. In Kohlbecker's system, keywords could not be shadowed by local bindings, and all macros were global, so its meaning must be the same as its meaning in the global environment of the expression being expanded. That implies `or.1` refers to the `or` macro. (This part of the algorithm would have to become more complicated if syntactic keywords could be shadowed by variable bindings, or if local macros and macros imported from separately compiled libraries were supported, as in the hygienic macro systems described in Sections 7, 9, and 11.)

Repeating the procedural macro expansion and time stamping, that nested use of `or` becomes

```
(let.2 ((temp.2 (<.0 temp.0 0)))
  (if.2 temp.2
        temp.2
        (or.2 (>.0 temp.0 UPPER_LIMIT.0))))
```

Continuing the recursive traversal, the macro expander encounters one last use of the `or` macro, which has only one argument so it just turns into

```
(>.0 temp.0 UPPER_LIMIT.0)
```

Putting this all together as the macro expander backs out of the recursion, we get

```
(let.0 ((temp.0 (read-temperature.0)))
  (if.0 (let.1 ((temp.1 (not.0 (real?.0 temp.0))))
          (if.1 temp.1
                temp.1
                (let.2 ((temp.2 (<.0 temp.0 0)))
                  (if.2 temp.2
                        temp.2
```

```
                                (>.0 temp.0 UPPER_LIMIT.0)))))
                (display.0 "Temperature out of range.\n")
                (process-temperature.0 temp.0)))
```

Now that the expression has been completely macro-expanded, Kohlbecker's algorithm traverses the entire macro-expanded code, keeping track of which identifiers have been bound locally and removing the time stamps from those whose time stamp is zero and from those that occur free within the expression. The time stamps that remain distinguish local variables that were introduced at different times during macro expansion. Because they were introduced at different times and do not occur free, variables with distinct time stamps could not have been intended to capture each other and should remain distinct variables, as is accomplished by allowing the time stamps to become part of their names in the macro-expanded code. The final result is

```
(let ((temp (read-temperature)))
  (if (let ((temp.1 (not (real? temp))))
        (if temp.1
            temp.1
            (let ((temp.2 (< temp 0)))
              (if temp.2
                  temp.2
                  (> temp UPPER_LIMIT)))))
      (display "Temperature out of range.\n")
      (process-temperature temp)))
```

As can be seen, Kohlbecker's hygiene condition is satisfied. The inadvertent capture of `temp` that would have occurred with naïve macro expansion has been avoided.

Time-stamping all identifiers at the beginning, even with a time stamp of 0 that will be removed at the end, allows the macro expander to distinguish identifiers that were already present in the subforms of a macro call from identifiers that were inserted during macro transciption.

4.4 Asymptotic Complexity

Because Kohlbecker's algorithm traverses the entire output of each procedural transcription, it runs in $\Omega(n^2)$ time for recursive macros such as `or` and `cond` with n clauses.

Kohlbecker's algorithm has often been said to run in quadratic time (e.g., [Clinger and Rees 1991a]), but it is easy to write artificial macros for which Kohlbecker's algorithm takes exponentially more time than naïve macro expansion or modern algorithms for hygienic macro expansion. That fact is not well known, and (so far as we know) no examples of the algorithm's exponential behavior have been published, so we give an example here:

```
(define-syntax slow-for-kohlbecker
  (syntax-rules ()
   ((slow-for-kohlbecker () x)
    0)
   ((slow-for-kohlbecker (a b ...) x)
    (slow-for-kohlbecker (b ...) (list x x)))))
```

Let $E_n \equiv$ `(slow-for-kohlbecker (1 ...` n`) '(a))`. E_n macro-expands to `0`, which will then be compiled or interpreted in constant time. In modern implementations of Scheme, E_n will macro-expand to `0` in linear time (because lists that match the pattern variable `x` would never be copied or traversed), but Kohlbecker's algorithm would traverse the list `(a)` exponentially many times.

(To see why, note that changing the macro by replacing the `0` by `x` would cause E_n to expand into a huge expression that, when evaluated, would produce a list in which the symbol `a` appears 2^n

times. Kohlbecker's algorithm traverses that huge expression following the next-to-last procedural transcription, just before the first of the two syntax rules discards that huge expression and delivers `0` as the final result of macro expansion. Modern algorithms never traverse any of the expressions that match the pattern variable `x`; they perform only a constant amount of work at each of the $O(n)$ transcription steps.)

The asymptotic inefficiency of Kohlbecker's algorithm limited its influence. Kohlbecker's system was added to Scheme84, and faculty at Indiana University said its asymptotic inefficiency was not apparent when hygienic macros were used in courses, but those reports did not convince the larger community that Kohlbecker's algorithm would be practical for larger programs.

4.5 Controlled Escape From Hygiene

Some traditional macros implicitly define or bind identifiers that are intended to capture identifiers that occur free within the context of use. (Some authors refer to such macros as *anaphoric* [Graham 1993].)

Consider, for example, a macro for defining new record types that implicitly defines field accessors and mutators whose names are obtained by combining the name of the record type with the name of a field [Steele 1990; Sperber et al. 2009b; Clinger 2009]. Those definitions would be worthless if those generated names could not be referenced within code that uses the macro.

As another example, consider a `loop` macro that executes some command repeatedly until an `exit` procedure is called with an argument that becomes the value of the loop expression. That macro is most easily defined by defining a local escape procedure named `exit` whose scope includes the body of the loop. Figure 3 shows an example of that macro being used in an expression that evaluates to 10.

As one last example, consider the `else` keyword of Scheme's `cond` syntax, which behaves somewhat as though it were bound to the boolean constant `#true`. It would be possible to approximate the intended behavior by defining `else` as a global constant or variable, but today's Scheme standards require `else` to behave as a syntactic keyword that is recognized specially only within expressions such as `cond` that use `else` as a keyword.

Kohlbecker's hygiene condition, as given above, would make those macros impossible to define. Because capturing is considered desirable in at least some macros such as those mentioned above, Kohlbecker modified his hygiene condition and extended his system so his original hygiene condition could be bypassed when necessary.

4.6 Kohlbecker's Modified Hygiene Condition

To support partially non-hygienic macros, Kohlbecker allowed macro definitions to list (1) auxiliary syntactic keywords such as `else`, and (2) identifiers the macro is intended to capture if they occur free within a use of the macro. With those extensions, Kohlbecker's algorithm no longer satisfies his hygiene condition. It does, however satisfy his *Modified Hygiene Condition* [Kohlbecker 1986, page 169]:

> Generated identifiers that become binding instances in a fully expanded macro call must bind only identifiers that are either apparent in the original macro call or generated during the same transcription step.

In other words, identifiers the macro is intended to capture will be captured only if they were already present within the original source code or were added during transcription of the macro, but will not be captured if they were added during transcription of some other macro or even during some other transcription of the same macro.

```
(let ((x 0)
      (y 5))
  (loop
   (set! x (+ x 2))
   (set! y (- y 1))
   (if (= y 0)
       (exit x))))
```

Fig. 3. Use of a loop macro that captures the reference to exit.

In Kohlbecker's system, the loop macro of Figure 3 could be defined as follows:

```
(extend-syntax (loop) (exit)
 ((loop cmd ...)
  (call-with-current-continuation
   ;; exit becomes bound to a one-argument procedure that
   ;; returns its argument to the continuation of the loop
   (lambda (exit)
     (define (the-loop)
       cmd ...
       (the-loop))
     (the-loop)))))
```

Because exit was listed in the first line of that definition as an identifier the macro is intended to capture, references to exit that occur free within the commands will refer to the escape procedure created by call-with-current-continuation and bound by the lambda expression.

Because the-loop was not listed as an identifier the macro is intended to capture, any free references to the-loop that might happen to occur within the commands will not refer to the zero-argument local procedure of that name whose definition is created by the macro.

A simplified version of the cond macro, similar to the definition in Kohlbecker's Figure 5.4 [Kohlbecker 1986], could be defined in Kohlbecker's system as follows:

```
(extend-syntax (cond else) ()
 ((cond)
  #false)
 ((cond (else exp1 exp2 ...))
  (begin exp1 exp2 ...))
 ((cond (exp0 exp1 exp2 ...) clause ...)
  (if exp0
      (begin exp1 exp2 ...)
      (cond clause ...))))
```

Listing else as a syntactic keyword along with cond prevented the matcher from treating else as a pattern variable. In Kohlbecker's system, we believe patterns that contain else would match a use of the macro if and only if the pattern's else is matched against an identifier whose original (un-time-stamped) name was else.

Kohlbecker's approach to syntactic keywords such as else was eventually added to syntax-rules (Section 7) and adopted by most subsequent systems. When matching keywords such as else, later systems took additional information into account so the else keyword in a pattern would not match a local variable named else or an identifier named else that was imported from a library where it means something other than what it means in the context of a macro definition that lists `else` as a keyword [Clinger and Rees 1991b; Kelsey et al. 1998; Sperber et al. 2009a; Shinn et al. 2013]; for details, see Section 14.2. In modern implementations of Scheme, `else` and similar keywords must therefore be exported from the `(rnrs base)` and `(scheme base)` libraries [Clinger 2007a].

Kohlbecker's approach to listing identifiers the macro is intended to capture had little influence, as it was judged unsatisfactory for at least these two reasons:

- Macros were unable to generate references that could be captured by bindings inserted by other macros. For example, a macro-generated use of the `loop` macro would be unable to insert a reference to `exit` that would refer to the binding that would be created during subsequent transcription of the `loop` macro.
- Macro-generated local macros, which Kohlbecker did not implement, would seldom be able to use the mechanism because the list of identifiers would itself be inserted during macro transcription, preventing their time stamps from matching the time stamps of any identifiers that occur free within the original source code.

4.7 Weaknesses of Kohlbecker's Algorithm

The most obvious problem with Kohlbecker's algorithm was its asymptotic inefficiency, as explained in Section 4.4.

Although Kohlbecker's algorithm had been designed to work with procedural macros, its advocates at Indiana were using Kohlbecker's pattern language, creating an erroneous but widespread belief that Kohlbecker's algorithm was limited by the expressive power of that pattern language.

On the other hand, Kohlbecker's algorithm *did* limit the power of procedural macros by making them hygienic by default, while providing a way to escape from hygiene that was not fully general. Furthermore, we believe Kohlbecker's mechanism for escaping from hygiene was implemented only for macros defined using his `extend-syntax` pattern language.

Kohlbecker's algorithm did not allow macros to define macros.

Kohlbecker's algorithm assumed syntactic keywords were reserved words that could not be shadowed by local bindings. It therefore solved only half of the capturing problem: Compare Kohlbecker's hygiene condition with the strong hygiene condition stated in Section 7.1.

Consequently, Kohlbecker's algorithm as described in his dissertation could not deal with local macros, which were already present in Common Lisp, nor would it be able to deal with module systems that allowed macros to be defined in scopes that differed from the global environment assumed by Kohlbecker's algorithm.

4.8 Friedman's Recollections

We asked Dan Friedman (Eugene Kohlbecker's thesis advisor) to share his recollections of how Kohlbecker's algorithm came about. We reproduce Friedman's recollections here, with our edits within square brackets.

> The HOPL authors, Will and Mitch, have asked me to write my recollections of what it was like before and when hygiene made an appearance, and for this, I wish to thank them. This is my story as best as I can remember it from over 35 years ago. I am grateful to Matthias [Felleisen] for sharing his reminiscences.
>
> Prior to December, 1975, [. . .] I was a devoted Lisp user. Then, when Mitch handed me a paper ([Sussman and Steele 1975]) along with the quote, "Dan, I think you are going to like this," everything changed. But, when I studied the code (without the multiprocessing) in that paper, where Gerry [Sussman] and

Guy [Steele] laid out Scheme and I understood that functions could be first-class values, that ended my infatuation with Lisp.

Fast forward to the Summer of 1983. Eugene taught c311 [(the undergraduate course on programming languages)] that summer. I had realized a pattern about macros where quasiquote and unquote in macros could virtually cancel out each other. Of course, it was just a simple observation. [. . .]

Eugene had already taken c311 (In those days both graduate and undergraduate students took this course.) During the Fall, Eugene was now in my advanced class. The first assignment was to implement fifteen macros, which were virtually Lisp macros. But, I gave Eugene special treatment. I asked him to take my simple observation and design a macro system that did not use quasiquote with unquote and work the same assignment as the others in my class. Eugene did his assignment and named his system "mkmac." [. . .]

In the Spring of 1984, Matthias took c311 with Bruce [Duba] as his associate instructor. At this point, we were using Eugene's mkmac. (But, hygiene had not yet come up as a topic.) The group of Eugene, Bruce, and Matthias was formed. In the Fall of 1985, Matthias and Bruce took my advanced class, which now relied on hygienic macros. I'm not exactly sure when Eugene came up with an extension of mkmac that included an algorithm for writing macros hygienically, but it clearly was before Eugene had enthusiastically showed anyone who was interested the magnificent or-macro. Discovering the "or-macro" problem and fixing it changed everything. This is when we all realized the significance of what we were witnessing. So my guess is this took place in the early months of 1985.

During the early days of the 1985 winterbreak, I called a group meeting to address an inkling I had that Eugene's algorithm had a bug. I had not known that Eugene had already headed home for the break. That afternoon, Matthias proclaimed, "There is a bug," and proceeded to demonstrate a small program that verified his claim. Then a couple of hours later, we fixed the bug. Fixing the bug for Eugene was a labor of love for the three of us. Because prior to our meeting, every test macro we threw at Eugene's hygienic system worked as expected, we had been quite delirious with hygienic macros and nothing changed our perspective with the discovery of this little bug.

[. . .]

Matthias [Felleisen] suggested we use the term *hygienic*, based on Barendregt's *hygienic variable convention.* After teaching many students about hygienic macros in my two programming languages courses over the preceding months, I felt that the ideas and implementation were now mature enough to share with the world. When Eugene presented in 1986 at the Lisp and Functional Programming Languages conference held at MIT, there was a comment during question time. An engineer from Symbolics observed, (paraphrase) "It's too bad we didn't have this idea built into the Lisp Machine. It would have saved lots of headaches."

Personally, I found the way my group worked together to be exhilarating. I understood that hygienic macros would change others perception about how to write macros, and I believed that we were opening a subfield that would never be closed and to this day, almost all systems with macros are assumed to be hygienic.

I used the powers of these hygienic macros for writing DSLs, which to my knowledge were not yet invented. This kind of power was well-known to Lispers,

> but *only* to those who knew where to place gensyms. Interestingly, Gerry [Sussman] had a great quote, "If I have to write a macro, I ask my local macrologist to write it for me." I think after every one realized the full power, in Mitch's and Eugene's *Macros-by-Example* paper, they became their own macrologist. It had been my goal early on to clean up the infestation of macros with quasiquotes and unquotes, but little did I know about the hygiene issues coming around the corner.
>
> I wish for everyone that they could be a part of something that will have had such a lasting inpact and will have had the opportunity to have led at least one such remarkable group.
>
> Sincerely,
> Dan
> October 7, 2019

5 NATIONAL AND INTERNATIONAL STANDARDS

As noted in Section 3.3, the design of Common Lisp had begun in 1981 under pressure from ARPA to consolidate the various major Lisp implementation efforts.

In December 1985, DARPA announced its decision that Common Lisp would become an American National Standard, with the standardization effort to be led by Bob Mathis, who had been convenor of the ISO working group on the Ada programming language [Steele and Gabriel 1993a]. Technical Committee X3J13 for Programming Language Common Lisp was formed soon thereafter.

As explained by Mathis, an official standard was needed because the European Lisp community, possibly feeling slighted because their views had not been solicited during the design of Common Lisp, planned to develop an ISO standard for Lisp that would probably be incompatible with Common Lisp. ISO standardization posed two distinct threats to Common Lisp: (1) US law required defense-related work to use ISO standards whenever possible, and (2) ISO rules would require developers of any competing national standards to suspend their work once ISO standardization had begun.

This turn of events made it important that the status of Scheme *vis-a-vis* Common Lisp be clarified.

5.1 Lisp-1 versus Lisp-2

As mentioned in Sections 3 and 3.4.3, Scheme's use of a single lexical environment for all variables and functions lay at the heart of its identity as a higher order language. In 1980s vernacular, Scheme was a *Lisp-1* language; Common Lisp, which used separate environments for variables and functions, was a *Lisp-2* language [Gabriel and Pitman 1988]. That was the key difference that stood in the way of merging those two languages [Steele and Gabriel 1993a]:

> The effort to merge the Scheme and Common Lisp communities was launched on two fronts. One was to come up with a solution to the macro problem that a Lisp-1 causes. This problem is that with only one namespace it is relatively easier to stumble across an unintended name conflict leading to incorrect code. The key Common Lisp leaders felt that if the macro problem could be solved, Common Lisp could survive a transition from Lisp-2 to Lisp-1. The other front was to convince the Scheme community that this was a good idea.

As Gabriel and Pitman [1988] put it:

> There are two ways to look at the arguments regarding macros and namespaces. The first is that a single namespace is of fundamental importance, and therefore

> macros are problematic. The second is that macros are fundamental, and therefore a single namespace is problematic.

In an email to Dick Gabriel, Clinger suggested X3J13 consider adopting a hygienic macro system based on Kohlbecker's algorithm. Gabriel pointed out that Kohlbecker's algorithm had not yet been implemented for Common Lisp. Furthermore, it was well known that Kohlbecker's algorithm ran in quadratic time for some common recursive macros such as cond.

As related below, the Scheme community would eventually invent efficient algorithms for hygienic macro expansion, but those algorithms came too late for X3J13 to consider transitioning from Lisp-2 to Lisp-1. Had those algorithms been discovered several years earlier, it might have been possible to merge Scheme into Common Lisp.

But probably not. The Common Lisp community was driven by the pragmatic advantages of agreeing upon a common language in the short term, while the Scheme community was driven by the academic desire to Do The Right Thing even if it takes forever.

For example: Common Lisp's separate environments for variables and functions don't actually solve the capturing problem. They just convert what would otherwise be a common bug into a less common bug. Actually solving the capturing problem was the Right Thing To Do, so the Scheme community did it, but it took a while.

5.2 ISO SC22/WG16

In the fall of 1987, the ISO effort got underway with the formation of the working group known as ISO/IEC JTC 1/SC 22/WG 16 (generally abbreviated to ISO SC22/WG16).

In a routine example of the political compromises needed for smooth development of official standards, France selected the convenor of ISO SC22/WG16 (Christian Queinnec) while X3J13 selected the editor (Richard Gabriel, founder of Lucid Inc), who would also serve as US representative to ISO SC22/WG16. Gabriel wanted Clinger, then serving as Tektronix's representative to X3J13, to accompany him as a semi-official co-editor, and X3J13 approved [Gabriel 1988; Steele and Gabriel 1993b].

By the end of 1987, X3J13 had decided Lisp was a family of languages, not a single language, and was supporting that position by citing Scheme as a viable language of the Lisp family that is decidedly distinct from Common Lisp.

The first meeting of ISO SC22/WG16 was held in Paris, on 24 and 25 February 1988 [NA ISO/JTC1/SC22 1988]. The first of its two days was devoted to an extended discussion of goals. That discussion often spilled over into arguments over the X3J13 position that Lisp is a family of languages, not a single language, frustrating those who were hoping to shut down the American effort to standardize Common Lisp. On that point, John McCarthy, Gabriel, and Clinger prevailed not only because the facts were on their side but also because the meeting was conducted in English, putting the French at a disadvantage, while convenor Queinnec was careful to avoid any appearance of favoring his countrymen. Following that first meeting, Clinger resigned as co-editor.

As it turned out, this dispute between the X3J13 and various European delegations continued for several years, centering on the name of the language to be standardized. If the name of that language was "Lisp", then Common Lisp might fall under the ISO umbrella. Eventually ISO SC22/WG16 adopted the name ISLisp, thus avoiding the threat to Common Lisp [SC22/WG16 1997].

5.3 IEEE Standard 1178-1990

Clyde Camp of Texas Instruments, chair of the IEEE's Microprocessor Standards Committee (MSC), sent email to Jonathan Rees, David Bartley, "and a few others" suggesting the RRRS authors consider the topic of standardization at their June 1987 meeting [Camp 1988]. In his tentative agenda for that meeting Bartley [1987a] listed standardization ("(IEEE, ACM, ANSI???)", with the parentheses and question marks in the original) as a meta-issue to be discussed, but the minutes of that meeting do not mention formal standardization [Clinger 1987].

In December 1987, Dick Gabriel sent a long message to the RRRS authors that began by saying [Gabriel 1987]:

> I have heard from my friend Will Clinger and read in a message from Chris Haynes that the Scheme group is considering pursuing an IEEE standard for Scheme. I made some remarks to Will about this by phone, but I would like to address them to the group as a whole.

Disclaiming "any deep feelings about the matter," Gabriel went on to discuss in some detail the following questions, which were raised in an earlier email from Chris Haynes:

(1) Is a Standard Necessary?
(2) Will Standardization Help Motivation and Discipline?
(3) Will a Standard Improve Portability?
(4) Will a Standard Increase the Likelihood of Difficult Implementation?
(5) Will Unnecessary and Flawed Features Creep In?
(6) Will Research Stop?
(7) Will the Wrong Folks Move In?
(8) What's the Right Thing?

In his discussion of question (5), Gabriel wrote:

> Here is a case in point: If you standardize Scheme in the next year or two, that standard cannot include any macro facility. This is because a PhD thesis on the topic has just been published, and no implementation of the latest, most final ideas has been in use for long enough. The Common Lisp macro facility is mightily flawed, but it codifies 20 years of practice, and its flaws are well-known. This situation is very much more desirable than casting as a standard something you just thought of.

Chris Haynes [1988b] agreed that macros were not ready for standardization:

> Perhaps there are sound technical reasons for a substantial delay (say a year or more). If so, I would like to hear them. However, my impression from last summer's meeting is that almost all of what we have now is relatively stable, and the new features we would like to consider adding (macros, modules, etc.) are all very much in the design stage. It will likely take at least a year (probably two or more) to refine these proposals and get them widely implemented, and another year of experience with them before we can be sure we like them. Only then (2 or 3 years hence) would it seem likely that major new features should be candidates for standardization.

Jonathan Rees [1988a], who (with Alan Bawden) had proposed macro systems that will be discussed in Section 6.5, agreed: "The macro proposal still needs more work."

Several RRRS authors expressed their opinion that standardization should wait for macros [Ramsdell 1987; Miller 1988]. Others favored standardization without macros [Haynes 1988b; Camp 1988; Abelson 1988; Freeman 1988].

On 17 May 1988, Haynes [1988a] reported:

> The proposal to form an IEEE Working Group on Scheme (let's call it the "working group" from now on) mentioned in my last note has been approved by the Microprocessor Standards Committee (MSC). There are a few more administrative steps

> before it is completely official, but no difficulty is expected. The standardization process has begun; wish us luck!
>
> The first meeting of the working group will be on Wednesday, July 27th, 1988, from 1:00 pm to 5:00 pm (or later if really necessary) following the Lisp and Functional Programming Conference at Snowbird, Utah....
>
> Very briefly, my current expectation (reflecting the wishes of the study group mentioned in my last note) is that the working group will serve to "filter" the work of the Revised^n Report authors, removing features deemed too experimental for standardization and adding nothing of substance to the language.... A "trial standard" should be completed in about two years, with a full standard perhaps two years after that.

The full standard was completed in two years, with Haynes chairing the working group and presiding over a remarkably quick and smooth process. David Bartley, Chris Hanson, and Jim Miller served as co-editors of *The IEEE Standard for the Scheme Programming Language*, IEEE Std 1178-1990, which was approved on 10 December 1990, and reaffirmed in 2008 [Bartley et al. 1991; IEEE 2008]. IEEE Std 1178-1990 also became an ANSI standard, as IEEE MSC routinely submits its approved standards to ANSI [Steele and Gabriel 1993a].

The IEEE standard was based on drafts of the R4RS, which was not completed until after the IEEE standard was approved, mainly because it was delayed by waiting for the low-level macro system described in its macro appendix to be redesigned [Clinger and Rees 1991b]. The IEEE standard includes two appendixes that provide guidance to implementers of Scheme's arithmetic system, but does not describe any macro facilities. That omission was explained in the standard's Foreward by a section that states the "Purpose of the Standard" and ends as follows [Bartley et al. 1991]:

> In particular, there are important linguistic design issues that are not discussed in the Scheme standard *precisely because* Scheme has sparked fruitful new approaches in these areas, and the authors of this report do not wish to discourage the further development of good ideas by taking a position on issues under active investigation. Some of these issues are macros, packaging, and object-oriented programming.
>
> The working group hopes that future revisions of the standard will be sensitive to the fact that good ideas need time to mature, and that exploration can often be stifled by the premature adoption of standards.

That prose had been drafted by Hal Abelson [1988]. Abelson also drafted a full paragraph for each of the three issues named in the excerpt above, but those paragraphs didn't make it into the standard. Here is what Abelson suggested the standard should say about macros:

> Macros: The scope of identifiers in macros has traditionally been problematic for Lisp. Scheme's adoption of lexical scoping and a single namespace for identifiers has stimulated the development of macro-expansion techniquies that are more elegant and robust than those appearing in other Lisp dialects. Although this report does not specify a macro-definition facility, most popular Scheme implementations allow users to define macros.

6 THE RISE AND FALL OF SYNTACTIC CLOSURES

Before returning to the June 1987 meeting of the RRRS authors and the rise of syntactic closures, we will say a few words about the cast of characters.

6.1 People Come and Go

The Scheme programming language had been invented at MIT, but T was implemented at Yale, and Indiana University (IU) became a new locus for research and teaching that used Scheme.

Mitch Wand and Dan Friedman joined the faculty at Indiana University in 1973. Wand had done his doctoral work in MIT's applied math program, and visited MIT's Project MAC Technical Report room every year. Wand read the original report on Scheme [Sussman and Steele 1975], realized it would make a great teaching language, and taught the language to Friedman; by 1978, they were using Scheme in courses. Although Clinger had done his doctoral work in MIT's pure math program, writing a dissertation related to computer science, he didn't become involved with Scheme until he joined Wand and Friedman at IU in 1981.

After finishing his PhD in 1986, Kohlbecker left IU (and computer science generally) to enter a Christian ministry. Matthias Felleisen finished his PhD at IU in 1987 and joined the faculty at Rice University. Wand left IU in 1985 to become associate dean of the College of Computer Science at Northeastern University. Clinger left IU in 1985 for Tektronix, which he left at the beginning of 1988 for the company that became known as Lightship Software (purveyors of MacScheme and MacScheme+Toolsmith), moving on to the University of Oregon in the fall of 1988.

Dan Friedman, Chris Haynes, and Kent Dybvig remained at IU [Dybvig et al. 1986, 1988], where they would soon be joined by PhD student Robert Hieb.

At Yale, Professor Paul Hudak turned his attention to lazy functional languages and the design of Haskell. His PhD students Richard Kelsey, David Kranz, and Norman Adams IV completed their degrees and went their separate ways; Adams briefly joined Clinger and Uwe Pleban at Tektronix before departing for a startup whose success allowed him to retire at a very young age.

Jonathan Rees had worked on T under Hudak's supervision and had edited the R3RS. He became a master's student at MIT, where he began to work on macros in collaboration with Alan Bawden and Chris Hanson, under the supervision of Gerry Sussman [NA Rees 1989].

6.2 Kohlbecker's Algorithm Considered Undisciplined

Kohlbecker's algorithm had come out of Indiana. How would it be received at MIT?

Bawden and Rees [1988] regarded Kohlbecker's algorithm as "very undisciplined" and hard to understand:

> In addition, it is difficult to comprehend how hygienic expansion operates and why it is correct.

As alternatives, they put forth a series of proposals culminating in the *syntactic closures* described in Section 6.5.

6.3 A Modest Macro Proposal

In a message sent to the RRRS authors near the end of March 1987, Rees [1987a] described "a modest macro proposal." His proposal was not inherently hygienic, but included workarounds for the capturing problem. His proposal did allow local macros to be defined, and introduced "preprocessed-expression" objects that might be regarded as a first step in the direction of syntax objects (to be described in Section 9):

> Kohlbecker's solution amounts to performing alpha-conversion and macro expansion at the same time. This is a lot of mechanism and breaks down in a few places. Here is a much simpler, low-tech solution.
>
> The solution is for the expander to introduce special expressions into the expansion that represent "absolute" or "global" references. Such references are not sensitive to the lexical environment.

That solution half-addresses the half of the capturing problem Kohlbecker had not addressed at all. To deal with the other half of the capturing problem (macros that introduce local variables), Rees proposed a gensym-like mechanism (as described at the end of Section 3.2).

In a significant break with tradition, macros would be expanded bottom-up instead of top-down.

Because the proposal was low-tech, it relied on programmers to get a lot of details right. Rees admitted it would be difficult to write correct macros using his proposal, but suggested it might be possible to build something like Kohlbecker's hygienic `extend-syntax` on top of his proposal.

Rees [1987b] acknowledged "serious objections from Gene Kohlbecker and/or Dan Friedman," but the nature of those objections is not recorded.

In subsequent discussion, David Bartley of TI suggested Scheme's status as a Lisp-1 could be reconsidered [Bartley 1987b]. In his response, Rees [1987c] acknowledged that T had switched from Lisp-1 to Lisp-2 "after a year or so in order to be compatible with MIT Scheme. I'll flame about this later if need be." Those remarks illustrate the importance of macros: As late as 1987, some were willing to compromise Scheme's identity as a higher-order language in order to reduce, without eliminating, bugs caused by the capturing problem.

In response to Bartley's observation that Kohlbecker and others had already addressed the capturing problem, Rees asked

> What is the computational complexity of the hygienic expansion algorithm, by the way?

In 1987, that question could shut down any consideration of hygienic macro expansion.

6.4 A Modified Macro Proposal

In May 1987, ten researchers gathered at MIT for a "Macro Summit": Alan Bawden, Chris Hanson, Eugene Kohlbecker, Richard Mlynarik, Jim Philbin, Kent Pitman, Jonathan Rees, Bill Rozas, Gerald Jay Sussman, and Mitch Wand [Bawden and Rees 1988].

Soon afterwards, Bawden [1987] offered a modification of Rees's modest proposal:

> Here, as promised at the Macro Summit Meeting, is a "Modified Macro Proposal"....The only real difference between this proposal and the last, is that it includes a mechanism that addresses the various variable capture problems.
>
> The basic idea is that the thing we usually call an "environment" (a map from identifiers to locations), can be viewed as the composition of two separate maps: First, a map from identifiers to "variables", followed by a second map from variables to locations....
>
> Now the idea is to make this static mapping from identifiers to variables part of syntax tables. This has a nice appeal to it since it makes syntax tables contain *all* of the static, contextual information necessary for interpreting the meaning of a particular piece of code. In one package you get both the mapping from keywords to their current meanings, as well as the mapping from identifiers to their current meanings....
>
> Perhaps the reason it works so well can been seen by thinking of the thing returned by PREPROCESS as a "syntactic closure", similar in nature to the closures returned by LAMBDA-expressions. In both cases you have an environment of some kind, a list of identifiers, and an expression. In both cases all identifiers in the expression are to be taken relative to the environment, *except* those in the given list. The identifiers in the list are to have their meanings determined later. In both cases it is a way of "parameterizing" the expression.

That was the origin of syntactic closures.

Later that month, "people interested in the continuing evolution of the Scheme programming language" met for the second time (after the October 1984 meeting at Brandeis) in a two-day meeting chaired by Mitch Wand. As communicated by Clinger [1988a]:

> Alan Bawden reported on recent research toward solving the problem of Scheme macros. His report grew out of an earlier workshop that brought together the leading researchers in this area.... It appears that Alan and his colleagues have designed an architecture capable of solving most of the problems that have heretofore plagued Lisp macro facilities. There is much more work to be done, but the macro issue seems to be in good hands.

6.5 Syntactic Closures

In block-structured programming languages, procedures are often represented by data structures called *closures*. A closure is simply an object that encapsulates both the code of the procedure (such as a lambda expression) and an *environment* that records the values denoted by any variables that occur free in the code.

Steele's use of thunks to reduce inadvertent capturing of free variables had relied on closures, which simple interpreters would create at run time.[2] Each closure remembered the intended bindings of the variables that occur free within the `lambda` expression from which it was created.

Bawden and Rees proposed moving that idea to macro expansion time. Most macro expanders, including Kohlbecker's, are implemented using a syntax table that records the core syntaxes and all macros that have been defined or are in scope. Bawden and Rees generalized those syntax tables into *syntactic environments* that macro writers could manipulate explicitly. For each syntactic form that is to appear in the output of a macro, the macro writer would specify which of its names should obtain their meanings from which syntactic environment (such as the syntactic environment that contains the standard meanings for all names pre-defined by the language, or the syntactic environment that gives meaning to all names that are in scope at the macro's definition). The macro writer would also specify which names would remain free so their meanings could be determined by some other syntactic environment (such as the syntactic environment of the macro's uses).

A syntactic closure then becomes a new kind of abstract syntax tree in which some or all of the names have their meanings fixed by a syntactic environment.

The `make-syntactic-closure` procedure takes three arguments and returns a syntactic closure. Its third argument is an expression or syntactic closure. If that argument is an expression, it can be regarded as a syntactic closure whose syntactic environment is empty and therefore does not determine the meanings of any names that occur within the expression. If that argument is a syntactic closure, then some of the names that occur within the syntactic closure's expression may have meanings that are already determined by the syntactic environment component(s) of the syntactic closure, but there may still be some names whose meaning has not yet been fixed. The second argument is a list of names whose meaning should remain unfixed within the syntactic closure returned by the procedure. The first argument is a syntactic environment that is used to fix the meaning of all names that do not appear within the second argument and whose meaning has not already been fixed by the syntactic closure passed as the third argument.

[2]Routine compiler optimizations can be relied upon to eliminate all run-time overhead of the thunks introduced by Steele's technique [Steele 1977], but that fact is not germane to the distinction between run time and macro expansion time that we are emphasizing here.

With syntactic closures, the push macro of Figure 1 could be written in the style of Hanson [1991]:

```
;;; (push <item> <stack>)

(define-syntax push
  (sc-transformer
   (lambda (exp env)
     (let* ((item-ast (cadr exp))
            (stack-ast (caddr exp))
            (item
             (make-syntactic-closure env '() item-ast))
            (stack
             (make-syntactic-closure env '() stack-ast)))
       `(set! ,stack (cons ,item ,stack))))))
```

The first argument of the lambda expression, exp, is the abstract syntax tree for a use of the push macro. (Parts of that abstract syntax tree might be represented as a syntactic closure.) The second argument, env, is the syntactic environment of that use. The macro transformer extracts abstract syntax trees for the <item> and <stack>, and closes those expressions in the syntactic environment of the use, leaving none of their names free to be determined by subsequent processing. The meanings of set! and cons are determined by the syntactic environment of the macro definition, so they cannot be affected by any non-standard meanings of those names that may be in effect within the syntactic environment of the use.

6.6 Hygiene Not Automatic

Because syntactic closures is a low-level system in which skillful programmers avoid inadvertent captures by taking care to close every part of the macro's output within the correct syntactic environment, writing macros that capture identifiers deliberately is just as easy (or hard) as writing hygienic macros.

Here is a definition of the loop macro from Figure 3 using syntactic closures [Hanson 1991]:

```
(define-syntax loop
  (sc-transformer
   (lambda (exp env)
     (let ((body (cdr exp)))
       `(call-with-current-continuation
         (lambda (exit)
           (define (the-loop)
             ,@(map (lambda (cmd)
                      (make-syntactic-closure env '(exit) cmd))
                    body)
             (the-loop))
           (the-loop)))))))
```

The highlighted call to make-syntactic-closure allows each command's references to exit to be captured by the highlighted lambda-binding of exit, while closing all other free variables of those commands within the syntactic environment passed to the transformer.

(The ,@ notation, known as unquote-splicing, is analogous to the unquote notation (indicated by a single comma) explained in Section 2. The expression following the ,@ notation is evaluated, as with unquote, but the result of that evaluation must be a list of things (instead of a single thing as with unquote), and the elements of that list are inserted into the quasiquoted expression. In this example, each command within the body of the loop will be converted into a syntactic closure that, during subsequent macro-expansion, will become the fully macro-expanded form of that command in the body of `the-loop`, with all of those macro-expanded commands preceding the tail-recursive call to `the-loop` at the end of its body.)

6.7 The Macro Subcommittee

A year later, in July 1988, the RRRS authors met again (at the ACM Conference on Lisp and Functional Programming in Snowbird, Utah), with about fifty people in attendance, to revise the R3RS [Clinger 1988b]:

> Most of the changes approved by the authors' meeting simply correct or clarify that report. The single most significant decision was to entrust a subcommittee with the task of designing a macro facility based on syntactic closures but with a syntax resembling that of `extend-syntax` without the `with` feature [Dybvig 1987]. The work of this subcommittee, comprised of Alan Bawden, Kent Dybvig, Bob Hieb, and Jonathan Rees, has been pre-approved. That is, whatever the subcommittee comes up with will go into the R4RS as an inessential feature provided it can deliver something reasonable in time, where reasonableness is presumably a concept to be negotiated between the subcommittee and the R4RS editors.
>
> Everyone seems to want macros, and the subcommittee arrangement shows that impatience is winning out over partisanship. Syntactic closures appear sound, but they need to be combined with a convenient language for defining macros such as is provided by `extend-syntax`. The Scheme community has accumulated a fair amount of experience with `extend-syntax`, and it is expected (hoped?) that any bugs that might eventually be found in the subcommittee's proposal can be fixed without requiring major changes in the defining language.

Chris Hanson worked with the macro subcommittee *ex officio*.

6.8 The Macro Subcommittee Stalls

In May 1989, Clinger [1989] recommended macros be omitted from the R4RS:

> Chris Hanson deserves great credit for developing a macro facility along the lines envisioned at Snowbird, but the well-specified part of it does not seem to be sufficiently powerful.... I think we are very close to having a well-specified system that does what we need, but I recommend that the R4RS not include a macro facility because we're not quite there yet.

Hal Abelson [1989] objected:

> I am very unhappy that...it will not say anything about macros. One of the major agreements at the Snowbird meeting was the agreements to go ahead with macros. The momentum of that decision has been lost, and we need to regain it....
>
> The macro committee should get its collective head together and write such a proposal. Here are two possible forms the proposal might take:
>
> (1) You guys can agree on something.
>
> (2) You can write TWO proposals. One would be along the lines of extend-syntax; the other based upon semantic [*sic*] closures. The full proposal should descibe [*sic*] both of these, together with a brief introduction that explains what

> the issues are, gives reasons why we are not yet ready to commit to either proposal (hopefully there are reasons, other than political ones), and sets forth some problems that we hope to solve in future R*RS documents. I nominate Jonathan Rees to write this overview.
>
> The main thing I want to avoid is for R4RS to appear with no macro proposal at all.

On 1 August 1989, twelve people attended the inaugural meeting of the Bay Area Scheme Hackers (BASH), at Stanford University, hosted by Daniel Weise. As recorded by Joel Bartlett [1989]:

> Jonathan [Rees] reported that there is an impasse between those wanting extend syntax and those concerned about capture problems. It is not clear how to resolve this to produce one macro proposal.
>
> Resolved: The macro committee should keep trying. R4RS should not go out without a macro proposal. It should not go out with two macro proposals.

Rees, Bawden, and Hanson were blocking Dybvig and Hieb's advocacy of `extend-syntax` and Kohlbecker's algorithm.

Rees, Bawden, and Hanson tried to design an easy-to-use, high-level pattern language for defining macros based on syntactic closures, but ran into a seemingly intractable technical roadblock: To define a macro using syntactic closures, the writer of the macro has to know which names must be left free so they can be bound by a binding inserted by the macro (as in the `let` macro above), with all other names being closed in an appropriate syntactic environment. With the pattern language of `extend-syntax`, there was no reliable way to determine which of the names that appear within the output of a macro would end up being bound by a binding inserted by that same macro. For example:

```
(extend-syntax (mylet1) ()
 ((mylet1 var val exp)
  (let ((var val)) exp)))
```

For a pattern-based macro expander to learn that free occurrences of `var` within `exp` will be bound by the `let`, it has to macro-expand the `let` that appears within the output of the rule. That's too late. With syntactic closures as defined by Bawden and Rees [1988], the `mylet1` macro has to decide which syntactic environments to use before the `let` is expanded.

As co-editor of the R4RS, Will Clinger would be responsible for incorporating the report of the macro subcommittee into that new document. On a visit to MIT, he met with Alan Bawden and Chris Hanson to find out why the macro subcommittee was taking so long to deliver its recommendation. They explained the problem described above, but suggested a high-level interface such as Kohlbecker's `extend-syntax` might not be necessary because any competent programmer should be able to define macros using the low-level interface for syntactic closures Bawden and Rees [1988]. Clinger suggested they use the room's whiteboard to show him how to write a simplified definition of Scheme's `let` macro, and checked his watch. After many erasures and edits, when they were reasonably sure they had gotten it right, Clinger noted that two of the world's three leading experts on syntactic closures had just taken fifteen minutes to define a macro that would have taken average programmers less than a minute to define using Kohlbecker's `extend-syntax`.

Here is how a simplified form of Scheme's `let` syntax can be defined using syntactic closures [Hanson 1991]:

```
(define-syntax let
  (sc-transformer
   (lambda (exp env)
     (let* ((bindings (cadr exp))
            (body (cddr exp))
            (vars (map car bindings))
            (vals (map cadr bindings)))
       `((lambda (,@vars)                 ; vars bound here
           ,@(map (lambda (exp)
                    ,(make-syntactic-closure env vars exp))
                  body))
         ,@(map (lambda (val)
                  (make-syntactic-closure env '() val))
                vals))))))
```

The first call to `make-syntactic-closure` passes the list of `vars` as a second argument because those variables must be left free to be captured by the lambda bindings of those variables in the output of the macro.

That definition of the `let` macro performs no syntax checking, so it will blow up with mysterious error messages when given a syntactically incorrect `let` expression to expand.

Here is an equally hygienic definition of that same macro using Kohlbecker's system, in which pattern matching accomplishes some syntax checking automatically:

```
(extend-syntax (let) ()
 ((let ((var val) ...) . body)
  ((lambda (var ...) . body)
   val ...)))
```

Clinger left MIT thinking non-experts might never be able to use syntactic closures reliably.

6.9 Syntactic Closures Fall Out of Favor

On 19 January 1990, IEEE/MCS/P1178 Working Group on Scheme met for the fourth time. Before voting unanimously to submit its next draft of the IEEE standard for public comment and balloting, the working group heard from Clinger and Gabriel [Haynes 1990]:

4. Reports on R4RS, macros, and ISO.

   A. Report on R4RS. (Will Clinger)

      R4RS is complete except for the appendix describing macros and …

   B. Report on the status of the macro committee. (Will Clinger)

      (1) Syntactic closures are out of favor. Chris Hanson attempted to implement EXTEND-SYNTAX using syntactic closures. The attempt was not entirely successful because of a previously unrecognized technical problem: in some cases an EXTEND-SYNTAX based on syntactic closures must know the syntactic role of an identifier before macro expansion has made that role apparent.

(2) A proposal based on a descendent of hygienic macro expansion has some members of the macro committee confident that a resolution to the macro impasse is at hand. The current proposal is due to Jonathan Rees, who modified a proposal by Will Clinger, who based his proposal on Eugene Kohlbecker's Ph.D thesis work.

C. Report on the International Standards Organization WG-16. (Dick Gabriel)

WG-16 is not currently interested in Scheme as a basis for an international standard.

The proposal mentioned in 4.B.(2) is described below (Section 7).

7 MACROS THAT WORK

After Bawden and Hanson had explained the technical problem that was delaying their report, Clinger realized the time-stamping and un-stamping performed by Kohlbecker's algorithm was a way to put off having to know which identifiers would be bound, and where, until macro expansion was sufficiently complete to reveal that information.

But it was not at all obvious how time-stamping could be added to syntactic closures.

Clinger began to wonder whether it might be easier to fix the inefficiency of Kohlbecker's algorithm than to design a high-level interface for syntactic closures.

The inefficiency of Kohlbecker's algorithm arose from its traversal of the entire output of every macro transcription, during which it added time-stamps to all identifiers that had been inserted by the macro. That traversal was necessary because Kohlbecker had designed his algorithm to work with the procedural macros of Common Lisp. By this time, however, Kohlbecker's algorithm was mainly being used in his pattern-based `extend-syntax` system, which had become popular among Scheme programmers at Indiana University and was beginning to catch on elsewhere. With a pattern-based system, it is easy for a time-stamping macro expander to traverse only the parts of the output that were not already present within the input.

Clinger wrote formal specifications for a modified version of Kohlbecker's algorithm that ran in linear time because it worked with the pattern-based rules instead of procedural syntax transformers. He described that algorithm in terms of renaming and environments rather than time-stamping, which was little more than a stylistic change, but it helped to emphasize the connection with α- and β-conversion in lambda calculus, from which Kohlbecker had borrowed the word "hygiene." (For a more detailed account of Clinger's specification, see Section 17; Section 7.3 shows how the algorithm works on a nontrivial example.)

Clinger sent his description of the algorithm to Gabriel and to several others, including Jonathan Rees. Gabriel said X3J13 could not consider new technologies that had not been in common use.

Within days of Clinger's email, Rees sent him a complete working implementation of the new algorithm, including referentially transparent local macros as described in Sections 7.2 and 7.3, all built on top of the compatible low-level procedural system described in Section 7.4. Their paper on "Macros That Work" was presented at POPL in January 1991 [Clinger and Rees 1991a].

In November of that year, a slight extension of their high-level `syntax-rules` facility was described in an appendix to the R4RS, along with a compatible low-level facility that eventually evolved into the `syntax-case` system described in Section 9 [Clinger and Rees 1991b].

7.1 Strong Hygiene Condition

Using syntax-rules as described in the R4RS, which differs in two minor ways from the syntax-rules syntax in "Macros That Work," the push macro of Figure 1 would be defined as follows [Clinger and Rees 1991a]:

```
(define-syntax push
  (syntax-rules ()
   ((push item stk)
    (set! stk (cons item stk)))))
```

The set! and cons inserted by the macro would be resolved in the environment of the macro definition, not in the environment of the use, so examples such as the following would work as intended:

```
(let ((pros (list "cheap" "fast"))
      (cons (list)))
  (push "unreliable" cons)
  (weigh pros cons))
```

Kohlbecker's Hygiene Condition had not guaranteed that references inserted by a macro would be resolved in the environment of the macro definition.

To describe the behavior of their system more accurately, Clinger and Rees stated this *Strong Hygiene Condition*:

(1) It is impossible to write a high-level macro that inserts a binding that can capture references other than those inserted by the macro.
(2) It is impossible to write a high-level macro that inserts a reference that can be captured by bindings other than those inserted by the macro.

Kohlbecker's Hygiene Condition stated only the first part of that.

7.2 Referentially Transparent Local Macros

A programming language is said to be *referentially transparent* "if we may replace one expression with another of equal value. . . without changing the meaning of the program" [Mitchell 2002, §4.4.2].

Clinger and Rees [1991a] made the following claim:

> Our algorithm supports referentially transparent local macros. We should note that these macros may not be referentially transparent in the traditional sense because of side effects and other problems caused by the programming language whose syntax they extend, but we blame the language for this and say that the macros themselves are referentially transparent.

To explain what that meant, they gave this example:

```
(let ((x "outer"))
  (let-syntax ((m (syntax-rules ()
                   ((m)
                    x))))
    (let ((x "inner"))
      (m))))
```

That example evaluates to "outer" because the x inserted by the local macro m is resolved in the environment of the macro definition, not in the environment of the macro use. The fully macro-expanded code is equivalent to

```
    (let ((x.1 "outer"))
      (let ((x.2 "inner"))
        x.1))
```

7.3 Local Macros That Define Local Macros

"Macros That Work" was the first algorithm to implement fully automatic hygiene for referentially transparent local macros. It was also the first algorithm to implement fully automatic hygiene for local macros that define local macros.

The following example was suggested by one of our HOPL reviewers:

```
(let ((x 0))                                          ; <- this one
  (let-syntax ((m (syntax-rules ()
                   ((m ?m2 ?body)
                    (let-syntax ((?m2 (syntax-rules ()
                                        ((?m2) x)))) ; <- reference to x
                      ?body)))))
    (let ((x 1))                                      ; <- not this one
      (m m2
         (let ((x 2))                                 ; <- not this one
           (m2))))))
```

With the "Macros That Work" algorithm, that example macro-expands hygienically into an expression that evaluates to 0. With macro expanders that are not fully hygienic, that expression and its analogues would probably expand into expressions that evaluate to 1 or 2.

We will now show the significant steps in the macro-expansion of that example, as confirmed by adding instrumentation to the implementation of "Macros That Work" described in Section 7.7.

Although Clinger's original specification of the algorithm employed a big-step semantics, our step-by-step walkthrough of the algorithm will mostly correspond to an equivalent small-step operational semantics in which the macro expander takes two arguments, an expression to be expanded and a syntactic environment that guides the expansion, and delivers two results: a macro-expanded or partially macro-expanded version of the expression, and a new syntactic environment that will guide any further expansion of the partially macro-expanded expression.

The syntactic environments map identifiers to syntactic denotations, which are of three kinds:

- *special* denotations for built-in syntaxes such as `lambda` and `let-syntax`
- *identifier* denotations, which name an alias to be substituted for any mentions of the identifier that might be encountered during macro expansion
- *macro* denotations, which include the rewrite rules for a macro together with a syntactic environment that remembers what the "free identifiers" of those rules meant in the context where the macro was defined. To avoid the asymptotic inefficiency of re-parsing a macro's rewrite rules every time a use of the macro is expanded, each macro's rules are translated into an efficient data structure when the macro is defined, and that data structure is recorded as part of the macro's denotation.

If a syntactic environment does not specify a binding for some identifier, that identifier is effectively global or undefined and is taken to be an alias for itself. Undefined identifiers can be used as pattern variables or quoted constants, so the macro expander can encounter undefined identifiers without having any reason to report an error. When the macro expander is used in an interactive read/eval/print loop, undefined identifiers can also refer to variables that will be defined later.

The macro expander begins by expanding the expression in a syntactic environment ρ_0 that contains bindings for all identifiers that are in scope. We will assume Scheme's standard libraries have been imported, so ρ_0 contains bindings for all of Scheme's standard syntaxes (e.g., `lambda`, `let-syntax`) and standard macros (e.g., `let`).

The first step of macro-expansion rewrites the original expression according to the `let` macro, yielding

```
((lambda.2
   (x)
   (let-syntax
     ((m (syntax-rules
           ()
           ((m ?m2 ?body)
            (let-syntax
              ((?m2 (syntax-rules () ((?m2) x))))
              ?body)))))
     (let ((x 1)) (m m2 (let ((x 2)) (m2))))))
 0)
```

(Why `lambda.2` instead of `lambda.1`? Because we are using the numbering of time stamps and environments produced by the implementation described in Section 7.7. Those numbers are not consecutive because we omit trivial and implementation-specific steps. We have also simplified the spelling of fresh identifiers, writing (for example) `x.7.11` instead of `|.x\|7\|11|`.)

Whenever some use of a macro is rewritten, the rewriting step also produces a syntactic environment that contains bindings for all identifiers that were introduced by the rewriting. In this case, the only introduced identifier is `lambda`. In the generated code, we need to make sure that the generated `lambda` refers to what `lambda` meant in the context where the `let` macro was defined, not to any local meaning the identifier `lambda` might have in a context where `let` is used.

To accomplish this, the algorithm renames the generated `lambda` to `lambda.2`, and expands the generated code in an environment in which `lambda.2` is bound to the meaning of `lambda` at the macro-definition site. That meaning is the standard meaning of `lambda`, which we write as λ.

So the resulting expression, shown above, should be expanded in the syntactic environment $\rho_4 = \rho_0[\texttt{lambda.2} \mapsto \lambda]$. This notation means that ρ_4 is the same as the original syntactic environment ρ_0 except ρ_4 also binds `lambda.2` to the standard meaning of `lambda` in Scheme.

This renaming of all identifiers introduced by macros, accompanied by the production of a new syntactic environment that extends the input environment by adding bindings for those new identifiers, allows the algorithm to achieve both parts of the Strong Hygiene Condition.

The macro expander recursively traverses the rewritten expression. It renames `lambda.2` back to `lambda`, as specified by the binding $[\texttt{lambda.2} \mapsto \lambda]$, and renames the bound variable `x` to `x.3`, thus converting the subexpression

```
(lambda.2
  (x)
  (let-syntax
    ((m (syntax-rules
          ()
          ((m ?m2 ?body)
           (let-syntax
             ((?m2 (syntax-rules () ((?m2) x))))
             ?body)))))
    (let ((x 1)) (m m2 (let ((x 2)) (m2))))))
```

into

```
(lambda
  (x.3)
  (let-syntax
    ((m (syntax-rules
          ()
          ((m ?m2 ?body)
           (let-syntax
             ((?m2 (syntax-rules () ((?m2) x))))
             ?body)))))
    (let ((x 1)) (m m2 (let ((x 2)) (m2))))))
```

and producing a new syntactic environment $\rho_6 = \rho_4[\texttt{x} \mapsto \texttt{x.3}] = \rho_0[\texttt{lambda.2} \mapsto \lambda][\texttt{x} \mapsto \texttt{x.3}]$ in which `x` is bound to the fresh identifier `x.3`.

Recursively traversing the body of this `lambda` expression in the syntactic environment ρ_6, the macro expander encounters the outer `let-syntax`. That `let-syntax` rewrites to its body

```
(let ((x 1)) (m m2 (let ((x 2)) (m2))))))
```

(The identifiers in this expression were in the body of the `let-syntax`, rather than being generated by the `let-syntax` macro, so they were not renamed.)

This expression will be expanded in a new syntactic environment $\rho_7 = \rho_6[\texttt{m} \mapsto M_1]$ where M_1 is a macro denotation that consists of two parts:

(1) the rewrite rules, obtained from the syntax rules for `m` as specified by that outer `let-syntax`
(2) the syntactic environment ρ_6 that specifies the intended meanings for all identifiers that might be introduced by those rewrite rules

Rewriting that `let` expression in syntactic environment ρ_7 yields

```
((lambda.5 (x) (m m2 (let ((x 2)) (m2)))) 1)
```

together with a new syntactic environment $\rho_{10} = \rho_7[\texttt{lambda.5} \mapsto \lambda]$.

The `(lambda.5 (x) ...)` subexpression is alpha-converted to

```
(lambda (x.6) (m m2 (let ((x 2)) (m2))))
```

together with a new syntactic environment $\rho_{12} = \rho_{10}[\texttt{x} \mapsto \texttt{x.6}]$.

Continuing its recursive traversal, the macro expander encounters

```
(m m2 (let ((x 2)) (m2)))
```

in the syntactic environment ρ_{12}. In that environment, `m` denotes the macro denotation M_1, which was closed in the syntactic environment ρ_6. Rewriting that use of `m` according to its rewrite rule proceeds as follows:

- The pattern variables `?m2` and `?body` match `m2` and `(let ((x 2)) (m2))`, respectively.
- The rewrite rule for `m` introduces identifiers `let-syntax`, `syntax-rules`, and `x`, for which the rewriting process introduces fresh aliases `let-syntax.7`, `syntax-rules.7`, and `x.7`.
- With those aliases and substitutions for the pattern variables of `m`, the use of macro `m` is rewritten into

  ```
  (let-syntax.7
    ((m2 (syntax-rules.7 () ((m2) x.7))))
    (let ((x 2)) (m2)))
  ```

  together with a new syntactic environment ρ_{14} that is obtained from ρ_{12} by:
  – binding `let-syntax.7` to the meaning of `let-syntax` in ρ_6 (which is the usual meaning of `let-syntax`, which we will write as *letsyntax*)

  - binding `syntax-rules.7` to the meaning of `syntax-rules` in ρ_6 (which is the usual meaning of `syntax-rules`, which we will write as *syntaxrules*)
  - binding `x.7` to the meaning of `x` in ρ_6 (which is `x.3`)
- So we get

$$\begin{aligned}\rho_{14} = \rho_{12}&[\texttt{let-syntax.7} \mapsto \textit{letsyntax}]\\ &[\texttt{syntax-rules.7} \mapsto \textit{syntaxrules}]\\ &[\texttt{x.7} \mapsto \texttt{x.3}]\end{aligned}$$

The macro expander then expands the `let-syntax.7` expression according to syntactic environment ρ_{14}. In ρ_{14}, `let-syntax.7` is bound to the usual meaning of `let-syntax`, so the expression is rewritten into its body

```
(let ((x 2)) (m2))
```

This `let` expression is expanded in a new syntactic environment $\rho_{15} = \rho_{14}[\texttt{m2} \mapsto M_2]$ where M_2 is a macro denotation that consists of the rewrite rules given by

```
(syntax-rules.7 () ((m2) x.7))
```

together with the syntactic environment ρ_{14}.

Expanding `(let ((x 2)) (m2))` in syntactic environment ρ_{15} yields

```
((lambda.9 (x) (m2)) 2)
```

together with a new syntactic environment $\rho_{19} = \rho_{15}[\texttt{lambda.9} \mapsto \lambda]$. That `lambda.9` expression is then alpha-converted into `(lambda (x.10) (m2))` together with a new syntactic environment $\rho_{20} = \rho_{19}[\texttt{x} \mapsto \texttt{x.10}]$.

Continuing its recursive traversal, the macro expander expands the expression `(m2)` in syntactic environment ρ_{20}. In that environment, `m2` is the macro denotation M_2, which was closed in environment ρ_{14}. The rewrite rule for `m2` says every use of `m2` should rewrite into a reference to `x.7`. Since `x.7` is not a pattern variable of the rule, the rule introduces `x.7`. As was seen above when the use of `m` was rewritten, the macro expander introduces fresh names for all introduced identifiers. The expression `(m2)` therefore becomes

```
x.7.11
```

together with a new syntactic environment $\rho_{22} = \rho_{20}[\texttt{x.7.11} \mapsto \texttt{x.3}]$, because in ρ_{14} (the environment part of M_2) `x.7` is bound to `x.3`.

Finally, expanding `x.7.11` in syntactic environment ρ_{22} yields a reference to `x.3`. As the macro expander backs out of its recursive traversal, it puts together the final macro-expanded version of the original expression:

```
((lambda (x.3)
   ((lambda (x.6)
      ((lambda (x.10) x.3) '2))
    '1))
 '0)
```

The final value of this expression is the value of `x.3`, so the expression evaluates to `0`, as it should.

7.4 Explicit Renaming

Jonathan Rees built his implementation of the "Macros That Work" algorithm on top of a new low-level (procedural) macro system he designed for the specific purpose of implementing the "Macros That Work" algorithm. That low-level system, known as *explicit renaming* [Clinger 1991a; Rees 1993], has become one of the more widely used systems for writing non-hygienic macros, partly because it interoperates perfectly with `syntax-rules` and can be made to interoperate perfectly with the `syntax-case` mechanism described in Section 9.

The main disadvantage of explicit renaming is that it requires programmers to perform all time-stamping themselves, which is tedious and error-prone.

Using explicit renaming in the implementation described in Section 7.7, the `loop` macro of Figure 3 can be defined as follows:

```
(define-syntax loop
  (transformer
   (lambda (x rename compare)
     (let ((cmds (cdr x))
           (call/cc (rename 'call-with-current-continuation))
           (lambda (rename 'lambda))
           (define (rename 'define))
           (the-loop (rename 'the-loop)))
       `(,call/cc
         (,lambda (exit)
          (,define (,the-loop)
           ,@cmds
           (,the-loop))
          (,the-loop)))))))
```

The three arguments of the procedure passed to `transformer` are

(1) an uninterpreted abstract syntax tree (`x`) for the use
(2) a 1-argument procedure (`rename`) that adds a time stamp to its argument; within any transcription step, all calls to the renaming procedure will use the same time stamp, but every expansion of a macro use will use a different time stamp
(3) a 2-argument procedure (`compare`) that compares two identifiers, returning true if and only they have the same denotation in the syntactic environment that guides expansion of the macro's use

In the `loop` macro above, identifiers that appear within the `cmds` subexpression will not be given new time stamps, but most of the other identifiers that will appear within the transcribed output are time-stamped using the rename procedure, just as they would be by the `syntax-rules` mechanism; the only difference is that `syntax-rules` does the time-stamping automatically, whereas explicit renaming requires the time-stamping to be done explicitly by the programmer.

The `exit` identifier is not renamed, however, because we want it to capture occurrences of `exit` that occur free within the list of commands `cmds`.

The `free-identifier=?` procedure used with `syntax-case` (to be described in Section 9) behaves like the `compare` procedure described above. In early implementations of explicit renaming, identifiers were represented as raw symbols, and the behavior of the `bound-identifier=?` procedure used with `syntax-case` could be obtained by using `eq?` to compare those symbols. André van Tonder added `bound-identifier=?` when the representation of identifiers became more sophisticated in his implementation of `syntax-case` on top of explicit renaming for SRFI 72 [van Tonder 2005b].

See also the discussion of *implicit renaming* in Section 12.3.1.

7.5 Summary of Hygienic Macro Expansion Through 1990

To summarize the history of hygienic macro expansion through 1990, we quote Clinger and Rees [1991a] at length:

The problems with naïve macro expansion have been recognized for many years, as have the traditional work-arounds [Brown 1974]. The capturing problems that afflict macro systems for block-structured languages were first solved by Kohlbecker's work on hygienic macro expansion. Bawden and Rees then proposed syntactic closures [Bawden and Rees 1988] as a more general and efficient but lower-level solution. Our algorithm unifies and extends this recent research, most of which has been directed toward the goal of developing a reliable macro system for Scheme.

At the 1988 meeting of the Scheme Report authors at Snowbird, Utah, a macro committee was charged with developing a macro facility akin to `extend-syntax` [Dybvig 1987] but based on syntactic closures. Chris Hanson implemented a prototype and discovered that an implementation based on syntactic closures must determine the syntactic roles of some identifiers before macro expansion based on textual pattern matching can make those roles apparent [personal communication]. Clinger observed that Kohlbecker's algorithm amounts to a technique for delaying this determination, and proposed a linear-time version of Kohlbecker's algorithm. Rees married syntactic closures to the modified Kohlbecker algorithm and implemented it all, twice. Bob Hieb found some bugs in the first version and proposed fixes.

A high-level macro system similar to that described here is currently implemented on top of a compatible low-level system that is not described in this paper. Bob Hieb and Kent Dybvig have redesigned this low-level system to make it more abstract and easier to use, and have constructed yet another implementation. It is expected that both the high-level and low-level macro facilities will be described in a future report on Scheme [Clinger and Rees 1991b].

7.6 R4RS

That "future report on Scheme" would be the R4RS. The R4RS was delayed while Hieb and Dybvig designed a low-level procedural macro system that would be compatible with and could be used to implement `syntax-rules` [Clinger and Rees 1991b]. Explicit renaming would have served that purpose, but the new system was expected to be better because it would be hygienic by default. Both `syntax-rules` and the low-level Hieb-Dybvig system were described in an appendix to the R4RS [Clinger and Rees 1991b].

The low-level macro system described in that appendix never saw much use, and would be dropped from the R5RS in 1998, but it was the precursor of `syntax-case`, which will be described in Section 9.

7.7 Portable Implementations of Macros That Work

Between 1989 and 1992, at the University of Oregon, Clinger wrote a new optimizing compiler for Scheme, which he called "Twobit" in homage to previous Scheme compilers such as "Rabbit" [Steele 1977], "Orbit" [Adams et al. 1986], and "Gambit" [NA Feeley (no date)]. Lars Hansen, a master's student who was interested in how generational garbage collection affects performance and how it should influence programming style, built several interchangeable garbage collectors. They combined those foundational components with library code derived from MacScheme, obtaining a new implementation of Scheme that became known as Larceny (because that's what "Lars's implementation of Scheme" begins to sound like after you've said it a few hundred times) [NA Hansen 1992; Clinger and Hansen 1994].

When Clinger re-implemented the “Macros That Work” algorithm for Larceny, his re-implementation remained compatible with and continued to support explicit renaming. That implementation was added to the SLIB suite of libraries [Jaffer 2007] and became the basis for most implementations of `syntax-rules` in R4RS and R5RS systems. Unfortunately, the SLIB implementation accumulated several bugs as various people, trying to make macro expansion run faster in systems based on slow interpreters, extended the macro expander to handle more syntaxes directly instead of implementing those syntaxes as hygienic macros. Two lessons to be learned from those bugs are:

- Slow interpreters encourage programmers to waste time rewriting code that would be fast enough in compiled systems.
- It is easier to give correct definitions of syntactic extensions using `syntax-rules` and related hygienic macro technology than it is to write correct low-level procedural code.

8 STRONG HYGIENE FOUND TO BE NOT SO STRONG AFTER ALL

Clinger and Rees [1991a] claimed, without proof, that their “Macros That Work” algorithm implemented a version of `syntax-rules` that satisfies the Strong Hygiene Condition stated in Section 7.

That claim was probably true.

Clinger [1991b] made a similar claim for the version of `syntax-rules` described in an appendix to the R4RS:

> The primary limitation of the hygienic macro system is that it is thoroughly hygienic, and thus cannot express macros that bind identifiers implicitly.

Whether that claim was true depends on exactly what is meant by “thoroughly hygienic” and “macros that bind identifiers implicitly.”

8.1 Petrofsky Extraction

The version of `syntax-rules` described by the R4RS differed from the version described in “Macros That Work” by the addition of one new feature: a list of syntactic keywords (such as `else`) whose occurrences within a pattern would not be interpreted as pattern variables, but would instead be matched against the corresponding part of a macro use. The semantics of that matching was left a bit fuzzy, but all plausible semantics allowed some intuitively non-hygienic macros, such as the `loop` macro of Figure 3, to be defined using `syntax-rules`.

Al Petrofsky discovered that macros defined using `syntax-rules` can exploit the special matching for syntactic keywords by searching the entire macro use for occurrences of those keywords [NA Petrofsky 1991]. Any occurrence it finds will be an identifier that has already been time-stamped. The macro can then expand into code that refers to that previously time-stamped identifier instead of inserting a newly time-stamped identifier of the same name. Although such macros do not actually violate the hygiene condition, they can behave as though they capture those previously time-stamped identifiers.

Even when applicable, Petrofsky extraction is not the best technique to use when defining macros such as `loop`:

- Petrofsky extraction is hard to use correctly. For example, getting nested loops to work correctly takes more care and is more complicated than we have shown here.
- Macros defined using Petrofsky extraction do not compose easily [NA Petrofsky 1991]: “The general rule, therefore, is that all macros that implicitly bind a particular identifier must be defined with knowledge of one another, so that a use of any one can create new local bindings for all the others as well as for itself.”
- As in Kohlbecker’s algorithm, traversing the entire use is slow. For example, nested uses of the `loop` macro would take time quadratic in the depth of the nesting.

Perhaps the most amazing thing about Petrofsky extraction is that it defines effectively non-hygienic macros *without actually violating the Strong Hygiene Condition.* In the `loop` macro, for example, the bindings inserted by the macro capture only the references that were inserted by the macro during its traversal of the use, and the macro does not insert any references that can be captured by bindings other than those inserted by the macro.

Petrofsky extraction led to more skeptical re-examination of the hygiene conditions and what they actually accomplish, along with renewed appreciation for the importance of giving rigorous proofs for claims that some specific version of the hygiene condition is achieved by some particular system. As Petrofsky himself wrote [NA Petrofsky 1992]:

> What was set out to be prevented was "inadvertent captures" [r5rs 4.3.2]. `Syntax-rules` is wildly successful at that. Thwarting deliberate captures was not a design goal. The seminal papers on hygiene all refer to deliberate capture as an occasionally useful and legitimate technique, not an evil to be extinguished.
>
> Oleg [Kiselyov] and I have been debating in email what the larger significance of our findings is. I tend to think it is not a whole lot. The lesson of the loop macro is that it's easy to take a vague definition of hygiene and wrongly conclude that some macro cannot be written hygienically. Once you nail down the definition, it becomes more apparent what is and is not possible.

Petrofsky extraction depends upon the macro's ability to examine and modify the entire abstract syntax tree for its use. In macro systems that supply a macro with opaque representations of the abstract syntax trees for its subcomponents, Petrofsky extraction would not work. Petrofsky extraction therefore serves as a (somewhat arcane) example of the additional expressive power afforded by allowing macros to examine the entire use, re-arranging it as desired. (Whether additional expressive power is good or bad is always open to debate.)

Petrofsky extraction helped to legitimize more reasonable mechanisms that would allow controlled escape from hygiene; some of those mechanisms are described below. If a syntax could be defined by writing some wild and crazy hygienic macro, why not make it easier to write a straightforward partially-hygienic macro that implements the same syntax?

8.2 Kiselyov Defilement

Ten years later, Oleg Kiselyov demonstrated that the hygiene conditions do not prevent Petrofsky extraction from being used for wholesale defilement of all standard binding constructs, starting with a seemingly non-hygienic version of `lambda` [Kiselyov 2001, 2002]:

> The overloaded `lambda` acts as if it was infected by a virus, which propagates through the `lambda`'s body infecting other `lambda`s in turn. . . . Although we eventually subvert all binding forms, we preserve the semantics of Scheme as given in R5RS.

The Kiselyov defilement works only for Scheme as defined by the IEEE and R5RS standards, which explicitly allow re-definition of `lambda` and all other special forms [Bartley et al. 1991; Kelsey et al. 1998]. Previous standards had not explicitly allowed such re-definitions [Clinger et al. 1985; Rees and Clinger 1986; Clinger and Rees 1991b]. The IEEE and R5RS standards explicitly allowed re-definitions of `lambda` *et cetera*, but required such re-definitions to have no effect on the behavior of special forms and procedures that were not redefined [Bartley et al. 1991; Kelsey et al. 1998]; full-scale Kiselyov defilement must therefore redefine all of Scheme's standard binding constructs, not just `lambda`. The R6RS standard allowed renaming but explicitly forbade re-definition or assignments to imported identifiers [Sperber et al. 2009a]. Although the R7RS standard neither forbids nor allows re-definition or assignments to imported identifiers, that is widely regarded as an oversight, and most implementations follow the R6RS in this matter; the R6RS `library` syntax and the R7RS `define-library` syntax then have essentially the same semantics, with the R7RS syntax adding a few new features while eliminating some arbitrary restrictions imposed by the R6RS syntax [Shinn et al. 2013; Clinger 2015].

9 SYNTAX-CASE

The most challenging problem facing a hygienic macro expander is determining which parts of the output will come from abstract syntax trees that were already present in the macro use (so they don't need any renaming) and which parts are being inserted by the macro (so their identifiers do need to be renamed to avoid conflict with identifiers in scope at the macro-call site).

Kohlbecker's algorithm solved this problem, even for procedural macros, by traversing the entire tree at every expansion, but that expansion might require quadratic or even exponential time. Syntactic closures failed because they required the macro writer to specify which was which before the necessary information was available. Macros That Work solved the problem for fully-hygienic pattern-based macros by having the pattern matcher return not only the transcribed (partially macro-expanded) expression but also an extended syntactic environment that guided any further expansion of the transcribed expression.

Wouldn't it be nice to combine the speed, convenience, and automatic hygiene of `syntax-rules` with the expressive power of procedural macros and the ability to escape from hygiene when needed?

Kent Dybvig, Bob Hieb, and Carl Bruggeman designed a facility called `syntax-case` that accomplished all of that [Hieb et al. 1992; Dybvig 1992; Dybvig et al. 1993; Dybvig 2000]. Their algorithm also maintained the correlation between original source code and macro-expanded code needed for good error reporting and debugging.

9.1 Syntax Objects

The key insight of their algorithm is that the abstract syntax trees manipulated by procedural macros do not have to be limited to the kinds of S-expressions the `read` procedure can produce from the original source code.

This paper introduced the notion of *syntax objects*, which constitute both the inputs and outputs of a macro expansion step. For Dybvig et al., a syntax object is an S-expression along with some additional information, called the *wrap*. The wrap must contain enough information to support the "lazy" variant of Kohlbecker's algorithm explained below. The wrap may also contain other information, such as the location in which the S-expression appeared in the source code, to allow for source-code tracking.

In this lazy algorithm, the traversal of an abstract syntax tree is deferred until the tree needs to be matched against a pattern or decomposed by a procedural macro. When a syntax object is matched against a pattern, the algorithm automatically unwraps just enough of the syntax object to do the matching, pushing the encapsulated information down into subtrees whose structure does not need to be examined by the matcher.

As in the `syntax-rules` system, a macro definition is specified through a sequence of patterns. For each pattern, the programmer writes procedural code that creates a syntax object as the output of the macro.

The output syntax object may contain newly inserted identifiers that need to be given new time stamps. Kohlbecker's algorithm would discover those identifiers by traversing the entire expression represented by the syntax object, but the Hieb-Dybvig-Bruggeman algorithm traverses only the unwrapped portions of the syntax object. The wrapped portions do not need to be traversed because they will always correspond to syntax objects that have already matched a pattern variable, so they cannot contain any newly inserted identifiers. As a result, the Hieb-Dybvig-Bruggeman algorithm avoids the repetitious traversals of subexpressions that are responsible for the asymptotic inefficiency of Kohlbecker's algorithm.

In simple cases, the syntax object returned by a `syntax-case` macro is created by writing `(syntax <template>)` or, equivalently, `#'<template>`, where `<template>` has the same form as the right hand side of a `syntax-rules` rule; any pattern variables that appear within the `<template>` will be replaced by the syntax objects they matched. For more complex cases, the programmer can create syntax objects using a backquote-like mechanism by writing `(quasisyntax <template>)` or, equivalently, ``#`<template>``. The `<template>` of a `quasisyntax` expression may contain `unsyntax` and `unsyntax-splicing` forms (usually abbreviated using the two-character sequence "`#,`" and the three-character sequence "`#,@`"), which are analogous to the `unquote` and `unquote-splicing` notations of the backquote notation for ordinary S-expressions in both Common Lisp and Scheme [Steele 1990, §22.1.3; Rees and Clinger 1986, §4.2.6; Sperber et al. 2009a, §11.17; Sperber et al. 2009b, §12.8; Shinn et al. 2013, §4.2.8].

Using `syntax-case`, the `loop` macro of Figure 3 can be defined by

```
(define-syntax loop
  (lambda (x)            ; x will be a syntax object
    (syntax-case x ()
     ((loop cmd ...)
      #`(call-with-current-continuation
         (lambda (#,(datum->syntax #'loop 'exit))
           (define (the-loop)
             cmd ...
             (the-loop))
           (the-loop)))))))
```

In that definition, `(datum->syntax #'loop 'exit)` attaches the time-stamp of the `loop` identifier that occurs within the macro use to the `exit` identifier whose free occurrences are within the commands we want to capture.

Note that `syntax-case` macros are procedural. The code that follows the pattern in the `loop` macro may look like a `syntax-rules` template with some mysterious # characters, but it is in fact ordinary Scheme code. The magic comes from the ``#` `` abbreviation for `quasiquote` and its ability to substitute for pattern variables `loop` and `cmd`, including any ellipses that may follow those pattern variables, and from `quasiquote`'s conversion of the list following the ``#` `` abbreviation for `quasiquote` into a syntax object that includes the time stamps and other information needed for lazy execution of a Kohlbecker-like algorithm.

Figure 4 shows a more substantial example of `syntax-case` being used in a non-hygienic macro. The syntactic layer of R6RS records is non-hygienic because record type definitions define predicates, accessors, and mutators whose names are constructed from the name of the record type and the names of the fields [Sperber et al. 2009b]. Figure 4 shows the non-hygienic part of that record system. It parses and checks the record type definition using the `syntax-case` pattern and several help procedures before creating the new names, which it passes on to another help procedure (`construct-record-type-definitions`) that returns the final syntax object. One of those new names is the predicate name, `pname`; its computation is shown in red. If the record type definition specifies a symbol for the predicate name, then the syntax object representing that symbol becomes the value of `pname`. Otherwise the value of `pname` is computed (non-hygienically!) by appending a question mark to the end of the record type's name-string, converting that string to a symbol, and using `datum->syntax` to convert that symbol into a syntax object that behaves as though it were

```
(syntax-case x ()
 ((_ explicit? rtd-name constructor-name predicate-name clause ...)
  (let* ((type-name (syntax->datum #'rtd-name))
         (type-name-string (symbol->string type-name))
         (clauses #'(clause ...))
         (fields-clause (clauses-assq 'fields clauses))
         (parent-clause (clauses-assq 'parent clauses))
         (protocol-clause (clauses-assq 'protocol clauses))
         (sealed-clause (clauses-assq 'sealed clauses))
         (opaque-clause (clauses-assq 'opaque clauses))
         (nongenerative-clause (clauses-assq 'nongenerative clauses))
         (parent-rtd-clause (clauses-assq 'parent-rtd clauses))
         (okay? <redacted for brevity>)
         (cname <redacted for brevity>)
         (pname
          (if (symbol? (syntax->datum #'predicate-name))
              #'predicate-name
              (datum->syntax
               #'rtd-name
               (string->symbol
                (string-append type-name-string "?")))))
         (field-specs <redacted for brevity>)

    (if (not okay?)
        (complain))
    (construct-record-type-definitions
     #'rtd-name
     cname
     pname
     type-name
     field-specs

     (and protocol-clause (cadr (syntax->datum protocol-clause)))
     (and sealed-clause (cadr (syntax->datum sealed-clause)))
     (and opaque-clause (cadr (syntax->datum opaque-clause)))

     (cond ((eq? nongenerative-clause #f)
            #f)
           ((null? (cdr nongenerative-clause))
            (gensym "uid"))
           (else
            (cadr nongenerative-clause)))
     (cond (parent-clause (cadr parent-clause))
           (parent-rtd-clause (cadr parent-rtd-clause))
           (else #f))
     (and parent-rtd-clause (caddr parent-rtd-clause))))))
```

Fig. 4. The syntactic layer of R6RS records is not fully hygienic: see computation of pname.

time-stamped at the same time as the syntax object `#'rtd-name` that represents the record type descriptor's name as specified within the definition of the record type.

9.2 Early Usage of `syntax-case`

In the 1990s, Chez Scheme and MzScheme were the primary implementations that used `syntax-case` [Dybvig 1987; Waddell and Dybvig 1999; Waddell 1999; Krishnamurthi et al. 2000a,b; Krishnamurthi 2001; Flatt 2002; Culpepper et al. 2007].

As noted in Section 3.4.2, Chez Scheme was implemented by Kent Dybvig.

MzScheme, an interpreter implemented by Matthew Flatt at Rice under the supervision of Matthias Felleisen, was a core component of the PLT Scheme system, which was later renamed to Racket.

Chez Scheme was closed-source at that time, while MzScheme/PLT/Racket's implementation was written in C++ and depended upon many implementation-specific aspects of the system. In the absence of a portable implentation of `syntax-case`, most other implementations of Scheme continued to use explicit renaming or naïve procedural macros as their low-level, non-hygienic alternative to `syntax-rules`.

In August 2000, Kent Dybvig and Oscar Waddell released the first portable implementation of `syntax-case` [Dybvig and Waddell 2000].

In 2005, André van Tonder described a new low-level procedural macro system that was compatible with and included both `syntax-case` and `syntax-rules`; its reference implementation was "strongly influenced by the explicit renaming system" [van Tonder 2005b].

In 2007, `syntax-case` became one of the standard libraries described by the R6RS libraries document [Dybvig 2005; Sperber et al. 2009b], leading to more widespread adoption of `syntax-case`. This story is continued in Section 11.3.

9.3 How It Works

In Dybvig's algorithm, syntax objects are an abstract data type with the following operations:

(1) Syntax objects are constructed using the form `(syntax` *template*`)`, which behaves like `quote`, except it preserves contextual information from the template.
(2) Syntax objects are observed using `syntax-case`. A use of `syntax-case` consists of a sequence of pattern-expression pairs; a pattern may be a pattern variable, a list structure, an identifier, or a constant. Each pattern binds its pattern variables and evaluates the expression in the resulting environment. Right-hand-side expressions are arbitrary Scheme expressions that return syntax objects.
(3) Syntax objects may be compared in two different ways: `free-identifier=?` is used to determine whether two identifiers would be equivalent if they appeared as free identifiers in the output of a transformer; `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 the system of the MTW paper, two identifiers x and y would be `free-identifier=?` in an environment e iff $e(x) = e(y)$; they would be `bound-identifier=?` iff they are the same x'_j from the same macro expansion.)
(4) Context information may be stripped from or added to a syntax object by the operations. `syntax-object->datum` and `datum->syntax-object` The former strips the context information from a syntax object. The latter takes a syntax object and a Scheme datum (usually, but not always, a Scheme identifier) and produces the syntax object that would have been created had that datum appeared in the same context as the syntax object. This is useful for

controlled escape from hygiene. There is no operation to extract the context information from a syntax object.

(5) A syntax object may be *marked*. It is assumed that there is an infinite collection of marks m. Marks have the property that if an object is marked with the mark m, and later marked with m again, the marks cancel.

(6) A syntax object is subject to substitution $subst(e, i, i')$, which returns a syntax object like e except that every occurrence of the identifier i (along with its context information) is replaced by i' (with its context information).

The key lines in the algorithm of Dybvig et al. [1993, Figure 4] are the following:

```
$expand : Exp \times Env \to ExpandedExp$
$expand(e, r) = \textbf{case}\ parse(e, r)\ \textbf{of}$
        $syntax\text{-}binding(i, e_1, e_2) \quad \to \quad expand(subst(e_2, i, s), r[s \to t])$
                                                where $t = eval(expand(e_1, r))$ and $s$ is fresh
        $macro\text{-}application(i, e) \quad \to \quad expand(mark(t(mark(e, m)), m), r)$
                                                where $t = r(i)$ and $m$ is fresh
```

The call to $parse(e, r)$ converts the S-expression e to abstract syntax. The procedure *parse* uses the environment r to determine whether an identifier is a variable, a "special" like `quote` or `lamba`, or the name of a macro; in the last case, r binds the name to its associated transformer.

The *syntax-binding* line corresponds to `(let-syntax ((`i e_1`))` e_2`)`. It macro-expands e_1 and then evaluates it; the result is presumably a function t from syntax objects to syntax objects.[3] It then generates a fresh name s for the macro and expands e_2 in an environment where the name s is bound to the function t; this enables the resulting call to *parse* to recognize that s now indicates a macro call.

When a macro application is expanded, every identifier in the application is marked by adding a fresh mark m. The transformation t is applied to the marked expression, and then the resulting expression is marked with m again. Since double marks cancel, the effect is that the arguments to the transformation remain unchanged. Any variable that appears in the output marked with m must have been generated during the expansion.

The complexity of the algorithm depends on the implementation of syntax objects. In an eager implementation, marks are attached directly to identifiers and substitutions are performed when they are invoked. Therefore the eager algorithm traverses the entire expression for every *subst* and *mark* operation. This leads to possibly exponential behavior, as in Kohlbecker's algorithm.

In a lazy implementation, *subst* and *mark* are treated as new constructors on the data type of syntax objects, reminiscent of explicit substitutions [Abadi et al. 1991]. It then becomes the responsibility of `syntax-case` to unwind these constructors just enough to do the pattern matching. This leads to linear behavior, like that of Clinger and Rees [1991a], at the expense of a more complicated pattern-matching algorithm.

The Dybvig et al. paper used the term "hygiene" without definition, other than to cite the KFFD paper. It defined "referentially transparent" as meaning that

> free identifiers appearing in the output of a local macro are scoped where the macro definition appears [Clinger and Rees 1991a] [citation in original].

[3]The paper does not specify an environment argument for this evaluation. This is the "expansion-time" environment, including whatever functions are necessary for computing the expansion of the macro. In order to guarantee hygiene, one might also want to assume that this function has no non-local state.

This formulation corresponds to the second part of the MTW paper's Strong Hygiene Condition, but avoids the hypothetical contrapositive of the MTW formulation.[4]

The paper makes no argument for the correctness of its algorithm, apparently relying on the reader to understand this from the code.

10 FROM R4RS TO R5RS

In this section and the next, we explain how macro technologies described in previous sections interacted with the development of new standards for Scheme.

Syntax objects and several other basic ideas of what later became `syntax-case` were already present in the low-level procedural macro system that Dybvig and Hieb designed for the macro appendix of the R4RS, which was published in 1991 after being held "hostage to the appendix on macros" (as Clinger and Rees [1991b] put it in their acknowledgements).

That R4RS macro appendix mentions "an alternative low-level macro facility" designed by Hanson and Bawden, which was based on syntactic closures. The macro committee, composed of Dybvig, Hieb, Bawden, and Rees, had "not endorsed any particular low-level facility, but does endorse the general concept of a low-level facility that is compatible with the high-level pattern language described in this appendix."

Shortly after the R4RS was published, Chris Hanson and Alan Bawden figured out how to modify syntactic closures so that system could support `syntax-rules` [Hanson 1991]. For more on the revival of syntactic closures, see Section 12.3.1.

Hieb and Dybvig continued to improve the low-level system they had designed for the R4RS appendix, notably by adding `syntax-case`, whose patterns were an extension of those in `syntax-rules`, with automatic substitution for pattern variables inside `syntax` and `quasisyntax` expressions [Hieb et al. 1992; Dybvig 1992; Dybvig et al. 1993].

Bob Hieb, whose work on `syntax-case` would have gone into his PhD dissertation, died in an automobile accident on 30 April 1992. IU awarded Hieb's PhD posthumously. The R4RS, to which Hieb contributed, had been "Dedicated to the Memory of ALGOL 60." The R5RS, dated 20 February 1998, was "Dedicated to the Memory of Robert Hieb."

With the arrival of `syntax-case` in 1992, there were three procedural macro systems contending for the role of low-level companion to `syntax-rules`: syntactic closures, explicit renaming, and `syntax-case`. With three candidates, it was not possible for the large group of R5RS authors to reach consensus on which one should be described by the report, so the R5RS did not describe any non-hygienic or low-level macro facilities. On the other hand, the high-level and hygienic `syntax-rules` system did make its way into the main body of the report, where it was perceived as the most important change made by the R5RS [NA Lawler 2004; Rettke 2008b; Soegaard 2008; Rettke 2008a; Ethier 2013].

During preparation of the R5RS, Clinger [1993] discovered "two bugs in the design of the high-level macro system described in the appendix to R4RS":

> (1) Many macro-defining macros that I have wanted to write do not work in the R4RS system because ellipses in the macro to be defined are expanded when the defining macro is transcribed. [...]
>
> (2) The QUASIQUOTE macro cannot be written in the high-level macro system of R4RS because it does not support vectors as full-fledged patterns.

The second bug was fixed by allowing vectors as patterns. The first bug was not fixed in the R5RS, but an escape syntax first proposed by Dybvig et al. [1993] became a popular extension to the R5RS macro facility and was eventually standardized in the R6RS [Dybvig 1993; Sperber et al. 2009a].

[4]But of course it begs the question of when an identifier appearing in the output of a local macro is free.

11 FROM R5RS TO R6RS

The R5RS had adopted high-level macros, pretty much as described in "Macros That Work" and in the macro appendix to the R4RS, but had dropped the low-level macro system of that appendix. The three most important things that were missing from the R5RS were a low-level macro system, a module system, and a record system. Hygienic macro technology became the foundation for all three.

11.1 From Consensus to Unanimity to Demise of RRRS Authors

During the heyday of the RRRS authors, a rule of consensus had prevailed. Over time, that rule came to be interpreted as requiring unanimity: a single author of the RRRS documents could veto any proposed change, and non-authors who showed up at a meeting of the RRRS authors or expressed their opinion via electronic mail were often allowed to veto changes as well.

At the 1998 Scheme workshop, the RRRS authors agreed to form a "Scheme Requests for Implementation" (SRFI) process, but were unable to come to any agreement on how to move forward toward an R6RS.

Some new process was needed.

11.2 A New Standards Process

In October 2002, in a session of the annual Scheme Workshop, attendees formed a "Scheme Strategy Committee" to come up with a new process [Bawden et al. 2004]. The members of that committee were Alan Bawden, William Clinger, Kent Dybvig, Matthew Flatt, Richard Kelsey, Manuel Serrano, and Michael Sperber. That committee proposed a draft charter and committee selection process, which were confirmed at the November 2003 Scheme Workshop. That charter called for creation of a "Scheme Language Steering Committee," which would oversee a committee of Editors, disjoint from the steering committee, who would produce a specification for core Scheme, a module system, a macro system, and a set of standard library modules.

In January 2004, the first Scheme Language Steering Committee, consisting of Alan Bawden (Brandeis), Guy Steele (Sun Microsystems), and Mitchell Wand (Northeastern University), named a committee of seven editors. Each was associated with a prominent implementation of Scheme:

- Marc Feeley, editor in chief (Université de Montréal; Gambit)
- Will Clinger (Northeastern University; Larceny)
- Kent Dybvig (Indiana University; Chez Scheme)
- Matthew Flatt (University of Utah; MzScheme)
- Richard Kelsey (Ember Corporation; Scheme 48)
- Manuel Serrano (INRIA; Bigloo)
- Mike Sperber (DeinProgramm; Scheme 48)

By the end of March 2004, the editors had formed lists of issues they might address, classifying five issues as "controversial or difficult but necessary" [Feeley 2004a,b]:

- module system
- non-hygienic macros
- records
- mechanism for new primitive types
- Unicode support

The "most pressing issue was the design of the module system" (which is described below in Section 11.4).

The editors, except for Kelsey, met face-to-face in September 2004 before the Scheme Workshop at Snowbird [Feeley 2004a,b]. By November 2004, some of the editors' email discussions had become rather heated, as editors who had been asked to develop specific proposals began to take offense when other editors suggested changes or questioned some of their design decisions. In April 2005, Kelsey resigned from the editors' committee and was replaced by Anton van Straaten. In May 2005, all of the editors met face-to-face at Northeastern University, where some discussions degenerated into angry disagreements. Feeley and Serrano resigned in January 2006. The Scheme Language Steering Committee named Dybvig as Chair of the editors' committee and named Sperber to the newly created position of Project Editor [Bawden et al. 2006]. The editors' committee ran more smoothly after Dybvig took over as chair, partly because Dybvig did a good job but also because (1) the committee's size had been reduced from seven to five, and (2) Dybvig, Flatt, and Sperber, whose ideas concerning the language Scheme should become were often compatible, now held a majority of the votes.

The status report Dybvig et al. [2006] put together in March said the R6RS should:

- make it possible to create "substantial programs and libraries. . . that run without modification in a variety of Scheme implementations"
- allow hygienic, hygiene-bending, and hygiene-breaking macros to be defined in any scope, and allow new unique datatypes to be defined in any scope
- "allow programmers to rely on a level of automatic run-time type and bounds checking sufficient to ensure type safety while also providing a standard way to declare whether such checks are desired"
- "allow implementations to generate efficient code, without requiring programmers to use implementation-specific operators or declarations"

The second of those goals falls squarely within the scope of our history. It was achieved by adding procedural `syntax-case` macros to the fully hygienic `syntax-rules` system of the R5RS.

The first and third goals sounded fairly harmless but became controversial when they were achieved via absolute requirements stated using "must", "must not", "required", "shall", and "shall not" as defined by RFC 2119 [Bradner 1997; Clinger 2015]. The fourth goal sometimes favored designs that might be more efficient in implementations associated with a majority of the remaining editors, but not in implementations that lacked represention on the editors' committee; for an example, see Section 11.5.3. Clinger, having decided he would vote against ratification of the R6RS draft the editors had developed, resigned from the editors' committee in May 2007.

As announced by the Scheme Language Steering Committee [Bawden et al. 2007a]:

> **Ratification Results**
>
> According to the Scheme Charter, at the end of the review process, the Steering Committee could choose either to finalize the submitted draft or to restart the review process. In order to be sure that the new revised Scheme standard enjoyed wide support among the Scheme community, the Steering Committee adopted a procedure in which there would be a vote on the question of whether the draft should be ratified, and they agreed to be bound by the results of that vote.
>
> 112 people registered to vote on the question. 102 of those electors cast their ballots before the poll closed on 12 August 2007. 67 electors voted to ratify draft 5.97 as R6RS. 35 electors were opposed to ratification. Thus 65.7% of those who voted, voted in favor of ratification. This is more than the 60% required for ratification.
>
> The Steering Committee therefore ratifies the draft numbered 5.97 as the official "Revised^6 Report on the Algorithmic Language Scheme". The Editors are directed to prepare the final text of the document, correcting only minor errata.

Of the issues that led more than a third of the electorate to oppose ratification, the matters discussed in Section 11.5.3 are most relevant to our history of hygienic macro technology.

11.3 Implementations of the R6RS

The R6RS standard remained controversial following its ratification. Much of that controversy had little to do with macros, but the new requirements for both `syntax-case` and `library` must have been intimidating to many implementers. Section 11.4.2 describes two portable implementations of `syntax-case` and the R6RS `library` system that greatly simplified those aspects of the task. Implementers of the R6RS soon found that its major challenges lay in its input/output system, support for Unicode, exception system, and (for those who had not already implemented the full numeric tower as described by the R5RS) the number system, with a substantial number of lesser chores lurking throughout the specifications.

On 30 October 2007, only five weeks after the ratified and corrected version of the R6RS was published online, Dybvig's PhD student Abdulaziz Ghuloum [NA 2007] announced the first public release of "Ikarus—the compiler of choice for R6RS hackers," saying it "supports over 80% of the most important features of R6RS." Eight days later, Clinger and Klock [NA 2007] announced the release of Larceny 0.95 "First Safety," saying it is "the first complete implementation of an R6RS-compatible system."

Larceny, Racket, Chez Scheme, Sagittarius, and Vicare eventually delivered implementations that were nearly 100% complete/conforming, as judged using the 8897 tests found in Racket's R6RS test suite (written by Matthew Flatt with contributions from Ghuloum and others) and a set of R6RS benchmarks collected or written by Clinger and distributed with Larceny's source code [Clinger 2015]. Clinger now estimates Ikarus 0.0.1 had been about 70% complete and Larceny 0.95 about 90% complete.

Racket itself never became an important implementation of the R6RS, partly because Racket adopted a native data type of immutable lists that are incompatible with the mutable lists of its R6RS mode, making it hard for R6RS programs to interoperate with Racket's standard libraries.

Ypsilon, which implemented a usable subset of the R6RS including hygienic macros, featured a "mostly concurrent" garbage collector that was developed to support applications such as computer games [NA Fujita (no date)].

IronScheme was said to implement "over 99% of the R6RS specification" on Microsoft's DLR [NA Pritchard (no date)]. Guile "mostly implements R6RS" [NA Jerram et al. (no date)]. BiwaScheme was said to offer "most features of the R6RS Base library" but did not support any part of the R6RS macro systems [NA Hara (no date)].

A couple of other partial implementations have been released, but were never completed and are no longer being maintained. Michael Sperber announced his intention to support R6RS in Scheme 48, but that ambition was never realized [Feeley 2007; Shinn 2011; Sperber 2012].

Ikarus became the most popular implementation of the R6RS, but Ghuloum stopped maintaining it after he completed his PhD in January 2009. Marco Maggi's fork of Ikarus, named Vicare, took over as the most popular implementation [NA Maggi (no date)]. After achieving a degree of conformance that allowed Vicare to pass more than 99.9% of the Racket tests, Maggi decided "full compatibility is of no real value" for the Vicare project and began to introduce changes that would "make some fully compatible R6RS code not to compile anymore" [NA Maggi 2016a,b].

In May 2016, Chez Scheme was released as open source; as of this writing, more than 700 forks have been created [NA Dybvig (no date)].

Those implementations show the extent to which `syntax-case` was implemented and available for use following its standardization in R6RS.

11.4 Library Systems

Hygienic macro technology is about resolving names that may have been defined in different scopes without creating inadvertent clashes. It was natural to apply that technology to module systems that allow selective import and export of names without clashes. For example: Variables/keywords with the same name can be defined within two different libraries and can still be imported into a program or library with appropriate renaming, so R6RS libraries are about avoiding clashes of variable names via alpha-conversion. In turn, allowing libraries to export hygienic macros or partially hygienic procedural macros required some extensions to algorithms we have described in previous sections.

Several module systems for Scheme had been proposed during the years following Kohlbecker's development of a hygienic macro expander. Although these proposals varied in many ways, most recognized the close connection between macros and modules [NA Rees 1989] [Curtis and Rauen 1990] [NA Queinnec and Padget 1990] [Queinnec and Padget 1991a,b] [NA Blume 1995] [Waddell and Dybvig 1999; Waddell 1999].

Chez Scheme, MzScheme, and Scheme 48 had been early adopters of hygienic macro technology. They were also among the first implementations of Scheme to support modules. The Chez Scheme and MzScheme module systems were implemented using the `syntax-case` technology, which was based on hygienic procedural macros. The design of the R6RS `library` system was based upon, but simpler than, the module systems of MzScheme and Chez Scheme [Flatt and Dybvig 2006; Sperber et al. 2009a].

The following library definition, adapted from an example given in the R6RS, illustrates several features of the R6RS library system:

```
(library (mymacros valuesStuff)
  (export mvlet
          contains-duplicates?)
  (import (rnrs base)
          (rnrs syntax-case)
          (for (rename (only (mymacros idStuff) find-dup)
                       (find-dup contains-duplicates?))
               expand))

  (define-syntax mvlet    ; multiple-value let
    (lambda (stx)
      (syntax-case stx ()
        [(_ [(id ...) expr] body0 body ...)
         ;; don't match if (id ...) contains duplicates
         (not (contains-duplicates? (syntax (id ...))))
         ;; return transcribed expression as syntax object
         (syntax
           (call-with-values
             (lambda () expr)
             (lambda (id ...) body0 body ...)))])))
```

That definition says:

- The name of the library is `(mymacros valuesStuff)`, which illustrates the fact that R6RS library names are sequences of identifiers surrounded by parentheses. Those identifiers usually spell out a relative path from some source directory to the file that contains the library's code.

- The library exports two identifiers, `mvlet` and `contains-duplicates?`, with the meanings they have within the library definition: `mvlet` is bound to the syntax defined by the macro definition in the body of the library, and `contains-duplicates?` is bound to the `find-dup` procedure exported by the `(mymacros idStuff)` library.
- The library imports all bindings that are exported by the standard R6RS libraries `(rnrs base)` and `(rnrs syntax-case)`.
- The library also imports from a library named `(mymacros idStuff)`, but imports only the `find-dup` procedure exported by that library. It also renames the `find-dup` procedure to `contains-duplicates?`, and declares that the `contains-duplicates?` procedure will be used only at macro expansion time; see Section 11.4.1 below.
- The library body contains one definition, which defines the `mvlet` macro that will be one of the library's two exports.

11.4.1 Phases. Procedural macros create issues of phasing and ordering: before some procedure definition could use a macro, the macro had to be defined; before that macro could be defined, its help procedures had to be defined; before they could be defined, the macros they use had to be defined; before those macros could be defined... and so on.

Following precedent established by MzScheme [Flatt 2002], the R6RS `library` syntax included the `for` feature, as was used in the example above, that allowed programmers to declare whether bindings imported from other libraries should be imported for run time or for macro expansion. A `meta` keyword dealt with more complicated situations such as a macro expansion phase that would be needed before the help procedures called by another phase of macro expansion could be defined. The only purpose of the `for` feature was to detect phase errors that could occur when using procedural macros such as `syntax-case`. R6RS programmers soon tired of fussing with these phase declarations, so most implementations of the R6RS no longer enforce them. As explained by Ghuloum and Dybvig [2007]:

> These declarations can become unwieldy. The declarations are also imprecise in nature, ...which in turn leads to unnecessary compile-time overhead as bindings are evaluated in some cases when they are not actually needed.
>
> A better alternative is to shift the burden of determining and declaring the phases at which each library's keyword and variable bindings must be evaluated from the user to the implementation....
>
> Having implemented and seen the benefits of implicit phasing, we lobbied the R^{6}RS editors to switch to implicit phasing. We got our way—partly. The subsequent draft...and each that followed allows implementations to support either implicit or explicit phasing. This means that programs to be run in an implementation that requires phase declarations must include them, but other programs can leave them out. A future version of the report, e.g., R^{7}RS, may mandate either implicit or explicit phasing once the community settles on a de facto standard, and we believe the relative simplicity and efficiency of implicit phasing will win out in the end.

As they predicted, the R7RS `define-library` syntax makes no provision for explicit phasing. All implementations of the R7RS that support procedural macros of any kind are using implicit phasing, and no programmers have complained about that *de facto* standard.

11.4.2 Implementations of the R6RS Library System. Abdulaziz Ghuloum and Kent Dybvig released their portable `psyntax` implementation of `syntax-case` and the R6RS `library` syntax, which was then used by most implementations of the R6RS [Ghuloum and Dybvig (no date)]. Independently, André van Tonder updated his own portable implementation to support the same features, plus explicit renaming [van Tonder (no date)].

11.5 Record Systems

Hygienic macro technology had succeeded so well that some began to perceive Scheme's syntactic extensibility as its most important feature, even at the expense of Scheme's historical identity as a higher-order language with first-class procedures and minimal syntax. That elevation of syntax over procedures can be seen in the R6RS record system, where its influence was not altogether benign.

11.5.1 Records Before R6RS. In SRFI 99, which proposed a replacement for the R6RS record system, Clinger [2009] reviewed the long history of record systems that have been proposed for Scheme. In the following quotation, we change the style of citations to match ours and elide most of a paragraph we will quote in Section 11.5.3.

> The importance of adding records to Scheme has been recognized for more than twenty years [Clinger 1987]. The basic idea behind the SRFI 9 and R6RS record systems was outlined by Norman Adams on 8 July 1987, following similar ideas that had been implemented in T and MIT CScheme [Adams 1987]. Jonathan Rees posted a revision of Adams's proposal on 26 May 1988 [Rees 1988b]. Pavel Curtis proposed an extension of Rees's proposal on 18 August 1989, noting that it had been approved by consensus at the first meeting of BASH (Bay Area Scheme Hackers?) [Curtis 1989]. The rrrs-authors archive includes several responses to these proposals that are worth reading.
>
> The Rees/Curtis proposal was revived in 1992 [Adams 1992]. When the RnRS authors met on 25 June 1992 in Palo Alto, they felt that this proposal needed more discussion [Clinger 1992]. Kent Dybvig objected to the proposal on several grounds, including the provision of inspection facilities, the inability to define immutable records, and the use of procedures instead of special forms. Although 9 authors favored adoption of the records proposal, 11 opposed it.
>
> The topic of records was revived again on 23 April 1996 by Bruce Duba, Matthew Flatt, and Shriram Krishnamurthi [Krishnamurthi 1996]. Alan Bawden and Richard Kelsey observed that the Duba/Flatt/Krishnamurthi proposal was essentially the same as Pavel Curtis's, which Kelsey reposted. Kent Dybvig objected once again, on the same three grounds. He also argued that procedural interfaces are difficult to compile efficiently, and that this inefficiency would create portability problems [Dybvig 1996b].
>
> In reality, however, procedural interfaces add no inefficiency....
>
> On 24 April 1996, Bill Rozas suggested the idea of having two separate APIs, one procedural and one syntactic, for the same record facility [Rozas 1996]. Two days later, Dybvig proposed a compromise along those lines [Dybvig 1996a] that incorporated several artificial restrictions, which were apparently motivated by concerns about the alleged extra load instruction [Dybvig 1996c]. Dybvig and Rozas continued to develop this proposal, and presented a summary of it following the 1998 Scheme Workshop [NA Clinger 1998]. I have been unable to locate a written or online copy of this proposal.
>
> SRFI 9, submitted by Richard Kelsey in July 1999, is a syntactic API in the tradition of the Rees, Curtis, and Duba/Flatt/Krishnamurthi proposals [Kelsey 1999].

Single inheritance was added by Larceny in 1998, and by Chez Scheme in 1999 [Clinger et al. 2005].

SRFI 57, submitted by Andre van Tonder in September 2004, features label polymorphism, which can be considered a form of structural subtyping and multiple inheritance [van Tonder 2005a].

11.5.2 Design of the R6RS Record System. For years, hygienic macro technology had both assisted and constrained the standardization and adoption of record systems for Scheme. Before R6RS, the most widely used record system was the one specified by SRFI 9, which could be implemented using `syntax-rules` because record type definitions had to list all of the names to be used for constructors, predicates, accessors, and mutators [Kelsey 1999]. With standardization of `syntax-case` in the R6RS, it became feasible to design record systems in which succinct descriptions of record types macro-expand into implicit definitions of record constructors, predicates, accessors, and mutators whose names are derived from the name of the record type and the names of the fields. Such implicit naming was not possible with the fully hygienic behavior of `syntax-rules`.

In 2005, the R6RS editors invited public comment on a proposed record system with single inheritance, comprising these four parts [Clinger et al. 2005]:

- a procedural API for creating record types
- a syntactic layer for defining record types whose constructors, predicates, accessors, and mutators would be explicitly named
- a syntactic layer for defining record types whose constructors, predicates, accessors, and mutators could be given implicit names derived from the name of the record type and the names of the fields
- a procedural API for reflection

That proposal's reference implementation used `syntax-rules` to define the explicit-naming syntactic layer via fully hygienic macro-expansion into the procedural API. The implicit-naming layer was implemented via macro-expansion into the explicit-naming layer, using either `syntax-case` or the explicit renaming system of Scheme 48 to convert from implicit to explicit names.

Although that proposal was criticized for being overly complicated, it was self-consistent: The four parts were compatible; in particular, record types created using the procedural API could be extended via inheritance using either of the syntactic layers, and vice versa.

By 2007, the editors had elevated the syntactic layer over the procedural API, making the most common cases of record definitions even more complicated while introducing an incompatibility between the syntactic and procedural layers that will be discussed in Section 11.5.3 below.

11.5.3 Causes and Consequences of the R6RS Record System Controversy. During the first round of formal comments on draft 5.91, Clinger [2006] noted that the procedural and syntactic layers were not orthogonal. The procedural layer was strictly more expressive than the syntactic, which would make sense if the syntactic layer were regarded as a convenience layer tailored to common cases, but in that case the syntactic layer should be simplified to make common cases more convenient. The editors responded by claiming the syntactic layer was present not as a convenience, but because it could "be implemented more efficiently" than the procedural layer. That claim was false [Clinger 2009; Keep and Dybvig 2012]. As Clinger [2009] explained in SRFI 99:

> It is now agreed that syntactic interfaces offer no advantages for generative records [Dybvig 2007]. Even for non-generative records, the claimed inefficiency consists of a single load instruction, which optimizing compilers can eliminate—along with the entire runtime check that includes the load instruction—for all but

> the first of a sequence of operations that access the same record. (That optimization is a straightforward extension of the optimization that eliminates the pair check when computing the `cdr` of a list whose `car` has already been computed.) Furthermore, it turns out that even the occasional load instruction is no harder to remove using a procedural interface than when using a syntactic interface [Clinger 2007c]. In R6RS library chapter 6, therefore, both of the statements that claim an advantage in efficiency for the syntactic layer have no basis in fact. (These two statements appear in the next-to-last paragraph before section 6.1, and in the note that follows the specification of `parent-rtd`.)

Before the editors acknowledged the syntactic layer was no more efficient than the procedural layer, the R6RS had been put up for ratification [Clinger 2007d; Dybvig 2007].

In a formal comment, van Tonder [2007b] complained about a serious incompatibility between the procedural and syntactic layers. In response, the editors acknowledged the problem and referred to a new feature they had added three days later, in the draft that was put up for ratification and would eventually be ratified. That new feature made things even worse [Clinger 2007e]:

> A last-minute change in the 5.97 draft attempted to fix things by piling yet another feature, a `parent-rtd` clause, on top of the syntactic layer [Sperber et al. 2007]. The presumed purpose of that `parent-rtd` clause was to address Andre van Tonder's observation that incompatibilities between the syntactic and procedural record layers create a modularity problem: You cannot define a new record type that inherits from an existing record type without knowing whether the base type was defined by the syntactic or by the procedural layer [van Tonder 2007a; Clinger 2007b].
>
> ...
>
> Although the editors acted with the best of intentions, their addition of the `parent-rtd` clause did not solve the problems it was intended to solve. Even with the `parent-rtd` feature, you **still** have to know whether the base record type was defined using the syntactic or record layer, and you **still** can't change a record definition from one layer to the other without running the risk of breaking client code.
>
> To make matters worse, the 5.97 draft added a couple of questionable statements that attempt to excuse the interoperability problems while asserting privileged status for the draft's syntactic layer. One of those statements is based upon a patently false claim.
>
> The editors have submitted this draft to the Steering Committee as a candidate for ratification, so there is no meaningful technical review of these last-minute changes apart from the ratification vote itself.

Clinger [2007e] foresaw three possible outcomes of that vote:

1. The vote is negative, which would give the editors an opportunity to get it right.
2. The draft is ratified, and everyone pretends to live happily ever after.
3. The draft is ratified, and the unhappy folk design alternative syntactic layers, probably written up as SRFIs, that build upon the R6RS procedural layer.

Clinger [2009] was one of the unhappy folk who designed an alternative syntactic layer, written up as SRFI 99, that built upon a procedural layer compatible with (but not the same as) the procedural layer of R6RS. That design led to several subsequent SRFIs described in Section 12.5.

Ten of the thirty-five who voted against ratification of the R6RS identified the design of its record system as a reason for their negative vote [Bawden et al. 2007b].

11.6 Contributions of R6RS

It would not be fair to condemn all of the R6RS for its slip-ups. The R6RS added procedural `syntax-case` macros to the fully hygienic `syntax-rules` macro system of the R5RS, and used hygienic macro technology to define a new and badly needed `library` system. Even though the R6RS record system would be replaced in subsequent standards, it demonstrated the power of hygienic macros to define ambitious new language constructs.

12 FROM R6RS TO R7RS

12.1 Election of New Steering Committee

Following ratification and journal publication of the R6RS, the Scheme Language Steering Committee announced its members were stepping down and outlined the process that would elect their replacements by a secret ballot [Bawden, Steele, and Wand 2009b]. To "make it more difficult for any particular faction to gain control of all three seats," they used "a form of single transferable vote proportional representation" that called for each voter to rank the nominated candidates. There were 116 voters, 14 more than had voted when the R6RS came up for ratification. The election resulted in a steering committee whose members were Marc Feeley, Jonathan A Rees, and William D Clinger [Bawden et al. 2009a].

All three had voted against ratification of the R6RS, which highlights a shortcoming of the ranked-choice algorithm when electing multiple members of a committee. Of the 12 nominated candidates, only 3 had voted in favor of ratifying the one and only R6RS draft that was put to a vote; 5 had voted against that draft; 4 had abstained. In the ranked-choice voting, a candidate who had voted against ratification was the first choice of 76 voters; a candidate who had voted in favor of ratification was the first choice of only 15 voters; a candidate who had not voted either way was the first choice of 25 voters. If we take those first-choice votes as crude proxy for hypothetical anti-R6RS and pro-R6RS factions it is easy to see how voters in that hypothetical anti-R6RS majority could rank their choices to give pro-R6RS candidates little chance of being elected unless the small minority of pro-R6RS voters could consolidate their votes behind a single candidate. With two strong pro-R6RS candidates in Dybvig and Sperber, and with two strong neutral candidates in Anton van Straaten and Olin Shivers, it seems the small pro-R6RS vote was too divided to elect a pro-R6RS or neutral candidate.

Hoping to expand the range of views represented on the steering committee, its new members voted to revise its governing charter to allow expansion of the committee to as many as five members [Bawden et al. 2010]. Rather than call a new election immediately after the one that had elected them, they used the election results to fill out the committee by adding Olin Shivers and Chris Hanson, whose candidacies had been the last to be eliminated by the ranked-choice algorithm [Clinger et al. 2009]. Hanson had voted against ratification, and Shivers had abstained, so that expansion failed to add any pro-R6RS members to the committee. Clinger, who had voted against ratification of the R6RS but was among the first to implement it, believes he has been the most pro-R6RS member of this second iteration of the Scheme Language Steering Committee. In retrospect, a wider range of views would have been achieved by adding van Straaten, Dybvig, or Sperber, who were the next three in line after Hanson.

12.2 Working Groups

As revised in 2010, the new charter called for standards to be produced by working groups created by the Steering Committee [Bawden et al. 2010]. That committee drafted a position statement that called for “two separate but compatible languages” that would become known as R7RS (small) and R7RS (large), and wrote charters for the two working groups that would design those languages [Shivers et al. 2009; Rees et al. 2010a,b; Rees and Clinger 2013a,b]. In January 2010, the committee solicited volunteers to serve on those working groups, and the groups started their work the following month [Feeley 2010; NA Shinn 2010].

Alex Shinn chaired Scheme Working Group 1, producing a progress report and nine drafts culminating in the Revised^7 Report on the Algorithmic Language Scheme [Shinn, Cowan, and Gleckler 2013; Cowan 2016]. That R7RS (small) standard was put to a vote in 2013, approved by 56 of the 63 who voted, and endorsed by the steering committee later that year [Rees and Clinger 2013a].

John Cowan is chairing Scheme Working Group 2, which is responsible for designing the R7RS (large) language. Cowan’s group is producing a color-coded series of interim “editions” that combine the R7RS (small) standard with a set of standard libraries that have been specified using the SRFI process [Rees and Clinger 2013b], [NA Cowan (no date)]. So far, working group 2 has voted to approve an R7RS Red Edition (June 2016) and an R7RS Tangerine Edition (February 2019) [Cowan 2017], [NA Cowan 2019]. The Tangerine Edition already describes a language that is substantially larger than the language described by the R6RS and its standard libraries, but is less complete than the R6RS with respect to macros and records.

12.3 R7RS Macro System

The R7RS (small) report requires `syntax-rules` but not `syntax-case` [Shinn et al. 2013].

Working Group 2 expects to add at least one procedural macro system. Candidate systems include `syntax-case`, explicit renaming, and syntactic closures [NA Cowan (no date)].

12.3.1 Implementations of R7RS Macro Systems. As of this writing, there are about a dozen implementations of the R7RS (small) standard [Clinger 2015]. Most incorporate Alex Shinn’s implementation of `syntax-rules` and the R7RS `define-library` system from Chibi Scheme [NA Shinn (no date)]. Sagittarius relies on Chibi Scheme’s implementation of `syntax-rules` in its R7RS mode but uses `psyntax` in its R6RS mode [Kato 2014; NA Kato (no date)]. Larceny has upgraded André van Tonder’s implementation to support both R7RS and R6RS libraries interchangeably [Clinger 2015; NA Larcenists (no date)]. Sagittarius and Larceny support both `syntax-case` and explicit renaming. Chibi Scheme’s macro system is based upon syntactic closures, and also supports explicit renaming.

Chicken and Picrin are implementations of the R7RS that support both explicit renaming and *implicit renaming*, which is like explicit renaming except the default behavior is reversed: Instead of telling the macro expander which identifiers to rename, the programmer tells the macro expander which identifiers should not be renamed; all others are renamed automatically [Lorenz 2009-2020].

12.3.2 Equipotence of `syntax-case` *and Explicit Renaming.* André van Tonder’s implementations prove explicit renaming is a viable way to implement `syntax-case` and the R6RS library system [van Tonder 2005b, (no date)]. As extended in Larceny, van Tonder’s implementation now supports `syntax-rules`, `syntax-case`, and both the R6RS and R7RS library systems along with a compatible explicit-renaming facility.

Explicit renaming (with `datum->syntax`) and `syntax-case` are essentially equipotent. Without `datum->syntax`, explicit renaming is probably weaker than `syntax-case`. André van Tonder’s implementation uses explicit renaming at its core, implements `syntax-case` on top of that core,

```
(define-syntax loop
  (er-transformer                                 ; was transformer
   (lambda (x rename compare)
     (let ((cmds (cdr x))
           (call/cc (rename 'call-with-current-continuation))
           (lambda (rename 'lambda))
           (define (rename 'define))
           (the-loop (rename 'the-loop))
           (exit (datum->syntax (car x) 'exit))) ; this line was added
       `(,call/cc
         (,lambda (,exit)                  ; was (exit) with no comma
          (,define (,the-loop)
           ,@cmds
           (,the-loop))
          (,the-loop)))))))
```

Fig. 5. Explicit renaming, made compatible with `syntax-case`.

and then uses `syntax-case` to define the explicit renaming library that is available for programs to import. That closes the circle of equivalence.

12.3.3 Interoperability Between `syntax-case` *and Explicit Renaming.* As explained in Section 9.1, the data type of syntax objects is similar to the data type of S-expressions, with one major difference: a syntax object encapsulates additional information about symbols so the macro system can preserve hygiene without having to keep track of auxiliary syntactic environments. A syntax object can be converted ("unwrapped") into a structurally similar S-expression by discarding that additional information, but it is not possible to convert an S-expression that contains symbols into a syntax object unless that additional information is known.

In particular, a raw symbol is not a syntax object, and no S-expression that contains raw symbols can be a syntax object.

The jargon of `syntax-case` therefore distinguishes symbols from *identifiers*: an identifier is a syntax object that, when unwrapped using `syntax->datum`, becomes a symbol. Equivalently, an identifier is a symbol that has been wrapped, as by `datum->syntax`, to convert it into a syntax object.

This terminology is confusing because early Scheme reports, before R6RS, used the word "identifier" as a near-synonym for symbol. In those reports, symbols were the standard internal representation for identifiers, which had a single external representation defined by the R6RS lexical syntax, but had multiple conceptual roles as names for symbols, variables, and syntaxes. The two most recent Scheme reports continue to use the word "identifier" with that original meaning in some contexts, while other contexts use the word with the more arcane meaning it acquired when `syntax-case` and syntax objects were added to the language. As might be expected, this overloading of the word "identifier" has led to disagreements concerning the meaning of various specifications, including the controversy we will describe in Section 14.1.

The distinction between symbols and identifiers that was introduced by `syntax-case` presents an obstacle to interoperation between explicit renaming and `syntax-case`. With explicit renaming as described in Section 7.4, procedural syntax transformers return S-expressions. With `syntax-case`, the abstract syntax tree created and returned by procedural syntax transformers must be a syntax object.

To allow explicit renaming to interoperate with `syntax-case`, procedural syntax transformers that use explicit renaming must return a syntax object. That invariant is enforced by van Tonder's implementation of explicit renaming [van Tonder (no date)].

The definition of `loop` given in Section 7.4 inserts `exit` into the output as a raw symbol, so it is incompatible with `syntax-case`. In André van Tonder's implementation of explicit renaming, which is now used by Larceny, the `loop` macro of Section 7.4 would be written as shown in Figure 5.

Explaining that definition in terms of time stamps, which is a bit of a simplification, `datum->syntax` takes the time-stamped identifier `loop` that appears within the macro use, extracted via `(car x)`, and attaches that `loop` identifier's time-stamp to the symbol `exit`, returning a wrapped symbol object instead of a raw symbol. That makes the time-stamped `exit` behave as though it had been inserted at the same time as `loop`, which is presumably the same time any references to `exit` that we want to capture were inserted as well.

12.4 R7RS Library System

The R7RS `define-library` system is essentially a syntactic variant of the R6RS `library` syntax, removing two features (explicit phasing and versioning) that most implementations of the R6RS were ignoring anyway [Ghuloum and Dybvig 2007; Courtès 2007a,b], removing some arbitrary restrictions that had proved burdensome, and adding a few new things (`include`, `cond-expand`, `features`) that don't concern us here.

12.5 R7RS Record Systems

Because Working Group 1 adopted the fully hygienic `syntax-rules` as the only macro facility required by R7RS (small), it could not consider record systems with implicit naming. The SRFI 9 record system, with its explicit naming, became the only record system required by R7RS (small) [Kelsey 1999; Shinn et al. 2013].

Working Group 2 may add one or more procedural record systems [NA Cowan (no date)]. The working group's consideration of syntactic record systems has been complicated by controversy concerning an interaction between records and hygienic macro technology; that controversy will be described below in Section 14.1.

13 BINDINGS AS SETS OF SCOPES

13.1 Motivation and Strategy

In January 1995, the developers of PLT Scheme began developing a Scheme-based programming environment designed for teaching computer science at various levels, from elementary school to university level. A key innovation was the development of a series of pedagogical languages, ranging from beginner- to professional-level. The features and (especially) the error messages for each language were tailored to the expertise and level of the intended audience.

Hygienic macro expansion was the primary tool for constructing these languages, and eventually the ecosystem (now named Racket) re-oriented itself as a tool for creating new languages [Felleisen et al. 2018].

Along the way, the Racket group added powerful features to the basic syntax-case system. Flatt *et al.* wrote:

> The Racket macro API exposes the compiler's general capability to bind and access compile-time information within a lexical scope, as well as the compiler's ability to expand a sub-expression's macros. This wider macro API enables language extensions that were technically impossible (or, at best, awkward to simulate) in the narrower macro API of earlier Scheme systems. Such extensions can

> be generally characterized as macros that cooperate by sharing compile-time information. [Flatt et al. 2012]

However, as the Racket group gained experience with the new API, it became evident that renaming-based hygiene algorithms had become cumbersome for the new applications:

> The analogy [to lexical scope] suggests specifying hygienic macro expansion as a kind of translation into lexical-scope machinery. In that view, variables must be renamed to match the mechanisms of lexical scope as macro expansion proceeds. A specification of hygiene in terms of renaming accommodates simple binding forms well, but it becomes unwieldy for recursive definition contexts (Flatt et al. 2012, section 3.8), especially for contexts that allow a mixture of macro and non-macro definitions. The renaming approach is also difficult to implement compactly and efficiently in a macro system that supports "hygiene bending" operations,such as `datum->syntax`, because a history of renamings must be recorded for replay on an arbitrary symbol. [Flatt 2016]

Flatt [2016] introduced a model of binding that was developed for use in the Racket macro expander. The goal was to accomodate all the new features of Racket macros, while maintaining sufficient compatibility with Racket's old expander.

The basis of the model was that all of the information in a syntax object could be modeled as a set of scopes. Flatt wrote:

> . . . Our new expander tracks binding through a *set of scopes* that an identifier acquires from both binding forms and macro expansions.

(Italics in original.)

The idea was that each binding form, whether present in the original program or generated as an effect of macro expansion, determines a *scope*, that is the portion of the program in which the binding form is effective (unless overridden). A variable reference is mapped to its matching binder by inspecting the set of scopes that the reference is within. In the classic case of lexical scope, the scopes are well-nested, so each scope set is uniquely determined by its innermost scope.

In general, however, the scope sets are not always well-nested. Individual scopes are created dynamically during macro expansion and may not even be associated with a binding form. In particular, every expansion of a use of a macro creates a new scope, called a *macro-invocation* scope. Thus variables generated by different macro expansion steps cannot shadow each other.

The major goal of the system was to create a model that would modularly accomodate all of the features of the Racket macro system, including modules, recursive macros, and macros that expand to definitions rather than expressions. Each of these led to significant complications, such as the necessity of creating a *macro-use* scope around the text of each macro use site in addition to the macro-invocation scope. It also became necessary to sometimes remove scopes from certain identifiers.

The resulting algorithm was formalized using Redex [Felleisen et al. 2009] in the style of Flatt et al. [2012], and a rendering using LaTeX fonts was included in the paper.

13.2 How It Works

In this section, we will work out a simple example. Our example is simplified in a number of ways:

- It exercises only macros in expression position, although the full system is designed to work with macros in definition contexts as well.
- The full model uses a both a syntactic environment and a syntactic store; we use a simple store.

- The full model deals with procedural macros and macro-defining macros, neither of which is exercised in this example.
- The full model also includes phasing.

Consider the following expression:

```
(let ((global 100))
    (let-syntax ([test (syntax-rules ()
                         [(_ x y)
                          (let ((local x))
                              (+ global y local))])])
      (let ((local 10)
            (global 1000)
            (x 10000))
        (+ x (test 1 local)))))
```

In this example, the instance of `local` in the macro application should not be captured by the instance of `local` in the macro definition; also, the instance of `global` in the macro definition should not be captured by the instance of `global` in the `let` preceding the macro application. So the result of the expression should be `(+ 10000 (+ 100 10 1))` or `10111`.

We will trace the major steps of the set-of-scopes algorithm as it operates on this expression.

13.2.1 Entering a Known Binding Form. When the set-of-scopes algorithm enters a known binding form, like `let` or `lambda`, it does the following:

- It creates a new scope.
- It paints every identifier in a binding position with the new scope.
- It extends a global table that maps a (symbol, scope set) pair to a representation of a binding (for example, a `gensym`) by adding an entry for each of the newly-painted identifiers.
- It paints every identifier in the region where the binding applies with the new scope. It must paint *every* identifier in the region, since it doesn't yet know which are binders and which are references. Painting each symbol with the new scope guarantees that if this symbol turns out to be a reference, it must refer to the current binding, unless some other binding intervenes.

In the examples below we write `x{1,2,3}` to indicate the symbol `x` annotated with scopes $\{s_1, s_2, s_3\}$.

In our example the first thing the algorithm sees is the `let` on the first line. It creates a new scope s_1, and paints the binding occurrence of `global` with s_1, yielding `global{1}`. It adds to the binding table the entry

```
global{1} -> global1
```

Then it paints every occurrence of `global` in the body of the `let` with s_1. This is done in constant time, by adding s_1 to the syntax object corresponding to the body of the `let`. For explanatory purposes, however, we will paint all the identifiers explicitly. The result is

```
(let ((global{1} 100))
  (let-syntax ([test{1} (syntax-rules ()
                         [(_ x{1} y{1})
                          (let ((local{1} x{1}))
                              (+ global{1} y{1} local{1}))])])
    (let ((local{1} 10)
          (global{1} 1000)
          (x{1} 10000))
      (+ x{1} (test{1} 1 local{1})))))
```

The algorithm then enters the `let-syntax` and creates a new scope s_2. It paints the bound variable `test` with s_2, adds the new macro definition to the global table, and paints all the symbols in its scope with s_2.[5]

So the global table looks like

```
global{1} -> global1
test{1,2} -> (syntax-rules () ...)
```

and the expression becomes

```
(let ((global{1} 100))
  (let ((local{1,2} 10)
        (global{1,2} 1000)
        (x{1,2} 10000))
      (+ x{1,2} (test{1,2} 1 local{1,2}))))
```

The algorithm then enters the inner `let`, creating a new scope s_3. It again paints the symbols in binding position, adds a new entry for each of them in the global binding table, and then paints each symbol in the body of the `let` with s_3. So the global table becomes

```
global{1} -> global1
test{1,2} -> (syntax-rules () ...)
local{1,2,3} -> local1
global{1,2,3} -> global2
x{1,2,3} -> x1
```

and the expression becomes

```
(let ((global1 100))
    (let ((local1 10)
          (global2 1000)
          (x1 100))
     (+ x{1,2,3} (test{1,2,3} 1 local{1,2,3}))))
```

13.2.2 Resolving a Reference. The algorithm next enters the body of the inner `let`. Here for the first time, the algorithm encounters a reference. The rule for references is that a reference resolves to the entry in the binding table with the same symbolic name as the reference, but whose set of scopes is the largest subset of the reference's own scopes. The idea is that the binding was created inside the scopes shown in the binding table, and has perhaps traveled through some other scopes to reach the reference.

The painted symbol `x{1,2,3}` matches exactly in the global binding table, so it resolves to `x1`.

What about `test{1,2,3}`? There is no exact match, but the sole occurrence of `test` in the binding table is at `test{1,2}`, so the largest-subset rule confirms this is the correct binder for `test{1,2,3}`.

13.2.3 Expanding a Macro Application. The binding table tells us that the meaning of `test{1,2}` is a macro, which is bound to a procedure that, when applied, matches

```
(syntax-rules ()
 [(_ x{1} y{1})
  (let ((local{1} x{1}))
      (+ global{1} y{1} local{1}))])
```

against the macro call.

[5]Since the system allows for procedural macros, what actually happens here is that the `(syntax-rules ...)` expression is evaluated, and `test{1,2}` is bound to a procedure that behaves like the `(syntax-rules ...)` expression.

The algorithm for expanding a macro call has three main steps:

(1) The macro call is painted with *two* new scopes, called the *macro-use* scope and the *macro-invocation* scope.
(2) The meaning of the macro (a procedure) is applied to the resulting syntax object.
(3) When the macro application returns, the marks from the macro invocation scope are flipped (like they are in `syntax-case`), and the resulting syntax object is fed back to the expander (as it is in all of these algorithms).

Let us see how this works for our example. The macro call is

```
(test{1,2,3} 1 local{1,2,3})
```

The algorithm creates the macro-use scope, s_4, the macro-invocation scope, s_5, and paints the call with both of these, yielding

```
(test{1,2,3,4,5} 1 local{1,2,3,4,5})
```

The value bound to the macro `test{1,2}` is a procedure which, when applied to a syntax object, matches it against

```
(syntax-rules ()
  [(_ x{1} y{1})
   (let ((local{1} x{1}))
       (+ global{1} y{1} local{1}))])
```

The match binds `x{1}` to 1 and `y{1}` to `local{1,2,3,4,5}`. Substituting these in the template gives

```
(let ((local{1} 1))
     (+ global{1} local{1,2,3,4,5} local{1}))
```

This is the value of the `syntax-rules`, so the next step is to xor the macro-invocation scope s_5 from this result. This gives

```
(let ((local{1,5} 1))
     (+ global{1,5} local{1,2,3,4} local{1,5}))
```

Observe that any identifiers marked with the macro-invocation scope s_5 must have been introduced by the macro; identifiers marked with macro-use scope s_4 must have come from the call site.

Now the algorithm expands this expression. It enters the let, creating s_6, so the final binding table is

```
global{1} -> global1
test{1,2} -> (syntax-rules () ...)
local{1,2,3} -> local1
global{1,2,3} -> global2
x{1,2,3} -> x1
local{1,5,6} -> local2
```

and the expression to be expanded is

```
(let ((local2 1))
     (+ global{1,5,6} local{1,2,3,6} local:{1,5,6}))
```

Now the algorithm is ready to work on the sum. According to the greatest-subset rule,

- `global{1,5,6}` refers to `global{1}`, which is `global1`
- `local{1,2,3,6}` refers to `local{1,2,3}`, which is `local1`
- `local{1,5,6}` refers to `local2`

Putting the expression back together yields

```
(let ((global1 100))
 (let ((local1 10)
       (global2 1000)
       (x1 10000))
   (+ x1
      (let ((local2 1))
       (+ global1 local1 local2)))))
```

which computes `(+ 10000 (+ 100 10 1))`, as intended.

14 SUBTLETIES OF HYGIENE

We have already described many subtle aspects of hygienic macro expansion in Sections 4.4, 4.7, 6.8, 7.1, 7.2, 7.3, 8, 9.1, 12.3.2, and 12.3.3. We have also shown how hygienic macros provide a design space wide enough to foment the controversy described in Section 11.5.

In this section, we describe two other controversies concerning subtle aspects of the design space. The first of those controversies remains unresolved, but the second story ends happily.

14.1 Are Field Names Symbols or Identifiers?

For most uses of the R7RS record system, it doesn't matter whether field names are symbols or something more arcane, but there are two circumstances in which it does matter:

- Macros that are intended to expand into record definitions might need to generate fresh names for the fields of those records, and there is a standard trick (invented by Clinger [1991b]) for getting `syntax-rules` to generate fresh identifiers without having to fall back on some procedural mechanism. If field names are symbols, that trick won't generate fresh field names.
- Inheritance becomes fragile unless field names are symbols.

As stated in R7RS section 2.1:

> Identifiers have two uses within Scheme programs:
> - Any identifier can be used as a variable or as a syntactic keyword (see sections 3.1 and 4.3).
> - When an identifier appears as a literal or within a literal (see section 4.1.2), it is being used to denote a symbol (see section 6.5).

When a record type is defined, identifiers are used to name its fields. Are those identifiers being used as variables, or as syntactic keywords, or as literals?

The R7RS doesn't say. The R7RS uses the word "identifier" only once in connection with field names, saying "It is an error for the same identifier to occur more than once as a field name." But that doesn't tell us whether the field names are to be taken as symbols or as identifiers (in the sense of `syntax-case` that was described in Section 12.3.3).

When the plain meaning of a document is insufficient to resolve issues such as this, we can look at historical context, authorial intent, and common practice. We can also fall back on the legal principle that says we should prefer interpretations that allow the document to achieve its purpose over interpretations that do not.

The R6RS says field names are identifiers, but disambiguates that by going on to say those identifiers "become, as symbols, the names of the fields."

Alex Shinn, chair of Working Group 1 and one of the three editors of the R7RS (small) document, insists the R7RS differs from the R6RS on this point, and that the change was made deliberately.

```
(define-library (foo)
  (export foo make-foo foo? foo.x foo.y)
  (import (scheme base))
  (begin
   (define x 1) ; x denotes a location that holds the value 1
   (define y 2) ; y denotes a location that holds the value 2

   (define-record-type foo    ; has fields named x and y
     (make-foo x y)           ; how you create a record of type foo
     foo?                     ; predicate to recognize such records
     (x foo.x)                ; foo.x is accessor for field x
     (y foo.y))               ; foo.y is accessor for field y

   ;; Example:

   (define f (make-foo x y))
   (list (foo.x f) (foo.y f)) ; evaluates to (1 2)
   ))

(define-library (bar)
  (export bar make-bar bar? bar.z)
  (import (scheme base)
          (scheme write)
          (srfi 99)
          (foo))
  (begin
   (define t 8)
   (define x 9)   ; this x denotes a different location from the x in foo
   (define y 10)  ; this y denotes a different location from the y in foo
   (define z 11)

   (define-record-type (bar foo)    ; inherits from foo
     (make-bar t x y z)             ; has fields t, x, y, z
     bar?
     (t bar.t)
     (z bar.z))

   ;; Example:

   (define b (make-bar t x y z))

   (write (list (bar.t b) (foo.x b) (foo.y b) (bar.z b)))
   (newline)))

(import (bar))
```

Fig. 6. If field names are symbols, this R7RS program prints (8 9 10 11).

Another of the three editors, John Cowan, responded to that discussion by denying "(on general principle) that the author of a document is an authority on its meaning" [NA Cowan et al. 2016].

According to Shinn, the field names of records are not symbols, but are instead identifiers in the sense of `syntax-case`, and should therefore be matched according to the rules for matching syntactic keywords and variables. As will be explained in Section 14.2, the semantics of that matching is subtle in itself, but it is clear that two identifiers defined as keywords or variables by distinct definitions should not match, even if they have the same name.

That can break inheritance of field names, as illustrated by the program in Figure 6. That program works if field names are symbols, but does not work if field names are identifiers in the sense of `syntax-case`.

If field names are not symbols, then the meaning of the field names `x` and `y` in the definition of record type `foo` is tied to the presence of the variables `x` and `y` that are defined in that same library but are not exported from that library. The library named `(bar)` defines different variables named `x` and `y`, so the `x` and `y` mentioned in the definition of record type `bar` don't have anything to do with the fields of record type `foo`. If field names are not symbols, we have to change the program of Figure 6 by

- renaming the variables `x` and `y` that happen to be defined within the `(foo)` library
- renaming the variables `x` and `y` that happen to be defined within the `(bar)` library
- adding field names `x` and `y` to the list of identifiers exported by the `(foo)` library

In June 2016, the program shown in Figure 6 and an equivalent R6RS program ran just fine in every implementation of the R7RS and R6RS tested by Clinger. One of those implementations was Chibi Scheme, which means field names were symbols even in Shinn's own implementation of the R7RS. Shinn responded by changing Chibi Scheme so the program in Figure 6 no longer runs, and Sagittarius has apparently followed Chibi's lead on this.

Clinger was allowed to add a post-finalization note to SRFI 99 clarifying that field names are symbols [Clinger 2016]. SRFI 131 was intended to be the subset of SRFI 99 that can be implemented on top of the R7RS (small) record system, but the controversy just described calls into question whether the intended semantics of SRFI 99 and SRFI 131 can be implemented portably using the R7RS (small) record system. Marc Nieper-Wißkirchen [2016] developed SRFI 136 as an alternative to SRFI 131 that finesses that portability problem by designing an inheritance mechanism that "does not rely on the identity of field names." Nieper-Wißkirchen [2018] developed SRFI 150 as a version of SRFI 131 that assumes field names are not symbols, and is therefore compatible with Shinn's interpretation of the R7RS (small) document. This proliferation of closely related standards was an unfortunate consequence of overloading the word "identifier" in the primary standards for Scheme and in the research literature on hygienic macro technology.

14.2 Matching of Keywords

SRFI 148 (Eager `syntax-rules`) is one of several SRFIs written by Marc Nieper-Wißkirchen [2017] that are related to macros. While porting that SRFI's sample implementation to Larceny, Clinger found and reported a bug in that sample implementation [Clinger 2017a]. By the next day, Nieper-Wißkirchen had convinced Clinger that it was really a bug in Larceny's implementation of hygiene [Clinger 2017b]:

> People often talk of renaming and hygiene as though it's all simple and obvious, but I don't know of any authoritative formal specification that explains how to deal with macro-defining macros, syntax-case, et cetera. (I know of several attempts, including my own, but they don't entirely agree with each other.)

> For example, we have to distinguish between tentative renaming (in some intermediate form) and ultimate renaming (in the fully expanded code). The real question here is which intermediate forms should match and which should not.
>
> The tracing I did yesterday convinced me that Larceny's macro expander was renaming too aggressively and/or failing to match often enough, however you want to look at it.

Another day later, Clinger decided Larceny had been right all along, and the bug really did lie within the sample implementation [Clinger 2017c].

That created a mystery: Why had the sample implementation worked in several other implementations? With help from Al Petrofsky and Takashi Kato (implementor of Sagittarius), we found that several implementations of the R7RS were incompatible with the R6RS when macros define other macros. It turned out that the sample implementations of SRFI 147 and SRFI 148 "work only in macro systems that are (intuitively) less hygienic than those used by R6RS systems such as Chez, Petite Chez, Racket, Vicare, Larceny, and the -r6 mode of Sagittarius." [Clinger 2017d,e]. We fixed those sample implementations, and Shinn and Kato changed their implementations (Chibi and Sagittarius) to make their behavior on macros that define local macros compatible with the R6RS and with Larceny [Shinn 2017].

The technical issue came down to a question of when a pattern literal `y` should match a literal `x` that occurs in some use of a macro. Examining André van Tonder's macro expander, Clinger discovered five plausible ways to perform that comparison. In decreasing order of hygiene:

```
(1)           (bound-identifier=? x y)
(2)           (free-identifier=? x y)
(3)           (free=? y (id-name x))
(4)           (eq? (id-name x)
                   (if (binding y)
                       (binding-name (binding y))
                       (id-name y)))
(5)           (eq? (id-name x) (id-name y))
```

The `bound-identifier=?` or `free-identifier=?` procedures, described in Section 7.4, were standardized by the R6RS; the other three alternatives are specific to van Tonder's implementation. As documented within the discussion of SRFI 148, the Scheme community has agreed that the most hygienic of those five alternatives (`bound-identifier=?`) should be used.

15 CLOJURE'S LESS NAÏVE MACRO EXPANSION

Naïve macro expansion makes it easy to write non-hygienic macros, but hard to write hygienic macros. Why not start with naïve macro expansion, adding new features that make hygienic macros easier to write?

The main problem with this approach is that it is hard to design a macro expander that makes fully hygienic macros easy to write while remaining compatible with naïve substitution as the default. Some designs have been more successful than others, however.

15.1 Macros in Clojure

Clojure is one of the more widely used programming languages in the Lisp family [NA Clojure web site (no date)]. Designed by Rich Hickey (who remains its "benevolent dictator for life") Clojure was originally implemented on top of the Java Virtual Machine (JVM). Its design draws from Common Lisp, Scheme, and Java.

In Clojure, macros are defined by writing procedural code that rewrites a use of the macro into its transcribed form [Volkmann 2009]. The general approach is similar to the naïve macro system of Common Lisp, as described in Section 2, but Clojure makes it easier to write hygienic and mostly-hygienic macros.

In Clojure, a hygienic binary `or2` macro can be defined as follows:

```
(defmacro or2 [a b]
  `(let [x# ~a]
    (if x#
      x#
      ~b)))
```

The backquote notation, known as *syntax-quote* in Clojure, is similar to the backquote notation of Common Lisp, writing unquote with tilde (~) instead of comma (`,`). But it introduces two important new behaviors. First, the three occurrences of `x#` evaluate to a generated symbol, known as a *gensym*, consisting of the base name `x_` concatenated with a time stamp. The syntax-quote notation automatically arranges for all three occurrences of `x#` to denote the same gensym. Second, the syntax-quote notation also resolves all unqualified symbols in the current namespace context, converting them into fully-qualified symbols. The unqualified symbols `let` and `if` turn into the qualified symbols `clojure.core/let` and `clojure.core/if`, which keeps the transcribed macro's free occurrences of `let` and `if` from being captured by any local bindings of those names that may be in scope at uses of the macro.

Omitting the gensyms, we would expect the following definition to be unhygienic because the binding of `x` might capture free references to `x` within the expression `b`:

```
(defmacro or2 [a b]
  `(let [x ~a]
    (if x
      x
      ~b)))
```

As it happens, that definition doesn't work at all because syntax-quote converts the unqualified symbol `x` into a fully qualified symbol, and Clojure's `let` syntax requires the symbols bound by the `let` to be unqualified. That restriction makes it harder for local bindings to shadow other bindings, and makes it harder to create local bindings that might result in inadvertent captures of free variables. These restrictions make it harder to write unhygienic macros by accident.

If we really want to write an unhygienic `or2` macro that captures free occurrences of `x` within `b`, we can do so by writing `~'x` to insert an unqualified symbol `x` into the transcribed output:

```
(defmacro or2 [a b]
  `(let [~'x ~a]
    (if ~'x
      ~'x
      ~b)))
```

Comparing this to the hygienic definition above, it becomes clear that the design of Clojure's macro system improves upon that of Common Lisp by making hygienic macros easier to write and by making unhygienic macros harder to write by accident.

15.2 Would Clojure-like Macros Have Been Good Enough?

One of our HOPL reviewers asked whether the Scheme community would have been content to standardize a Clojure-like macro facility if that technology had been demonstrated during the 1980s.

We doubt it. Clojure's macro system resembles Rees's "modest macro proposal" of March 1987, described in Section 6.3 (although we have no reason to think Clojure's designers knew of Rees's proposal). As recounted in Sections 6.4 and 6.5, that proposal was replaced less than three months later by a "Modified Macro Proposal" that soon evolved into syntactic closures. Had Clojure's macro system been proposed for Scheme during the 1980s, we suspect the history of hygienic macro technology would have played out much as it did following Rees's similar proposal.

So long as we are speculating, however, demonstration of a Clojure-like macro system during the 1980s might conceivably have led to that technology becoming popular in languages with conventional syntax. As will be seen in our next section, the use of hygienic macro technology in such languages is a recent development.

16 HYGIENIC MACROS WITH CONVENTIONAL SYNTAX

Several languages with more conventional syntax now include macro systems that make it possible to define hygienic or partially hygienic macros. We examine three of those macro systems here, together with an early prototype of hygienic macros for Modula-2. For each system, we ask the following questions:

- What language is used to define a macro?
- Can macros examine the structure of their arguments?
- Are macros expanded top-down or bottom-up?
- What representations are manipulated by macros?
- How are uses of a macro parsed/matched/delimited?
- Does the macro system satisfy Kohlbecker's Hygiene Condition?
- Does the macro system satisfy the Strong Hygiene Condition?

16.1 Modula-2

The first implementation of hygienic macros for a language with Algol-like syntax was constructed by Brian Reistad as part of an undergraduate research project at the University of Oregon, supervised by Clinger. This system was a research prototype, tested only by Reistad and Clinger. We quote the entire abstract of Reistad's 20-page paper [NA Reistad 1992]:

> A macro facility was developed for Modula-2 based on the modified Kohlbecker algorithm for hygienic macro expansion of Clinger and Rees. This paper describes the resulting system and the problems encountered in developing a macro facility for Modula-2. The problems arose from locating the macro arguments in the Algol-like syntax of Modula-2 and the extended syntax defined by the macro writer. Any macro facility for a language with an Algol-like syntax will face these syntax problems. The problems were solved by placing a type system on macros. A working macro facility for Modula-2 was implemented.

What language is used to define a macro? Here is the `case` macro defined and used in Reistad's Figure 3:

```
MODULE case;
FROM InOut IMPORT WriteString,ReadInt,WriteLn;
MACRO case:STMT
      RULE case ?x:ID ?e:EXPR : ?do:STMT ;
                ?rest:REPEAT ?e:EXPR : ?do:STMT ; ENDREPEAT
                else ?otherwise:STMT ;
      BECOMES
             IF ?x = ?e THEN
                  ?do
             ELSE
                  case ?x ?rest else ?otherwise ;
             END;
      RULE case ?x:ID else ?otherwise:STMT ;
      BECOMES ?otherwise
ENDMACRO;
VAR a:INTEGER;
BEGIN
     WriteString(``Enter an integer:  ``);
     ReadInt(a);
     WriteLn;
     case a 0 :  WriteString("foo") ;
            5 :  WriteString("bar") ;
            else WriteString("!!!");
     WriteLn;
END case.
```

Pattern variables are explicitly declared to have one of the following syntactic "types": expression, statement, declaration, block, identifier. The REPEAT pseudotype, together with the ENDREPEAT, corresponds to Scheme's ellipsis.

In Reistad's implementation, a macro could be used only within the module in which it was defined. He explained how relaxing that restriction would provide the benefits of Ada generics.

Can macros examine the structure of their arguments? Yes, to about the same extent as in Scheme's `syntax-rules`.

Are macros expanded top-down or bottom-up? Bottom-up. Macros are expanded during LL(1) parsing, so the expressions and statements of the `case` macro above will be macro-expanded during the matching/parsing process.

What representations are manipulated by macros? Abstract syntax trees as produced by the LL(1) parser.

How are uses of a macro parsed/matched/delimited? Defining a macro extends the LL(1) grammar of Modula-2 by adding a special production for that macro. LL(1) director sets are not re-calculated when a macro is defined, so "macro arguments should be separated by keywords chosen from the already existing follow sets." In the `case` macro, those separators include colon and semicolon together with the reserved word `else`.

The extended LL(1) grammar is used to parse the macro argument that matches a pattern variable according to the syntactic category declared for that pattern variable. In the `case` macro, for example, the macro argument that matches the pattern variable `?otherwise` is found by the LL(1) parser for statements.

Each macro's rules are tried in order, discarding abstract syntax trees created during failed attempts to match a rule.

Does the macro system satisfy Kohlbecker's Hygiene Condition? Yes.

Does the macro system satisfy the Strong Hygiene Condition? Yes.

16.2 Scala

The Scala language combines object-oriented and higher-order functional programming with a rich static type system and syntax [Scala (no date)]. Its most distinctive feature is the use of *traits*, which combine interfaces and superclasses, giving the effect of a sophisticated multiple-inheritance system. It targets the JVM, enabling programmers to utilize Java's huge library.

Scala was first released in 2004. Macros were included as an experimental feature of Scala in version 2.10, which was released in January 2013. Our discussion here is based on two papers [Burmako and Odersky 2012; Burmako 2013] and our own experiments with the the `def`-macros of version 2.12.6, released in April 2018. Those `def`-macros are the oldest "flavor" of macro available in Scala and appear to be the most commonly used, but the Scala papers and documentation mention several other flavors of macro we have not examined.

What language is used to define a macro? Macros are written in Scala, making use of dependent types and a reification facility that supports hygienic recreation of abstract syntax trees [Burmako and Odersky 2012].

The basic operation of a Scala `def`-macro is illustrated by the following code, which defines and uses a macro with a local binding, similar to the example in section 13.

```
object Macros {

  def global = 100
  def get_global () = global

  def test (x: Int, y: Int):Int = macro testImpl
  def testImpl(c:Context)(x: c.Expr[Int], y: c.Expr[Int]) =
  {import c.universe._
    (q"""
             val local = $x
             global + $y + local
            """)}
}
```

The macro's definition is divided into two parts. The declaration of `test` shows the types of the macro arguments as they will be in any use of the macro, while the declaration of the macro's implementation, `test1_Impl`, shows the types of the values that will be manipulated by the macro's implementation.

The macro's implementation is called with two sets of parameters. The single parameter in the first set, a `Context`, encapsulates types and methods that represent the context of a macro use as understood by the compiler. The parameters in the second set represent encapsulations of abstract syntax trees that represent the arguments of a macro use; if the macro expects an argument of type

Int, as in this example, then its implementation will be passed an encapsulated abstract syntax tree of type c.Expr[Int].

The body of the macro implementation above imports c.universe._ and uses a quasiquote-like operator q to construct an expression that will evaluate the two subexpressions from left to right, binding those results to temporary variables x and y.

Quasiquotation as in the above example is just one of the techniques that can be used to construct the macro-expanded code.

All macros must be compiled before any code that uses a macro is compiled. We believe this effectively rules out macros that define macros.

Can macros examine the structure of their arguments? Yes. Burmako [2013] gives an example of this in his Section 4.5.

Are macros expanded top-down or bottom-up? Expansion is done bottom-up.

What representations are manipulated by macros? The implementation of a macro manipulates encapsulated abstract syntax trees using types and methods made available through the context argument.

How are uses of a macro parsed/matched/delimited? A use of a def-macro looks like a function call and is parsed as such, so that kind of macro does not really extend the context-free syntax of Scala.

Does the macro system satisfy Kohlbecker's Hygiene Condition? Yes.

Does the macro system satisfy the Strong Hygiene Condition? No [Burmako and Odersky 2012]. Names inserted by a macro can be captured by bindings in force where the macro is used. In our example, if we exercise the macro by running

```
object Main extends App {

  import Macros._

  def local = 10
  def global = 1000
  def x = 10000

  def correct_answer = x + get_global() + local + 1
  def answer = x + test(1, local)

  printf("hygienic answer is %d%n", correct_answer);
  printf("actual   answer is %d", answer)
}
```

The result is

```
hygienic answer is 10111
actual   answer is 11011
```

indicating that the name global in the macro body (with value 100) was captured by the value of global at the macro use (with value 1000).

The recommended workaround is to insert qualified names such as Macros.wrapper or, better yet, to use absolute qualified names such as _root_.Macros.wrapper.

Burmako and Odersky [2012] go on to describe a `reify` macro that "packages the result expression tree with the types and values of all free references that occur in it. . . . As a consequence, macros that generate trees only by the means of passing expressions to `reify` are hygienic."

16.3 Rust

Rust is a systems programming language, developed at Mozilla [Klabnik and Nichols 2018]. It is intended to be a language for highly concurrent systems. Like most of the languages considered in this survey, it supports functional, imperative, and object-oriented programming. One of its goals is to produce code with performance comparable to that of C++. Unlike most of the languages considered in this survey, Rust does not rely on a garbage collector to ensure memory safety. Instead, it uses a system of *ownership types*: every piece of mutable data is owned by exactly one variable; when that variable goes out of scope, the memory for that data is reclaimed. Assignment to a variable that does not implement the `Copy` trait is interpreted as renaming, not copying. For example, in

```
    let s1 = String::from("hello");
    let s2 = s1;
    // ...more...
```

the variable `s1` is out of scope following the assignment to `s2`. Since the deallocation rule is based on static scope, it can be determined at compile time.

Rust first appeared in July, 2010; the first advertised stable release was on August 2, 2018.

Like Scala, Rust is in active development. Our discussion is based on the second edition of *The Rust Programming Language* [Klabnik and Nichols 2018] and the observed behavior of Rust version 1.41.0 (2020-01-27) In what follows, we consider Rust's partially hygienic macros based on pattern matching; Rust also supports totally unhygienic procedural macros.

What language is used to define a macro? Macros are written in Rust.

Unlike the situation in Scala, an invocation of a Rust macro can expand to an expression, a statement, a pattern, or zero or more methods or items; this is determined by the context of the macro use.

Macro definitions can be in the same compilation unit as their uses. It is possible to write macros that define other macros; Stansifer [NA 2016] gives an example of this.

Can macros examine the structure of their arguments? Not directly, but macro writers may be able to achieve much the same purpose by embedding their macro arguments within a suitable context.

Are macros expanded top-down or bottom-up? Top-down. Macro expansion occurs before name resolution or type-checking.

What representations are manipulated by macros? Macros manipulate untyped abstract syntax trees, after tokenization and limited parsing (as in Lisp or Scheme), but before name resolution or type-checking.

The basic operation of Rust macros is illustrated by the following code, which defines a macro similar to the macro defined in section 13.2. In this code, `global` is defined as a procedure, since a module may not export a plain variable.

```
#[macro_use]
mod macros {

    pub fn global () -> i32 {100}
    macro_rules! test {
    ($x:expr, $y:expr) => {
        {let local = $x;
         global() + $y + local}}
    }
}
```

How are uses of a macro parsed/matched/delimited? Macro uses are marked by writing ! immediately after the name of the macro, and its arguments are bracketed by parentheses. The macro arguments may be written as any sequence of valid Rust tokens in which parentheses, brackets, and braces are balanced and properly nested.

Pattern matching is used to select one of several rules to expand a macro invocation. Each macro pattern consists of a sequence of tokens, pattern variables, and repetitions. Pattern variables are marked with a *fragment specifier*, which is a primitive syntactic type like `ident`, `expr`, or `ty`. Repetition is specified using a Kleene star, written as a postfix “`*`” following a separator character such as “`,`”, which must then be used to separate multiple items. For example, the following macro takes a sequence of zero or more comma-separated expressions:

```
macro_rules! rec_macro {
    () => {println!("no arguments")};
    ($e1:expr) => {println!("last one: {}", $e1)};
    ($e1:expr, $($es:expr),*) => {{
        println!("one argument: {}", $e1);
        rec_macro!($($es),*);
    }}
}
```

Then

```
rec_macro!();
rec_macro!(100,200,300);
```

prints

```
no arguments
one argument: 100
one argument: 200
last one: 300
```

This example also illustrates recursive macros, which work roughly the way they do in `syntax-rules`.

In any macro, the patterns must form a (nearly) deterministic grammar. (There is no simple description of the allowed grammars, and backtracking is limited.) Once the parser begins consuming tokens for a capture, it cannot stop or backtrack, so the second rule of the following macro cannot ever match [Keep (no date)]:

```
macro_rules! dead_rule {
  ($e:expr)    => { ... };
  ($i:ident +) => { ... };
}
```

Does the macro system satisfy Kohlbecker's Hygiene Condition? It apparently satisfies the Hygiene Condition for `let` bindings in the macro body, but not in general. Consider this use of the `test` macro defined above:

```
fn main() {
    let local = 10;
    fn global () -> i32 {1000}
    let x = 10000;
    println!("{}", x + test!(1,local));
}
```

This prints `11011`. This indicates that the reference to `local` in the macro call (which was equal to 10) was not captured by the occurrence of `let local = ...` in the macro definition, as required by Kohlbecker's Hygiene Condition.

On the other hand, this also indicates that the reference to `global()` in the macro body, which should have returned 100, was captured by the definition of `global()` at the macro application site, which returned 1000. So the Strong Hygiene Condition is not satisfied.

However, if we bind `local` with a `fn` declaration instead of a `let`, the behavior is different. Consider the following example, which changes `local` to be a procedure, rather than a scalar, and binds it with `fun`.

```
#[macro_use]
mod macros {

    pub fn global () -> i32 {100}
    macro_rules! test {
    ($x:expr, $y:expr) => {
        {fn local () -> i32 {$x}
         (global() + $y + local())}}
    }
}

fn main() {

    fn local ()  -> i32 {100000}  // a big number
    fn global () -> i32 {1000}
    let x = 10000;
    println!("{}", x + test!(3,local()));
}
```

This prints `11006`, indicating that `$y`, which should have been bound to the procedure `local` at the macro call site (returning 100000), was overridden by the value of `local` at the definition site (returning 3). So the sum included the use-site value of `global()` (non-hygienic) and the definition-site value of `local()` (also non-hygienic).[6]

Does the macro system satisfy the Strong Hygiene Condition? No, as shown by the example above. As in Scala, qualified names such as `macros::global` can be used to work around this problem, and other workarounds can be used as well.

[6]Apparently, hygiene was implemented for `let` bindings but never extended to other binding forms. [NA Clements 2020]

16.4 Javascript with `sweet.js`

Sweet is probably the best example of a hygienic macro system for a programming language other than Scheme. It is a hygienic macro system for Javascript; its slogan is [NA Sweet (no date)a]:

Sweet brings the hygienic macros of languages like Scheme and Rust to JavaScript.

Sweet was developed by Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan [Disney et al. 2014]. Disney was a student of Cormac Flanagan, who was in turn a student of Matthias Felleisen; David Herman was a student of Mitchell Wand.

Our discussion here is based upon the Sweet tutorial [NA Sweet (no date)b] and our own testing of its examples and similar examples. Sweet does not provide a `--version` option; our testing used what we believe to be version 1.0, committed to github on August 4, 2017.

What language is used to define a macro? Sweet macros are written in Javascript, using an API for parsing and ES2015-like templates for constructing syntax trees. Templates return a list of syntax objects.

The first published paper on Sweet (Disney et al. [2014]) included both pattern-based and procedural macros, analogous to Scheme's `syntax-rules` and `syntax-case` macros (respectively). However, support for pattern-based macros was removed in version 1.0.

Macro definitions can be in the same compilation unit as their uses. It is possible to write macros that define other macros. Phasing is supported by allowing the import of values into the compile-time environment.

Hygiene is implemented using the set-of-scopes algorithm Flatt [2016], adapted for use with Sweet's parser [Disney 2015].

How are uses of a macro parsed/matched/delimited? Parsing and delimiting of macro arguments is accomplished using *enforestation* [Rafkind and Flatt 2012], an adaptation of top-down operator precence parsing [Pratt 1973]. In enforestation, parsing proceeds in two stages: first, the input is parsed into nested structures, using parentheses, braces, and the like as fixed delimiters. Within each level of nesting, infix notation is parsed using a top-down operator precedence algorithm (see [Rafkind and Flatt 2012; Disney et al. 2014] for details). When a token denoting a macro is encountered, the macro's definition is passed an iterator representing the remainder of the tokens in the current level; thus, a macro invocation cannot extend past the end of the current nesting level.

Can macros examine the structure of their arguments? Yes.

Are macros expanded top-down or bottom-up? Top-down.

What representations are manipulated by macros? In Scheme, the argument passed to a `syntax-case` macro is a single syntax object that is usually examined and decomposed via pattern matching.

In Sweet, the argument is an iterator, called the *macro context*, which represents the as-yet-unparsed portion of the JavaScript program. Calling the `next` method of the iterator does two things:

- It returns the first syntax tree representing the first item in the unparsed portion of the program, and a boolean `done` indicating whether there are more items in the context.
- It advances to the next item in the unparsed version.

A macro that wants to constrain its next argument to some syntactic category, such as an expression, can do so by calling the `expand` method of its macro context and passing the name of the desired syntactic category as an argument. For example, calling `.expand('expr')` consumes tokens from the context until an expression is consumed, and returns the syntax object for the expression as its value.

As an illustration, here is the example macro from Section 13.2:

```
let global = 100;

syntax test = function (ctx) {
    // sample usage:  test(expr1,expr2)
    let the_args = ctx.next().value
    // the_args is now a syntax object representing the tokens in
    // the list (expr1,expr2)
    let args_ctx = ctx.contextify(the_args);
    // call .contextify to create an iterator for the arguments
    // args_ctx now points to the first argument.
    let x = args_ctx.expand('expr').value;
    // parse the first argument as an expr
    args_ctx.next();   // consume the comma
    // consume the next argument
    let y = args_ctx.expand('expr').value;
    // ignore any more arguments
    return #`((local = ${x}) => global + ${y} + local)()`;
}

{let local = 10;
 let global = 1000;
 let x = 10000;
 console.log(x + test(1,,local));
}
```

A Sweet macro whose uses will pass multiple arguments can obtain the syntax objects for those arguments by calling the next method of the iterator until the done value is true.

Does the macro system satisfy Kohlbecker's Hygiene Condition? Yes.

Does the macro system satisfy the Strong Hygiene Condition? Yes.

16.5 Some Other Systems

ExJS [Arai and Wakita 2010; Wakita et al. 2014] is a prototype for another hygienic macro system for JavaScript. Instead of constructing an entirely new macro expander, it translates JavaScript programs that use macros into Scheme, uses the Ypsilon implementation of R6RS Scheme [NA Fujita (no date)] to expand macros hygienically, and then translates the macro-expanded code back into JavaScript.

The Pure programming language includes a macro system that satisfies the Strong Hygiene Condition [Gräf 2018].

There are some other languages that are often mentioned in connection with macro-enabled languages, but that do not have all the attributes of the other systems considered here.

Cardelli et al. [1993], in their paper "Extensible Grammars for Language Specialization," described a system for extensible concrete syntax based on attribute grammars. That system was primarily concerned with top-level notational definitions. The notations expand into a fixed target language with explicit scoping. Although the paper recognized the issue of hygiene, the system was not itself hygienic; it was left to the person defining the macro to introduce gensyms where necessary.

Ganz et al. [2001] presented a system for typed macros in an extension of ML called MacroML. In this system, macros were hygienic, so one could define macros to delay or force a computation by writing

```
mac $ e = fn x => e
mac ? e = e ()
```

and the x was guaranteed not to capture any occurrences of x in e. Furthermore, the system allowed macros to define new binding forms, such as

```
mac (let seq x = e1 in e2) =
   $(let val x = ?e1 in $e2)
```

to define a new binding form `let seq` in terms of the existing binding form `let val`.

The paper defined a core language in which macros were defined using syntax like

$$\mathsf{letmac}\ f(\sim x_0, x_1, \lambda x.x_2) = \ldots$$

This indicated that x_0 was an "early parameter" (an expansion-time value), x_1 was a "late parameter" (an expansion-time expression), and that x was to be bound in x_2.

The paper did not give any details about the translation from the full language to the core language. In our terminology, the core language was a "specified-binding" system like that of Griffin [1988].

17 FORMAL MODELS OF HYGIENE

The two properties of a hygienic macro system, as originally stated by Kohlbecker and refined by Clinger and Rees [1991a], appear reasonably clear, and the various algorithms discussed above make plausible claims to satisfy those criteria. But few researchers have attempted to prove these claims with any rigor. This is likely due to the fact that the researchers who developed the algorithms were more concerned with applicability than with mathematics.

17.1 What Does It Mean For an Expansion Algorithm to Be Correct?

The closest thing we have to a specification of hygiene is Clinger and Rees's *Strong Hygiene Condition* [Clinger and Rees 1991a] (which we have already mentioned in Section 7.1):

(1) It is impossible to write a high-level macro that inserts a binding that can capture references other than those inserted by the macro.
(2) It is impossible to write a high-level macro that inserts a reference that can be captured by bindings other than those inserted by the macro.

In principle, these criteria seem clear. But pinning them down is hard. How do we know that any particular binding or reference has been "inserted"? At best, one can define what these terms mean for a particular macro expansion algorithm. But if there is to be a notion of "hygiene" in general, one would like a formulation that is independent of any particular expansion algorithm, or is at least applicable to some class of expansion algorithms.

In this section, we will trace some attempts to prove that particular expansion algorithms have properties like this.

Attempts to formalize these notions can be divided into two broad classes: algorithms in which the binding structure of the macro is specified *a priori*, and those in which the binding structure must be inferred during expansion. We refer to the former as the *specified-binding* case, and to the latter as the *inferred-binding* case.

$$\begin{aligned} a \Leftrightarrow b &\overset{def}{=} (a \Rightarrow b) \wedge (b \Rightarrow a) \\ \forall_1 x \in a.b^x &\overset{def}{=} \forall x.(x \in a) \Rightarrow b \end{aligned}$$

Fig. 7. Examples of notational definition (from Griffin [1988], slightly modified).

17.2 Hygiene with Specified Binding

We begin with papers that consider the somewhat easier case of expansion in which the binding structure of the macro call is known in advance.

17.2.1 Griffin [1988]: Notational Definition — a Formal Account. This paper is set in the context of interactive proof checkers, where notational shortcuts are essential for usability. Figure 7 shows two examples of notational definitions, taken from the paper. The notation b^x signifies that b is in the scope of x (whether or not x actually occurs in b). The right-hand sides of these definitions must be closed (relative to any previous definitions), so these definitions are effectively "at the top level."[7]

The paper's goal is to provide a way of interpreting notational definitions in such a way that the "basic operations" (substitution, computation of free variables, and change of bound variables) can be applied to terms using the new notation, without expanding them. Furthermore, when expansion is necessary, it is desirable to so in a way that preserves variable names whenever possible.

This is accomplished by representing uses of notational definitions using higher-order abstract syntax (what Griffin calls "representing languages à la Church"). So, for example, an instance of the $\forall_1$ notation above might be represented as a lambda-term of the form $\forall_1(A, \lambda z.B)$, where A, z, and B instantiate the variables a, x, and b. Enclosing B in the lambda expression manifestly puts B in the scope of z. $\forall_1$ becomes a new constant with a reduction rule (a so-called δ-rule) so that any term of the form $\forall_1(A, \lambda z.B)$ reduces to a term α-equivalent to $\forall z.(z \in A) \Rightarrow B$. Since $\forall_1$ is just another new constant, the basic operations are well-defined on abbreviated terms.

Although simply reducing the term in this way is correct (by definition), ordinary $\beta\delta$-reduction will rename variables excessively. Therefore Griffin presented an expansion algorithm Exp^{Δ} that does this without renaming so many variables.

This paper is widely cited, but it seems to have had relatively little impact on either the theory or practice of hygienic macros that followed. Worthy of note, however, are the work of Bove and Arbilla [1992], which extended this model by using a variant of deBruijn indices and explicit substitution [Abadi et al. 1990, 1991] as a model for freshness, and that of Taha and Johann [2003], which applies the model to the multi-stage language MacroML [Ganz et al. 2001]. Interestingly, Griffin's paper cites Kohlbecker et al. [1986], saying that it addresses "a problem somewhat similar to the one addressed above for notational definitions."

17.2.2 Herman & Wand [2008]: A Theory of Hygienic Macros. This paper starts with the observation that the informal notion of hygiene is tied to α-equivalence, but

> ...The binding structure of a Scheme expression does not become apparent until after it has been fully expanded to core Scheme. Thus α-equivalence is only well-defined for Scheme programs that have been fully expanded.... So if the conventional wisdom is correct, the definition of hygienic macro expansion relies on α-equivalence, but the definition of α-equivalence relies on the results of macro expansion!

[7]The paper does not deal with how this mathematical notation is represented in text. Also, it overloads the symbol $\forall$; we have written $\forall_1$ to distinguish the new notation from the presumably built-in $\forall$.

> ...
> But observe that in practice, well-behaved macros follow regular binding disciplines consistently.... Programmers rely on knowing the binding structure of `let` without actually thinking about its expansion....
> ...We argue that *the binding structure of a macro is a part of its interface.*

The paper introduces a language of *shape types*, which express the intended shape of a macro call along with its intended binding structure. Shape types extend those of Culpepper and Felleisen [2004] by adding information about the binding relations among the macro arguments.

The following example gives a flavor of the type system:

$$\begin{aligned}&\texttt{macro}\ (?a\ ?b\ ?e_1\ ?e_2)\ :\ (\langle 0\rangle\langle 1\rangle\ \mathbf{expr}^{0}\ \mathbf{expr}^{0,1})\\ &\qquad\Rightarrow \lambda ?a.((\lambda ?b.?e_2)\ ?e_1)\end{aligned}$$

This tells us that if this macro is associated with the name m, then an invocation of m should be of the form $(m\,x\,y\,E_1\,E_2)$. The *binder address* $\langle 0\rangle$ says that the corresponding position ($?a$) should be filled with an identifier (locally named 0); similarly, $?b$ should be matched by another identifier. The type $\mathbf{expr}^0$ indicates a position that should be filled by an expression that will be within the scope of the identifier at $\langle 0\rangle$, and the type $\mathbf{expr}^{0,1}$ indicates a position that should be filled by an expression that will be within the scope of the identifiers at $\langle 0\rangle$ and $\langle 1\rangle$, with the identifier at $\langle 0\rangle$ possibly shadowed by the identifier at $\langle 1\rangle$. So here E_1 will be within the scope of x, and E_2 will be within the scope of x and y. In this case, the scoping relationships could be easily inferred from the template $\lambda ?a.((\lambda ?b.?e_2)\ ?e_1)$, but of course this will not always be so obvious. One benefit of the proposed system is that mismatches between the shape types and the template can be detected statically.

The paper then developed a theory of shape-directed α-equivalence. This was based on the nominal syntax of Gabbay and Pitts [2002]. In that theory, α-equivalence is defined using a notion of variable swapping, defined by

$$\begin{array}{lcl}(v_1 \leftrightarrow v_2)\bullet v_1 & = & v_2\\ (v_1 \leftrightarrow v_2)\bullet v_2 & = & v_1\\ (v_1 \leftrightarrow v_2)\bullet v & = & v \quad \text{otherwise}\\ (v_1 \leftrightarrow v_2)\bullet \lambda x.e & = & \lambda((v_1 \leftrightarrow v_2)\bullet x).((v_1 \leftrightarrow v_2)\bullet e)\\ \mathit{etc.} & & \end{array}$$

For the pure λ-calculus, α-equivalence is then defined by the rules in Figure 8. In the rule for abstractions, the hypothesis $z\#e, e'$ means that the name z is distinct from any name in e or e', and the rule says that to see if two abstractions are α-equivalent, swap the bound variable on each side for the same fresh variable, and see if the bodies are α-equivalent.[8]

Herman and Wand [2008] generalized this notion to arbitrary shape types using a notion of *shape-directed swapping*.

The paper presented a type system, whose job it was to check that each macro definition conforms to its specification and that each use of a macro conforms to its interface. The paper showed that shape-directed α-equivalence preserves these types.

The paper gave an expansion system, which took the form of a reduction system with three rules:

(1) Move the template of a macro m to an invocation site of m, provided the invocation matches the pattern and there are no variable conflicts.
(2) Perform a shape-directed substitution of a macro definition for a macro invocation.

[8] A meta-theorem says that the answer does not depend on the choice of z.

$$\frac{}{v =_\alpha v} \qquad \frac{e_1 =_\alpha e_1' \qquad e_2 =_\alpha e_2'}{(e_1\, e_2) =_\alpha (e_1'\, e_2')} \qquad \frac{\begin{array}{c} z \# e, e' \\ (z \leftrightarrow x) \bullet e =_\alpha (z \leftrightarrow x') \bullet e' \end{array}}{\lambda x.e =_\alpha \lambda x'.e'}$$

Fig. 8. α-equivalence for the λ-calculus, based on swapping [Gabbay and Pitts 2002].

(3) Alpha-convert to resolve variable conflicts.[9]

It is shown that a pattern-template macro that is well-typed will always expand to a well-typed term of the specified shape type.

The paper shows that the expansion algorithm is confluent, that is, any two complete expansions of the same term are α-equivalent. From this we can conclude that the system is hygienic, that is, expansion preserves shape-directed α-equivalence. The argument is that, given two α-equivalent input terms, we can apply the α-conversion rule to go from one to the other; then apply confluence.

The system was limited in its applicability: it covered only macros consisting of a single pattern-template pair, and the shape types had limited expressiveness, because they could not deal with lists or other recursive structures.

The concept of α-preservation as a formulation for hygiene has now been widely adopted, for example by Erdweg et al. [2014].

17.2.3 Herman [2010]: A Theory of Typed Hygienic Macros. In his dissertation, Herman extended the work in [Herman and Wand 2008] with multiple patterns and recursive shape types (for list structures of indefinite length, such as argument lists). Recursive shape types were technically difficult: they needed a more sophisticated notion of binder address and required dealing with subtyping.

To illustrate these difficulties, consider the expression `(let* ((x` e_1`) (y` e_2`) (z` e_3`))` e_4`)`. Here

- e_1 is within the scope of some outer bindings B_0;
- e_2 is within the scope of a set of bindings we might describe as $B_0 \triangleleft \mathtt{x}$, which means that this `x` shadows any previous binding for an `x` in B_0;
- e_3 is within the scope of $(B_0 \triangleleft \mathtt{x}) \triangleleft \mathtt{y}$; and
- e_4 is within the scope of $((B_0 \triangleleft \mathtt{x}) \triangleleft \mathtt{y}) \triangleleft z$

To describe this, Herman extended the shape types of Herman and Wand [2008] to become an attribute grammar with a single attribute, whose values are sets of bindings. A grammatical phrase could both import bindings as an inherited attribute and export them as a synthesized attribute. Thus in the example above, the phrase `(y` e_2`)` imports the bindings $B_0 \triangleleft \mathtt{x}$ and exports the bindings $(B_0 \triangleleft \mathtt{x}) \triangleleft \mathtt{y}$.

The shape types are a domain-specific language for the attribute grammars. The grammar of shape types (extracted and simplified from Herman [2010, Figure 3.1]) is given by:

$$\sigma \quad ::= \quad \mathsf{expr} \mid \mathsf{bvar} \mid () \mid (\sigma \otimes \sigma) \mid \sigma \downarrow \beta \mid \sigma \uparrow \beta \mid \mu X.\sigma \mid X \mid \cup\{\overline{\sigma_i}\}$$

Here

- expr matches any expression;
- bvar matches any identifier and interprets it as a binder;
- $(\sigma \otimes \sigma')$ matches the `cons` of anything matching σ and anything matching σ';

[9]It appears that the transform, as defined, is global, whereas a real expander can only alpha-convert within the term it is expanding. The dissertation has the same bug (Sec 4.4.2.3).

$$\begin{array}{lcl}
\mathsf{clauses} & = & \mu X.\cup\{(\,)\!\uparrow\!\varepsilon \\
 & & \qquad\qquad ((\mathsf{bvar}\ \mathsf{expr}) \otimes \langle X\!\downarrow\!\{\mathsf{AA}\}\rangle)\!\uparrow\!\mathsf{D}@\{\mathsf{AA}\}{::}\varepsilon\} \\
\mathsf{let}{*} & = & (\mathsf{clauses}\ \mathsf{expr}\!\downarrow\!\mathsf{A}@\varepsilon) \rightarrow \mathsf{expr}
\end{array}$$

Fig. 9. Types for `let*` (simplified from [Herman 2010, page 47]). In the second line $\langle \dots \rangle$ is a grouping symbol intended to help the reader parse the example.

- $\sigma \downarrow \beta$ indicates that σ imports bindings described by β, where β ranges over another sublanguage describing bindings; and
- $\sigma \uparrow \beta$ indicates that σ exports bindings described by β.

The last three productions allow recursion and alternation in the grammar.

Figure 9 shows the types for `let*`. A clauses is either an empty list, which exports nothing or the cons of a 2-list and another clauses X. X imports the identifier in the bvar position (the cAAr) of the phrase. The entire clause exports whatever is exported by its cDr plus the bvar (the cAAr of the phrase). A let* is a 2-list consisting of a clauses and an expr; the expr imports the bindings produced by the cAr of phrase (that is, the clauses).[10]

The main results of the thesis were the same as for [Herman and Wand 2008], extended to the new system. The details were extremely technical, and, as is so often the case, neither Herman nor Wand had the energy to condense and submit the expanded system for publication.

17.2.4 Stansifer & Wand [2014; 2016]: Romeo — A System for More Flexible Binding-Safe Programming. The aim of Stansifer's work was to extend the pattern-matching macros of Herman and Wand [2008]; Herman [2010] to procedural macros. To do this, Stansifer introduced a domain-specific language, called Romeo, for binder-safe meta-programming. The Romeo language was based on Pure FreshML [Pottier 2007]. In this system, macros were defined externally: rather than writing macros as part of the program, one writes a custom expander in Romeo that takes a term in the source language and expands away any macro calls. Thus the system is more like notational definitions [Griffin 1988] than full-fledged macros.[11]

Romeo had the following features:

- In Romeo, values were "plain old data": atoms arranged in abstract syntax trees without binding information. The missing binding information was provided by a type system.
- In Romeo, a type determined a set of values and an α-equivalence on those values. Just as in [Herman 2010], the values of the type were determined by an attribute grammar with a single attribute. The values of the attribute were sets of names representing bindings, and a language of *binding combinators* was used to specify the flow of this attribute between sibling nodes in the parse tree.
- The α-equivalence was derived by a set of inference rules based on the grammar. Alpha equivalence was complicated by the fact that a term both had a value and could export a set of bindings. Two terms were taken to be α-equivalent if they both exported identical bindings and if local (non-exported) bindings could be renamed along with the names that reference them to make the terms identical. The first was a straightforward induction on the term; the latter was done in the nominal style, but required a type-directed set of rules to collect the corresponding binding instances.
- The execution semantics of Romeo guaranteed that a fault is produced whenever a name escaped the context in which it is defined.

[10] As with so many theoretical dissertations, usability took a back seat.
[11] The same simplification is made in [Kohlbecker et al. 1986].

$$\begin{aligned} \mathit{LetStarClauses} ::=& \mathrm{Prod}() \\ & \mid \mathrm{Prod}^{\Uparrow 1 \triangleright 0}(\mathrm{Prod}^{\Uparrow 0}(\mathrm{BAtom}, \mathit{Expr}), \mathit{LetStarClauses}{\downarrow}0) \end{aligned}$$

Fig. 10. Types for let* (from Stansifer and Wand [2014]). Compare to Figure 9.

- A type system was defined, in which "well-typed programs don't go wrong," that is, a well-typed program would never generate a fault. The type system generated verification conditions that were resolved by calling out to a theorem-prover [Muehlboeck 2013].
- Last, the paper showed that programs in Romeo indeed preserved α-equivalence.

Figure 10 shows the definition of `let*` clauses in [Stansifer and Wand 2014]. To explain this, we quote from that paper:

> The grammar for *LetStarClauses* says that a set of `let*`-clauses is either the empty list or the cons of a single clause and the rest of the clauses. In the second production for *LetStarClauses*, the $\mathrm{Prod}^{\Uparrow 0}(\mathrm{BAtom}, \mathit{Expr})$ indicates that the first clause exports the binder from its position 0 (the BAtom), but that this name is not in scope in the *Expr*. The $\mathit{LetStarClauses}{\downarrow}0$ indicates that the names exported from the first clause are in scope in the remainder of the clauses. The $\Uparrow 1 \triangleright 0$ indicates that the entire set of clauses exports all the binders exported by either the first clause or the rest of the clauses, and that names in the rest of the clauses override those from the first clause.

As part of his dissertation [Stansifer 2016], Stansifer adapted Romeo to integrate with the Redex prototyping system [Felleisen et al. 2009]. As of June, 2019, there were approximately 200 hits on Github indicating a use of this feature of Redex.[12]

17.2.5 Pombrio, Krishnamurthi & Wand [2017]: Inferring Scope Through Syntactic Sugar. The goal of this paper was to *infer* binding patterns based on pattern/templates. More precisely, given a target language with a set of scoping rules and a set of global macro definitions in pattern/template form, the paper showed how to derive a set of scoping rules for the source language so that expansion preserves α-equivalence. This was intended to be done at language-definition time; the resulting set of rules was intended be in a form that could be given to other elements of the language tool chain, such as analyzers or debuggers.

As in Herman [2010]; Stansifer and Wand [2014], scope was conceptualized as an attribute grammar. Each non-terminal had two attributes: the set of bindings it imports and the set it exports. The propagation rules were given in textual form, as a set of annotations. The most common annotations were:

- **import i**: the i-th child imports the bindings imported by its parent. This behavior was the default in Herman [2010]; Stansifer and Wand [2014]. In the new system, omitting this annotation allowed a parent to hide all its bindings from a child.
- **bind i in j**: make the i-th child's bindings available in the j-th child. In particular, any declarations exported by child i were to be imported by child j.
- **export i**: The i-th child's exports are added to the exports of its parent.

For example, the scope rules for `(lambda` $\bullet_1$ $\bullet_2$`)` would be (**import 1**, **import 2**, **bind 1 in 2**).

Given these annotations and a program, a set of rules generated a preorder on the phrases of the program, so that $p \leq q$ meant that the bindings of q (imports or exports) flowed to p. So $x^R \leq x^D$

[12] https://github.com/search?q=binding-forms+refers-to&type=Code, accessed 10 June 2019.

meant that the reference x^R was in the scope of the declaration x^D.[13] Further rules defined a notion of α-equivalence relative to this preorder.

The main result of the paper was an algorithm *inferScope* that, when given a core language with scoping rules Σ_{core} and a set of global macro definitions in pattern/template form, produced a set of scoping rules Σ_{surf} with the property that

$$\Sigma_{surf} \vdash s =_\alpha t \implies \Sigma_{core} \vdash f(s) =_\alpha f(t)$$

where f is the expansion function determined by the macro definitions, that is, expansion preserved α-equivalence.

The algorithm worked roughly as follows: the macros were given in pattern/template form, that is, a set of rewrite rules $C \Rightarrow C'$. From each such rule, the algorithm generated a set of constraints of the form

$$F_1 \wedge F_2 \wedge \ldots F_n \iff F'_1 \wedge F'_2 \wedge \ldots F'_m$$

where each F_i and F'_j is either a fact about the core language, such as "**bind 1 in 2** $\in \Sigma$[`Lambda`]" or a fact about the surface language, such as "**bind 2 in 1** $\in \Sigma$[`Let`]". The algorithm then simplified these constraints by applying the rules "in reverse" to determine the least set of surface rules that satisfied the constraints. These became the inferred scoping rules for the macros.

The paper then applied the algorithm to a variety of well-known notational definitions:

- All of the macros that bind values in the Pyret language [NA Lerner et al. (no date)];
- Haskell list comprehensions, including guards, generators, and local bindings; and
- All of the global macros in R5RS Scheme [Kelsey et al. 1998] that bind values, including `let`, `let*`, `letrec`, and `do`

Some of the macro definitions had to be adjusted to eliminate ellipses; after this adjustment, the algorithm succeeded on all of the macros except for `do`. The algorithm failed on `do` because the binding language could not express lists in which the bindings are visible in some subexpressions but not others.

17.3 Hygiene with Inferred Binding

The key problem in inferring binding is that of associating each identifier or S-expression with the information that represents the lexical context in which it occurs and with the imperative data that represents the particular macro expansion in which it was generated.

In this section, we will review a few attempts to formalize this process.

17.3.1 Kohlbecker, Friedman, Felleisen & Duba: [1986]: Hygienic Macro Expansion. This is the original paper describing hygienic expansion. We will refer to it as "the KFFD paper."

The paper considered three main classes of S-expressions:

- An S-expression built from variables was termed an *stree*; variables were defined as "Lisp symbols that are used as identifier names" and were assumed disjoint from core tokens. Lambda terms were represented as a subset of the *strees*.
- An S-expression built from ordinary variables and (possibly) time-stamped variables was termed a *tsstree*. A time-stamped variable was assumed to be a pair consisting of a variable and a natural number; this may be regarded, in retrospect, as a primitive notion of a syntax object.

[13]The direction of the ordering was a matter of heated discussion among the authors; the direction was eventually chosen to be consistent with that used by Visser, who was on Pombrio's thesis committee [Neron, Tolmach, Visser, and Wachsmuth 2015].

- An S-expression consisting of a macro token (one of a distinguished set of tokens) and a sequence of *tsstrees* was termed a *tsmstree*.

A macro expansion is defined by a transformation called a "syntax table," whose type is described as

$$\theta \in ST = tsmstree \to tsstree$$

A syntax table θ takes a macro invocation (possibly containing time-stamped variables) and produces a time-stamped S-expression. Presumably this function would dispatch on the first symbol in the macro invocation and choose a transformation accordingly, but this is not required.

The full expansion algorithm $\mathcal{E}_{hyg}$ takes an *stree* and a syntax table and produces a lambda-term. The algorithm had four phases:

- The first phase ($\mathcal{T}$ in the paper) time-stamps all variables with 0.
- The second phase ($\mathcal{E}$ in the paper) is the key procedure. It takes a time-stamped S-expression (initially the result of the first pass), a syntax table θ, and a value for the clock (initially 1). If the S-expression is anything other than a macro invocation, it does a shallow copy, recurring on each subexpression.
  If the S-expression is a macro invocation, it applies the syntax table to it, yielding a new *tsstree*, and time-stamps "all macro-generated identifiers" with the current clock value. The algorithm determines which identifiers are macro-generated by assuming that all identifiers in the output of θ are either "pure identifiers or tokens that already occurred in [the input to θ.]" [Kohlbecker et al. 1986, p. 157].
  The phase $\mathcal{E}$ then recurs on the resulting S-expression with an incremented value for the clock.
- The third phase α-converts all time-stamped variables to unstamped ones, generating a fresh variable for each bound time-stamped variable.
- The fourth pass strips the time stamps from any remaining free variables.

The paper defines the desired hygiene condition as follows:

> **Hygiene Condition for Macro Expansion.**
> *Generated identifiers that become binding instances must only bind variables that are generated at the same transcription step.*

The KFFD paper is notable in that it states a theorem claiming the correctness of their algorithm. While it is unreasonable to expect the first paper on a subject to also include a perfectly formulated correctness theorem, in retrospect the paper offers some insights into the subtleties of the concept of hygiene and the difficulties of proving any useful theorems about that concept.

The theorem is stated as follows:

> **Theorem.** *Let θ be a syntax table and let θ' be the induced syntax table. Then, for all strees P, if $\mathcal{E}_{naive}[\![P]\!]\theta$ expands into a λterm, then $\mathcal{E}_{hyg}[\![P]\!]\theta'$ expands into a structurally equivalent term which satisfies the hygiene condition for macro expansion.*

That theorem and its proof have numerous problems. We list them in approximate order of importance, starting with the most important.

- To say that a term satisfies the hygiene condition is a category error. The hygiene condition, as stated, is a condition not on a term, but on the process by which that term was generated: it talks about "transcription steps" at which variables are "generated." In other words, it is an *intensional* property of terms, not an extensional one. Thus, unless the term carries with it some information about its history, there is no way to tell whether it satisfies the hygiene condition or not. Furthermore, in the KFFD paper, whatever information might be present is systematically erased in the third and fourth phases.

- The proof and associated text make unstated assumptions about the syntax table θ. The definition of the "induced" syntax table is highly unclear.
  For example, a θ might return a term containing a fixed symbol, like `temp`. This is normal, as in the case of `or`. It is not clear, however, whether the definition of the induced table would allow it to return a *tsstree* containing `temp` at a fixed time-stamp, say `temp:42`. This would satisfy the equation on page 155 of the KFFD paper, yet falsify the assertion in the proof that "The variables in $\theta' f$ are either pure identifiers or tokens that already occurred in f." (p. 157). This could in turn lead to a possible loss of hygiene.
- The property of "structural equivalence" is very weak. The paper says:
  > Two $\lambda terms$ are structurally equivalent if they are equal after replacing all bound variables by the symbol X.

  But that condition would be satisfied if we defined the hygienic expansion as taking the naïve expansion and then replacing each identifier (whether binding or bound) by a freshly generated identifier, so that no identifier appears more than once (except perhaps for free identifiers). The resulting term would be structurally equivalent to the naïvely expanded term, and it would trivially satisfy any reasonable interpretation of the Hygiene Condition, since no identifier would bind any other.[14]

17.3.2 Clinger and Rees [1991a]: Macros That Work. This paper was the first to deal with local macros. We will refer to it as "the MTW paper." It introduced the Strong Hygiene Condition, which is stated as:

> (1) It is impossible to write a high-level macro that inserts a binding that can capture references other than those inserted by the macro.
> (2) It is impossible to write a high-level macro that inserts a reference that can be captured by bindings other than those inserted by the macro.

The algorithm takes two arguments, a term E, and an environment e, which maps each source code name x into the name by which it should be replaced in the expanded term.[15] The algorithm is stated in a big-step rules-and-axioms style, where each assertion is of the form $e \vdash E \rightarrow E'$, which asserts that E' is the full expansion of E in environment e.[16]

Two sample rules (simplified) might be:

$$\frac{e(x) = x'}{e \vdash x \rightarrow x'}$$

$$\frac{e[x \mapsto x'] \vdash E \rightarrow E' \qquad x' \text{ fresh}}{e \vdash \texttt{(lambda (}x\texttt{) }E\texttt{)} \rightarrow \texttt{(lambda (}x'\texttt{) }E'\texttt{)}}$$

These two rules say that when the expansion algorithm encounters a reference x, it should be return the name specified by the environment e. If the algorithm encounters a lambda-expression with bound variable x, it should return a lambda expression whose bound variable is a fresh variable x', and the body should be expanded in an extended environment that replaces each occurrence of x by x'.

When the expander reaches a macro definition, the definition, along with its environment ("the definition environment") is saved as a closure in the environment:

[14]The definition of structural equivalence does not specify whether binding occurrences, reference occurrences, or both are to replaced. We interpret it as meaning both, since that is the intepretation that seems to make the most sense.
[15]Clinger says that this was merely a "stylistic change" from time-stamping; see Section 7.
[16]The paper also provides for renaming of the keywords `lambda` and `let-syntax`; we elide that complication here.

$$\frac{e[m \mapsto (\Delta, e)] \vdash E \to E'}{e \vdash \texttt{(let-syntax ((}m\ \Delta\texttt{))}\ E\texttt{)} \to E'}$$

Macros are defined by pattern-template pairs (l, r). Patterns are S-expressions containing names and pattern variables (assumed disjoint). During matching, the names in the pattern l other than pattern variables are interpreted as pattern literals in the definition environment e_{def}; names in the term being matched (the "scrutinee") are interpreted in the environment e_{use} at the point of use. Thus a name x in the pattern matches a name x' in the scrutinee if and only if $e_{def}(x) = e_{use}(x')$.

If the scrutinee matches the pattern, then a fresh name x'_j is generated for each name x_j ($j = 1, \ldots, n$) in the template r, the results of the match are substituted for pattern variables in the template r, and the resulting term is expanded in the environment

$$e_{use}[x'_1 \mapsto e_{def}(x_1)][x'_2 \mapsto e_{def}(x_2)] \cdots [x'_n \mapsto e_{def}(x_n)]$$

So new variables are generated at every macro expansion, but the new variables are given the meanings the old variables had *at definition time*.

The paper did not claim any theorems, but gave a plausibility argument. The argument relied on the fact that the variables generated by a transcription step are precisely the freshenings of the variables in the template r. By this argument, the process would satisfy Kohlbecker's version of the hygiene condition:

> **Hygiene Condition for Macro Expansion.**
> *Generated identifiers that become binding instances must only bind variables that are generated at the same transcription step.*

A limitation of this argument is that the status of pattern variables is ambiguous. The paper said: "We are not completely certain that this refinement eliminates all problems with macros that define other macros. [Clinger and Rees 1991a, p. 162]" As it turned out, the "refinement" mentioned in that paragraph did seem to have eliminated all problems with macros that define other macros, except for the trivial syntactic problem with ellipses whose solution is described at the end of Section 10. Nevertheless, the meaning of the Hygiene Condition remains subtle and subject to disagreement, as discussed in Section 14.

It is worth noting that the conditions of the Strong Hygiene Condition as stated are more complex than those of the original, since they involve a hypothetical contrapositive rather than the safety condition of Kohlbecker's version.

17.3.3 Adams [2015]: Towards the Essence of Hygiene. The stated goal of this paper was to present "a precise, mathematical definition of hygiene."

After numerous examples, the paper's new material begins in Sections 4.7–4.8, which present an expansion algorithm.

In Adams's system, the expander takes an S-expression over syntax objects (called the *U-syntax*), and produces a syntax tree with ordinary Scheme identifers (called the *K-syntax*).

The syntax objects in Adams's system are "diatomic" identifiers. A diatomic identifier is a syntax object $\langle r, b \rangle$ consisting of two atoms: a reference atom r and a binder atom b, chosen from disjoint sets. The reference atom is the name that is used if this identifier is used as a reference. The binder atom is "used to determine what identifiers are captured when the identifier ends up in a binding position." This means that an identifier $\langle r, b \rangle$ that is found in a binding position will capture an identifier $\langle r', b' \rangle$ that is found in a reference position if and only if $b = b'$.[17]

[17]Thus, `free-identifier=?` corresponds to equality of reference components; `bound-identifier=?` corresponds to equality of binder components.

This is analogous to an environment entry $[b \mapsto r]$ in the system of Clinger and Rees [1991a]. Adams's system, however, uses a substitution-based model, in which substitutions are performed eagerly. So a typical rewrite rule is

```
#(<lambda,b0> <r,b> exp) =>
   (lambda r' #exp[<_,b> -> <r',b>])
```

where # marks the boundary between the K-syntax and the U-syntax, and `exp[<_,b> -> <r',b>]` denotes denotes the U-syntax object obtained from the U-syntax `exp` by substituting $\langle r', b\rangle$ for every occurrence of a syntax object with binding atom b.

The actual algorithm is presented in Adams [2015, Figure 5]. Most of the cases follow the pattern for `lambda` above. The case of macro expansion is given by:

$$(\langle r, b\rangle\ \overrightarrow{args}) \rightsquigarrow f(\langle r, b\rangle\ \overrightarrow{args})$$

"where f is the currently in-scope transformer bound to r and all identifiers introduced by f have freshly generated binder parts."

However, Figure 5 does not indicate how a `let-syntax` or `letrec-syntax` creates such an f or brings it in scope, nor does it specify how to identify "the identifiers introduced by f." It appears that the reader is intended to regard Figure 5 as a framework for an expansion algorithm, and that the definitions of hygiene in later sections are to be interpreted over any expansion algorithm that fits into this framework.

Figure 5 presents an eager model, which may lead to exponential behavior, like Kohlbecker's algorithm, but the paper suggests that the algorithm of Figure 5 could also be interpeted lazily, as in Dybvig et al. [1993]. This would presumably lead to the same linear behavior as that of Dybvig et al. [1993].

Section 5 defines α-equivalence $\simeq$ on the K-syntax, in a nominal style much like the definition of α-equivalence given above in Figure 8. In the system of this paper, permutation is defined on a per-atom, not a per-identifier basis, so permutation of identifiers is defined by:

$$\pi \bullet \langle r, b\rangle = \langle \pi \bullet r, \pi \bullet b\rangle$$

With this definition in place, the paper states a definition of hygiene with two conditions corresponding to the two conditions of the Strong Hygiene Condition.

> **Criterion 1** (Reference Hygiene). *A macro expansion step is hygienic with regard to references iff for any partially expanded expressions e_1, e_2 and e_1' where $e_1 \simeq e_2$ and $e_1 \rightsquigarrow e_1'$, there exists an e_2' such that $e_1' \simeq e_2'$ and $e_2 \rightsquigarrow e_2'$. A macro system is hygienic with regard to references iff all its expansion steps are hygienic with regard to references.*
> **Criterion 2** (Binder Hygiene) *A macro transformer is hygienic with respect to binders iff it is equivariant with respect to binder atoms. A macro system is hygienic with respect to binders if all its transformers are hygienic with respect to binders.*

The first condition is natural; it corresponds to the intuition in [Herman and Wand 2008] that changing the names of bound variables should not change the binding pattern of the expanded code. In particular this implies that a binder in the macro definition cannot accidently capture a subsequently-bound reference in the macro use (as in the classic case of `or`), because that would not preserve α-equivalence.

The second condition is less obvious. It says that for any macro transformer f, any syntax object x and any permutation π that permutes only binder atoms,

$$f(\pi \bullet x) = \pi \bullet f(x)$$

This condition prevents f from forging a fixed binder atom (like a time-stamped identifier), as discussed on page 94, because a permutation π that changed that binder atom would be a counterexample to the equivariance condition.[18]

The equivariance condition as stated seems far removed from the conventional statement of Binder Hygiene. However, if f is equivariant on binders, then it would be possible to assert something like "the variables in $f(x)$ are either pure identifiers or tokens that already occurred in x" (*cf.* page 94). From this one could conclude that it would be impossible for the macro use context to capture a variable from the definition context (unless that variable was already in scope at the use context). This connection is not worked out in the paper.

The paper proposes a construct called `fresh` for generating the needed fresh binder atoms. Its operational semantics are unclear, but this could probably be worked out using additional operations from nominal logic [Gabbay and Pitts 2002] or deBruijn indices [Gasbichler 2006].

17.4 What Can We Prove About Hygiene?

After all this, what progress have we made on proving any of these expansion algorithms correct? What are the prospects for a correctness proof?

The first question is what a correctness theorem might even look like. We need to distinguish between an "internal" view of correctness and an "external" one.

The internal view asks the question: given a macro definition and an invocation of that macro, what are the acceptable expansions of that invocation? The internal view is exemplified by the Strong Hygiene Condition and its variants: an expansion is acceptable iff it can be produced by some *process* that satisfies the Strong Hygiene Condition.

The external view asks the question: given a program with some macro definitions and invocations, what properties can the containing program expect the macro system to guarantee? The external view is exemplified by the "explicit binding" papers in section 17.2: the interface of the macro says it guarantees that any invocation of the macro will behave as if it satisfied the specified scope rule.

In an ideal world, one might like to have a meta-theorem that would allow a system like DeepSpec [Appel et al. 2016] to work on languages with macros. Presumably the "internal" correctness theorem would be an important lemma in such a system.

There are formidable technical challenges in the face of any of these goals:

- As suggested in Section 14.1, the details of applying any theoretical result to a real language are fraught. But even for an idealized language, the obstacles are difficult:
- As suggested in Section 14.2, there can be a fair amount of disagreement about what hygiene requires in the presence of macro-defining macros (a common case).
- That means the strategy of using a reference implementation can be used only for the relatively small subset of macro definitions on which there is a consensus about the desired behavior.
- For the specified-binding cases, proofs have been obtained, but they are sufficiently complex that even verifying the proof is difficult. It is necessary both to check that the proof is correct, and to verify that the statement of the theorem is what is wanted.
- All existing theory seems to be tightly coupled to particular expansion algorithms and to particular macro-definition languages. A good theory should deal at least with some reasonably general class of expanders and of macro-definition languages.

We hope the perspective offered by this survey may give researchers some clues going forward.

[18]The paper also observes that the *mark* operation of Dybvig et al. [1993] is essentially an opaque permutation, lazily-applied.

18 CONCLUSION

Hygienic macro technology solves the capturing problem, and pattern languages make hygienic macros easier to write and to understand.

Hygienic macro technology has been a standard part of Scheme for twenty years, allowing average programmers to write routine macros with confidence and expert programmers to implement significant extensions to the language. Hygienic macro technology has had a strong influence on the design of Scheme's library and record systems, and is the core technology used in their implementations.

Implementing hygienic macros for languages with conventional syntax poses additional problems, but several of those languages have begun to offer macro facilities with at least some support for partial hygiene. We hope this history of hygienic macro technology will increase awareness and acceptance of hygienic macros among programmers and designers, while encouraging designers and implementers to improve their support for hygiene.

Fig. 11. *The Right Way to Go.* (Photograph taken by Jed Clinger while leaning out the passenger window of a rental car stopped on the eastbound shoulder of Colorado State Highway 66, 22 August 2018.)

ACKNOWLEDGMENTS

We thank our shepherd, Guy Steele, his co-chair Dick Gabriel, and the rest of the program committee and external reviewers for their many detailed and valuable comments.

We also thank John Clements, Jed Clinger, John Cowan, Ryan Culpepper, Matthew Flatt, Dan Friedman, Chris Hanson, Jonathan Rees, Paul Stansifer, and Ming-Ho Yin for their assistance during preparation of this paper.

All opinions, unsubstantiated claims, and outright errors in this paper are of course ours, but this paper contains fewer errors than it would have without their help.

REFERENCES

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1990. Explicit Substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*. 31–46. https://doi.org/10.1145/96709.96712

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1991. Explicit Substitutions. *J. Funct. Program.* 1, 4, 375–416. https://doi.org/10.1017/S0956796800000186

Hal Abelson. 1988. RRRS-AUTHORS email with Subject: Can we standardize Scheme without killing it? 6 Feb 1988. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Hal Abelson. 1989. RRRS-AUTHORS email with Subject: macros. 26 May 1989. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Harold Abelson, Gerald Jay Sussman, and with Julie Sussman. 1985. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA.

Michael D. Adams. 2015. Towards the Essence of Hygiene. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 457–469. https://doi.org/10.1145/2676726.2677013

Norman Adams. 1987. RRRS-AUTHORS email with Subject: structures. 8 Jul 1987. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Norman Adams. 1992. RRRS-AUTHORS email with Subject: record proprosal. 17 May 1992. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. 1986. ORBIT: an optimizing compiler for Scheme. In *SIGPLAN '86 Proceedings of the the 1986 SIGPLAN Symposium on Compiler Construction.* (Aug), 219–233. https://dl.acm.org/doi/10.1145/12276.13333 Also at https://cpsc.yale.edu/sites/default/files/files/tr445.pdf

Torben Amtoft. 1993. Minimal Thunkification. In *WSA '93 Proceedings of the Third International Workshop on Static Analysis* (*WSA âĂŹ93*). Springer-Verlag, Berlin, Heidelberg (Sep), 218–229. https://dl.acm.org/doi/10.5555/647164.717823

Andrew Appel, Adam Chlipala, Benjamin Pierce, Zhong Shao, Stephanie Weirich, Steve Zdancewic, and Lennart Beringer. 2016. DeepSpec: the Science of Deep Specification. NSF Grant. Archived at Internet Archive: https://web.archive.org/web/20190616201913/https://deepspec.org/main

Hiroshi Arai and Ken Wakita. 2010. An Implementation of a Hygienic Syntactic Macro System for JavaScript: A Preliminary Report. In *Workshop on Self-Sustaining Systems* (Tokyo, Japan) (*S3 âĂŹ10*). Association for Computing Machinery, 30âĂŞ40. https://doi.org/10.1145/1942793.1942798

The article describes an implementation scheme of a hygienic syntactic macro system for JavaScript. Instead of implementing the complex logic of a hygienic macro system from scratch, the proposed method heavily relies on an existing Scheme implementation of its hygienic syntactic macro system.

H. P. Barendregt. 1985. *The Lambda Calculus—Its Syntax and Semantics*. Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland.

H. P. Barendregt. 1992. Lambda calculi with types. In *Handbook of Logic in Computer Science (Volume 2)*. Oxford University Press, 117–309.

Joel Bartlett. 1989. RRRS-AUTHORS email with Subject: First BASH Meeting. 8 Aug 1989. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

David Bartley. 1987a. RRRS-AUTHORS email with Subject: Agenda. 10 Jun 1987. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

David Bartley. 1987b. RRRS-AUTHORS email with Subject: macros. 10 Apr 1987. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

David Bartley, Chris Hanson, and Jim Miller. 1991. *IEEE Standard for the Scheme Programming Language*. IEEE. 1178-1990.

Alan Bawden. 1987. RRRS-AUTHORS email with Subject: Better late than never. 9 Jun 1987. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Alan Bawden, William Clinger, Kent Dybvig, Matthew Flatt, Richard Kelsey, Manuel Serrano, and Michael Sperber. 2004. Scheme Standardization. Jan 2004. NON-ARCHIVAL http://www.r6rs.org/charter/charter-mar-2006.txt (also at Internet Archive 26 Sept. 2019 06:53:41).

Alan Bawden, William Clinger, Kent Dybvig, Matthew Flatt, Richard Kelsey, Manuel Serrano, Michael Sperber, Marc Feeley, and Jonathan Rees. 2010. Scheme Standardization. Nov 2010. NON-ARCHIVAL http://www.scheme-reports.org/2009/charter.html (also at Internet Archive 17 April 2019 12:12:45).

Alan Bawden and Jonathan Rees. 1988. Syntactic closures. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*. ACM (July), 86–95. https://dl.acm.org/doi/abs/10.1145/62678.62687

Alan Bawden, Guy Steele, and Mitchell Wand. 2006. Scheme Language Steering Committee Report to the Community. Mar 2006. NON-ARCHIVAL http://lambda-the-ultimate.org/node/1342 (also at Internet Archive 7 March 2020 14:08:49).

Alan Bawden, Guy Steele, and Mitchell Wand. 2007a. R6RS Ratification. Aug 2007. NON-ARCHIVAL http://www.r6rs.org/ratification/ (also at Internet Archive 25 Sept. 2019 02:18:36).

Alan Bawden, Guy Steele, and Mitchell Wand. 2007b. R6RS Ratification Vote. Aug 2007. NON-ARCHIVAL http://www.r6rs.org/ratification/results.html (also at Internet Archive 6 June 2019 12:36:23).

Alan Bawden, Guy Steele, and Mitchell Wand. 2009a. Steering Committee Election Results. 2 Mar 2009. NON-ARCHIVAL http://www.r6rs.org/steering-committee/election/results.html (also at Internet Archive 25 Sept. 2019 02:19:22).

Alan Bawden, Guy Steele, and Mitchell Wand. 2009b. Steering Committee Replacement. Jan 2009. NON-ARCHIVAL http://www.r6rs.org/steering-committee/election/announcement.html (also at Internet Archive 25 Sept. 2019 02:18:56).

Daniel G. Bobrow and Daniel L. Murphy. 1967. Structure of a LISP system using two-level storage. In *Communications of the ACM*, Vol. 10. (March), 155–159. https://dl.acm.org/doi/10.1145/363162.363185 The footnote on page 158 suggests that a shallow binding technique for implementing dynamic scoping solves the FUNARG problem, which is not true.

Ana Bove and Laura Arbilla. 1992. A confluent calculus of Macro expansion and evaluation. In *1992 ACM Conference on LISP and Functional Programming*. http://doi.acm.org/10.1145/141471.141562

S. Bradner. 1997. Key words for use in RFCs to Indicate Requirement Levels. Mar 1997. https://tools.ietf.org/html/rfc2119 (also at Internet Archive 26 June 2019 13:24:34).

Gary Brooks. 1984. RRRS-AUTHORS email with Subject: Environments and Macros. 20 Oct 1984. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

P. J. Brown. 1974. *Macro Processors and Techniques for Portable Software*. Wiley.

Eugene Burmako. 2013. Scala Macros: Let Our Powers Combine! On How Rich Syntax and Static Types Work with Metaprogramming. In *SCALA '13*. NON-ARCHIVAL http://scalamacros.org/papers.html (also at Internet Archive 24 June 2018 03:30:25).

Eugene Burmako and Martin Odersky. 2012. Scala Macros, a Technical Report. In *Third International Valentin Turchin Workshop on Metacomputation*. NON-ARCHIVAL http://scalamacros.org/papers.html (also at Internet Archive 24 June 2018 03:30:25).

Clyde Camp. 1988. RRRS-AUTHORS email with Subject: Scheme Standardization. 6 Jan 1988. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Luca Cardelli, Florian Matthes, and Martín Abadi. 1993. Extensible Grammars for Language Specialization. In *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages, Manhattan, New York City, USA, 30 August - 1 September 1993* (*Workshops in Computing*), Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha (Eds.). Springer, 11–31. https://doi.org/10.1007/978-1-4471-3564-7_2

T. E. Cheatham, Jr. 1966. The introduction of definitional facilities into higher level programming languages. In *AFIPS Proceedings—Fall Joint Computer Conference 29*. 623–637. https://dl.acm.org/doi/10.1145/1464291.1464359

Will Clinger. 1984a. RRRS-AUTHORS email with Subject: feature list for workshop (long message). 15 Oct 1984. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Will Clinger. 1984b. RRRS-AUTHORS email with Subject: questions for workshop (long message). 15 Oct 1984. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Will Clinger. 1984c. RRRS-AUTHORS email with Subject: TI position (long message). 17 Oct 1984. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Will Clinger. 1985. RRRS-AUTHORS email with Subject: DRAFT of Revised Revised Report (LONG message). 18 Mar 1985. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Will Clinger. 1986. RRRS-AUTHORS email with Subject: Minutes from lunch 5 August 1986. 14 Aug 1986. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Will Clinger. 1987. RRRS-AUTHORS email with Subject: Minutes of the Scheme meeting etc. 10 Jul 1987. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

William Clinger. 1988a. The June 1987 Meeting. *ACM SIGPLAN Lisp Pointers* I, 5 (Dec-Jan-Feb-Mar), 25–27. https://doi.org/10.1145/1317273.1317276

William Clinger. 1988b. Standardization at Snowbird. *ACM SIGPLAN Lisp Pointers* II, 2 (Oct-Nov-Dec), 43–47. https://doi.org/10.1145/1317250.1317254

William Clinger. 1989. RRRS-AUTHORS email with Subject: planned changes to R3.95RS. 25 May 1989. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

William Clinger. 1991a. Hygienic macros through explicit renaming. *ACM SIGPLAN Lisp Pointers* IV, 4 (Oct-Nov-Dec), 25–28. https://doi.org/10.1145/1317265.1317269 Also at NON-ARCHIVAL ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/prop/exrename.ps.gz

William Clinger. 1991b. Macros in Scheme. *ACM SIGPLAN Lisp Pointers* IV, 4 (December), 17–23. ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/prop/macinsch.ps.gz (also at Internet Archive 19 Nov. 2018 09:05:25).

William Clinger. 1992. RRRS-AUTHORS email with Subject: draft minutes of June 1992 meeting. 9 Oct 1992. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

William Clinger. 1993. RRRS-AUTHORS email with Subject: proposals for R5RS (sorry). 11 Apr 1993. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

William Clinger, Hal Abelson, Norman Adams, David Bartley, Gary Brooks, Dan Friedman, Robert Halstead, Chris Hanson, Chris Haynes, Eugene Kohlbecker, Don Oxley, Kent Pitman, Jonathan Rees, Bill Rozas, Gerald Jay Sussman, and Mitchell Wand. 1985. The revised revised report on Scheme, or an uncommon Lisp. *MIT AI Memo 848* (August). https://dspace.mit.edu/handle/1721.1/5600

Will Clinger, R. Kent Dybvig, Michael Sperber, and Anton van Straaten. 2005. SRFI 76: R6RS Records. Sep 2005. https://srfi.schemers.org/srfi-76/ (also at Internet Archive 27 Nov. 2018 01:15:17).

Will Clinger, Marc Feeley, Chris Hanson, Jonathan Rees, and Olin Shivers. 2009. Announcement of `scheme-reports.org` web site. Posted to `scheme-reports`. 20 Oct 2009. NON-ARCHIVAL http://www.scheme-reports.org/2009/announcement.html (also at Internet Archive 17 April 2019 12:12:37).

William Clinger and Jonathan Rees. 1991a. Macros That Work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Orlando, Florida, USA) (*POPL âĂŹ91*). Association for Computing Machinery, New York, NY, USA (January), 155âĂŞ–162. https://doi.org/10.1145/99583.99607 PDF also archived at https://web.archive.org/web/20191027100429/https://3e8.org/pub/pdf-t1/macros_that_work.pdf

William Clinger and Jonathan Rees. 1991b. The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers* 4, 3, 1–55. https://dl.acm.org/doi/10.1145/382130.382133; Also at NON-ARCHIVAL https://bitbucket.org/cowan/r7rs/raw/4c27517de187142ad2cf4bcd8cb9199ae1e48c09/rnrs/r4rs.pdf

William D Clinger. 2006. Formal comment #90: Record layers are not orthogonal. Nov 2006. NON-ARCHIVAL http://www.r6rs.org/formal-comments/comment-90.txt (also at Internet Archive 18 May 2013 01:48:12).

William D Clinger. 2007a. Formal comment #267: syntactic sugar causes cancer of the exports. Apr 2007. NON-ARCHIVAL http://www.r6rs.org/formal-comments/comment-267.txt (also at Internet Archive 8 Aug. 2008 12:20:09).

William D Clinger. 2007b. Rationale issues. Posted to `r6rs-discuss`. 27 Jun 2007. NON-ARCHIVAL http://lists.r6rs.org/pipermail/r6rs-discuss/2007-June/002889.html (also at Internet Archive 5 July 2008 18:03:01). Response to van Tonder [2007a].

William D Clinger. 2007c. Re: [r6rs-discuss] an essay on language design. Posted to `r6rs-discuss`. 25 Jul 2007. NON-ARCHIVAL http://lists.r6rs.org/pipermail/r6rs-discuss/2007-July/003125.html (also at Internet Archive 28 Aug. 2008 11:51:04). Repairs a truncated email that had been sent on 10 July.

William D Clinger. 2007d. Re: [r6rs-discuss] an essay on language design. Posted to `r6rs-discuss`. 24 Jul 2007. Archived at Internet Archive: https://web.archive.org/web/20070915235512/http://lists.r6rs.org/pipermail/r6rs-discuss/2007-July.txt.gz

This is a bitter complaint about two paragraphs of “tendentious nonsense” that were added to the R$^{5.97}$RS draft, which was up for ratification and would be ratified in August 2007.

William D Clinger. 2007e. Vote against ratification of R6RS. Aug 2007. NON-ARCHIVAL http://www.r6rs.org/ratification/results.html#X101 (also at Internet Archive 6 June 2019 12:36:23).

William D Clinger. 2008. Scheme@33. In *LISP50: Celebrating the 50th Anniversary of Lisp* (Nashville, Tennessee) (*LISP50*). Association for Computing Machinery, New York, NY, USA, Article Article 7, 5 pages. https://doi.org/10.1145/1529966.1529973

William D Clinger. 2009. SRFI 99: ERR5RS Records. Oct 2009. https://srfi.schemers.org/srfi-99/ (also at Internet Archive 3 Aug. 2017 12:43:58).

William D Clinger. 2015. R7RS Considered Unifier of Previous Standards. In *Scheme and Functional Programming Workshop 2015*. Archived at Internet Archive: https://web.archive.org/web/20160911102749/http://schemeworkshop.org/2015/sfpw1-2015-clinger.pdf

William D Clinger. 2016. clarifying that field names are symbols. Posted to SRFI 99 discussion list. 21 Jun 2016. https://srfi-email.schemers.org/srfi-99/msg/3905499/ (also at Internet Archive 7 March 2020 15:17:22).

William D Clinger. 2017a. apparent bug in sample implementation of SRFI 148. Posted to SRFI 148 discussion list. 18 Jul 2017. NON-ARCHIVAL https://srfi-email.schemers.org/srfi-148/msg/6084267/ (also at Internet Archive 7 March 2020 18:15:45).

William D Clinger. 2017b. Re: apparent bug in sample implementation of SRFI 148. Posted to SRFI 148 discussion list. 19 Jul 2017. NON-ARCHIVAL https://srfi-email.schemers.org/srfi-148/msg/6090613/ (also at Internet Archive 7 March 2020 18:18:52).

William D Clinger. 2017c. Re: apparent bug in sample implementation of SRFI 148. Posted to SRFI 148 discussion list. 20 Jul 2017. NON-ARCHIVAL https://srfi-email.schemers.org/srfi-148/msg/6091649/ (also at Internet Archive 7 March 2020 18:21:56).

William D Clinger. 2017d. Re: apparent bug in sample implementation of SRFI 148. Posted to SRFI 148 discussion list. 20 Jul 2017. NON-ARCHIVAL https://srfi-email.schemers.org/srfi-148/msg/6091691/ (also at Internet Archive 7 March 2020

18:20:35).

William D Clinger. 2017e. Re: apparent bug in sample implementation of SRFI 148. Posted to SRFI 148 discussion list. 20 Jul 2017. NON-ARCHIVAL https://srfi-email.schemers.org/srfi-148/msg/6092367/ (also at Internet Archive 7 March 2020 18:25:07).

William D Clinger and Lars Thomas Hansen. 1994. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In *Proceedings of the 1994 ACM conference on LISP and Functional Programming*, Vol. VII. Association for Computing Machinery, New York, NY, USA (July), 128–139. https://doi.org/10.1145/182590.156786 PDF archived at https://web.archive.org/web/20191027101720/http://3e8.org/pub/scheme/doc/lisp-pointers/v7i3/p128-clinger.pdf

Ludovic Courtès. 2007a. R6RS Formal Comment #264: Versioned names for base libraries considered harmful. 14 Jun 2007. NON-ARCHIVAL http://www.r6rs.org/formal-comments/comment-264.txt (also at Internet Archive 4 July 2008 13:45:40).

Ludovic Courtès. 2007b. Versioned standard libraries. Posted to `r6rs-discuss`. 24 Sep 2007. NON-ARCHIVAL http://lists.r6rs.org/pipermail/r6rs-discuss/2007-September/003321.html (also at Internet Archive 28 Aug. 2008 19:26:42).

John Cowan. 2016. R7RS Small Errata (unofficial). Nov 2016. NON-ARCHIVAL https://small.r7rs.org/wiki/R7RSSmallErrata/ (also at Internet Archive 31 Dec. 2019 22:51:41).

John Cowan. 2017. RedEdition Version 8. 28 Aug 2017. NON-ARCHIVAL https://small.r7rs.org/wiki/RedEdition/ (also at Internet Archive 8 March 2020 22:52:40).

Ryan Culpepper and Matthias Felleisen. 2004. Taming Macros. In *Generative Programming and Component Engineering (GPCE 2004).* (October). https://doi.org/10.1007/978-3-540-30175-2_12 Also at NON-ARCHIVAL http://www.ccs.neu.edu/scheme/pubs/gpce2004-cf.ps PDF: NON-ARCHIVAL http://www.ccs.neu.edu/scheme/pubs/gpce2004-cf.pdf

Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. 2007. Advanced Macrology and the Implementation of Typed Scheme. In *2007 Workshop on Scheme and Functional Programming.* (September). https://dl.acm.org/doi/abs/10.1145/1328438.1328486 https://www.semanticscholar.org/paper/Advanced-Macrology-and-the-Implementation-of-Typed-Culpepper-Tobin-Hochstadt/88d82e5cf6e3bd91cba77b48c90a242404e33a55

Pavel Curtis. 1989. RRRS-AUTHORS email with Subject: Programmer-defined data types. 18 Aug 1989. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Pavel Curtis and James Rauen. 1990. A module system for Scheme. In *Proceedings of the 1990 ACM conference on LISP and Functional Programming* (Nice, France) (*LFP âĂŽ90*). Association for Computing Machinery, New York, NY, USA (Jun), 13–19. https://doi.org/10.1145/91556.91573

Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. 2014. Sweeten your JavaScript: hygienic macros for ES5. In *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014.* 35–44. https://doi.org/10.1145/2661088.2661097 Also at NON-ARCHIVAL https://www.disnetdev.com/papers/sweeten-your-javascript/

Timothy Charles Disney. 2015. *Hygienic Macros for JavaScript.* Ph.D. Dissertation. UC Santa Cruz.

Kent Dybvig. 1985. RRRS-AUTHORS email with Subject: Chez Scheme. 4 Feb 1985. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Kent Dybvig. 2005. SRFI 93: R6RS Syntax-Case Macros. Sep 2005. https://srfi.schemers.org/srfi-93/ (also at Internet Archive 27 Nov. 2018 01:15:15).

Kent Dybvig, Will Clinger, Matthew Flatt, Mike Sperber, and Anton van Straaten. 2006. R6RS Status Report. Feb 2006. NON-ARCHIVAL https://schemers.org/Documents/Standards/Charter/status-mar-2006.html (also at Internet Archive 27 Nov. 2018 01:12:56).

R. Kent Dybvig. 1987. *The Scheme Programming Language.* Prentice Hall.

R. Kent Dybvig. 1992. *Writing Hygienic Macros in Scheme with Syntax-Case.* Technical Report 356. Indiana University Computer Science Department (June). https://www.cs.indiana.edu/ftp/techreports/TR356.pdf

R. Kent Dybvig. 1993. RRRS-AUTHORS email with Subject: Re: proposals for R5RS (sorry). 12 Apr 1993. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

R. Kent Dybvig. 1996a. RRRS-AUTHORS email with Subject: compromise record-type proposal. 26 Apr 1996. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

R. Kent Dybvig. 1996b. RRRS-AUTHORS email with Subject: Re: Generative record types. 24 Apr 1996. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

R. Kent Dybvig. 1996c. RRRS-AUTHORS email with Subject: Re: Record proposal. 29 Apr 1996. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

R. Kent Dybvig. 2000. From Macrogeneration to Syntactic Abstraction. *Higher Order Symbol. Comput.* 13, 1âĂŞ-2 (April), 57âĂŞ63. https://doi.org/10.1023/A:1010041423101

R. Kent Dybvig. 2007. Re: [r6rs-discuss] an essay on language design. Posted to `r6rs-discuss`. 24 Jul 2007. Archived at Internet Archive: https://web.archive.org/web/20070915235512/http://lists.r6rs.org/pipermail/r6rs-discuss/2007-July.txt.gz

This short note promises to correct wording that had been attacked by Clinger [2007d].

R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. 1986. Expansion-Passing style: Beyond Conventional Macros. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) (*LFP âĂŹ86*). Association for Computing Machinery, New York, NY, USA, 143âĂŞ150. https://doi.org/10.1145/319838.319858

R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. 1988. Expansion-Passing Style: A General Macro Mechanism. *LISP and Symbolic Computation* 1, 1 (June). https://doi.org/10.1007/BF01806176 Also at NON-ARCHIVAL http://www.brics.dk/~hosc/local/LaSC-1-1-pp53-75.pdf

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. 1993. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation* 5, 4 (December). https://dl.acm.org/doi/10.1007/BF01806308 Also at NON-ARCHIVAL http://www.cs.indiana.edu/~dyb/papers/syntactic.ps.gz

R. Kent Dybvig and Oscar Waddell. 2000. Portable syntax-case. Aug 2000. NON-ARCHIVAL https://www.cs.indiana.edu/syntax-case/old-psyntax.html (also at Internet Archive 14 April 2016 07:30:20).

Sebastian Erdweg, Tijs van der Storm, and Yi Dai. 2014. Capture-Avoiding and Hygienic Program Transformations. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings.* 489–514. https://doi.org/10.1007/978-3-662-44202-9_20

Marc Feeley. 2004a. The R6RS Status Report. In *Proceedings of the Fifth Workshop on Scheme and Functional Programming*. https://www.cs.indiana.edu/ftp/techreports/TR600.pdf (also at Internet Archive 5 July 2017 10:28:33).

Marc Feeley. 2004b. The Revised R6RS Status Report. In *Proceedings of the Fifth Workshop on Scheme and Functional Programming*. (Sep). https://schemers.org/Documents/Standards/Charter/2004-10-13.pdf (also at Internet Archive 27 Nov. 2018 01:12:56). There are two versions of the proceedings for that workshop, differing mainly by whether they include Feeley's original R6RS status report or this revision of that status report.

Marc Feeley. 2007. Implementors' intentions concerning R6RS. Posted to `r6rs-discuss`. 26 Oct 2007. Archived at Internet Archive: https://web.archive.org/web/20110727195419/http://lists.r6rs.org/pipermail/r6rs-discuss/2007-October/003351.html

Marc Feeley. 2010. Volunteering for Scheme language standardization working groups ***deadline January 8***. Forwarded to `r6rs-discuss` by Jonathan Rees. 5 Jan 2010. http://scheme-reports.org/mail/scheme-reports/msg02360.html (also at Internet Archive 7 March 2020 13:49:56).

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex.* MIT Press, Cambridge, MA.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. 2018. A programmable programming language. *Commun. ACM* 61, 3, 62–71. https://doi.org/10.1145/3127323

Matthew Flatt. 2002. Composable and Compilable Macros: You Want it When?. In *International Conference on Functional Programming (ICFP'2002).* https://dl.acm.org/doi/10.1145/583852.581486 Also at NON-ARCHIVAL http://www.cs.utah.edu/plt/publications/macromod.pdf

Matthew Flatt. 2016. Binding as sets of scopes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* 705–717. https://doi.org/10.1145/2837614.2837620

Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. 2012. Macros that Work Together - Compile-time bindings, partial expansion, and definition contexts. *J. Funct. Program.* 22, 2, 181–216. https://doi.org/10.1017/S0956796812000093

Matthew Flatt and Kent Dybvig. 2006. SRFI 83: R6RS Library Syntax. Sep 2006. https://srfi.schemers.org/srfi-83/ (also at Internet Archive 11 Feb. 2019 12:01:20).

Andy Freeman. 1988. RRRS-AUTHORS email with Subject: Re: Standardization. 6 Feb 1988. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Murdoch Gabbay and Andrew M. Pitts. 2002. A New Approach to Abstract Syntax with Variable Binding. *Formal Asp. Comput.* 13, 3-5, 341–363. https://doi.org/10.1007/s001650200016

Dick Gabriel. 1987. RRRS-AUTHORS email with Subject: Some Remarks on Standardization (by someone who has been there). 22 Dec 1987. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Dick Gabriel. 1988. RRRS-AUTHORS email with Subject: Clarifications. 30 Jan 1988. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Richard P. Gabriel and Kent M. Pitman. 1988. Technical Issues of Separation in Function Cells and Value Cells. *Lisp and Symbolic Computation* 1, 1 (Jun), 81–101. https://doi.org/10.1007/BF01806178 Also at NON-ARCHIVAL http://www.nhplace.com/kent/Papers/Technical-Issues.html

Steven E. Ganz, Amr Sabry, and Walid Taha. 2001. Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.* 74–85. https://doi.org/10.1145/507635.507646

Josef Martin Gasbichler. 2006. *Fully-parameterized, first-class modules with hygienic macros.* Ph.D. Dissertation. University of Tübingen, Germany. NON-ARCHIVAL http://w210.ub.uni-tuebingen.de/dbt/volltexte/2006/2423/index.html

Abdulaziz Ghuloum and R. Kent Dybvig. 2007. Implicit Phasing for R6RS Libraries. In *Proceedings of the 2007 International Conference on Functional Programming*, Vol. 42. Association for Computing Machinery, New York, NY, USA (Oct), 303–313. https://doi.org/10.1145/1291220.1291197

Abdulaziz Ghuloum and R. Kent Dybvig. (no date). Portable syntax-case. NON-ARCHIVAL http://www.cs.indiana.edu/chezscheme/r6rs-libraries (also at Internet Archive 26 April 2017 09:42:10).

Albert Gräf. 2018. The Pure Manual (subsection on Macro Hygiene). 15 April 2018. NON-ARCHIVAL https://agraef.github.io/pure-docs/pure.html#macro-hygiene (also at Internet Archive 26 Dec. 2018 19:18:05).

Paul Graham. 1993. *On Lisp: Advanced Techniques for Common Lisp.* Prentice-Hall. Text at NON-ARCHIVAL http://www.paulgraham.com/onlisp.html (also at Internet Archive 1 Feb. 2020 10:38:36).

Timothy Griffin. 1988. Notational definition-a formal account. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988.* 372–383. https://doi.org/10.1109/LICS.1988.5134

Chris Hanson. 1984. RRRS-AUTHORS email with Subject: Scheme meeting. 13 Oct 1984. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Chris Hanson. 1991. A Syntactic Closures Macro Facility. *ACM SIGPLAN Lisp Pointers* IV, 4 (December). https://dl.acm.org/doi/10.1145/1317265.1317267 Also at NON-ARCHIVAL http://www.swiss.ai.mit.edu/ftpdir/users/cph/macros/prop.ps.gz

Timothy P. Hart. 1963. *MACRO Definitions for LISP.* Technical Report 57. (Oct). https://dspace.mit.edu/handle/1721.1/6111

Chris Haynes. 1988a. RRRS-AUTHORS email with Subject: Scheme standardization meeting. 17 May 1988. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Chris Haynes. 1988b. RRRS-AUTHORS email with Subject: standardization. 5 Jan 1988. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Chris Haynes. 1990. RRRS-AUTHORS email with Subject: Minutes of the 4th Scheme standarization meeting. 31 Jan 1990. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

David Herman. 2010. *A Theory of Typed Hygienic Macros.* Ph.D. dissertation. Northeastern University, Boston, Massachusetts (May). https://dl.acm.org/doi/book/10.5555/1925552

David Herman and Mitchell Wand. 2008. A Theory of Hygienic Macros. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* 48–62. https://doi.org/10.1007/978-3-540-78739-6_4

Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. 1992. *Syntactic Abstraction in Scheme.* Technical Report TR-355. Indiana University Computer Science Department (June). ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/pubs/iucstr355.ps.gz

IEEE. 2008. IEEE 1178-1990-IEEE Standard for the Scheme Programming Language. https://standards.ieee.org/standard/1178-1990.html

Aubrey Jaffer. 2007. SRFI 96: SLIB Prerequisites. Jun 2007. https://srfi.schemers.org/srfi-96/ (also at Internet Archive 3 Aug. 2017 11:04:51).

Takashi Kato. 2014. Implementing R7RS on an R6RS Scheme system. In *Scheme and Functional Programming Workshop 2014.* Archived at Internet Archive: https://web.archive.org/web/20190829133853/http://www.schemeworkshop.org/2014/papers/Kato2014.pdf

Andrew W. Keep and R. Kent Dybvig. 2012. A sufficiently smart compiler for procedural records. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming* (Copenhagen, Denmark) (*Scheme âĂŹ12*). Association for Computing Machinery (Sep), 36âĂŞ–46. https://doi.org/10.1145/2661103.2661107

Daniel Keep. (no date). The Little Book of Rust Macros, Captures and Expansion Redux. NON-ARCHIVAL https://danielkeep.github.io/tlborm/book/mbe-min-captures-and-expansion-redux.html (also at Internet Archive 3 Nov. 2019 03:14:53).

Richard Kelsey. 1989. *Compilation by Program Transformation.* Ph.D. Dissertation. Yale University (May). https://dl.acm.org/doi/book/10.5555/916124

Richard Kelsey. 1999. SRFI 9: Defining Record Types. Sep 1999. https://srfi.schemers.org/srfi-9/ (also at Internet Archive 26 Nov. 2018 07:02:14).

Richard Kelsey, William Clinger, and Jonathan Rees. 1998. The revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices* 33, 9 (September), 26–76. https://dl.acm.org/doi/10.1145/290229.290234; Also at NON-ARCHIVAL https://bitbucket.org/cowan/r7rs/raw/4c27517de187142ad2cf4bcd8cb9199ae1e48c09/rnrs/r5rs.pdf

Richard Kelsey and Paul Hudak. 1989. Realistic compilation by program transformation (detailed summary). In *POPL '89 Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* (Jan), 281–292. https://dl.acm.org/doi/abs/10.1145/75277.75302

Oleg Kiselyov. 2001. Re-writing abstractions, or Lambda: the ultimate pattern macro. Posted to `comp.lang.functional` and `comp.lang.scheme`. December 2001. NON-ARCHIVAL http://okmij.org/ftp/Computation/rewriting-rule-lambda.txt (also at Internet Archive 8 Aug. 2019 23:59:12).

Oleg Kiselyov. 2002. How to write seemingly unhygienic and referentially opaque macros with syntax-rules. In *Workshop on Scheme and Functional Programming (2002).* (October). NON-ARCHIVAL http://okmij.org/ftp/Scheme/Dirty-Macros.pdf (also at Internet Archive 12 Jan. 2020 07:40:53). See also NON-ARCHIVAL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.3405

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language.* No Starch Press (Jun). NON-ARCHIVAL https://doc.rust-lang.org/book/ (also at Internet Archive 29 Feb. 2020 15:32:06).

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming.* https://dl.acm.org/doi/10.1145/319838.319859 Also at https://help.luddy.indiana.edu/techreports/TRNNN.cgi?trnum=TR194

Eugene E. Kohlbecker and Mitchell Wand. 1987. Macro-by-example: Deriving syntactic transformations from their specifications. In *Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of programming languages.* (January). https://dl.acm.org/doi/10.1145/41625.41632 Also at https://help.luddy.indiana.edu/techreports/TRNNN.cgi?trnum=TR206

Eugene E. Kohlbecker, Jr. 1986. *Syntactic Extensions in the Programming Language Lisp.* Ph.D. Dissertation. Indiana University. https://help.luddy.indiana.edu/techreports/TRNNN.cgi?trnum=TR199 Archived at Internet Archive: https://web.archive.org/web/20130522065120/http://repository.readscheme.org/ftp/papers/kohlbecker_phdthesis.pdf

David Kranz. 1988. *ORBIT: an optimizing compiler for Scheme.* Ph.D. Dissertation. Yale University (Feb). https://cpsc.yale.edu/sites/default/files/files/tr632.pdf

Shriram Krishnamurthi. 1996. RRRS-AUTHORS email with Subject: Generative record types. 23 Apr 1996. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Shriram Krishnamurthi. 2001. *Linguistic Reuse.* Ph.D. Dissertation. Rice University. https://scholarship.rice.edu/handle/1911/17993

Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. 2000a. *From Macros to Reusable Generative Programming.* Technical Report TR 00-364. http://www.ccs.neu.edu/scheme/pubs/tr00-364.ps.gz (also at Internet Archive 10 Dec. 2005 14:42:23).

Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. 2000b. From Macros to Reusable Generative Programming. In *Generative and Component-Based Software Engineering. GCSE 1999. Lecture Notes in Computer Science*, Vol. 1799. 105–120. https://doi.org/10.1007/3-540-40048-6_9

B. M. Leavenworth. 1966. Syntax macros and extended translation. *Commun. ACM* 9, 11 (Nov), 790–793. https://dl.acm.org/doi/10.1145/365876.365879

Juergen Lorenz. 2009-2020. Mini-tutorial on explicit (and implicit) renaming macros in CHICKEN. NON-ARCHIVAL https://wiki.call-cc.org/explicit-renaming-macros#implicit-renaming-macros Archived at https://web.archive.org/web/20191026053510/https://wiki.call-cc.org/explicit-renaming-macros

M. Donald MacLaren. 1969. Macro processing in EPS. *SIGPLAN* 4, 8 (Aug), 32–36. https://dl.acm.org/doi/abs/10.1145/1115858.1115866

Masinter, van Roggen, and Barrett. 1990. Issue LISP-SYMBOL-REDEFINITION Writeup. Jun 1990. http://clhs.lisp.se/Issues/iss214_w.htm (also at Internet Archive 6 Sept. 2015 05:31:59).

John McCarthy. 1978. History of Lisp. In *History of Programming Languages.* ACM (June), 173–185. https://dl.acm.org/doi/10.1145/960118.808387

John McCarthy. 1981. History of LISP. In *History of Programming Languages*, Richard L Wexelblat (Ed.). Academic Press, 173–197.

John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. 1962. *Lisp 1.5 Programmer's Manual.* MIT Press.

James Miller. 1988. RRRS-AUTHORS email with Subject: Standards. 28 Feb 1988. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

John C. Mitchell. 2002. *Concepts in Programming Languages.* Cambridge University Press.

Joel Moses. 1970. The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. *ACM SIGSAM Bulletin* 15 (July), 13–27. http://dspace.mit.edu/handle/1721.1/5854

Fabian Muehlboeck. 2013. *Checking binding hygiene statically.* Master's thesis. Northeastern University, Boston, MA. http://hdl.handle.net/2047/d20003134

Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

Marc Nieper-Wißkirchen. 2016. SRFI 136: Extensible record types. Feb 2016. https://srfi.schemers.org/srfi-136/ (also at Internet Archive 3 Aug. 2017 10:45:33).

Marc Nieper-Wißkirchen. 2017. SRFI 148: Eager `syntax-rules`. Aug 2017. https://srfi.schemers.org/srfi-148/ (also at Internet Archive 2 Aug. 2017 22:33:00).

Marc Nieper-Wißkirchen. 2018. SRFI 150: Hygienic ERR5RS Record Syntax (reduced). Jan 2018. https://srfi.schemers.org/srfi-150/ (also at Internet Archive 3 Aug. 2017 10:47:29).

David M. R. Park, Daniel P. Friedman, David S. Wise, and Guy L Steele, Jr (Eds.). 1982. *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming, LFP 1980, August 15-18, 1982, Pittsburgh, PA, USA*. ACM. https://dl.acm.org/citation.cfm?id=800068

Kent Pitman. 1983. *The Revised MacLisp Manual* (the PitMANUAL, draft #14 ed.). MIT. http://www.maclisp.info/pitmanual/ (also at Internet Archive 18 Feb. 2008 22:55:59).

Justin Pombrio, Shriram Krishnamurthi, and Mitchell Wand. 2017. Inferring scope through syntactic sugar. *PACMPL* 1, ICFP, 44:1–44:28. https://doi.org/10.1145/3110288

Francois Pottier. 2007. Static Name Control for FreshML. *LICS*. https://doi.org/10.1109/LICS.2007.44 Long version at NON-ARCHIVAL http://gallium.inria.fr/~fpottier/publis/fpottier-pure-freshml-long.pdf

Vaughan R. Pratt. 1973. Top Down Operator Precedence. In *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, Patrick C. Fischer and Jeffrey D. Ullman (Eds.). ACM Press, 41–51. https://doi.org/10.1145/512927.512931

Christian Queinnec and Julian Padget. 1991a. Modules, macros and Lisp. In *Computer Science Research and Applications*, Ricardo Baeza-Yates and Udi Manber (Eds.). Plenum Publishing Corporation, USA United States (1 10), 109–122. https://doi.org/10.1007/978-1-4613-6513-6

Christian Queinnec and Julian Padget. 1991b. A proposal for a modular Lisp with macros and dynamic evaluation. In *Journées de Travail sur l'Analyse Statique en Programmation Équationnelle, Fonctionnelle et Logique*. BIGRE: Bulletin d'information du Groupe de recherche sur les outils de conception et d'ÃĺcritureSdes systÃĺmes opÃĺratoires (1 10), 1–8. https://researchportal.bath.ac.uk/en/publications/a-proposal-for-a-modular-lisp-with-macros-and-dynamic-evaluation

Jon Rafkind and Matthew Flatt. 2012. Honu: syntactic extension for algebraic notation through enforestation. In *GPCE '12 Proceedings of the 11th International Conference on Generative Programming and Component Engineering* (Dresden, Germany) (*GPCE âĂŹ12*). Association for Computing Machinery (Sep), 122âĂŞ131. https://doi.org/10.1145/2371401.2371420

John D. Ramsdell. 1987. RRRS-AUTHORS email with Subject: A vote against standardization. 24 Dec 1987. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Jonathan Rees. 1993. Implementing lexically scoped macros. *ACM SIGPLAN Lisp Pointers*. https://dl.acm.org/doi/10.1145/173770.173774; Also at NON-ARCHIVAL http://mumble.net/~jar/pubs/scheme-of-things/easy-macros.ps

Jonathan Rees. (no date). The T Project. NON-ARCHIVAL http://mumble.net/~jar/tproject/ (also at Internet Archive 28 Nov. 2019 02:31:52).

Jonathan Rees and William Clinger. 1986. The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices* 21, 12 (December), 37–79. https://dl.acm.org/doi/10.1145/15042.15043; Also at NON-ARCHIVAL https://bitbucket.org/cowan/r7rs/raw/4c27517de187142ad2cf4bcd8cb9199ae1e48c09/rnrs/r3rs.pdf

Jonathan Rees and William Clinger. 2013a. Scheme Working Group 1. NON-ARCHIVAL http://www.scheme-reports.org/2015/working-group-1.html (also at Internet Archive 3 July 2019 11:41:55).

Jonathan Rees and William Clinger. 2013b. Scheme Working Group 2. NON-ARCHIVAL http://www.scheme-reports.org/2015/working-group-2.html (also at Internet Archive 31 Dec. 2019 22:51:39).

Jonathan Rees, Olin Shivers, William Clinger, Marc Feeley, and Chris Hanson. 2010a. Charter for working group 1. 20 Jan 2010. NON-ARCHIVAL http://www.scheme-reports.org/2009/working-group-1-charter.html (also at Internet Archive 17 April 2019 12:12:40).

Jonathan Rees, Olin Shivers, William Clinger, Marc Feeley, and Chris Hanson. 2010b. Charter for working group 2. 3 Mar 2010. NON-ARCHIVAL http://www.scheme-reports.org/2009/working-group-2-charter.html (also at Internet Archive 26 Oct. 2019 05:20:46).

Jonathan A Rees. 1987a. RRRS-AUTHORS email with Subject: a modest macro proposal. 28 Mar 1987. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Jonathan A Rees. 1987b. RRRS-AUTHORS email with Subject: macros. 7 Apr 1987. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Jonathan A Rees. 1987c. RRRS-AUTHORS email with Subject: macros. 10 Apr 1987. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Jonathan A Rees. 1988a. RRRS-AUTHORS email with Subject: DO in Scheme. 16 Feb 1988. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Jonathan A Rees. 1988b. RRRS-AUTHORS email with Subject: opaque type proposal. 26 May 1988. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Jonathan A. Rees and Norman I. Adams, IV. 1982. T: a dialect of Lisp or, Lambda: the ultimate software tool. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh, Pennsylvania, USA) (*LFP âĂŹ82*). Association for Computing Machinery (Aug), 114–122. https://doi.org/10.1145/800068.802142

Guillermo J. Rozas. 1996. RRRS-AUTHORS email with Subject: Re: Generative record types. 24 Apr 1996. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

ISO SC22/WG16. 1997. Programming Language ISLISP: ISLISP Working Draft 20.3 (Public Domain). http://islisp.info/Documents/PDF/islisp-1997-03-31-pd-v20.pdf

Scala. (no date). The Scala Programming Language. NON-ARCHIVAL https://www.scala-lang.org/ (also at Internet Archive 30 June 2019 22:36:28).

Alex Shinn. 2011. initial results of implementor intention poll. Posted to `scheme-reports`. 23 Oct 2011. NON-ARCHIVAL http://www.scheme-reports.org/mail/scheme-reports/msg00372.html (also at Internet Archive 9 March 2020 17:08:27).

Alex Shinn. 2017. Re: apparent bug in sample implementation of SRFI 148. Posted to SRFI 148 discussion list. 20 Jul 2017. NON-ARCHIVAL https://srfi-email.schemers.org/srfi-148/msg/6102883/ (also at Internet Archive 7 March 2020 18:27:32).

Alex Shinn, John Cowan, and Arthur A Gleckler. 2013. Revised^7 report on the algorithmic language Scheme. July 2013. NON-ARCHIVAL https://bitbucket.org/cowan/r7rs/raw/4c27517de187142ad2cf4bcd8cb9199ae1e48c09/rnrs/r7rs-official.pdf (also at Internet Archive 31 Dec. 2019 22:51:38).

Olin Shivers. (no date). History of T. NON-ARCHIVAL http://www.paulgraham.com/thist.html (also at Internet Archive 23 Feb. 2020 06:29:59).

Olin Shivers, William Clinger, Marc Feeley, Chris Hanson, and Jonathan Rees. 2009. Scheme Steering Committee Position Statement. 20 Aug 2009. NON-ARCHIVAL http://scheme-reports.org/2009/position-statement.html (also at Internet Archive 21 July 2019 21:27:35).

Michael Sperber. 2012. Formal Response #456: Adoption of R6RS. Posted to `scheme-reports`. 13 Oct 2012. NON-ARCHIVAL http://www.scheme-reports.org/mail/scheme-reports/msg00945.html (also at Internet Archive 9 March 2020 17:32:16).

Michael Sperber, William Clinger, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. 2007. Revised$^{5.97}$ report on the algorithmic language Scheme—Standard Libraries. 30 Jun 2007. NON-ARCHIVAL http://www.r6rs.org/versions/r5.97rs-lib.pdf (also at Internet Archive 28 Sept. 2018 12:20:57). This draft was ratified with only minor changes.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. 2009a. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming* 19, S1 (August), 1–301. http://www.r6rs.org/final/r6rs.pdf (also at Internet Archive 29 Dec. 2019 12:53:33). HTML version at NON-ARCHIVAL http://www.r6rs.org/

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. 2009b. Revised^6 report on the algorithmic language Scheme—Standard Libraries. *Journal of Functional Programming* 19, S1 (August), 1–301. NON-ARCHIVAL http://www.r6rs.org/final/r6rs-lib.pdf (also at Internet Archive 29 Dec. 2019 12:53:33). HTML version at NON-ARCHIVAL http://www.r6rs.org/

Paul Stansifer. 2016. *Flexible binding-safe programming*. Ph.D. Dissertation. Northeastern University. https://repository.library.northeastern.edu/files/neu:cj82mb52h

Paul Stansifer and Mitchell Wand. 2014. Romeo: a system for more flexible binding-safe programming. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. 53–65. https://doi.org/10.1145/2628136.2628162

Paul Stansifer and Mitchell Wand. 2016. Romeo: A system for more flexible binding-safe programming. *J. Funct. Program.* 26, e13. https://doi.org/10.1017/S0956796816000137

Paul Steckler and Mitchell Wand. 1994. Selective Thunkification. In *International Static Analysis Symposium*. 162–178. https://doi.org/10.1007/3-540-58485-4_39

Guy Lewis Steele, Jr. 1977. *RABBIT: a compiler for Scheme.* Master's thesis. MIT (May). Published as MIT AI Memo 474 [Steele 1978].

Guy Lewis Steele, Jr. 1978. *RABBIT: a compiler for Scheme.* Technical Report 474. (May). https://dspace.mit.edu/handle/1721.1/6913. Also at ftp://publications.ai.mit.edu/ai-publications/pdf/AITR-474.pdf. MIT AI Memo 474.

Guy L. Steele, Jr. 1982. An Overview of Common Lisp. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh, Pennsylvania, USA) (*LFP âĂŹ82*). Association for Computing Machinery (Aug), 98–107. https://doi.org/10.1145/800068.802140

Guy L. Steele, Jr. 1990. *Common Lisp the Language, 2nd Edition.* Digital Press. https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/clm.html

Guy L. Steele, Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb. 1984. *Common Lisp the Language.* Digital Press.

Guy L. Steele, Jr. and Richard P. Gabriel. 1993a. The evolution of Lisp. In *HOPL-II The second ACM SIGPLAN conference on History of programming languages.* ACM (April), 231–270. https://dl.acm.org/doi/10.1145/155360.155373

Guy L. Steele, Jr. and Richard P. Gabriel. 1993b. The evolution of Lisp. NON-ARCHIVAL https://www.dreamsongs.com/Files/HOPL2-Uncut.pdf (also at Internet Archive 20 Dec. 2019 13:43:27). "Uncut" draft of Steele and Gabriel [1993a].

Guy Lewis Steele, Jr. and Gerald Jay Sussman. 1978. The revised report on Scheme, a dialect of Lisp. *MIT AI Memo 452* (January). https://dspace.mit.edu/handle/1721.1/6283

Gerald Jay Sussman and Guy Lewis Steele, Jr. 1975. *Scheme: an interpreter for extended lambda calculus.* Technical Report 349. (Dec). https://dspace.mit.edu/handle/1721.1/5794 MIT AI Memo 349. See also the journal version [Sussman and Steele 1998].

Gerald Jay Sussman and Guy Lewis Steele, Jr. 1998. Scheme: an interpreter for extended lambda calculus, Vol. 11. (Dec), *Higher-Order and Symbolic Computation* 11. https://doi.org/10.1023/A:1010035624696 Journal version of MIT AI Memo 349 [Sussman and Steele 1975].

Walid Taha and Patricia Johann. 2003. Staged Notational Definitions. In *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*. 97–116. https://doi.org/10.1007/978-3-540-39815-8_6

Warren Teitelman. 1974. *INTERLISP Reference Manual.* Xerox PARC and BBN. http://www.softwarepreservation.org/projects/LISP/interlisp/Interlisp-Oct_1974.pdf/view

André van Tonder. 2005a. SRFI 57: Records. Mar 2005. https://srfi.schemers.org/srfi-57/ (also at Internet Archive 3 Aug. 2017 01:24:35).

André van Tonder. 2005b. SRFI 72: Hygienic Macros. Sep 2005. https://srfi.schemers.org/srfi-72/ (also at Internet Archive 3 Aug. 2017 13:09:43).

André van Tonder. 2007a. `r6rs-discuss` email of 26 Jun 2007. Subject: Rationale issues. 26 Jun 2007. NON-ARCHIVAL http://lists.r6rs.org/pipermail/r6rs-discuss/2007-June/002825.html (also at Internet Archive 5 July 2008 18:17:28).

André van Tonder. 2007b. Formal comment #276: Rationale 15.1. 27 Jun 2007. NON-ARCHIVAL http://www.r6rs.org/formal-comments/comment-276.txt (also at Internet Archive 5 July 2008 20:20:05).

André van Tonder. (no date). R6RS Libraries and Macros. NON-ARCHIVAL http://www.het.brown.edu/people/andre/macros/ (also at Internet Archive 26 Sept. 2019 06:39:25).

R. Mark Volkmann. 2009. Clojure—Functional Programming for the JVM. March 2009. https://objectcomputing.com/resources/publications/sett/march-2009-clojure-functional-programming-for-the-jvm (also at Internet Archive 7 March 2020 20:42:35).

Oscar Waddell. 1999. *Extending the Scope of Syntactic Abstraction.* Ph.D. Dissertation. Indiana University Computer Science Department (August). NON-ARCHIVAL http://www.cs.indiana.edu/~owaddell/papers/thesis.ps.gz

Oscar Waddell and R. Kent Dybvig. 1999. Extending the Scope of Syntactic Abstraction. In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* (January). https://dl.acm.org/doi/10.1145/292540.292559. Also at NON-ARCHIVAL http://www.cs.indiana.edu/~dyb/papers/popl99.ps.gz

Ken Wakita, Kanako Homizu, and Akira Sasaki. 2014. Hygienic Macro System for JavaScript and Its Light-Weight Implementation Framework. In *Proceedings of ILC 2014 on 8th International Lisp Conference* (Montreal, QC, Canada) (*ILC âĂŹ14*). Association for Computing Machinery, 12âĂŞ21. https://doi.org/10.1145/2635648.2635653

ExJS, our prototype implementation of a hygienic macro system for JavaScript, is implemented in less than 2,000 lines of JavaScript and Scheme, and exhibits its flexible syntactic extensibility.

Mitchell Wand. 1984a. RRRS-AUTHORS email with Subject: Revised invitation list. 16 Oct 1984. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Mitchell Wand. 1984b. RRRS-AUTHORS email with Subject: Scheme meeting. 10 Oct 1984. Archived at Internet Archive: https://web.archive.org/web/20100413124637/https://groups.csail.mit.edu/mac/ftpdir/scheme-mail/

Gerald M. Weinberg. 1985. *The Psychology of Computer Programming.* John Wiley and Sons.

Daniel Weinreb and David Moon. 1981. *Lisp Machine Manual* (third ed.). MIT. Second edition at http://www.softwarepreservation.org/projects/LISP/MIT/Weinreb_Moon-Lisp_Machine_Manual_Jan_1979.pdf/view

Joseph Weizenbaum. 1968. The Funarg Problem Explained. http://www.softwarepreservation.org/projects/LISP/MIT/Weizenbaum-FUNARG_Problem_Explained-1968.pdf

NON-ARCHIVAL REFERENCES

David Bartley. 1984a. Sep 1984. Email of 14 September 1984, sent to members of the agenda committee.

David Bartley. 1984b. TI-CSL Position on "Standardizing" SCHEME. Oct 1984. Email of 17 October 1984.

Matthias Blume. 1995. *Refining Hygienic Macros for Modules and Separate Compilation.* Technical Report ATR Technical Report TR-H-171. http://ttic.uchicago.edu/~blume/papers/hygmac.pdf

John Clements. 2020. Personal communication.

William Clinger and Felix S Klock, II. 2007. ANN: Larceny v0.95 "First Safety". Posted to the `comp.lang.scheme` newsgroup. 8 Nov 2007. NON-ARCHIVAL https://groups.google.com/d/msg/comp.lang.scheme/fUVX1SYurOQ/MHq4-6DFofoJ

William D Clinger. 1984a. Aug 1984. A long note, dated 28 August 1984, sent to the six members of the agenda committee before the 1984 meeting at Brandeis, listing known differences between T, Scheme 84, Scheme 312, and MIT Scheme.

William D Clinger. 1984b. Sep 1984. Email of September 1984, responding to [NA Bartley 1984a].

William D Clinger. 1984c. Oct 1984. Agenda for the October 1984 meeting at Brandeis.

William D Clinger. 1984d. Oct 1984. Email to Mitchell Wand on 27 Oct 1984 listing answers to questions considered at the workshop.

William D Clinger. 1998. my notes from the Scheme workshop at ICFP98. Posted to `comp.lang.scheme`. 19 Oct 1998.

John Cowan. 2019. Tangerine Edition final results available. Posted to `scheme-reports-wg1`. 2 Feb 2019. NON-ARCHIVAL https://groups.google.com/d/msg/scheme-reports-wg1/Vk8oqyUoLHg/pWW14lcuGAAJ

John Cowan. (no date). ColorDockets. NON-ARCHIVAL https://bitbucket.org/cowan/r7rs-wg1-infra/

John Cowan, Will Clinger, Marc Nieper-Wißkirchen, and Alex Shinn. 2016. R7RS-large discussion: Miscellaneous. Posted to `scheme-reports-wg2` (Google Groups). Jun 2016. NON-ARCHIVAL https://groups.google.com/d/topic/scheme-reports-wg2/oKuhgwaM45w/discussion

This thread was dominated by the controversy described in Section 14.1.

Kent Dybvig. (no date). (chez (chez scheme)). NON-ARCHIVAL https://www.scheme.com/

Justin Ethier. 2013. 10 Sep 2013. NON-ARCHIVAL https://stackoverflow.com/questions/18713254/is-there-a-digest-for-different-rnrs-scheme-standards Answer given at Stack Overflow on 10 Sep 2013 to the question "Is there a digest for different 'RnRS' Scheme standards?".

Marc Feeley. (no date). GitHub repository for Gambit. NON-ARCHIVAL https://github.com/gambit/gambit

Carol Fessenden, William Clinger, Daniel P Friedman, and Christopher Haynes. 1983. *Scheme 311 version 4 reference manual.* Technical Report 137. Indiana University Computer Science Department (Feb).

John K. Foderaro. 1980. The Franz Lisp Manual: a document in four movements.

Daniel P. Friedman, Christopher Haynes, Eugene Kohlbecker, and Mitchell Wand. 1984. *Scheme 84 Reference Manual Version 0.* Technical Report 153. (Feb). This version of TR 153 was superseded almost a year later by an "interim" manual that was also published as TR 153.

Daniel P. Friedman, Christopher Haynes, Eugene Kohlbecker, and Mitchell Wand. 1985. *Scheme 84 Interim Reference Manual.* Technical Report 153. (Jan). This version of TR 153 was published almost a year after the previous version of TR 153, Version 0.

Yoshikatsu Fujita. (no date). Ypsilon. NON-ARCHIVAL https://code.google.com/archive/p/ypsilon/

Implements much of the R6RS standard for Scheme.

Abdulaziz Ghuloum. 2007. [ANN] Initial release of Ikarus—the compiler of choice for R6RS hackers. Posted to the `comp.lang.scheme` newsgroup. 31 Oct 2007. NON-ARCHIVAL https://groups.google.com/d/msg/comp.lang.scheme/TrR5TmBUDAo/sknYDNQlVu8J

Lars Thomas Hansen. 1992. *The Impact of Programming Style on the Performance of Scheme Programs.* Master's thesis. University of Oregon (Aug).

Chris Hanson. 1984. Oct 1984. Email of 4 October 1984, responding to [NA Bartley 1984a] and [NA Clinger 1984b].

Chris Hanson. 1986-2019. MIT/GNU Scheme. NON-ARCHIVAL https://www.gnu.org/software/mit-scheme/

Yutaka Hara. (no date). Biwa Scheme. NON-ARCHIVAL https://github.com/biwascheme/biwascheme/

Implements part of the R6RS standard for Scheme.

Rich Hickey. (no date). The Clojure Programming Language. NON-ARCHIVAL https://clojure.org

Secretariat ISO/JTC1/SC22. 1988. Draft Report of the first meeting of SC22/WG16-Lisp held in Paris France on 1988-02-24/25. Apr 1988. SC22 N494, cross referencing SC22 N453.

Neil Jerram, Marius Vollmer, Martin Grabmueller, Ludovic Courtès, and Andy Wingo. (no date). Guile Manual §7.6: R6RS Support. NON-ARCHIVAL https://www.gnu.org/software/guile/manual/html_node/R6RS-Support.html

Takashi Kato. (no date). sagittarius-scheme. NON-ARCHIVAL https://bitbucket.org/ktakashi/sagittarius-scheme/wiki/Home

Eugene Kohlbecker. 1984. Position Statement on Macros. Dated 19 October 1984, marked DRAFT, and apparently never published or posted to any mailing list.

Various Larcenists. (no date). Larceny. Web site at NON-ARCHIVAL http://larcenists.org/

Steve Lawler. 2004. R4Rs [*sic*] vs. R5RS. Posted to a newsgroup. 2 Jun 2004. NON-ARCHIVAL http://computer-programming-forum.com/40-scheme/e431f99fab9fd6fe.htm

As far as I can gather, the big differences b/n R4 and R5 are the addition of hygienic macros…

Ben Lerner, Joe Gibbs Politz, Daniel Patterson, and Dorai Sitaram. (no date). NON-ARCHIVAL pyret.org

Pyret is a programming language designed to serve as an outstanding choice for programming education while exploring the confluence of scripting and functional programming.

Lightship Software. 1985-1990. *MacScheme*. Lightship Software.

Marco Maggi. 2016a. Hard choices, R6RS, harder to write code. 5 Dec 2016. NON-ARCHIVAL http://marcomaggi.github.io/weblog/weblog-2016/Dec-05.html#Dec-05

These changes will make some fully compatible r6rs code not to compile anymore or raise runâĂŞtime errors. Life is hard.

Marco Maggi. 2016b. On naming Vicare. 7 Dec 2016. NON-ARCHIVAL http://marcomaggi.github.io/weblog/weblog-2016/Dec-07.html#Dec-07

But the future of Vicare lies with the typed language, which has incompatibilities with the r6rs language that was the official one in Vicare.

Marco Maggi. (no date). Vicare Scheme. NON-ARCHIVAL http://marcomaggi.github.io/vicare.html

MIT. 1981. MIT Scheme, Version 1.

Ronald B. Ohlander. 1984. Workshop on Common Lisp. Invitation mailed to 'key language designers, hardware manufacturers, and research groups'. 16 Aug 1984.

Al Petrofsky. 1991. How to write seemingly unhygienic macros using syntax-rules. Posted to the `comp.lang.scheme` newsgroup. 19 Nov 1991. https://groups.google.com/d/msg/comp.lang.scheme/wyYJ5PwSxSM/cZ9Lrj3ROFQJ

Al Petrofsky. 1992. Holey macros! (was Re: choice for embedding Scheme implementation?). Posted to the `comp.lang.scheme` newsgroup. 22 May 1992. NON-ARCHIVAL https://groups.google.com/d/msg/comp.lang.scheme/KM3P9QAOsqQ/kWiOThjr7IwJ

Kent Pitman. 1984. Oct 1984. Email of 16 October 1984, listing questions to be resolved at the Brandeis meeting.

Llewellyn Pritchard. (no date). IronScheme. NON-ARCHIVAL https://archive.codeplex.com/?p=IronScheme

Implements part of the R6RS standard for Scheme.

Christian Queinnec and Julian Padget. 1990. *A deterministic model for modules and macros*. Technical Report 90-36. http://pagesperso-systeme.lip6.fr/Christian.Queinnec/Papers/modmac2.ps.gz

Jonathan Rees. 1989. *Modular Macros*. Master's thesis. MIT.

Jonathan A. Rees, Norman I. Adams, IV, and James R Meehan. 1984. *The T Manual* (4th ed.). Technical Report. Yale University Computer Science (Jan).

Brian Reistad. 1992. Macros That Work in Modula-2. 5 Oct 1992. Undergraduate research project at the University of Oregon.

Grant Rettke. 2008a. Re: Was there a schism between R4RS and R5RS? Posted to `comp.lang.scheme`. 9 Oct 2008. NON-ARCHIVAL https://groups.google.com/d/msg/comp.lang.scheme/KXVg8WRDI-M/fdEdFzQW_KQJ

Timeline of macro research from R4RS to R5RS.

Grant Rettke. 2008b. Was there a schism between R4RS and R5RS? Posted to `comp.lang.scheme`. 6 Oct 2008. NON-ARCHIVAL https://groups.google.com/d/msg/comp.lang.scheme/KXVg8WRDI-M/KLIGBcFpzFAJ

R4RS only suggested macros, though, and didn't include them in the standard. Why then, did they get added to the standard in R5RS?…macros seem like a "big" addition.

Alex Shinn. 2010. welcome to working group 1. Posted to `scheme-reports-wg1`. 9 Feb 2010. NON-ARCHIVAL https://groups.google.com/d/msg/scheme-reports-wg1/Dh6avGU4rqI/F7Xw_qkpa4wJ

Alex Shinn. (no date). chibi-scheme. NON-ARCHIVAL http://synthcode.com/wiki/chibi-scheme

Jens Axel Soegaard. 2008. Re: Was there a schism between R4RS and R5RS? Posted to `comp.lang.scheme`. 6 Oct 2008. NON-ARCHIVAL https://groups.google.com/d/msg/comp.lang.scheme/KXVg8WRDI-M/EYJh18Q3erkJ

At the time R4RS was written there were quite a few persons doing research in macros. Instead of choosing one low-level macro mechanism and fearing the choice had to be remade later, one simply omitted choosing a low-level macro mechanism.

Paul Stansifer. 2016. GitHub comment. 24 May 2016. NON-ARCHIVAL https://github.com/rust-lang/rfcs/pull/1561 Gives an example of a macro-defining macro in Rust.

Sweet. (no date)a. Build your dream language. NON-ARCHIVAL https://www.sweetjs.org/

Sweet. (no date)b. `sweet.js` tutorial. NON-ARCHIVAL https://www.sweetjs.org/doc/tutorial.html

Texas Instruments. 1987. PC Scheme source code.

Texas Instruments. 1990. *PC Scheme User Guide and Software* (trade edition ed.). MIT Press.