SYNTACTIC EXTENSIONS

IN THE

PROGRAMMING LANGUAGE LISP

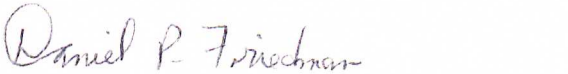
Eugene Edmund Kohlbecker, Jr.

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
of the degree
Doctor of Philosophy
in the Department of Computer Science,
Indiana University

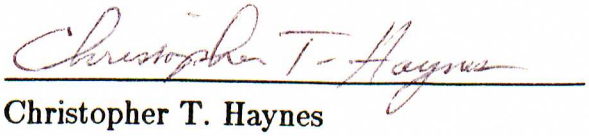
September 1986

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree Doctor of Philosophy.

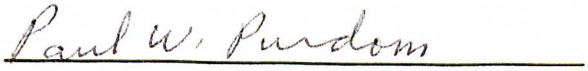
Doctoral Committee:



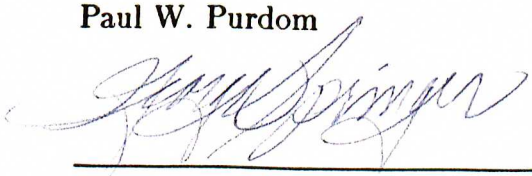
Daniel P. Friedman, Ph.D.



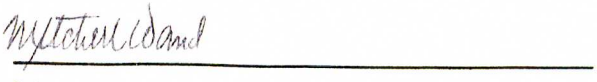
Christopher T. Haynes



Paul W. Purdom



George Springer



Mitchell Wand

21 June 1986

To my mother Billie

and my sister Karel

in memory of my father.

Acknowledgements

I have been successful in this effort through the grace of God.

Many people are agents of this grace. Foremost is my advisor and friend Dan Friedman. His enthusiasm and encouragement have been invaluable. I thank him for his guidance and support; I am honored to be his student. He has always been there for me, ready to help in many ways.

I appreciate the efforts of the other members of my advisory committee: Chris Haynes, Paul Purdom, George Springer, and, especially, Mitch Wand. Through the four years that I have been working on syntactic extensions in Scheme, they have been with me all along, suggesting new features, debating the merits of existing ones, and helping hone my macro tools to their present state. Mitch was one of the first to recognize the relevance of my work on the capturing problem. His suggestions and criticisms have greatly improved my work.

Other friends have been involved in the development of this work. I owe thanks to Bruce Duba, Kent Dybvig, Matthias Felleisen, and John Gateley. Each of these has been an important person off whom to bounce ideas. I also thank Matthias and Jeff Perotti for proof-reading this dissertation.

I have been extremely fortunate to have had a large user community in the form of the introductory and advanced programming languages courses at Indiana University. In particular, to those students taking the advanced course in Fall 1984, when an early implementation of the hygienic expansion algorithm was being debugged, I say thanks for putting up with it all.

During my research I have been supported financially by the Computer Science Department at I.U., the National Science Foundation (under grant MCS 83-04567), Texas Instruments, and the IBM Corporation. I gratefully acknowledge the role each has played in the pursuit of my degree.

Abstract

The traditional macro processing systems used in Lisp-family languages have a number of shortcomings. We identify five problems with the declaration tools customarily available to programmers. First, the declarations themselves are hard to read and write. Second, the declarations provide little explicit information about the form macro calls are to take. Third, syntactic checking of macro calls is usually ignored. Fourth, the notion of a macro binding for an identifier gives rise to a poor understanding of what macros really should be. Fifth, the unrestricted capabilities of the language used to declare macros cause some to take advantage of macros in ways inconsistent with their role as textual abstractions. 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.

We examine several existing macro processors, both inside and outside of the Lisp-family. We then enumerate a set of design principles for macro processing systems. These principles are general enough that they apply to the organization of macro processing systems for a large number of high-level languages. Taking our principles as guidelines, we design a new macro processing system for Lisp. The new macro declaration tool addresses each of the five problems from which the traditional tools suffer. A description of the use of our tool and an annotated presentation of its implementation are provided. We also develop a new macro expansion algorithm that eliminates the capturing problem. The macro expander has the responsibility for avoiding the unwanted capture of identifiers appearing within macro calls.

Table of Contents

1. Introduction 1
   1.1. The Need for Textual Abstraction 3
   1.2. Basic Definitions 7
   1.3. The Problems with Syntactic Extensions 8
      1.3.1. Declaring Syntactic Extensions 9
      1.3.2. Expanding Syntactic Extensions 14
   1.4. The Organization of this Dissertation 17
2. Related Work 20
   2.1. A Survey of Textual Abstraction Systems 22
      2.1.1. General-purpose macro systems 22
      2.1.2. Special-purpose macro systems 26
      2.1.3. The C Preprocessor 29
      2.1.4. Common Lisp 29
   2.2. Textual and Syntactic Abstractions 33
   2.3. Macro Extensions to Languages in the Algol Family 33
      2.3.1. McIlroy's proposals 33
      2.3.2. Pre run-time macros 36
      2.3.3. Leavenworth's syntax-macros 38
      2.3.4. Further developments in Algol-family languages 39
   2.4. A Pattern Matching Macro Processor 41
   2.5. Design Principles 43
3. Syntactic Extensions in Lisp 44
   3.1. FEXPRs 44
   3.2. Macros 47
   3.3. The Use of Macros in Lisp 57
      3.3.1. Abbreviating function calls 58
      3.3.2. Supplying arbitrary numbers of arguments 58
      3.3.3. Controlling the order of evaluation of expressions 59
      3.3.4. Using syntactic forms that are not expressions 60

3.3.5. Performing cosmetics 60
3.3.6. Aliasing 61
3.3.7. Changing the semantics of special forms 62
3.3.8. Hiding implementation details 63
3.3.9. Avoiding the duplication of text 63
3.3.10. Simulating dynamic variables 64
3.3.11. Supporting a philosophical view of program structure 65
4. Design Principles 67
4.1. What the Macro Writer Does 67
4.2. Syntactic Extensions 69
4.3. Syntactic Domains 70
4.4. The Core Language and Syntactic Extensions 72
4.5. The Syntax Table 74
4.5.1. Reserved Words 74
4.5.2. The Recognition of Macro Calls 77
4.6. Macro Languages 78
4.7. When to Expand 80
4.7.1. Top-level calls 80
4.7.2. Calls occurring within other macro calls 82
4.7.3. Isolated versus mixed expansion 84
4.8. The Composition of STFs 84
4.9. What the Macro Processor Does 86
4.10. Triance and Layzell's Design Principles 87
4.11. A Summary of Our Design Decisions 90
5. Declaring Syntactic Extensions 92
5.1. The `extend-syntax` Manual 92
5.1.1. Production rules 95
5.1.2. New variables 101
5.1.3. Repeated symbols 106
5.1.4. Nested `extend-syntax` calls 107
5.1.5. Related special forms 109
5.2. A Formal Semantics of Ellipsis 111
5.3. An Implementation of a Syntax Table 119
5.4. The Implementation of `extend-syntax` 120
5.4.1. The principal function 128
5.4.2. The generation of pattern matchers 143
5.4.3. Partial processing 145
5.4.4. Concluding remarks about the implementation 145

6. Expanding Syntactic Extensions 148
6.1. Naïve Macro Expansion 148
6.1.1. A formal description of the naïve expansion algorithm 150
6.1.2. Some ways of avoiding the capturing problem 150
6.2. Hygienic Macro Expansion 152
6.3. A Formal Description of the Hygienic Expansion Algorithm 157
6.4. The Modified Hygienic Expansion Algorithm 165
6.5. Enlarging the Host Language 170
6.6. Scheme Implementation Details 172
7. Conclusion 175
7.1. The Ramifications of `extend-syntax` 175
7.2. The Ramifications of Hygienic Macro Expansion 177
7.2.1. The argument "The default is not backwards." 178
7.2.2. The argument "The algorithm is too complicated." 179
7.2.3. The argument "The algorithm is too expensive." 180
7.2.4. The argument "The renaming of identifiers is bad." 181
7.3. Possibilities for Further Research 182
7.3.1. Semantics of multiple ellipses 182
7.3.2. New notations for declarations 183
7.3.3. Syntax macros and enhanced declarations 183
7.3.4. Scoped macro declarations 184
7.3.5. Error reporting 186
7.4. Final Remarks 186
Appendix A: Twelve Design Principles 188
Appendix B: Scheme Source Code for `extend-syntax` 190
Bibliography and References 210

List of Figures

1.1. *A nested conditional in a Pascal program* 4
1.2. *The function* `dispatch` *written using a form of* `COND` 5
1.3. *The Lisp macro* `let` 9
1.4. *The declaration of* `macro` 10
1.5. *The syntactic extension* (`or` *exp1 exp2*) 15
1.6. *The syntactic extension* `inf-loop` *and necessary auxiliary macros* . . . 16
1.7. *The partial expansion of an* `inf-loop` *expression* 18
2.1. *McIlroy's declaration of an Algol macro* 36
2.2. *The classification of various macro systems* 38
2.3. *An example of Leavenworth's syntax-macros* 40
3.1. *The Definitions of three FEXPRs* 46
3.2. *An Extended BNF Grammar for a subset of Scheme* 49
3.3. *A parser for a subset of Scheme* 50
3.4. *A syntax table production rule* 51
3.5. *The macro* `let` *declared without and with use of backquote* 54
3.6. *The Scheme macros* `and` *and* `or` 60
3.7. *The Scheme macro* `cond` 61
3.8. *The Scheme macro* `let*` 61
3.9. *The Scheme macro* `recur` 63
3.10. *A syntactic extension which creates an object data type* 64
3.11. *The Scheme macro* `letrec` 65
4.1. *A variant on the Scheme macro* `case` 82
5.1. *An if-then-else macro* 93
5.2. `extend-syntax` *declarations of some common Scheme macros* 96
5.3. `extend-syntax` *production rules* 96
5.4. *A declaration of* `cond` 97
5.5. *A declaration of* `case` 98
5.6. *Transcription specifications* 102
5.7. *A macro for Fibonacci numbers* 103

5.8. *An illustration of* `withrec` 104
5.9. *A second declaration of* `writeln` 105
5.10. *Two special forms that share a cons cell* 106
5.11. *A declaration of* `letrec` *without using* `let` 107
5.12. *The generated* STF *for the macro* `let` 110
5.13. *Another illustration of* `extend-syntax` *produced code* 111
5.14. *Formal semantics of ellipsis* (1) 112
5.15. *Formal semantics of ellipsis* (2) 114
5.16. *Formal semantics of ellipsis* (3) 116
5.17. *Top-level* `extend-syntax` 121
5.18. *Two more preparatory functions* 122
5.19. *The prod-rule-function* 125
5.20. *Regrouping and partial processing of new variable declarations* 127
5.21. *The principal function* `mk-expander` 130
5.22. *The formal parameters of* `mk-expander` 131
5.23. *The atom processing function* `mk-atom` 132
5.24. *The function* `mk-one-liner` 132
5.25. *Processing transcriptions without ellipses in* `mk-regular-body` . . . 133
5.26. *A skeleton of the function* `mk-ellipsis-body` 134
5.27. *Values computed during* `mk-ellipsis-body` 135
5.28. *The specification of* `find-dls` 136
5.29. *The recursive call in* `mk-expander` 138
5.30. *Computing the locations of the* `dotted-lists` 139
5.31. *Computing the* `map` *expression* 141
5.32. *Computing the code for the second argument in a call to* `map` 142
5.33. *Putting together the entire call to* `map` 143
5.34. *The function* `mm`, *which helps generate pattern matchers* 145
5.35. *The* `cond` *clause from the function* `mm` 146
5.36. *The definition of partial-processing* 147
6.1. *Naïve macro expansion* 151
6.2. *Three specifications of the macro* (`or` *exp1 exp2*) 152
6.3. *Hygienic macro expansion* (1) 158
6.4. *Hygienic macro expansion* (2) 159
6.5. *Hygienic macro expansion* (3) 160
6.6. *An* `extend-syntax` *declaration of the Scheme standard macro* `or` . . 161
6.7. *Tracing the expansion of* (`or (zero? v) v`) 162
6.8. *Tracing the expansion of* (`start b`) *and* (`start a`) 168
6.9. *A trace of the extended, modified hygienic algorithm involving* `quote` . 172

1. Introduction

Abstraction mechanisms have always been a part of programming languages. After the first simple machine languages were designed, those who worked with them realized that programming was easier if some abstract form of machine instruction was used. They found they could cope with mnemonic devices that described the instructions and memory locations better than they could with bit sequences. So, names were given to these entities. It was also recognized that certain sequences of instructions were being repeated. People wanted some way to encode algorithms that avoided the duplication of program text. Two forms of abstraction, procedural and textual, were developed. The former was implemented as functions, procedures, and subroutines; the latter as macros.

Procedural abstraction works on the semantic level. We think of the procedure as an independent program that can be called upon as needed while running a main program. In addition to simplifying the original program, the same procedure can be used to simplify other programs. All that is needed is a way to inform the host system which procedures are available during run-time. For example, a procedure that swaps the contents of two memory locations might be written as

```
SWAP: DEFINE SWAP(X,Y)
      LOAD X            ; Load accumulator with contents of X
      MOVE Y,X          ; Move contents of Y to location X
      STORE Y           ; Store contents of accumulator in Y
      RETURN
```

Textual abstraction is designed to work on a syntactic level. The goals for its use are much the same as those for using procedural abstraction, but we think of the abstracted text as being replaced in-line whenever a macro call is encountered during the syntactic analysis phase of program processing. When the program is actually executed, all forms of textual abstraction in the current instruction are gone. The swapping routine might be written as a macro as

```
MACRO SWAP #1,#2
LOAD #1
MOVE #2,#1
STORE #2
ENDMACRO
```

In this work we are concerned with textual abstraction and the form it takes within lexically scoped programming languages. We argue for the inclusion of textual abstraction mechanisms in high-level programming languages in Section 1.1. A section giving definitions basic to our research follows that. Section 1.3 discusses the specific problems we address. Section 1.4 concludes the chapter, describing the contents of the rest of this dissertation.

The seeds of a portion of our thought are contained in a paper by Leavenworth on syntax-macros [42]. We quote his introduction in full:

> The procedure concept in programming has been developed into an elegant and powerful tool. However, the potential extension of current procedural languages into special purpose areas seems to require a more flexible facility than offered by the procedure. On the other hand, the conventional macro, also considerably developed, has still been based essentially on symbolic assembly code. The purpose of this paper is to suggest a generalization of the macro concept to high-level languages, which allows the programmer to extend the syntax and semantics of a given base language by new statements or expressions. The constituents of the new type of macro, called a syntax-macro, are syntactic elements, or constructs of the base language, rather than fragments of machine instructions as is the case with conventional macros. The objective of this work is to develop

> powerful modes of expression and reference by using flexible syntax and multiple levels of definition. ([42], p. 790)

In extending the syntax of a programming language, the programmer forms abstractions of semantic operations. The person who adds the `SWAP` macro to his assembly language no longer thinks of `SWAP` as being comprised of lower-level operations. Instead, he sees his new structure as a semantic entity in itself. This is a human phenomenon; it is what makes abstractions important. The macro would be of little value if we could not attach a semantics directly to macro calls without expanding them. If we always had to do textual translation in order to understand macros, we would soon give up on them.

1.1. The Need for Textual Abstraction

The merits of procedural abstraction are clear: every general purpose, high-level language we know of provides a way to form semantic abstractions over the operations that are performed during program execution. Not many of them, however, provide textual abstraction; those that do include C (Kernighan and Ritchie, [37]), Ada [62], PL/I [34], and the languages in the Lisp family (Steele, [70]; McCarthy, *et al.*, [47]). In this section we argue for the inclusion of a textual abstraction mechanism in all high-level languages.

If we reconsider our assembly language swapping routine and make the assumption that in the procedure the two parameters can represent only memory locations, we see that the subroutine cannot be used to exchange the contents of two registers. However, because the macro version is a textual operation that, in general, does not depend upon the types of its actual parameters, it could be used to exchange the register values. For example, in the macro call `SWAP R0,R1`, the macro parameters `#1` and `#2` represent the registers `R0` and `R1`.

The restriction of user-named entities to only memory locations in this example

```
.
.
TYPE
  function_name = PACKED ARRAY [1 .. 10] of char;
.
.
PROCEDURE dispatch (a : function_name);
   IF a = 'car'
      THEN apply_car
      ELSE IF a = 'cdr'
              THEN apply_cdr
              ELSE IF a = 'cons'
                      THEN apply_cons
                      ELSE IF a = 'pair?'
                              THEN apply_pair?
                              ELSE error(a);
.
.
```

Figure 1.1. *A nested conditional in a Pascal program.*

is admittedly contrived. It illustrates that macros can do things procedures cannot, in this case the swapping of the contents of two registers. Yet there are other, not as concocted, situations in which we might want to abstract over text instead of over operations.

Consider the programming language Pascal (Jensen and Wirth, [36]; Wirth, [79]). Within many programs, IF-THEN-ELSE statements are deeply nested, causing those who read the code to struggle with its logical structure. The CASE statement is designed to remedy this, but there are often situations in which the selection of an alternative cannot be based upon the value of a scalar or subrange expression. For example, to dispatch on the value of a character string, we might write the procedure of Figure 1.1.

```
PROCEDURE dispatch (a : function_name);
   COND
      a = 'car'       : apply_car;
      a = 'cdr'       : apply_cdr;
      a = 'cons'      : apply_cons;
      a = 'pair?'     : apply_pair?;
      otherwise       : error(a)
   END;
```

Figure 1.2. *The function* `dispatch` *written using a form of* `COND`.

A nested conditional is forced by Pascal's rules. Our point is the deeper the nesting, the more inconvenient and unreadable our programs become. If we were free to design our own syntax in Pascal, we could implement something similar to Lisp's `cond`. We would then be able to write the version of `dispatch` contained in Figure 1.2.

The deeply nested `IF` statement occurs with enough frequency to warrant a better syntax. But, `COND` cannot be defined as a function or procedure. The parts of the conditional statement that we want to parameterize—the predicates and their consequents—cannot be expressed as the arguments to a function or procedure; only some of them should be evaluated. Because Pascal does not transmit parameters by-name, each would be evaluated during the function or procedure call. Without a macro facility it is impossible for the typical user of Pascal to create the more convenient conditional expression. Only those willing to modify their compilers or to write a new language front-end can extend the language. This is not done very often; the job is too big.

We are not content with a language that prescribes the set of syntactic forms available to us any more than we would be with a language that decrees and limits

the set of functions and procedures. It is important to be able to recognize repeated syntactic components in a program and abstract over them. Thereby effort is saved in writing programs, in reading them, and in ensuring their correctness. Meaningful, well-designed syntax greatly aids in the understanding of code. Moreover, once we have formed a textual abstraction and its corresponding mental image, that same abstraction can be used in contexts other than the original one. We gain the same benefits that any form of abstraction provides: the ability to suppress the details of a recurring pattern.

Without a textual abstraction system, once a language designer selects his syntax for the semantic constructs in his language, there is no room for adaptation. Leavenworth recognized this in the passage already quoted. No language designer should be so vain as to believe that his language is the best possible, even with respect to a limited applications area. We appreciate the quotation Wegbreit used as the front-piece of his dissertation:

> ... a good notation has a subtlety and suggestiveness which at times make it seem almost like a live teacher ... a perfect notation would be a substitute for thought.
>
> Bertrand Russell
> in the introduction to Wittgenstein's
> *Tractatus Logico-Philosophicus*

All language users should be participants in the search for a perfect notation.

At this point we want to comment on what many believe to be the major weakness of macro processing systems in programming languages: the code that is interpreted or compiled is different from the source code. This is obvious, but it causes problems. Errors detected during the compilation or execution of macro-expanded code should be reported in terms of the source code. Often the compiler or run-time processor cannot do this because the macro system has already done

its work, replacing the source text. We are concerned with this problem, but it lies outside of the scope of our present work.

Cheatham ([14], p. 637) sums up our thoughts:

> In conclusion, it is our very strong feeling that languages with powerful definitional facilities must be placed in the hands of users. Indeed, the appropriate representation—something natural to the individual—of a problem (or solution) has more to do with the issue of solving problems than merely providing a nicety. To those who question this position, I suggest the following problem in long division:
>
> XLVI | MCXXIV

1.2. Basic Definitions

We identify the notion of a textual abstraction mechanism with the general idea of macros. Textual abstraction consists of recognizing that certain pieces of text can be lifted out of their context. These pieces can be compared, resulting in a description of their common structure and in the specification of a set of textual parameters. In the source text, the pieces are replaced by macro calls containing actual values for the parameters. Later, the original, unabstracted version of the text can be restored by the macro processing system.

The virtual machine that carries out the substitution of one piece of text for another is the *macro processor.* Its inputs can be divided into two classes: the *macro declarations* and the *source texts* that are to be transformed by the processor. For textual macros, each declaration specifies a character string that is to be replaced by another character string. The processor analyzes the source text, looking for character strings that match those specified by the coexistent set of declarations. Usually a matching string contains some special *keyword* that is taken to be the *macro name.* There may also be other fixed keywords that appear as syntactic components of the macro call.

A segment of text that matches a macro declaration's input specification is a *macro call.* On finding a macro call, the processor replaces it with the corresponding *transcription* as described by a particular declaration. Portions of text appearing in the macro call are *apparent* and that those fragments introduced during transcription are *generated.*

Completely expanded text is written in the *base* or *host* language of the macro processing system. This language may be, for example, a natural language, a text processing language such as the primitives of TEX (Knuth, [38]), or Common Lisp (Steele, [70]) stripped of all its macros.

The action of replacing one macro call with its transcription is *transcribing* the macro call. One transcription may include further macro calls. Completely removing all macro calls from a piece of text is called *macro expansion.* A macro processor that does so is a *macro expander.*

1.3. The Problems with Syntactic Extensions

There are two components of a syntactic extension mechanism. They are reflected in the two inputs to the macro processor: the set of declared macros and the source text. Building the set of macros requires a macro declaration facility; expanding the macro calls in the source text requires a transcription operation. We discuss each of these, in turn.

There are two significant features of the Lisp family's macro system. The first is that the expansion specification is itself a program that describes how to effect a desired transcription. This program is called a *syntactic transform function* (an STF) or, in the terminology of the Common Lisp book, the "expansion function."

Figure 1.3 contains an example of an STF for the common macro `let`. The argument to an STF is the entire macro call. Instead of being a macro declaration in which replacement text is explicitly specified, as in assembly language macros, Lisp

```
(macro let
   (lambda (form)
      (cons (cons 'lambda
                  (cons (map car (cadr form))
                        (cddr form)))
            (map cadr (cadr form)))))
```

Figure 1.3. *The Lisp macro* `let`.

STFs are instructions for the building of the transcription. This form of declaration gives great power and flexibility to the macro writer, but it suffers because the STFs themselves bear little resemblance to the texts they produce.

The second important feature is that Lisp expansion operates upon syntax trees instead of character streams, as most other macro systems do. Lisp programs may be typed as character streams, but by the time syntactic processing begins, the stream has been transformed by the Lisp reader into a tree structure. Macro calls occupy branches of the tree. The transcribing of a call prunes the branch containing the call and grafts onto the same spot a subtree representing the transcription.

1.3.1. DECLARING SYNTACTIC EXTENSIONS. We use two tools to declare new macros. First, we assume our system has a function `extend-macro-env` that takes two arguments: a macro name and an STF. It extends the macro environment with the appropriate association. And second, the special form

(`macro` *name STF*)

is itself a syntactic abbreviation for

(`extend-macro-env (quote` *name*) *STF*).

The actual declaration of this form is in Figure 1.4.

The declarations for both `let` (Figure 1.3) and `macro` are relatively easy to

```
(extend-macro-env 'macro
   (lambda (form)
      (list
         'extend-macro-env
         (list 'quote (cadr form))
         (caddr form))))
```

Figure 1.4. *The declaration of* `macro`.

understand. However, macro specifications in this form usually involve extensive car-ing, cdr-ing, mapping, and cons-ing; the result is often an STF whose purpose is hard to discern. The difficulty in reading STFs comes from the reader having to mentally execute the code in order to determine the intended transcription. The reading of assembly language macros is so much easier because the parameterized expansion text is in full view. Various attempts have been made to alleviate this problem; some of them are examined in Section 3.2.

A similar problem is that these declarations contain only implicit information as to the syntax of macro calls. The clever reader can perhaps deduce the syntax of a call to `macro` by examining Figure 1.4, but doing so is a demanding activity, requiring knowledge of the semantics of the generated text to know just what form the call must take. Macro users should be able to form macro calls without reference to the STF or the replacement text. However, the macro writer needs to be familiar with both. His task is made harder by the lack of self-documentation in declarations.

Another difficulty with the STF model stems from the unwillingness of macro writers to include detailed syntactic checking of calling forms within the STF. Some systems may provide rudimentary built-in checks, such as "providing some degree of explicit error checking on the number of argument forms in the macro call" (Steele, [70], p. 147). However these checks will not intercept such errors as a

symbol appearing in the macro call where a list is expected. Cryptic, illegal car and cdr errors are also commonplace when people use macros. The macro writer should be careful enough to provide complete, explicit syntactic checks within his STF. But in practice, this is seldom done. None of the macro declarations made in this chapter are correct in this sense.

We see two other problems with this model; both are attitudinal, rather than technical. Working with a macro environment, the user thinks of the macro name as an identifier with a special macro binding. In spite of such warnings as:

> In COMMON LISP, macros are not functions. In particular, macros cannot be used as functional arguments to such functions as `apply`, `funcall`, or `map`; in such situations the list representing the "original macro call" does not exist, and cannot exist, because in some sense the arguments have already been evaluated. ([70], p. 144)

some users try to do exactly what Steele says will not work. Furthermore, among those who understand the restriction, the wish is sometimes expressed that macros be first-class objects. Regarding the macro name as a component of a macro call, we do not believe that it has any significance outside of a legal macro expression. From this point of view, trying to use a macro name in any other context (outside of quoted data) is meaningless. Yet, the reality of what the user sees—the associations in the macro environment—persuades some that the macro name should have an independent existence.

The macro name does serve as the name of the equivalence class of all syntactically legal expressions with that symbol as their first component. The reserved words of the core forms have the same function. Thus we informally speak of "using `set!`" when we mean "using a `set!` expression" and of "using `let`" when we mean "using a `let` expression."

The last problem concerns the power of the STFs to interact with the state of

the machine. This interaction occurs in two ways: (1) accessing values and data structures in existence during transcription aside from the functions and special forms used to build the output expression and (2) actual changes made to the current state. Even though "macros should be written in such a way as to depend as little as possible on the execution environment" (Steele, [70], p. 143), the form that the STF takes encourages both sorts of interaction. Users feel free to do whatever they want within the STFs. A commonly cited use for the first kind of interaction is the gathering of compiler statistics; for the second, the storing of pieces of program text. Both move the user away from looking at macros as syntactic abstractions and toward seeing them as procedures that effect results.

Thus, we perceive five problems with the standard Lisp methods for the declaration of syntactic extensions: (1) STFs are hard to read and write, (2) the STF model provides only implicit information about the syntax of macro calls, (3) error checking is usually neglected, (4) macro environments promote the notion of a macro binding for the macro name, and (5) unrestricted capabilities in the bodies of the STFs allow users to view macros as other than pure syntactic abstraction mechanisms.

In response to these five problems with the customary tools for declaring Lisp macros, we have designed and implemented a program for the declaration of syntactic extensions. It frees the macro writer from having to actually construct an STF; it is consistent with the view that the writer should think of syntactic extensions as tabulated in a set of grammatical productions; it provides an easier-to-read form for the declaration of macros than does an STF; it incorporates syntactic error checking without the writer having to worry about it; and it enforces the discipline that syntactic extensions should be used for extending the syntax of a language and not for other purposes.

Our mechanism, the special form `extend-syntax`, builds an STF, including error checking routines, based upon user-supplied pattern and transcription specifications for a new macro.[1]

The user of `extend-syntax` need not know about STFs. He designs the syntax for calls to a new macro, identifies the parameters within those calls, and creates a transcription specification based on the intended semantics of the new macro. With this work done, he turns to `extend-syntax`. For example, to declare the macro `let`, one writes

```
(extend-syntax (let) ()
   [(let ([i e] ...) b ...)
    (for-all symbol? '(i ...))
    ((lambda (i ...) b ...) e ...)]).
```

The second and fourth lines of this specification state that expressions of the form

```
(let ([i e] ...)  b ...)
```

are to be transcribed into expressions of the form

```
((lambda (i ...)  b ...)  e ...).
```

The symbols `i`, `e`, and `b` are the pattern variables. The third line guarantees that the components of the macro calls referred to by the pattern variable `i` are legal identifiers. Pattern variables are recognized as such by their presence in both pattern and transcription parts. In the declaration

```
(extend-syntax (begin) ()
   [(begin e1 e2 ...)
    ((lambda () e1 e2 ...))])
```

[1] The decision to use the STF model was originally made so that syntactic extensions declared with `extend-syntax` would be compatible with those declared in other ways, for example with `macro`. STFs are a convenient means of operationally expressing the program transformation that must take place during macro transcription.

the pattern part is

```
(begin e1 e2 ...),
```

the transcription part is

```
((lambda () e1 e2 ...)),
```

and the pattern variables are `e1` and `e2`.

1.3.2. EXPANDING SYNTACTIC EXTENSIONS. The simplest form of macro transcription in Lisp is brought about by applying the STF to the macro call. We call this *naïve transcription.* When a transcription replaces a macro call, no attention is paid to the *context* in which the call appears. The context of an expression is the entire program text which surrounds it.[2]

The problem with naïve transcription is concisely stated as:

> *Apparent identifiers within a macro call intended to occur free within that call's expansion may be inadvertently captured by generated, binding instances of the same identifiers.*

This is the *capturing problem.* It is easy to see what happens by looking at an example. Let the syntactic extension `(or` *exp1 exp2*`)` have the transcription

```
(let ([v exp1])
  (if v v exp2)).
```

The `let` expression binds the local identifier `v` to the value of the first sub-expression *exp1*. The actual declaration is given in Figure 1.5. All is well unless a `v` happens to appear in the second sub-expression *exp2* of an `or` macro call. Transcribing the `or` expression in

```
(let ([v 1]) (or (zero?  v) v))
```

[2] A closely related definition of "context" is given by Barendregt ([3], p. 29).

```
(macro or
   (lambda (form)
      (list 'let
         (list (list 'v (cadr form)))
         (list 'if 'v 'v (caddr form)))))
```

Figure 1.5. *The syntactic extension* `(or` *exp1 exp2*`)`.

produces

```
(let ([v 1])
   (let ([v (zero? v)])
      (if v v v))),
```

an expression whose value is `false`, whereas the expected value of the original expression is `1`. The discrepancy occurs because the apparent instance of the identifier `v` as *exp2* in the macro call gets placed in the transcription where it is captured by the binding instance of `v` generated by the STF.

There are times when capturing is desirable. For example, a looping construct

`(inf-loop` *exp1 exp2 ...*`)`

that permits exit only by passing a value to the local procedure `exit-with-value` might be transcribed as

```
(call/cc
   (lambda (exit-with-value)
      (while true exp1 exp2 ...))).
```

Figure 1.6 contains the actual declaration for this macro along with declarations for `begin`, `rec`, and `while`.

When using `inf-loop` expressions, we expect to be able to write function calls to `exit-with-value` within the expressions making up the body of the loop. Apparent uses of the identifier `exit-with-value` are to be caught by the generated,

```
(macro inf-loop
   (lambda (form)
      (list 'call/cc
         (list 'lambda (list 'exit-with-value)
            (cons 'while
               (cons 'true (cdr form)))))))

(macro while
   (lambda (form)
      (list
         (list 'rec
            'loop
            (list 'lambda '()
               (list 'if
                  (cadr form)
                  (cons 'begin
                     (append (cddr form)
                        (list (cons 'loop '()))))
                  nil))))))

(macro rec
   (lambda (form)
      (list (list
               'lambda
               (list (cadr form))
               (cons 'set! (cdr form))
               (cadr form))
            (list 'quote '*))))

(macro begin
   (lambda (form)
      (list (cons 'lambda
               (cons '() (cdr form))))))
```

Figure 1.6. *The syntactic extension* `inf-loop` *and necessary auxiliary macros.*

binding occurrence in the `call/cc` expression. On the other hand, the identifier `loop` is in the same category as the identifier `v` in `or` expressions: an undesired binding occurs when we eventually expand a generated `while` expression. It is illuminating to trace (Figure 1.7) the expansion of the `inf-loop` expression in

```
(let ([loop 10])
   (inf-loop
      (if (positive? loop)
          (set! loop (sub1 loop))
          (exit-with-value loop)))).
```

When the `rec` expression is generated, the binding troubles of `loop` are manifested.

Identifiers with roles like that of `exit-with-value` are *key identifiers*. They are not keywords, but they are part of the protocol a programmer must know in order to use the macros with which they are associated. It is our experience that the potential for the inadvertent capture of apparent identifiers occurs with much greater frequency than the opposite situation in which key identifiers are needed. Hence, we suggest that naïve macro expansion mechanisms are poorly designed; they provide capturing as the default. These macro expanders work well for the rarer case of having key identifiers and poorly for the more common case of inadvertent capturing. This is the opposite of how it should be. The hygienic expansion algorithm of Chapter 6 switches the defaults.

1.4. The Organization of this Dissertation

Our primary goal is to present a reasoned approach to the inclusion of a syntactic extension mechanism in a lexically scoped, high-level language. This includes designing and implementing facilities for macro declaration and expansion as well as providing a rationalization for our design decisions.

Chapter 2 defines some technical terms in the course of presenting a survey of macro processing systems. It also includes a discussion of the distinction between

```
(let ([loop 10])
   (inf-loop
      (if (positive? loop)
          (set! loop (sub1 loop))
          (exit-with-value loop))))

⇒ (let ([loop 10])
       (call/cc
          (lambda (exit-with-value)
             (while true
                (if (positive? loop)
                    (set! loop (sub1 loop))
                    (exit-with-value loop))))))

⇒ (let ([loop 10])
       (call/cc
          (lambda (exit-with-value)
             ((rec loop
                 (lambda ()
                    (if true
                        (begin (if (positive? loop)
                                   (set! loop (sub1 loop))
                                   (exit-with-value loop))
                               (loop))
                        nil)))))))
```

Figure 1.7. *The partial expansion of an* `inf-loop` *expression.*

textual abstraction and syntactic abstraction within programming languages. A partial history of the efforts toward the goal of a syntactic extension facility in Algol family languages is given.

Chapter 3 provides a similar account of what has been done in Lisp systems prior to our work. It also has a description of some of the uses of macros in Lisp programming.

Chapter 4 contains the philosophical foundation of our work. It explains the design decisions made in implementing our system. A large part of what we have to say is not tied to Scheme, but relevant to all high-level languages. We state our major tenets as a series of Principles. More design principles are brought out as we consider the differences between expansion times, the scoping mechanisms of the base language, and the behavior of the macro processor.

Chapters 5 and 6 give the details of our implemented system. The system is in two parts: a tool for declaring syntactic extensions and a processor for expanding them. Both components have been operational for over two years and have withstood testing by many programmers, especially students in the programming languages courses at Indiana University.

The tool for declaring syntactic extensions is described in Chapter 5. The first part presents a manual for its use, the second part gives a formal semantics for simple occurrences of ellipses within `extend-syntax`, and the last part describes the current implementation of `extend-syntax`.

The expansion of syntactic extensions is taken up in Chapter 6. It explains the hygienic expansion algorithm and sketches a proof of its correctness.

We conclude with Chapter 7, highlighting and summarizing our work, evaluating its place in the development of Scheme. We also discuss local macro declarations and suggest some areas for further work.

2. Related Work

In this chapter we survey a representative set of macro processing systems chosen because they illustrate concepts important in the design of our system. We also consider some of the syntactic extension mechanisms proposed for languages in the Algol family.

Macros are an aspect of extensible programming languages. Wegbreit defines an extensible language as "a higher level language which includes mechanisms with which the user can extend the language to facilitate its use in various application areas" ([76], p. 20). In 1969 and again in 1971 conferences were held on extensible languages (Christensen, [16]; Schuman, [65]).

Solntseff [68] provides a classification scheme for extensible languages, and Standish [69] gives a good summary and analysis of the work done in the field up to 1975. A thorough bibliography appeared in 1979 [50]. Layzell gives a short history of macro processors in high-level languages in his paper of 1985 [41]. Henderson and Gimson discuss the use of macros to achieve program modularization [33]. We follow Brown's book on macro processors [8] in the early parts of this chapter.

A number of macro systems and extensible languages have been proposed. In 1975, Standish counted twenty-seven extensible languages ([69], p. 19). Most of these have not seen widespread use; in fact, some of them have never been implemented. One widely distributed macro processor attached to a high-level language

is the PL/I compile-time facility [34]. Macro processors have also been used with Pascal (Brown and Ogden, [11]; Comer, [19]) and FORTRAN (Nagata, [56]; Munn and Stewart, [55]; Macleod, [46]).

Recently there has been renewed effort on extensible languages especially with regard to semantic and type structure extensions. A large part of modern program language design is concerned with finding mathematically sound, controlled approaches to providing users with ways of extending a given base language. The work on polymorphic types and type constructors (Harland, [31]), modules (Wirth, [80]), clusters (Liskov, *et al.*, [44]) and so forth are efforts at giving the user more power in a well-reasoned structure. Currently there is a project at the Massachusetts Institute of Technology Laboratory for Computer Science on designing a fully extensible language (Schooler, [64]).

Wegbreit recognizes three components that constitute an extensible language:

> (1) the base language, its theoretical foundations, its design, and its specification,
> (2) the extension faciltities, their mechanism, and their theory,
> (3) the definition sets, their creation and interaction. ([76], p. 11)

We are concerned with syntactic extensions, their mechanism, and their theory as they relate to lexically scoped languages. Because of the research we wish to pursue in the design and implementation of programming languages, we select a base language that contains the semantic equivalents of Landin's four fundamental operators: function application, function abstraction (λ), variable assignment ($\Leftarrow$), and escape (J) [40]. Scheme is such a language (Rees and Clinger, [60]; Steele and Sussman, [71]). Unless we state otherwise, it is the dialect of Lisp we use in our programming examples.

One of our major research interests is the embedding of other languages within Scheme. The availability of a macro facility directly affects this work, enabling

and empowering our thinking. A variety of Indiana University Computer Science Department technical reports show this interest. For example, there is work on Data Flow (Chen and Friedman, [15]), 2-Lisp (Halpern, [29]; Smith, [66]), Prolog (Felleisen, [21]), and import and export statements (Felleisen and Friedman, [22]). Examples of related work by others are found in Lindstrom's paper [43] and Chapters 9 and 10 of Brooks' book [6].

2.1. A Survey of Textual Abstraction Systems

There are two basic kinds of textual abstraction systems: special-purpose and general-purpose. The first macro processors were tied to particular assemblers and assembly languages; they are special-purpose. Some later macro processors were designed to work on any string of characters; they are general-purpose. In working with programming languages, the special-purpose processors are keyed to a specific host language, the general-purpose systems may be used with any base. We will examine each kind of system, using our discussion to provide and illustrate needed definitions. Then we will look briefly at two modern programming languages with macro processors: C and Common Lisp.

2.1.1. GENERAL-PURPOSE MACRO SYSTEMS. Some textual abstraction facilities are designed to work with any host system. They may function as a preprocessing stage for a programming language implementation, or they may produce text that stands as is without need of further processing such as a letter written by a system which composes form-letters. A general-purpose macro system may be used to abstract over the general form of the letter and such specific information as the return address and closing. The parameters to such an abstraction would be used for such information as varies from letter to letter, for example, the inside address, greeting, and the recipient's name for the spot in the body where it is included to give that extra, personal touch.

Whether general or special purpose, macro calls must be distinguishable from the rest of the text. There are two main ways of accomplishing this: *warning mode* and *free mode*. They are best illustrated by example.

An early general-purpose macro processor is Strachey's GPM [73]. Using it, macro calls are identified by the presence of the special symbol §. All macro calls begin with the § character, and whenever the § character appears, it signals the beginning of a macro call. This symbol is the *warning marker*; GPM runs under warning mode. A macro call has the form

$$\S name,\ arg_1,\ arg_2, \ldots,\ arg_n;$$

Users are limited to ten parameters, denoted `~0` through `~9`, within the expansion text. The first, `~0`, represents the macro name, the remaining nine, the arguments of a macro call.

Another example of warning mode is contained in the TRAC language (Mooers and Deutsch, [53]; Mooers, [51]; Mooers, [52]). The symbol # is used to denote function calls, and macros are expanded by calling a built-in function. This function's arguments are the components of a macro call. The macro call is written

$$\texttt{\#(cl,}\ name,\ arg_1,\ arg_2, \ldots,\ arg_n\texttt{)}$$

where `cl` is a built-in TRAC function that transcribes a macro. In a sense, TRAC has a warning string "`#(cl,`" instead of a warning character.

Those macro processors that do not require the use of a warning marker are said to run under free mode. Examples are Brown's ML/I [7] and Waite's STAGE2 [75].[3]

Of additional interest is the delimiter structure each macro system possesses. In general, macros are declared with parameters as a means of incorporating apparent

[3] Actually, the user of ML/I has the option of whether to run in free mode or warning mode. But Brown states that in practical usage, free mode has been chosen almost exclusively over warning mode ([8], p. 57).

text within the transcription and of selecting which of several possible transcriptions may be produced. The text corresponding to each parameter is set off within the macro call by the *delimiter structure.* It includes the macro name and any special symbols that mark the end of the macro call.

In GPM, actual text arguments within a macro call are delimited by commas. There are special string quoting symbols, the left and right angle brackets, so that commas may appear within arguments. The entire macro call is terminated by a semi-colon. In TRAC, commas also serve as delimiters. Because the entire macro call takes place within the context of a function call, the arguments in the macro call are regarded as arguments to the function.

In ML/I, there is no fixed delimiter structure; each macro may have its own. In a macro declaration, the delimiters are literals. The regions between delimiters are indicated with ellipses. For example, in

```
SET...=...+...
```

the three delimiters are SET, =, and +.

It is also possible to specify alternatives for delimiters. Subsequently, during macro transcription a delimiter may be used to select one of several possible expansions. Furthermore, each delimiter structure describes a directed graph. Certain spots within the graph may be designated as nodes, and each node may be defined as the successor of any delimiter. In this way it is possible to declare optional or repeated arguments.

ML/I bases its macro facility on atoms instead of characters. Any sequence of alpha-numeric characters set off by spaces or other delimiters makes up an atom. When an atom is encountered during macro expansion that matches the name of a macro, the macro processor begins analyzing the remaining text looking for a correspondence with the macro's delimiter structure. The SET macro would be

called only if the name appears as an atom. Symbols such as ASSET and SETF would not cause the SET macro to be called. Furthermore, when the atom SET appears within comments, character strings, or other specially designated sections of text, the macro is not called.

ML/I is limited because each macro call must begin with the macro name; macros are written in a prefix notation. The STAGE2 macro system solves this problem by recognizing macro calls by pattern matching. The lines of source text are analyzed separately. Each macro has a template made up of some fixed strings and the gaps between them. The fixed strings serve as delimiters, and the gaps correspond to the arguments. The source lines are compared with all the declared templates, looking for the one that fits. If more than one template fits, the one that matches the most fixed-string characters is selected. For example, the source text line

```
IF (n = 0) THEN true ELSE even?(n - 1) FI
```

would match the templates

```
IF ' THEN ' ELSE ' FI

IF ' THEN ' FI

' THEN ' ELSE '

IF '
```

where the single quote represents an argument. The first template gives the best fit; it would be the one selected for our example macro call. MP/1 [Macleod, 46] uses a similar and slightly extended method for matching macro calls.

Corresponding to each STAGE2 template is an expansion text. When a template is matched, local character variables are created and given the argument character strings as initial values. The system limits the macro writer to nine such local variables, and hence, to nine gaps in each template. However, there is also a set

of global character values maintained in an associative memory. These may be associated with any text and used as needed within expansion texts.

The use of local and global character values in STAGE2 is related to the concept of *local* and *global* macros. Just as it is possible to create procedures in a lexically scoped language that have lexical scope, and hence are local to some region of program text, in some macro systems it is possible to create macros that are local to some region of text and unknown outside.

2.1.2. SPECIAL-PURPOSE MACRO SYSTEMS. Three examples of special-purpose macro systems are macro-assemblers, the typesetting language TEX, and the languages of the Lisp family.

Macro-assemblers are used to translate assembly language code into machine language. The source text may contain macro declarations interspersed with assembly language code that contains macro uses. Because macros must be declared before they can be recognized as such, some protocol about the order of declaration and use must be established. Some systems enforce the restriction that a declaration must physically precede any macro call. In these systems, only one pass through the source text is needed to effect complete expansion. Other systems may permit the macro declarations to appear at any location within the program. They are easily implemented with two passes through the program: the first to gather the declarations and the second to expand any calls.

When a macro-assembler requires the presence of a macro name in a special field within each line, it is using a variation on warning mode. However, this use is a little different from that in GPM. For that system the presence of the warning marker indicates the undisputed existence of an immediately following macro call. For a macro-assembler, the special field, usually the operand field, may or may not contain the name of a macro. The space or whatever delimits the operand field acts

as the warning marker. This limits macro calls to replacing full lines of assembly language text. The macro calls may not, for example, be located in the operands.

A more extensive discussion of macro assemblers lies outside the bounds of this work. The final word has not been given on this topic; from time to time papers appear describing some new aspect of assembly language macros. The interested reader is referred to the books by Campbell-Kelly [13] and Cole [18] as well as the papers by Flores ([23]; [24]) and Revesz [61].

TEX is a special-purpose macro processing language designed for the typesetting of books and papers. It contains about three hundred low-level control sequences which are designated as "primitive." The language of these primitives may be extended with further control sequences by individual macro definitions. Each non-primitive control sequence is a macro.

The system uses warning mode. The warning character, what Knuth calls the "escape character," is generally the backslash, although the user is free to change it whenever desired.

The primitive control sequence `\def` is used to declare new macros. It is followed immediately by the macro name and a description of the delimiter structure for the macro being declared. Finally comes the (parameterized) *transcription text* or *body* of the macro declaration, describing the transcription to be made when a macro call is encountered. It is enclosed within TEX's grouping symbols, the left and right brace. An example of a macro declaration for a control sequence that produces two copies of a given argument separated by the symbol and is

```
\def\doubleand#1{#1 and #1}.
```

TEX allows a flexible delimiter structure; the macro writer is free to choose whatever structure he desires for each macro he declares. Bodies are limited to nine parameters, designated `#1` through `#9`.

Argument text is treated as is, fit into the transcription text in place of the formal parameters where specified. Macro calls within the arguments are not expanded until the transcription is itself processed. This is *call-by-name* macro transcription.

The alternative is *call-by-value* transcription. Macro calls within the arguments are expanded first, before they are associated with the outer macro call's formal parameters. This method is used in Strachey's GPM.

Another consideration is the time at which macro calls within the declared bodies are expanded. In TEX, they are not expanded until after they have been copied into transcribed text. This is how most macro processors behave. However, it is possible to have macro calls within the body expanded at macro declaration time. This is accomplished using `\edef` instead of `\def`. Borrowing an example from *The TEXbook* ([38], p. 215), we consider

```
\def\double#1{#1#1}
\edef\a{\double{xy}}
\edef\a{\double\a}.
```

The first call to `\edef` is equivalent to

```
\def\a{xyxy},
```

while the second is equivalent to

```
\def\a{xyxyxyxy}.
```

The body may contain transcription-time directions about what to produce. In a great many macro processors these consist of some sort of conditional tests that may be performed during transcription. TEX, for example, contains an elaborate set of conditional tests.

The language used within the bodies is all of TEX itself. For other macroprocessors, this need not be the case. Leavenworth's syntax-macro system [42], defined as an extension to an Algol-like language, contains conditional forms not part of that language. There are really two languages involved: the *host language*, which

contains the core forms, macro calls, and literals, and the *macro language*, in which the bodies of macro declarations are written.

2.1.3. THE C PREPROCESSOR. Kernighan and Ritchie's language C includes a preprocessor. It has several components; we describe only the textual abstraction mechanism.

Macro declarations consist of the special symbol `#define`, a macro name, an optional parameter list, and the replacement text. The replacement text normally extends to the end of the line, although it may be continued on subsequent lines by use of a special character. C's macro language is sparse; all that is allowed within a declaration body is a parameterized replacement string.

The macro processor runs in free mode, and macros may appear anywhere within the source text. The macro call is transcribed before compilation, during the preprocessing phase. We illustrate this with an example from Kernighan and Ritchie's book ([37], p. 87). The declaration of a macro named `max` appears as

```
#define max(A, B) ((A) > (B) ? (A) : (B)).
```

The source program line

```
x = max (p+q, r+s);
```

is replaced by the transcription

```
x = ((p+q) > (r+s) ?  (p+q) :  (r+s));
```

Similar to our assembly language macro SWAP in Chapter 1, as "long as the arguments are treated consistently, this macro will serve for any data type; there is no need for different kinds of `max` for different data types, as there would be with functions." ([37], p. 87)

2.1.4. COMMON LISP. There are two kinds of macros in Common Lisp: *read macros* and *ordinary macros* They are distinguished by the different source languages each treat.

Macro characters are symbols that trigger special behavior in the Lisp reader, such as building lists and recognizing comments. The set of macro characters normally includes the left-parenthesis, right-parenthesis, single-quote, double-quote, and semi-colon. The user is free to alter the particular reader behavior associated with each macro character.

If a general-purpose macro system is attached to a language, its macros may be expanded at read-time. Text processing occurs without reference to the lexical structure of the language—input is treated as a character stream. Because read macros, in a sense, have more to do with the characters of the text rather than with the syntactic tokens of the program, we are not concerned with them in this work.

The other kind of Lisp macro acts on the syntax trees produced by the reader. The macros of Section 1.3 are ordinary macros. Sometimes these macros are divided into two classes based on the time in language processing during which macro calls are expanded. These two groups are the run-time and compile-time macros.

Assuming a language is implemented by compilation, all source language macros must be either read-time or compile-time macros. However, for a language implemented by interpretation, it is still possible to have a syntactic analysis phase in the language processing. This phase does not translate the source language into another language, but it may accomplish a syntactic check on the program and provide for the expansion of macros into the core language.

A different refinement of the set of ordinary macros is possible. Macros that do not access the run-time state of the program's executing machine are *safe* macros. Those that do access the run-time state are *unsafe*. Safe macros can always be

expanded during a syntactic analysis phase of language processing.

As an example of an unsafe Lisp macro, consider a macro `ornd` whose STF has local state. This local state is used to determine whether the macro has been called an even or an odd number of times. If the macro has been called an even number times, it expands as a call to the macro `and`. If it has been called an odd number of times, it expands as a call to the macro `or`. Since, in general the meaning of the expression

```
(begin (ornd a b) (ornd c d))
```

should not depend upon which of the two sub-expressions is expanded first, it is impossible to give a semantics for this expression. The macro `ornd` is unsafe because it accesses the run-time state of the machine making the transcription. Other possibilities for unsafe macros include those that access global and dynamic variables and those that (as part of transcription) modify the macro environment.

To finish this section, we tabulate the attributes of the Common Lisp macro system. As we have indicated, there are reader macros available in Common Lisp systems, but we concern ourselves here with the ordinary macros that are expanded during a later phase of language processing.

1. The macro processor is a Lisp function known as `macroexpand`. It effects a complete expansion. There is also another function `macroexpand1` that accomplishes a single transcription.
2. Macro declarations take the form of STFs. Literal information within the STF must be quoted, or some special tool like backquote must be used.
3. The macro language is the same as the host language at the time of macro declaration.
4. Each macro call takes the form of a Lisp pair (a "cons" in Common Lisp book terminology) of which the first component is the name of a macro. This first

component is a keyword.

5. Any other special symbols that are integral parts of a macro call are *auxiliary keywords*. If an STF expects an auxiliary keyword and it is not present, the STF itself is responsible for reporting the error.
6. Macros are often expanded during run-time. It is possible to write unsafe macros that depend upon the run-time state of interpreting machine for control of expansion, but it is also possible to write safe macros that do not depend on this state. Most common macros are safe.
7. Macro calls are made using free mode; there is no warning marker. Excepting the presence of the initial keyword, macro calls are syntactically indistinguishable from function applications.
8. The list structure of the macro call provides the delimiter structure. It and any keywords act as delimiters because they are detected by the STF.
9. Each STF takes two arguments, an entire macro call and a macro environment containing the macro bindings of the currently declared macros. Components of a macro call that need to be accessed within an STF are referenced by appropriate car-ing and cdr-ing.
10. Local macros are available using the special form `macrolet`.
11. Expansion is call-by-name.
12. Macro calls generated by an STF are expanded after transcribing of the original macro call, not at macro declaration time.
13. The macro language is the same as the host language, containing all macro definitions that have been made prior to the declaration of the macro currently being declared.
14. As part of the transcription of any macro call, any subexpression of the original call may be expanded prior to the generation of the transcription of the

original call. This is done by calling either of the functions `macroexpand` or `macroexpand1` within the STF.

2.2. Textual and Syntactic Abstractions

The primary difference between textual abstraction and syntactic abstraction is whether information about the structure of a piece of text is used in performing the abstraction. The TEX macro processor implements textual abstraction because it treats its text input as a token stream. The same is true of GPM, TRAC, and ML/I. The macro processors in the Lisp family implement syntactic abstraction because they treat their input as syntax trees, objects with a discernible structure.[4] Most of them read their input character by character—thus treating it as a stream. But they replace one syntactic unit, the macro call, with another unit that makes sense in the location.

When working with syntactic abstraction, we prefer the name "syntactic extension" to "macro," although we use both, interchangeably. We shy away from the name "macros" because macros have earned a bad reputation within the programming language community. We restrict some of their generality in our work, hoping to provide a still-useful yet not-so-capricious macro system.

2.3. Macro Extensions to Languages in the Algol Family

As Wegbreit observes, many extensions have been proposed for the language Algol 60 ([76], p. 3; Naur, [57]), some of them are suggestions for syntactic extension packages. In this section we examine a few of these, giving a little of the early history of macro proposals in the Algol family. The systems we choose to discuss contain ideas central to our work on a syntactic extension system for Lisp.

[4] The main reason why it is possible to think of Lisp programs as trees instead of streams is the presence of the character macros and the Lisp reader.

2.3.1. McIlroy's proposals. As early as 1960, macros were being proposed as an extension to Algol. McIlroy describes a general purpose macro system that can be "readily implemented for a wide variety of programming systems," and which constitutes "a powerful tool for extending source languages conveniently and at will" ([48], p. 214). He bases most of his paper on adding macros to an assembly-level language but in the appendix describes a macro system built around Algol. The paper deals more with the declaration of macros than with the techniques for their expansion. Five features of his macro system are noteworthy:

- *Pyramided definitions.* The capability of defining one macro transcription in terms of others is useful in building a large set of macros. The specification of `inf-loop` in terms of `while` (Section 1.3.2) is an example of a pyramided definition.
- *Conditional macros.* Most macro systems contain a way to designate alternative expansions based on conditions detected at macro expansion time. These conditions may involve testing the arguments to a macro call.
- *Created symbols.* Generated symbols are frequently produced during macro expansion; names are created that are unique to the expansion of each macro call. Since a created symbol has "no significance outside the macro, its naming should not be of concern to the programmer" ([48], p. 215). Created names are "generated sequentially as needed from some special alphabet that the programmer is forbidden to use" (p. 215). McIlroy would use a created symbol for the identifier `v` in our macro `or` of Section 1.3.2. However, as is clear in his examples from Algol, he also uses created symbols as part of the control structures which produce replacement texts.
- *Nested definitions.* The capability of one macro to declare a new macro as part of producing its replacement text is the "nesting of definitions." By including

this feature, macros not only generate replacement text but also perform other tasks. McIlroy states the principles: "Allow anything in the body of a definition that is acceptable outside," and "Statements effective at object time should have counterparts effective at compile time."

- *Repetition over a list.* There is a means of defining a new macro with one parameter that represents a list of the varying arguments from a set of macro calls. In the declaration body, a special control structure maps part of the body over the list, thus providing a shorthand for making a set of macro calls.

McIlroy does not say this, but it is an easy extension of the first two features to allow *recursive* macro declarations. These declarations cause the generation of macro calls to the same macro. The generated calls, of course, must be in a sense simpler than the original call or expansion will never terminate. The use of conditional expansion enables the detection of the simpler cases. Repetitions over lists eliminate much, though not all, of the need for recursive macro declarations.

McIlroy's declarations consist more of expansion-time directions for producing the transcription than is common in assembly language macros. In this respect his system is more like Lisp. In fact, there is a **text** declaration form that specifies the actual replacement text to be generated. We illustrate this with a macro from the paper that specifies an "alternative" statement (Figure 2.1). This statement takes two lists of expressions. The first list is an ordered set of tests, the second a corresponding set of result expressions. It is used, for example, as

```
alternative ((a>0, a<0, a=0),
             (y:=a+2, y:=a-2, y:=0))
```

with the intention of assigning one of the values `a+2`, `a-2`, or `0` to the variable `y` according to the algebraic sign of `a`.

The alternative macro has two parameters (B and S) and three created symbols (X, Y, and Z). The created symbols X and Y are used entirely during the production

```
macro alternative (B, S; X, Y, Z) :=
begin
    for X := B, Y := S
     text if X
      begin Y;  go to Z end
    text Z;
end alternative
```

Figure 2.1. *McIlroy's declaration of an Algol macro.*

of the transcription; the created symbol Z appears in the output text as the label of the empty statement at the end of a series of **if** statements. Our example might transcribe as

```
        if a>0
          begin y:=a+2; go to 10001 end
        if a<0
          begin y:=a-2; go to 10001 end
        if a=0
          begin y:=0; go to 10001 end
10001   ;
```

It becomes harder to determine the intent of the macro the more macro language instructions there are in the declaration.

2.3.2. PRE RUN-TIME MACROS. Cheatham [14] identifies three times during compilation where macros might be expanded: preceding syntactic analysis, during syntactic analysis, and following syntactic analysis. As mentioned earlier, macros expanded during the first time correspond to the reader macros of Lisp.

A macro that is expanded following syntactic analysis is called a computational macro. Its replacement text is compiled once, at macro declaration time. The result is a sequence of pseudo-instructions in the target language. A call to a computational macro is not expanded during the syntactic analysis phase; it is left

in the host language representation of the program until actual compilation. Then, the call is replaced in-line by the compiled transcription, substituting the compiled arguments to the call for the parameters. Computation macros and other forms that produce compiled code do not interest us here.[5]

Cheatham divides the macros expanded during syntactic analysis into two categories, MACROS and SMACROS. The two kinds of syntactic macros are further distinguished by calling MACROS in warning mode and SMACROS in free mode. MACROS are similar to other warning-mode macros. The difference is that their arguments, instead of being arbitrary strings, have a specified syntactic type. They can be used anywhere within program text, but they have the potential of trapping type errors during macro expansion.

SMACROS can only be called within specific syntactic contexts. These contexts are expressed as syntactic types when the SMACRO is declared. Thus some SMACRO calls may be replaced by expressions, some by commands, some by type specifications, some by complete declarations, and so forth.

A notation for describing the kind of transcriptions effected by various macro systems may clear this up. We designate four different languages:

1. *Source.* The original language, the one in which the programmer writes.
2. *Syntax-tree.* A language in which the program is not viewed as a character string but as a syntactic structure.
3. *Intermediate.* A language midway between *Source* and *Target.* It may be the same as *Syntax-tree*, but it may also reflect further processing, such as having all identifiers replaced by bindings.
4. *Target.* The language that is interpreted by a virtual machine during program

[5] The MAD definition facility (Arden, Galler, and Graham, [2]) contains computation macros. In LISP/VM (Alberga, [1], pp. 23–24) one can write a macro that is processed during syntactic analysis but expands to compiled code.

McIlroy's macros	[*Source* → *Source*]
Cheatham's text macros	[*Source* → *Source*]
Lisp reader macros	[*Source* → *Syntax-tree*]
Cheatham's **macros** and **smacros**	[*Intermediate* → *Intermediate*]
Ordinary Lisp macros	[*Syntax-tree* → *Syntax-tree*]
Cheatham's computation macros	[*Intermediate* → *Target*]

Figure 2.2. *The classification of various macro systems.*

execution. The code generation phase of a compiler is a translation from *Intermediate* to *Target*.

Figure 2.2 contains a description of several of the kinds of macros we have discussed. For ordinary Lisp macros in an interpreted implementation, the language *Syntax-tree* is the same as the language *Target*.

2.3.3. LEAVENWORTH'S SYNTAX-MACROS. About the time that Cheatham made his proposals (late 1966), Leavenworth was thinking along the same lines as the SMACROS [42]. He implemented a system of "syntax-macros" in which there are statement macros, declared with **smacro**, and function macros, declared with **fmacro**. Statement macros can be used anywhere a statement is legal; function macros can appear anywhere an expression is allowed. The syntactic types of the macro arguments are specified. During expansion the types are compared with the actual arguments; a mismatch is an error.

Of interest in Leavenworth's paper are:

- Syntax-macros have two parts, a structure that describes the syntax of the macro calls and a definition that describes the transcription.
- Top-down syntactic analysis of the source program for macro expansion is pre-

ferred. In this way, macro expansion can be separated from the rest of parsing. When a macro call is detected, the macro expander takes over control, returning a syntactic unit that is then fed back to the parser.

- The capturing problem is recognized. After giving an example of an fmacro **sum** in which there is a generated, binding occurrence of the identifier t, he notes:

> 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 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. ([42], p. 792)

- A string-based transcription method is used. This is why the declarations appear so simple; Leavenworth has only to describe the replacement text by writing a parameterized string. Conditional macros are provided, but the notation for them does not correspond to the syntax of conditional statements in his host language. The "full power of compile-time imperatives" is not implemented.

We classify Leavenworth's syntax-macros as [*Intermediate* → *Intermediate*].

The declaration of a statement macro of a simple **for** statement, taken from Leavenworth's paper, is found in Figure 2.3. Parameters in the macro definition are prefixed by a dollar sign.

2.3.4. FURTHER DEVELOPMENTS IN ALGOL-FAMILY LANGUAGES. The late 1960's saw a surge of effort on extending Algol, in much the same way the past ten years has seen work on extending Pascal. Many of the proposals for Algol involve syntactic extensions; the interested reader is referred to the two extensible language conference proceedings already mentioned ([16]; [65]) as well as the papers by Galler and Perlis [27] and Irons [35]. We briefly discuss one other proposal, which has its

```
smacro for variable ← express to express do statement
    define begin $1 ← $2;
      L1 : if $1 ≤ $3 then
              begin $4; $1 ← $1 + 1;
                 go to L1
              end
           end
           endmacro
```

Figure 2.3. *An example of Leavenworth's syntax-macros.*

roots in the papers we have considered.

MacLaren's paper on [*Intermediate* → *Intermediate*] macros in EPS [45] is relevant for two reasons. First, he observes three differences between his macros and those found in assembly languages. They are:

> *First.* The computational facilities provided are more general than is typical. Integer and floating point arithmetic, list structures, compound expressions, loops, etc. all can be used at compile time.
>
> *Second.* Macro processing takes place after syntactic analysis and concurrent with the processing of declarations and consequent interpretation of expressions. This means that the macro programmer is freed from concern with the details of syntactic analysis. Instead he works directly with entities that seem natural for programming, e.g., integers, identifiers, expressions. More importantly, the syntactic structure of the source program and the information contained in declarations can be exploited in macro processing. For example, a macro definition can be made local to a **begin** block, a macro procedure can learn the attributes of its actual parameters.
>
> *Third.* Macros ultimately expand not into source language phrases but rather into phrases in an abstract object language. In this object 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. (p. 32)

The second interesting thing about MacLaren's paper is that he recognizes the

problem with conflicts between apparent and generated identifiers. He realizes that the block structure of his base language complicates the problem. As in Leavenworth's system, the burden of avoiding this difficulty lies with MacLaren's macro processor. He says:

> 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 entity to which it is bound.* (p. 34)

2.4. A Pattern Matching Macro Processor

Starting with syntax macros, ML/I, STAGE2, and MP/1, Sassa develops a [*Source* → *Source*] macro processor, which he calls PM [63].

The declaration component of his macro processing system is similar to ours. A macro definition is made up of pattern declarations, which describe the syntax of macro calls, and macro bodies, which operationally specify the generation of macro transcriptions. A typical macro declaration looks like

macro *macro pattern* 1 = *macro body* 1,
macro pattern 2 = *macro body* 2, . . . ;

Macro patterns are defined using regular expressions. The analysis of macro calls is made by pattern matching against these declarations. Sassa's use of regular expressions gives him improved expressive power over Leavenworth's syntax-macros, where unlimited repetitions within the macro call are not allowed, and over ML/I, where optional components are not supported.

An example of a macro pattern is

'case' enq **'in'** unit ⟨**','** unit⟩ (**'out'** outunit|/) **'esac'**

where the angle brackets indicate zero or more repetitions of the enclosed form and the (|/) structure indicates an optional part. The delimiters are surrounded by

single quotes, and the pattern variables, that is, the formal macro parameters, lie outside the quotes.

Within the transcription text, PM allows variables local to that text and several expansion-time operations, such as assignments, logical and arithmetic operations, conditions, and loops. Pattern variables from the pattern component of the declaration are referred to in the macro body by their names rather than by some numbering scheme, as in TEX.

The expander component of Sassa's macro processor fits the ML/I model. Source text is treated as a stream of tokens. Encountering a macro name triggers the search for a matching macro pattern, taking into account the balancing of grouping symbols such as pairs of parentheses. The macro body is executed causing the generation of a transcription. The transcription is then scanned for generated macro calls. Macro calls appearing as arguments to other calls are processed first; PM is call-by-value.

Actual arguments are associated with the formal parameters that make up the macro pattern. In the sample **case** macro pattern, the symbols enq, unit, and outunit are formal parameters. As indicated, there can be one or more occurrences of unit expressions in an actual macro call. They are referenced within the macro body as unit(1), unit(2), and so forth. There are also some built-in system functions that yield values such as the number of occurrences of a specified macro parameter in a macro call and the n^{th} delimiter given an expression that computes n. A set of macro language constructs is available for generating output text, performing conditional tests, making assignments to macro variables, and looping.

Macros are scoped similarly to identifiers in lexically scoped programming languages. The only difference being that a macro is in effect from the lexical point of its declaration to the end of the block but not before. This is because PM is a one-pass macro processor.

Sassa concludes his paper with a discussion of the weaknesses and limitations of his macro processing system. He notes the deficiency of his system for error checking and recovery, the impossibility of infix macro calls, and the designation of macro names as reserved words.

2.5. Design Principles

In a paper published in 1985 [74], Triance and Layzell espouse a set of design principles for macro processors in high-level languages. Their work has much the same flavor as our Chapter 4; they identify fifteen design principles that are to be regarded as guidelines for any language enhancement scheme. Furthermore, they report that they have designed and implemented a macro processor for COBOL in accordance with their recommendations. In Section 4.10, we present their principles and discuss the relationship of their work to ours.

3. Syntactic Extensions in Lisp

The rise of macro systems in the Lisp-family came out of specific needs for expressing operations that could not be expressed with ordinary functions. In this chapter, we trace that development, culminating with a discussion of the details of the syntactic extension system in Common Lisp. Along with the historical information, we discuss some of the uses to which macros are put in Lisp programming, emphasizing why they are needed.

3.1. FEXPRs

The inadequacy of functions for expressing all desired programming abstractions was apparent early on to those working with Lisp. The first attempt at remedying this deficiency was the creation of FEXPRs (McCarthy, *et al.*, [47]). For example, it is impossible to define `QUOTE` with the standard function mechanism because when such a function is applied its argument is evaluated. Additionally, such forms as `AND` and `OR` that not only determine boolean values but also specify that some arguments may not be evaluated are not possible with standard function application.

In early Lisp systems, the values of identifiers are stored on a property list. Special properties are selected whenever an identifier appears in the function position of an application. A normal function, represented by a `lambda` expression, is stored under the `EXPR` property. A FEXPR, also represented by a `lambda` expression, is stored under the `FEXPR` property. It is significant that Lisp attaches the notion of

EXPR or FEXPR to the identifier. Thus one speaks of "the FEXPR `foo`" or of "the EXPR `bar`." An identifier can either be an EXPR or a FEXPR, but not both.[6]

During the evaluation of a non-atomic expression with a FEXPR in the first position, the `lambda` expression is applied to the unevaluated list of actual parameters, taken as one argument. This provides a way of having an arbitrary number of arguments and of explicitly controlling when they are evaluated. Examples of the definitions of the three FEXPRs `QUOTE`, `AND`, and `OR` are given in Figure 3.1.[7]

Although they provide additional expressive power, FEXPRs are unsatisfactory in at least two respects. Principally, it is impossible to give a compositional denotational semantics of a program that contains calls to `EVAL`. Secondarily, there is a dynamic binding problem with FEXPRs. Suppose that calls to `EVAL` are made so that the most recent bindings of all identifiers comprise the environment that `EVAL` itself uses.[8] Then, if an assignment is made to the variable used as the formal parameter of a FEXPR, subsequent instances of that variable refer to the new value and not to the original FEXPR input.

For example, consider the behavior of the FEXPR `AND` (Figure 3.1), when used as

```
(AND (PROGN (SET! X '(1 2 3)) T) 4).
```

[6] In some Lisp systems, for example Franz Lisp (Wilensky, [78]; Foderaro, *et al.*, [25]), FEXPRs are implemented as an `nlambda` expression instead of being a regular `lambda` expression attached to a special property. An `nlambda` expression is similar to a `lambda` expression except that there can only be one formal parameter. When an `nlambda` expression is applied to some arguments, they are placed, unevaluated, in a list. This list is bound to the formal parameter of the `nlambda` expression, and its body is evaluated.

[7] For improved readability, we frequently use brackets instead of parentheses in Lisp or Scheme code when the surrounded text is not an application. The two are operationally equivalent.

[8] This is, in fact, what happens in a dynamically scoped Lisp. In lexically scoped Common Lisp, `EVAL` uses the current dynamic environment and a null lexical environment.

```
(DEFINE-FEXPR QUOTE
   (LAMBDA (X) (CAR X)))

(DEFINE-FEXPR AND
   (LAMBDA (X)
      (COND [(NULL? X) TRUE]
            [(NULL? (CDR X)) (EVAL (CAR X))]
            [(EVAL (CAR X)) (EVAL (CONS 'AND (CDR X)))]
            [ELSE FALSE])))

(DEFINE-FEXPR OR
   (LAMBDA (X)
      (COND [(NULL? X) FALSE]
            [(NULL? (CDR X)) (EVAL (CAR X))]
            [(EVAL (CAR X))]
            [ELSE FALSE])
```

Figure 3.1. *The Definitions of three FEXPRs.*

The third COND clause causes the evaluation of the expression

```
(EVAL (CAR X))
```

where X is the list

```
((PROGN (SET! X '(1 2 3)) T) 4).
```

EVAL causes the variable X to be set to the list (1 2 3). Then the evaluation of the result part of the clause is equivalent to evaluating (AND 2 3). The original AND expression has been lost.

The version of EVAL described in the *Lisp 1.5 Programmer's Manual* (McCarthy, *et al.*, [47]) does not have this second problem. It takes two arguments, the expression to be evaluated and the environment in which this evaluation is to occur. The same two arguments are passed to the FEXPRs, the second one being the dynamic environment in existence at the time of the FEXPR call. The environment is sup-

plied by the system during the interpretation of FEXPR calls. It is then passed to EVAL within the FEXPR body. This assures that there is no confusion about which variable bindings are intended.

3.2. Macros

Macros provide a solution to the problems caused by having explicit calls to EVAL within user code of FEXPRs. Unfortunately, in older Lisp systems, they are perceived as another kind of function; for property-list-based Lisp systems, a third function property is included, MACRO. Macros are described as identifiers associated with functions of one argument, the entire unevaluated macro call. These functions are STFs.

The original Lisp macro processing systems are similar to the simple system we present in this section. However, in those Lisp systems implemented by interpretation, a separate syntactic analysis phase is often not included. Macros are expanded as the interpreter encounters them. Other variations on macro processing systems are possible; of interest are the ones described in the Common Lisp book (Steele, [70], pp. 143–152) and in the Scheme book (Dybvig, [20]). Harrison describes the list-processing language BALM, based on Lisp 1.5, that contains a macro system [32].

With respect to the steps in program analysis, there are several ways the implementation of a language can be organized. Usually, at least one intermediate language is designed. This language represents the program midway between syntactic analysis and either interpretation or compilation. For Lisp systems the translation from *Source* to *Intermediate* includes another language, *Syntax-tree*. The sequence of translations might proceed as

$$Source \xrightarrow{\text{Reader}} Syntax\text{-}tree \xrightarrow{\text{Parser}} Intermediate.$$

The parser expands syntactic extensions and performs other kinds of syntactic anal-

ysis, producing a program representation that can either be interpreted or compiled. In creating such a system, specific constructions in the language are recognized as fundamental to the definition of the language. These essential operations make up the *core* of the language. They are what is left after expansion of syntactic extensions. For example, the goto, the assignment statement, and some form of conditional operation are core forms in several languages. However, if a language implementor decides to implement the **while** statement

while ⟨expression⟩ **do** ⟨statement⟩

by first transcribing it into a semantically equivalent labeled statement using a conditional test and a goto, then we say that *for that implementation* the **while** statement is not a core form. It is implemented by a syntactic extension to the core language.

Here, we select a Scheme core with the following capabilities: identifier dereferencing; `quote` expressions, enabling the inclusion of constants within programs; `lambda` expressions, making functional abstraction possible; `if` expressions, permitting conditional evaluation of expressions; `set!` expressions, allowing for the assignment of values to lexically bound identifiers; and applications. The concrete syntax of this language is presented in a modified BNF grammar in Figure 3.2. The ellipsis indicates zero or more occurrences of the immediately preceding form; its use here is equivalent to the Kleene star of regular expressions.

The grammar is ambiguous; special forms have the same syntactic structure as applications. We resolve the ambiguity by treating the four symbols `quote`, `lambda`, `if`, and `set!` as reserved words. However, there is no requirement that all special forms be recognized by a reserved symbol as their first component. It is just as acceptable from a language designer's point of view to allow any expression syntactically distinct from identifier references and applications to be a special form.

```
⟨expression⟩ ::=
    ⟨identifier⟩ |
    (quote ⟨s-expression⟩) |
    (lambda ⟨formals declaration⟩ ⟨expression⟩ ⟨expression⟩ ... ) |
    (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩) |
    (set! ⟨identifier⟩ ⟨expression⟩) |
    (⟨expression⟩ ⟨expression⟩ ... ) |
    ⟨syntactic extension⟩

⟨formals declaration⟩ ::= (⟨identifier⟩ ... )

⟨syntactic extension⟩ ::= (⟨keyword⟩ ⟨syntactic component⟩ ... )
```

Figure 3.2. *An Extended BNF Grammar for a subset of Scheme.*

We choose not to permit this for two reasons: it is more convenient to have all special forms signaled in the customary fashion, and programs are easier to read if the type of any legal special form can be determined by looking at its first position. In any event, we reserve the symbols used to indicate special forms, not permitting them to be used as identifiers.

We assume that the abstract syntax of our core Scheme is the same as its concrete syntax. A simple parser for our language that includes a mechanism for the transcription of syntactic extensions is in Figure 3.3. The function `type-of` is responsible for checking the top-level syntax of the input expression. For example, it checks that a `set!` expression is composed of three parts: the keyword `set!`, an identifier, and a sub-expression. It does not check whether that sub-expression is legally formed. If an expression passes this cursory syntax check, `type-of` returns a tag indicating the kind of expression.

```
(define parse
   (lambda (exp)
      (case (type-of exp)
         [identifier exp]
         [quote-expression exp]
         [lambda-expression
            `(lambda ,(formals exp) ,(parse (body exp)))]
         [if-expression `(if ,(parse (test-part exp))
                             ,(parse (then-part exp))
                             ,(parse (else-part exp)))]
         [set!-expression
            `(set! ,(identifier-part exp) ,(parse (body exp)))]
         [application (map parse exp)]
         [syntactic-extension (parse (transcribe exp))]
         [else (error "◈nrecognized expression: " exp)])))
```

Figure 3.3. *A parser for a subset of Scheme.*

Our model for a syntactic extension mechanism is based upon thinking of the entire set of syntactic extensions as a grammar. Each production in the grammar corresponds to a particular syntactic extension. The left-hand side of each production specifies the syntax used by a programmer in a macro call. The right-hand side of each production describes a program semantically equivalent to the macro call in terms of the language's core syntax or other syntactic extensions. This set of productions comprises the *syntax table*. For example, if we were to implement the `while` statement as a syntactic extension, it might show up in the syntax table as the production of Figure 3.4.

The implementation of a syntactic extension system can be made so that users think of declaring macros only in terms of extending the syntax table. However, most Lisp systems do not foster this perspective. Instead, the process of declaring new macros involves writing STFs. The user thinks of a *macro environment* that

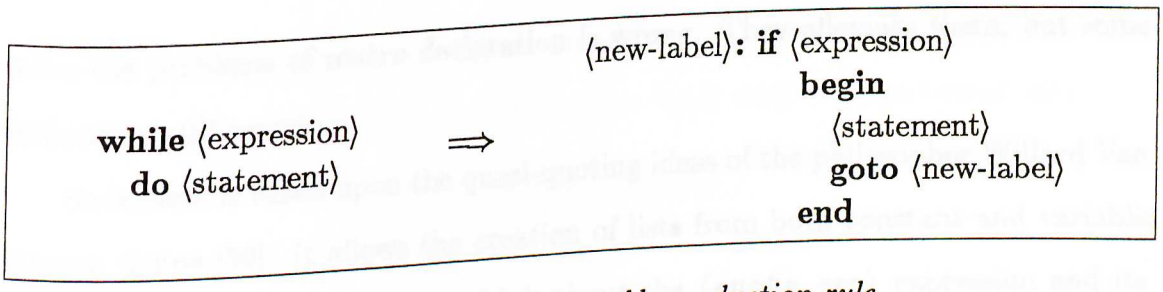


Figure 3.4. *A syntax table production rule.*

contains associations between macro names and STFs. Temporarily adopting this point of view, we can define the function `transcribe`, which effects the naïve transcription of macro calls in Figure 3.3, as

```
(define transcribe
   (lambda (form)
      ((macro-env-lookup (macro-name form))
       form)))
```

Macros are usually declared in much the same manner as we have illustrated. The problems with readability that we previously noted have always plagued macro writers and readers. After presenting the macro declaration of a `FOR` looping construct, the Lisp Machine book goes on to say:

> The main problem with the definition for the `for` macro is that it is verbose and clumsy. If it is that hard to write a macro to do a simple iteration construct, one would wonder how anyone could write macros of any real sophistication.
>
> There are two things that make the definition so inelegant. One is that the programmer must write things like "`(cadr x)`" and "`(cddddr x)`" to refer to the parts of form he wants to do things with. The other problem is that the long chains of calls to the `list` and `cons` functions are very hard to read.
>
> Two features are provided to solve these problems. The `defmacro` macro solves the former, and the "backquote" (`) reader macro solves the latter. (Moon, *et al.*, [54], p. 250)

Both `defmacro` and backquote have found their way into Common Lisp where they, too, are the principal tools used for specifying STFs. The claim that these two tools

solve the problems of macro declaration is wrong. They alleviate them, but some difficulties still remain.

Backquote is based upon the quasi-quoting ideas of the philosopher Willard Van Orman Quine [59]. It allows the creation of lists from both constant and variable parts. To understand it we first think about the `(quote` *exp*) expression and its reader abbreviation *'exp*.

There are several ways to build the list `(a b c)`. The one that is relevant to us is

```
(list 'a 'b 'c).
```

Since the use of `quote` expressions becomes cumbersome, even when abbreviated, this kind of list building is often written as `'(a b c)`.[9]

The building of a list containing some constants and some variable parts, as

```
(list 'a (cons x y) 'b z),                    (1)
```

also has a special reader abbreviation. Not having to quote individual components of a list is useful in building a list of constants. A similar facility is desirable for lists with variable parts. We use the character macros ` and , to write

```
`(a ,(cons x y) b ,z).                        (2)
```

In essence, backquote quotes everything except those forms preceded by a comma. The forms (1) and (2) are equivalent in the sense that the reader transforms the backquote expression into a function call that, when run, builds the appropriate list. However, not all implementations of backquote work the same way. Each makes lists with `list`, `cons`, `append`, and other list building functions as it sees

[9] The two are not necessarily operationally equivalent. The first causes the Lisp system to allocate new memory cells each time it is evaluated, thus creating many similar lists that are not `eq?` to one another.

fit. Thus the operational aspects of backquote expressions may vary from system to system. For example, the expression `‘(,a b c)` may be transformed into

```
(cons a '(b c))
```

or

```
(list a 'b 'c).
```

There is not enough standardization among Lisp systems to be certain of what one will get. Furthermore, if the system bases its reader transcriptions on functions that can be redefined by the user, disaster may strike when these functions are changed.

The atsign character `@` has a special meaning within backquoted expressions. The character sequence `,@` must be followed by an expression that evaluates to a list. The list is spliced into the list being built by backquote. For example, if `a` is bound to the list `(1 2 3)`, then `‘(0 ,a 4)` represents something like

```
(list 0 a 4),
```

whereas `‘(0 ,@a 4)` stands for an expression similar to

```
(cons 0 (append a (list 4))).
```

The construction dot-comma may be used to splice a list onto the end of a backquote expression, as in `‘(0 . ,a)`. In some Lisp systems there is also comma-dot that means the same thing as comma-atsign, but appends destructively using `append!` instead of `append`.

As the Lisp Machine book says, “This is generally found to be pretty confusing by most people.” Yet, there are people who, with practice, have become quite proficient at the use of backquote. They produce expressions in which the backquotes are nested several levels deep. It all works, but the shifting of levels between what is constant and what is not is tricky. We have seen that there are possible variations in the meanings of one-level backquote expressions; the situation is even worse

```
(macro let
   (lambda (form)
      (cons (cons 'lambda
                  (cons (map car (cadr form))
                        (cddr form)))
            (map cadr (cadr form)))))

(macro let
   (lambda (form)
      `((lambda ,(map car (cadr form)) ,@(cddr form))
        ,@(map cadr (cadr form)))))

(macro let
   (lambda (form)
      (apply
         (lambda (defs . bodies)
            `((lambda ,(map car defs) ,@bodies)
              ,@(map cadr defs)))
         (cdr form))))
```

Figure 3.5. *The macro* `let` *declared without and with use of backquote.*

when backquotes are nested. Because of these uncertainties, we do not recommend the backquote for anything but the simplest list building, for example, as used in the parser of Figure 3.3.

Furthermore, while providing some relief from hard to read STFs, backquote does not help when portions of the transcription need to be built by a mapping function. For an example, consider the three declarations of the macro `let` in Figure 3.5. We repeat the version from Figure 1.3 for ease of comparison with the others. The second version uses backquote; the readability is somewhat improved over that of the first. The third version uses the destructuring capability of function application to eliminate some of the car-ing and cdr-ing.

The special form `defmacro` comes in a variety of versions, depending upon which Lisp system one is using. We describe a simplified form of the one in the Common Lisp book, ignoring the optional declaration and documentation parts.

Our `defmacro` expression has the syntax

(`defmacro` *name lambda-list STFform ...*)

where, once again, the ellipsis indicates zero or more occurrences of *STFform*. The *lambda-list* is a formal parameter list, although for `defmacro` the notion is extended beyond what is normally used as a formals list for Common Lisp `lambda` expressions. All the expressiveness of the `lambda` expression list is maintained; the "lambda-list keywords" `&optional`, `&rest`, `&key`, `&allow-other-keys`, and `&aux` can all be present. There are three additional markers: `&body`, `&whole`, and `&environment`. The first of these is identical in purpose to `&rest`, but indicates the remainder of the form is to be indented for output and editing in a special way. The next, `&whole`, allows a formal parameter to be associated with the entire macro call during transcription. The last enables the specification of a lexical environment in which macro calls are to be transcribed.

The difference between `lambda` expression lambda-lists and `defmacro` lambda-lists is that those for `defmacro` provide for destructuring. We quote from the Common Lisp book:

> Anywhere in the lambda-list where a parameter name may appear, and where ordinary lambda-list syntax ... does not otherwise allow a list, a lambda-list may appear in place of the parameter name. When this is done, then the argument form that would match the parameter is treated as a (possibly dotted) list, to be used as an argument forms list for satisfying the parameters in the embedded lambda-list. ([70], p. 146)

Steele's point is that the lambda-list declares the formal parameters used in the body of the STF—that part described by *form* in the `defmacro` expression. These

formal parameters are associated with the components of a macro call in much the same way the formal parameters are bound to the numbers by using `apply` in the expression

```
(apply (lambda (a b c) whatever) '(1 2 3)).
```

The difference is that more than mere top-level destructuring is possible. For example, assuming this is implemented by a function `defmacro-apply`, the expression

```
(defmacro-apply (lambda (a (b c) . d) whatever) '(1 (2 3) 4 5 6))
```

binds `a` to 1, `b` to 2, `c` to 3, and `d` to the list `(4 5 6)`.

With this assumption, and ignoring the complication of the lambda-list keywords, we can specify the transcription of

```
(defmacro name lambda-list STFform ...)
```

as

```
(macro name
   (lambda (form)
      (defmacro-apply
         (lambda lambda-list STFform ...)
         (cdr form)))).
```

The destructuring capability goes a long way toward eliminating the unpleasant car-ing and cdr-ing that appears in STFs. For example, with `defmacro` we can recast our declaration of the form `let` as

```
(defmacro let (defs . bodies)
   (cons (cons 'lambda
               (cons (map car defs) bodies))
         (map cadr defs))).
```

Using both backquote and `defmacro`, we write it as

```
(defmacro let (defs . bodies)
   `((lambda ,(map car defs) ,@bodies) ,@(map cadr defs))).
```

We still have not gotten rid of the calls to `map`, nor does this form give us explicit information about the syntax of macro calls. Furthermore, we cannot easily see that the `defs` component consists of identifier-expression pairs.

3.3. The Use of Macros in Lisp

Pitman [58] describes the use of special forms—macros, `nlambda` expressions, and FEXPRs—in Lisp. In this section we identify eleven uses of syntactic extensions that are commonly found in Scheme programming. When another programming language is embedded in Scheme, many of the other language's structures cannot be implemented conveniently or quickly without using macros. Functional abstraction, as powerful a tool as it is, is insufficient.

We ignore macros which interact with their expansion environment, the unsafe ones. We noted two possible uses for such macros in Section 1.3.1. The macros that are more than syntactic abstractions, those that do more than produce a transcription based on source text and known macro declarations, are dangerous. They have contributed to the disrepute macros have gained. If a compiler writer is tempted to use macros to gather compilation statistics, we suggest that he should find another means to do so, leaving macros as a tool for the user of his language.

The macro writer extends the base language that subsequent users, including other macro writers, see. It is, from the standpoint of these latter folk, as if the new special form had been part of the core language all along. The macro writer alters the language he started with.[10]

The notation for specifying macros consists of describing a sample macro call, in the form of a pattern, and using the pattern variables—indicated by italics—within a transcription specification. This form for the description of macros, which

[10] In our terminology, the *special forms* of a language comprise the core forms and any syntactic extensions. This differs from that used in the Common Lisp book.

has been in common use for some time (for example, a variant of it was used in the Revised Report on Scheme [71]), avoids the writing of STFs. As in previous examples of this sort, the ellipsis means that zero or more occurrences of the form immediately preceding it appear in macro calls.

3.3.1. ABBREVIATING FUNCTION CALLS. When a function, such as `parse` (Figure 3.3), is repeatedly tested at the keyboard, one finds it inconvenient to quote the test expression and to spell out the function name on each function call. A macro can be defined to remove the necessity of typing the quote. For example, the macro

(p *exp*)

can be declared to transcribe to

(parse '*exp*).

This technique applies to many other situations in which the user wants to avoid typing the same thing over and over. We are distinguishing this case from the other uses of syntactic abstraction because of the way it arises. It usually comes up when convenience is desired. It is the same sort of abstraction students make for themselves when forming private abbreviations during note-taking.

3.3.2. SUPPLYING ARBITRARY NUMBERS OF ARGUMENTS. Unless we choose to use rest parameters, the number of arguments to our functions is set at the time they are closed. With macros, the number of components of a macro call is arbitrary. We can declare a macro

(plus *n1 n2* ...)

to sum a sequence of numbers of any length. The transcription of each macro call is based on the number of numbers present. If there is just one, as in

(plus *n1*),

the transcription is *n1*. If there are more than one, the transcription is

```
(+ n1 (plus n2 ...)).
```

Because of rest parameters, we do not recommend the declaration of the particular syntactic extension `plus`. It is an abuse of syntactic extensions to employ them when other language features will serve. The expression

```
(apply plus '(1 2 3))
```

fails to have a value if `plus` is a macro. When the sole reason for declaring a macro is to have an unlimited number of arguments, all of the same type, we suggest defining a function using rest parameters.

3.3.3. CONTROLLING THE ORDER OF EVALUATION OF EXPRESSIONS. We have already seen an example of a macro whose arguments are Scheme expressions that are to be evaluated in a specific order: `begin` (Figure 1.6). Two other special forms that fit this category are `or` and `and` (Figure 3.6). They cannot be defined as functions because all arguments to applications are evaluated, and this evaluation occurs in an unspecified order.

The current version of `or` uses a gensym to avoid the capturing problem. The italic "*gsym*" is a *new pattern variable*. It does not appear in the corresponding pattern part of the declaration, yet it is used in the transcription part as if it did. We intend it to represent a system-generated symbol, one that is guaranteed not to be identical to any user-created identifiers.

The special form `define` is another notation for `set!`. As such, the first component—an identifier subexpression—is not evaluated. We declare

```
(define id exp)
```

to be equivalent to

```
(set! id exp).
```

```
          (and) ⇒ true
       (and exp) ⇒ exp
(and exp1 exp2 ...) ⇒ (if exp1 (and exp2 ...) false)

          (or) ⇒ false
       (or exp) ⇒ exp
(or exp1 exp2 ...) ⇒ (let ([gsym exp1]) (if gsym gsym (or exp2 ...)))
```

Figure 3.6. *The Scheme macros* and *and* or.

The evaluation is controlled so the identifier part is not evaluated.

As a final example, consider the macro when. Often, using a conditional expression that contains no alternative part is desirable. We declare

(when *predicate exp1 exp2* ...)

to transcribe to

(if *predicate* (begin *exp1 exp2* ...)).

3.3.4. USING SYNTACTIC FORMS THAT ARE NOT EXPRESSIONS. The Lisp form cond uses list components—the cond clauses—that are not themselves expressions. They cannot be evaluated as arguments to a function. However, we can declare cond as a syntactic extension (Figure 3.7).

This macro is also interesting because of the presence of the auxiliary keyword else. Some macros employ symbols other than the macro as part of their syntactic structure. Unlike macro names, auxiliary keywords need not appear in all macro calls. However, when present, they distinguish the call as being of a particular form and affect transcription accordingly.

3.3.5. PERFORMING COSMETICS. Sometimes the addition of a few non-operational keywords or the rearranging of the parts of an expression makes a big difference

```
                                (cond) ⇒ unspecified
          (cond [else exp1 exp2 ...]) ⇒ (begin exp1 exp2 ...)
              (cond [exp] clause ...) ⇒ (or exp (cond clause ...))
(cond [exp1 exp2 exp3 ...] clause ...)
                                    ⇒ (if exp1
                                          (begin exp2 exp3 ...)
                                          (cond clause ...))
```

Figure 3.7. *The Scheme macro* `cond`.

```
(let* ([id exp]) body ...)
    ⇒(let ([id exp]) body ...)

(let* ([id1 exp1] [id2 exp2] ...) body ...)
    ⇒(let ([id1 exp1])
        (let* ([id2 exp2] ...) body ...)
```

Figure 3.8. *The Scheme macro* `let*`.

in the readability of code. At other times, particularly when embedding a language in Scheme, we want to create special forms reminiscent of the components of that language. These situations call for the use of macros for cosmetic purposes to improve the appearance of the program text. The macros `let` and `cond` are good examples of cosmetic macros. Another is the macro `let*` (Figure 3.8), a more convenient way of writing a series of dependent declarations than explicitly nesting `let` expressions.

3.3.6. ALIASING. Related to cosmetics is the issue of renaming macros and core constructs. Implementors of models of imperative programming languages may wish to use the symbol `:=` for an assignment operator instead of Scheme's `set!`.

The non-evaluation of the identifier that is assigned prohibits a functional alias for `set!`. Therefore, a macro is needed; it is very similar to the macro `define`; in fact, `define` is itself an alias for `set!`.

A natural question concerning aliasing is "What about aliasing the names of functions?" For example, many people prefer to rename the functions `car` and `cdr` as `first` and `rest`. They insist that they do not want to incur the cost of the multiple function calls they would get each time they called `first` if it were defined as `(define first (lambda (x) (car x)))`. The way around this is to write `(define first car)` instead of declaring `first` to be a macro that expands to a function call to `car`. In Scheme 84 [26], the two forms are identical.

Aliasing functions is an often-cited use for macros, yet we feel it is ill-advised. It blurs the distinction between syntactic and semantic extensions to a language. The user of the macro `first` wonders why `(map first some-list)` does not work while `(map car some-list)` does. The symbol `first` is meaningless by itself. The macro is `(first` *exp*`)`; the symbol `first` only makes sense when used as the initial component of a macro call.

3.3.7. CHANGING THE SEMANTICS OF SPECIAL FORMS. At least one version of Scheme, Scheme 84 [26], allows the declaration of macros that replace the existing core forms. Using it, we can create a `lambda` macro semantically equivalent to the accustomed form except that it prints tracing information whenever a closure formed from evaluating an expression that uses it is applied. We recompile the code in which functions that we want to trace are closed. The text of the function definitions does not have to be changed.

Related to changing the behavior of core forms is altering the behavior of existing macros. Because a syntactic extension mechanism is in the hands of the user, he is free to revise what each macro does. He is even able to remove the declarations

```
(recur name ([id form] ...) exp ...)
    ⇒ ((rec name (lambda (id ...) exp ...))
       form...)
```

Figure 3.9. *The Scheme macro* `recur`.

of specific macros.

3.3.8. HIDING IMPLEMENTATION DETAILS. It is often unnecessary to burden the users of a particular system with the details of how specific language structures are designed. For example, the semantics of `let` expression can be described without recourse to the fact that they are implemented by an application of a closure to some arguments. The user can think about identifier declarations and formulate the scoping rules for declared bindings in the `let` expression. This is even more obvious with macros such as `inf-loop` (Figure 1.6) and `recur` (Figure 3.9).

We can think of `recur` in terms of its underlying application and assignment expressions. The assignments show up when we expand the `rec` expression (Figure 1.6). But, most people find it more natural to think of the iteration of a portion of code. Macros enable the forming of mental abstractions. Expansion specifications have their value, but mental abstractions are the tools of good programmers.

3.3.9. AVOIDING THE DUPLICATION OF TEXT. Sometimes several structures are built out of the same pattern. A good example is the object data type. The basic schema is the same whether one is defining a stack, queue, set, or any other object data type. So, it is possible to textually abstract that pattern from the instances of it. That abstraction is used to declare a macro that generates object data types when supplied with the appropriate functions and local identifiers (Figure 3.10).

```
(make-odt name init [msg action] ...)
        ⇒ (define name
            (lambda ()
              (rec self
                (let (init)
                  (let ([msg action] ...)
                    (lambda (m)
                      (cond [(eq? m 'msg) msg] ...)))))))
```

Figure 3.10. *A syntactic extension which creates an object data type.*

For example, to create a stack object data type, we write

```
(make-odt stack
   [stk '()]
   [push! (lambda (x) (set! stk (cons x stk)) self)]
   [pop! (lambda ()
           (if (null? stk)
               (error "Attempt to pop empty stack")
               (let ([v (car stk)])
                 (set! stk (cdr stk))
                 v)))]
   [empty? (lambda () (null? stk))]).
```

To define a queue, record, or other data structure, we change the name, the initialization component, and the access functions. The initialization component describes the internal representation of the structure. The names of the access functions become the messages sent to the object data type.

3.3.10. SIMULATING DYNAMIC VARIABLES. It is possible to use syntactic abstraction and Scheme's lexical scoping to give the appearance of dynamically scoped identifiers. Any free identifiers in the expansion of a macro can be lexically caught within the scope of identifiers bound in the context surrounding the macro call.[11]

[11] This is not the capturing problem. There, the troublesome bindings occur

```
(letrec ([id exp] ...) body ...)
⇒ (let ([id '*])
    (set! id exp) ...
    body ...)
```

Figure 3.11. *The Scheme macro* `letrec`.

A simple example illustrates this. Supposing the macro `dynamic-a` is declared as

```
(dynamic-a) ⇒ (bar a).
```

We can place a call to `dynamic-a` in the context

```
(let ([a 7]) (dynamic-a)).
```

Then, blurring the distinction between function and macro calls, we can think of the identifier `a` within the expansion of `(dynamic-a)` getting the most recent value of any binding to `a`, in this case 7.

3.3.11. SUPPORTING A PHILOSOPHICAL VIEW OF PROGRAM STRUCTURE. Some people prefer to program without the use of side effects, specifically avoiding `set!` expressions. Therefore, when it is desired to form circular structures such as the environments for recursive functions, some other means must be found for expressing their definition. Yet, in Scheme, a common way to build circular structures is to use the assignment expression. To resolve this paradox special forms like `rec` and `letrec` (Figure 3.11) have been designed. The user of these forms ignores the assignment expressions in their expansions, thus programming from his philosophical viewpoint. It is an accepted reality of programming in Scheme that the implementation details require side effects, but the programmer is free to believe

---

within the generated expansion of a macro, not the surrounding context of a macro call.

that he is not using them. Macros again are being used to control the set of mental abstractions possessed by the programmer.

4. Design Principles

We begin the design of our macro processing system for Scheme by enumerating twelve principles. We treat them as recommendations, using them to guide the implementation of our declaration and expansion system components in Chapters 5 and 6. Also included is a section devoted to similar design decision work done by Triance and Layzell [74]. Appendix A contains a complete list of the design principles stated in this chapter.

4.1. What the Macro Writer Does

We start with a discussion of the activities of someone who wishes to use a syntactic extension package with which he is already familiar. There are two approaches to the creation of new macros: *abstraction over repeated textual forms* and *designing new language components.*

To create the object data type macro of Section 3.3.9 the repeated form evident in many different data types is abstracted. We recognize a repeated structural pattern, anticipating its later recurrence. Wishing to avoid duplicating that structure, we compare existing portions of code that fit its pattern determining what varies from one piece to another and what is fixed. This results in the formulation of a set of parameters and a structure for the desired macro transcription. The macro is then implemented using the system's declaration facilities.

Designing new language components usually arises out of different circumstances. For example, a programmer accustomed to using a `while` loop may find his expressive powers diminished when he trying to use a system that does not have one. A new syntactic extension with the desired semantics would restore his writing skills. In this situation, the macro writer thinks in terms of providing an operational semantics for his new form. Identification of the parts that change from one use of the macro to another and production of a transcription structure are still part of the process of declaring macros.

The boundary between the two approaches is not firm. They are two ways of viewing the same process. The distinction is in the motivation for declaring a new syntactic extension. In any case, there are three steps to do before ever declaring a macro: (1) recognizing the variable parts of the form, (2) designing the transcription structure, noting where the variable parts fit in the fixed form, and (3) designing a new syntax for the macro call enabling the later incorporation of actual arguments.

The process is complicated when the macro language is not suited for easy description of transcriptions or when the code for the bodies of macro declarations must contain conditional tests, variable assignments, iteration constructs, and so forth. We have already discussed some of the Lisp tools used in transcription texts; in the next chapter we present our tool. We state

Principle 1. *Since syntactic extension declarations always involve the specification of calling forms, transcription structures, and the pattern variables used in both, they should be made in such a way that the writer uses these three components as directly as possible.*

For example, a Lisp macro system that requires the explicit description of mapping operations in an STF, as in Figure 3.5, is not as good a system as one in which the mapping expressions are concealed. The implementor of a macro package should insure that declarations can be made as easily as possible. This involves studying

the typical uses of macros in his language, trying to figure out the best notation for the expression of transform functions. A different host language might require different macro declaration tools.

In some macro processors, like TEX and GPM, a pattern variable is chosen from a limited set of formal parameters. In others, like PM, a pattern variable is any symbol. The generality of systems like PM is better.

4.2. Syntactic Extensions

In designing and implementing a macro system for a programming language, a decision must be made whether macros are to be based on character strings or on syntax trees.

Basing the system on strings has the advantage of great flexibility. Macro writers are free to do as much as they please, creating macros that can be called from any location within program text. But this is too haphazard for a programming language based macro facility. Textual substitutions based on character strings are too permissive; they discourage clean code. The macro writer shares responsibility of sound language composition with the language's original designer. Hence, macros should fit into the overall mold of the host language. Macros should not be syntactic clutter imposed on top of a language.

A more sound approach is to disallow macro calls within identifiers, numbers, and so forth. Macro systems based on tokens, like ML/I and PM, have this restriction. However, they ignore the rest of the syntactic structure of the host language. A macro call can be made in a location in source code that ought to hold an expression, yet the macro could expand, for instance, into a command or part of an expression. In a string-based system designed to preprocess Algol, it might be possible to have one macro begin a block and another terminate it. This sort of thing happens frequently in TEX, one macro beginning a group and another ending it.

Another approach is to allow macros that replace only certain portions of source code such as the line-oriented macro processors like STAGE2. The trouble with line-oriented macros is that most modern programming languages do not have lines as syntactic units. To base a macro processor for such a language on lines of text emphasizes the syntactic role of lines, even where they may not have had any role other than making it easier for us to read our programs.

The best we can do toward integrating a macro processor and its host programming language is to have the processor operate on the syntactic structures of that language. Since these structures are more easily detected when the program is represented as a syntax tree than as a character string, we favor a macro processor based on syntax trees. We conclude that

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.*

This is the intent of Cheatham's SMACROS and Leavenworth's syntax-macros.

4.3. Syntactic Domains

When the semantics of a programming language is given in terms of a denotational semantics [72], various *syntactic domains* are specified for the syntactic constructs in the language. Typically, these include the domains of Commands, Declarations, and Expressions.

In the core language (Section 3.2), we recognize two syntactic domains: Declarations and Expressions. Figure 3.2 describes Expressions. The one place a declaration appears is in the formal parameter list of the `lambda` expression.[12] We have described the syntactic extensions to Scheme as if they were all syntactic extensions to the domain of Expressions. This restriction is implemented by not recursively

[12] A third domain is present in the grammar, the domain of S-expressions used as data within `quote` expressions.

invoking the parser on the formal parameter list or on the constant component of the `quote` expression.

Syntactic extensions can be made to the domain of Declarations or, for that matter, to the domain of S-expressions. If any syntactic domain is present in a language, we can create a syntactic extension facility for it. This involves more elaborate parsing, probably utilizing different parser functions, such as `parse-expression`, `parse-declaration`, and `parse-command`.

Following Leavenworth, if we allow macros for different domains, we expect their forms to be distinct. Just as there are **smacros** for statement macros and **fmacros** for function macros, we require unique declaration and expansion mechanisms for each kind of syntactic extension. We will not extend our language in this way, remaining content with expression syntactic extensions. But it is possible, and a design principle is

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.*

This means that an expression syntactic extension can never be used in place of the formal parameters list of a `lambda` expression, nor can one appear as the "argument" of `quote`. It also means that one cannot be used in place of the identifier within a `set!` expression. Even though identifiers are expressions, expression syntactic extensions can only be used where *any* expression is legal.

As noted in the previous section, macros based on syntactic entities are preferable to string-based substitutions. However, an expression macro should not expand into anything other than an expression. Mixing syntactic domains is not good. Thus

Principle 4. *A syntactic extension from domain X should expand to a complete term from domain X.*

We choose to limit ourselves to syntactic extensions over the domain of Scheme

Expressions because this is the convention for macro systems in Lisp. Expressions are the dominant syntactic category used in Lisp programming. The other types of syntactic extension do not have as full a set of common applications as do expression extensions. All the Lisp macros in the last chapter are expression macros.

4.4. The Core Language and Syntactic Extensions

When a language is implemented without a macro processor, the user of the language has a fixed set of special form constructs of which he needs to be aware. Once a macro processor is added, the set is no longer fixed. Every syntactic extension adds to the number of special forms.

There are instances of macros that introduce new free identifiers into the code that is to be interpreted or compiled. If an error condition involves one of these generated identifiers, the user needs to be cognizant of its source. The language implementor has the responsibility of informing the user of the existence of the macro processor, and the macro writer has the responsibility of informing him of the behavior of the macros if they produce unexpected effects.

The generated identifiers that cause the troubles are those that are generated free, as described in Section 3.3.10 on simulating dynamic variables. Binding, generated identifiers, provided that they do not cause the capturing problem, are not a hindrance. The user need not be aware that special form such-and-such is a macro unless it generates free instances of identifiers. Even with the knowledge that our implementation of Scheme has a macro processor, the user need not know that `let`, `and`, `or`, `cond`, `begin`, `rec`, `letrec`, and `recur` are included *as macros*. He needs to know that `dynamic-a` generates a free instance of the identifier `a`.

This leads us to consider what syntactic domain macro calls should resemble. We prefer that macro calls look like terms in the domain they represent:

Principle 5. *Syntactic extensions to domain X should be called in the same syntactic style as terms in domain X.*

Command syntactic extensions to Pascal comprising more than one statement should have an initial keyword corresponding to **begin** or **case**, should have statements separated by semi-colons, and should conclude with the keyword **end**. They should fit the established model for commands in Pascal.

If there are expression syntactic extensions in Scheme, they should be similar in structure to other expressions. After all, a reader of Scheme code who has created suitable mental abstractions for the macros need not be informed that any macros are present (besides those generating free identifiers). One should be able to read code and not be tripped up by special syntax that indicates a macro call. Macro calls in Scheme should look like other special forms; they are all expressions.

Ideally, any syntactic structure conforming to the style of the host language and which is not a core form should be available as a potential macro call. Macro calls do not need keywords in the first position of a list, nor do they even have to be lists. A syntactic extension could be represented as a single reserved symbol or as a list in which the keyword appears in some other position than the head. The expression `sum23` might be replaced by its transcription `(+ 2 3)` and the expression `(4 times 5)`, where `times` is a keyword, might be replaced by its transcription `(* 4 5)`. Nevertheless, we will restrict macro calls to be in the form of a list with a keyword—the macro name—in the first position.

The real purpose of a warning marker is to enable the easy detection of macro calls by the language processor. If such a marker is inconsistent with the style of the base language, its use violates Principle 5. A warning marker only tells the reader that a macro is coming. Besides, in most Scheme programming, a large part of the code is made up of calls to special forms that are usually implemented as macros. Forms such as `let`, `letrec`, `cond`, `and`, `or`, `rec`, and `recur` are often implemented as macros in Scheme systems. The programmer should not have to

declare each time, "I'm going to use a macro now." We conclude that macro calls in a programming language should be made in free mode.

4.5. The Syntax Table

Section 3.2 introduced the idea of a syntax table for macros. The usual way to implement a macro system in Lisp involves having a macro environment in which macro names are bound to STFs. It is fine to know this as a detail of the implementation, but it is not what the macro writer should think of. Instead, syntactic extensions should be regarded as being specified by a set of productions. The macro writer should not have to worry about the details of how each transcription is effected. He should not have to know whether a list is built with the function `list` or with repeated calls to the function `cons`. We say that

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.*

The specifications of the Lisp macros presented in Chapter 3 adhere to this idea. Our primary concern is always with the syntax table abstraction and not with the details of STFs. In Section 4.8 we discuss the composition of STFs.

4.5.1. RESERVED WORDS. There are four major points of view concerning the status of keywords as identifiers.

1. *The set of keywords is disjoint from the set of identifiers.* When a macro is declared, the host language is extended, changed. Part of this change is the introduction of new reserved words. No matter how much we would like to use the symbol `else` as an identifier, once we have stated it is part of the syntactic structure of our language, it is that and nothing more. Even outside of `cond` expressions, the symbol `else` should not appear as an identifier. Moreover, the set of keywords is always finite; the set of identifiers is unbounded.

2. *Macro names and the words indicating core forms may not be identifiers but auxiliary keywords may.* This attitude is useful when the macro processor is based on streams of tokens, as in ML/I and PM. There are two difficulties with it in a Lisp macro system. First, it makes the macro name something special. Even though a macro pattern matcher might use the name to make a quicker search of the syntax table, the keyword is just part of the syntax of macro calls. Despite the implementation distinction between macro names and auxiliaries, their roles are the same in a syntax-tree based system. Their presence in certain spots within list structures designates the structures as syntactically legal expressions. In ML/I and MP the macro name is fundamentally different from the other markers used as delimiters. The second problem is that it gives rise to a possible confusion of intention, as in the use of `else` in

```
(let ([else false])
  (cond [else else])).
```

3. *Macro keywords may be used as identifiers but core form keywords may not.* This point of view stresses too much the distinction between macros and core forms. As mentioned before, the only time this disparity is important to the language user is when macros generate free instances of identifiers. Whether we choose to implement the special form `let` as a macro or as a core form should be immaterial to the programmer.
4. *All symbols can be used as identifiers.* Proponents of this opinion feel any other is too restrictive. They like the expressive power of being able to use any symbol as an identifier, that is any symbol other than the few character strings they have in their set of "improper symbols." In spite of the confusion that must be resolved when reading code such as

```
((lambda (lambda)
   (lambda (a b) (lambda (a) b)))
 (lambda (f x) (g y))),
```

they feel their expressive power is being reduced when restrictions are imposed. They are also worried about the growth of the set of reserved words. We believe that there are design problems with any language in which the set of reserved words is overlarge. Care must be taken that any new syntactic extensions are justified in terms of the expense of having more special forms.

Considering the merits of each of these positions, we conclude that

Principle 7. *The set of keywords should be disjoint from the set of identifiers.*

Declaring a new macro removes at least one symbol from the set of identifiers and reserves it as a keyword. There also may be auxiliary keywords such as the keyword `else` in the declaration of `cond` (Figure 3.7). They, too, are not identifiers.

It is unfortunate that most terminals cannot readily display symbols in two different fonts, as in papers and descriptions of Algol and Pascal. Then it would always be clear which symbols were reserved and which could be used as identifiers. The symbol **lambda** could be the reserved word, the symbol `lambda` the identifier. If the objections people raise to a particular point of view are eliminated by the possibility of using two different fonts, we discount their argument, writing it off to inadequate equipment. This is the *two font test.* Until satisfactory hardware is available for all systems using whatever language we are working with, we satisfy ourselves with the rule that keywords can never be identifiers.

The macro system should keep track of all reserved words, for example with a predicate `reserved-word?` defined over symbols. This set of keywords only changes when new macros are being declared or when the declarations of macros no longer needed are being removed. Thus

Principle 8. *During the pre-execution processing of any program text, the set of keywords and the contents of the syntax table should remain constant.*

This principle denies the possibility of macros local to specific parts of program

expressions, macros declared with a facility similar to Common Lisp's `macrolet`. We discuss this issue in Section 7.3.4.

4.5.2. THE RECOGNITION OF MACRO CALLS. Macro calls are recognized by matching the potential call against all of the left-hand sides of the productions in the syntax table. The keywords and the list-structure of the call play a major role in determining whether a potential call is in fact a call and what transcription belongs to the call. Since this is a form of pattern matching, we refer to the left-hand sides as *patterns*. Therefore

Principle 9. *Macro calls are matched against all declared patterns in the syntax table in some specified order.*

The specified order may be a best fit situation, as for the macros of STAGE2, or according to some other plan. The implementor of the macro system is free to choose what he thinks appropriate.

We have chosen the following plan because of the ease of implementing it and its relative efficiency in finding a successfully matched pattern. Our method avoids ambiguity and is easy to understand. A search through an unorganized, large set of production rules looking for the best fit requires examining each pattern. To speed this process up, all declared patterns with the same macro name are grouped together in the syntax table. Attempts at pattern matching a potential macro call against the patterns with the same macro name proceed until the first successful match is found. The corresponding production is taken as the one that describes the transcription.

The macro writer needs control over the order the productions are entered in each group. For example, the productions for the Scheme macros `and`, `or`, and `cond` must appear in their respective groups in the order described in Figures 3.6 and 3.7. For both `or` and `and`, the pattern of the second production is a special case of the third.

The keywords and the list structure of the macro call serve as delimiters. The pattern variables serve as formal parameters. Thus, there is no limit on the number of formal parameters.

4.6. Macro Languages

The issue of the composition of the macro language came up several times in Chapter 2. In the simplest macro assemblers, this language has no special forms. Macro transcriptions are declared by writing a character string that is not even parameterized. The next simplest macro processors provide for parameters but still specify transcriptions essentially by an implicit quasi-quoting of text, where the escaped variables are the formal macro parameters.

Moving toward more complexity, we see that some macro processors include forms for conditional expansion, the selection of alternate quasi-quoted texts based upon conditions during macro transcription. In some macro languages these forms are the same as corresponding forms in the host language. In others, there are special macro language commands that control expansion. Examples of this last kind are found in McIlroy's and Leavenworth's systems. McIlroy goes so far as to assert that anything that can be done during program execution should be possible during macro expansion.

Brown's SUPERMAC[13] concept ([10]; [9]) is based on abandoning "the idea of the separate stand-alone macro processor," and "bringing the macros to the users" by "embedding them within the user's own favourite programming language" ([9], pp. 433–434). Using whatever host language he chooses, Brown creates what are the equivalent of Lisp's STFs.

The macro language of TEX is TEX itself. Compared to the number of macro

[13] Coincidentally, "Supermac" is also the name of Henderson and Gimson's macro-processor [33].

calls made while using TEX, the number of calls to primitives is small. This gives the user the impression that almost everything is either literal text or a macro call. It is due to the primitives' extreme low level.

A weak feature of TEX is the lack of syntactic distinction between primitive control sequences and the replacement text of a macro. For example, the condition-als are usually not part of generated text. Instead, as part of the macro language, they control the flow of the transcription. However, other primitives and macro control sequences are part of the replacement text. Knuth only casually implies this difference by stating that the expansion of a conditional is empty ([38], p. 213). He does not discriminate between the expansion of macros and the interpretation of control sequences; the distinction between the low-level control sequences that actually typeset something and those that control the flow of further processing is not made clear. Most other macro processors that incorporate their host language as their macro language give a way to explicitly indicate what is control text and what is replacement text. All this confusion is compounded in TEX by the mixing of expansion and interpretation.

The macro language for Lisp macro systems is also the same as the host lan-guage. There is great freedom of expression in the STFs, but also the potential for problems, particularly in the user's ability to declare unsafe macros. There are other problems, chiefly stemming from a user's capacity to change the meaning of STFs by changing the functions used in them. These capabilities are a direct result of the coincidence of the macro and host languages. See Sections 4.7.1 and 4.8.

The identification of host and macro languages in Lisp is more than syntactic and semantic correspondence. These languages *coincide* because not only do ex-pressions in them appear the same and have the same meaning, but also expressions are evaluated sharing the same processor and global environment. In an interpreted

Lisp, the same interpreter evaluates the body of an STF as runs the expanded code.

4.7. When to Expand

Macro expansion must take place late enough in language processing to take advantage of the syntactic structure of the program. Now we deal with exactly when the expansion of macro calls should take place. There are three aspects to this question: how early should macro calls be transcribed, should they be completely expanded or only transcribed, and when should macro calls occurring as "arguments" to other macro calls be transcribed.

We break our discussion up into two parts. The first concerns *top-level* macro calls, or those that appear in program text not surrounded by any other macro calls. The second concerns those calls that appear nested as "arguments" to encompassing macro calls.

Top-level calls may be distinguished by a recursive descent parser (Figure 3.3). A tree can be built showing the calls the parser makes. The tree has the original program at its root. Each recursive call introduces a new branch consisting of the subexpression then being parsed. We call this tree the *parser tree* since it describes the recursive calls the parser performs. A *top-level macro call* is a macro call for which there are no other macro calls in the parser tree between it and the root. The recursive level of the call is the first one at which the function `transcribe` is invoked.

4.7.1. TOP-LEVEL CALLS. Two possible times for syntactic extension transcription to consider are compile-time and run-time. Safe macros exhibit the same behavior when expanded at either time. Some unsafe ones show differences. It is these differences that we want to consider here, leading us to some criteria for selecting an expansion time. Obviously, in a language implemented by a compiler, expansions must occur at compile-time. So, the question is, when they are implemented by an

interpreter, what are the effects of expanding macros during syntactic analysis as opposed to expanding them during interpretation. To answer this, we examine the different forms unsafe macros may take. All of them rely upon the host language coinciding with the macro language of the transcription texts.

A macro declaration within a declaration body may enable a macro to modify itself. This not only produces an unsafe macro, but also one that is in violation of Principle 8. A recursive macro such as and could change its expansion for the calls it generates to itself. An instance of a macro call in one part of a syntax tree could change the transcription of an identical macro call in another part. Since different implementations may analyze the same program text in different orders, the same set of declarations and source code may be expanded into different programs.

Within bodies, the values of global identifiers and of dynamically bound variables may be used to control expansion of run-time syntactic extensions.[14] The difficulty with global identifiers is caused by the coincidence of host and macro languages. As before, the problem is that no reliable semantics can be ascribed to programs when macros expand based on run-time values. If a compile-time macro accesses a global variable in order to determine a replacement text, its value is known before the analysis of the program containing the macro call begins.

The problems with macros that are unsafe and in violation of our principles are more acute when expansion takes place at run-time than when it occurs at compile-time. Thus

Principle 10. *In order to preserve program semantics, the complete expansion of syntactic extensions should take place prior to program interpretation or code generation.*

We are imposing a restriction on the programmer—he cannot rely on run-time

[14] As illustrated in Section 2.1.4, identifiers local to and lexically bound within bodies can also cause problems. However, these problems are common to both compile-time and run-time expansions.

```
(case tag-exp [tag exp1 exp2 ...] ... [else expa expb ...])
   ⇒ (let ([gsym tag-exp])
        (cond [(eqv? gsym (quote (tag ...))) exp1 exp2 ...] ...
              [else expa expb ...]))
```

Figure 4.1. *A variant on the Scheme macro* `case`.

expansion tricks—to gain more control on the semantics of programs.

In addition, conforming to this principle results in the more efficient evaluation of interpreted code. For if a `cond` expression must be expanded each time a recursive function is called, the execution takes longer than if the macro call were expanded once.

4.7.2. CALLS OCCURRING WITHIN OTHER MACRO CALLS. Lisp macro systems are usually call-by-name. A macro call occurring within the body of a `let` expression is not expanded until after the `let` itself is transcribed. We say a macro call *occurs within* a lexically surrounding macro call if it is located in an expression subposition in the outer call. Unless a part of the declaration of syntactic extensions is to declare which components of a macro call are themselves expressions, this cannot be determined by the system. For example, when Lisp macros are declared with STFs, there is no way for the macro system to determine that the cond-clauses of the `cond` macro are not expressions.

As an illustration of macro calls occurring and not occurring within others consider a version of the Scheme macro `case`. Its specification is given in Figure 4.1. We look at a call that might be used during the syntactic analysis of a Scheme

program:

```
(case (keyword exp)
   [and (conjunction exp) (continue)]
   [or (disjunction exp) (continue)]
   [begin (sequence exp) (continue)]
   [else (other-syntactic-extension exp) (continue)]).
```

It has already been determined that the expression `exp` is a macro call with a keyword in the first position. We imagine the function `continue` to have something to do with the rest of language processing.

The first `case`-clause is syntactically an and expression. Yet, because clauses themselves are not expressions within calls to case, it cannot be a macro call. Instead, `(conjunction exp)` and `(continue)` are the sub-expressions. The symbol `and` is quoted data, not the keyword of an expression. What appear to be macro calls, in reality, are not. The clauses are *not* instances of macro calls occurring within a `case` expression. Macro calls only occur within other expressions if they appear in a position that accepts expressions. The inability of some systems to determine whether a piece of a program is an expression is demonstrated by those printing routines that display

```
(case tag 'x)
```

for

```
(case tag [quote x]).
```

Adopting Leavenworth's ideas, we would specify the syntactic domain of all expression components. Then, the macro system could be a partial call-by-value system. However, the benefits of call-by-value are not clear to us. And, while we think Leavenworth's approach is the correct one, because of the preponderance of expressions in Lisp programs, to the virtual exclusion of terms from every other syntactic domain, we do not deem it worthwhile. If not expressions themselves,

terms within macros are usually pairs or sequences of expressions, as in `case` and `cond` clauses and `let` and `letrec` declarations. Without formulating it as a design principle, we decide that macro expansion should be by-name.

4.7.3. ISOLATED VERSUS MIXED EXPANSION. We assume that parsing is done by recursive descent. There are other methods, but they do not necessarily lend themselves to by-name macro expansion. Bottom-up parsing discovers the macro calls occurring within an outer call before it reaches the outer call.

When a macro call is encountered during syntactic analysis, its transcription is made. At that point, the macro processor can pass the transcription back to the parser, or it can perform its own recursive analysis. Passing the transcription back without expanding generated macro calls is *mixed* expansion. The alternative, expanding all generated calls so that a fully expanded syntax tree is returned to the parser, is *isolated* expansion. When expansion is done naïvely, it does not matter which is chosen. Isolated expansion essentially requires two passes through code, the first to obtain the complete expansion and the second to parse it.

To adopt fully the syntactic designations of Leavenworth's syntax macros entails performing syntactic checks on each macro argument. An isolated expander would have to either duplicate the activities of the parser as regards error checking or interleave its actions with those of the parser.

In any case, we modify the parser (Figure 3.3) so that the line that takes care of macro transcription becomes

```
[syntactic-extension (parse (expand exp))].
```

It is left up to the function expand whether a full expansion or a single-step transcription is produced.

4.8. The Composition of STFs

The coincidence of the macro language with the host language is of great convenience

to the language implementor. There is a problem: as long as macro writers can declare unsafe macros, a reliable semantics for programs using them cannot be given. One way to discourage unsafe macros and encourage safe ones is to separate the macro language and the host language, or, more properly, the global environments of the two languages. If a Lisp STF uses the identifier `cons`, it should be bound to a different function than that identifier in the host language. Otherwise, the user might redefine `cons` and destroy the macro. If an STF accesses values in a global environment, these should be values in the expand-time global environment, not the run-time global environment. And, an STF should not be able to redeclare the macro it transcribes. This last claim makes sense if the syntax table is part of the host language, and transcription texts are written in the macro language. A declaration, made in the host language, contains the macro body; but the body is written in the macro language. Thus

Principle 11. *The global environment of the macro language and the host language should not coincide.*

There is, however, at least one legitimate interaction between the host language and the macro language—the saving of text in connection with the generation of a transcription. If a function-defining macro can save source code in the host language environment, then that code can later be accessed for editing or printing. Thus, the one permissible connection between the two languages is the ability in the macro language to change the current state of host language's processor. The idea is that the macro language can transmit information to the host language but it cannot receive information from it. Since the functions cons, list, car, cdr, and so forth are host language data, the macro language should not be able to access these values. Once we have separated the macro language from the host language, there are no longer any problems as far as macros are concerned with the redefinition of system primitives and special forms.

4.9. What the Macro Processor Does

We are now ready to consider what happens to a text from a semantic point of view during macro expansion. We touched upon this slightly while presenting the capturing problem.

There are two alternative ways to specify the semantics of a syntactic extension. First, the semantics can be described without resort to macro expansion. It can be written for the form as if it were part of the core language. Alternatively, the semantics can be described operationally by writing the macro's transcription in terms of forms for which a known semantics already exists. We call the first semantics the *primary semantics* of a form, the second, the *expansion semantics.* In a correct implementation of a macro, the two semantics are mathematically equivalent.

We wish to make a stronger statement about macro calls in context, macro calls occurring as sub-expressions within our programs. It is not enough to know that macro calls occurring out of a program context are correct. We must be able to make claims about programs that contain macro calls. Extending the notion of primary and expansion semantics to programs containing macro calls, we assert

Principle 12. *The expansion semantics of a program P should be mathematically equivalent to the primary semantics of P.*

A violation of this principle was given in Section 1.3.2 when we considered the expression

```
(let ([v 1]) (or (zero?  v) v)).
```

The primary semantics of this program is `1`; the expansion semantics (under naïve macro expansion) is `false`. It is precisely this principle that is violated by naïve expansion. A macro expansion algorithm should not inadvertently change the semantics of the programs it processes.

The recognition of a scoping mechanism during syntactic processing adds more texture to the syntax trees that represent programs. Once it is taken into account, the capturing problem arises. No matter whether the scoping is lexical or dynamic, it is the presence of `lambda` bindings—block structuring—that causes the trouble.

4.10. Triance and Layzell's Design Principles

The recommendations of Triance and Layzell [74] concerning the design of a macro processing system are similar to ours. In this section each of their design principles is discussed.

Their first principle is that a macro processing system should be designed for a single base language. That is, the system's design should be dependent upon the host language and not be general purpose. We tacitly assume this principle in our work on Scheme. Most of the ideas of this chapter are host language independent, but starting in the next section our efforts are directed toward building a Scheme macro processing system.

The second principle concerns the complexity of the macro processing system. Triance and Layzell believe that the requirements placed on the system should be limited. It should not be expected to support a wide variety of language enhancements. A clear statement of exactly what the macro processor does is required. We have made such statements about our work. When we decide that the Scheme macro system will only include expression syntactic extensions, we are declaring that the macro processor is not responsible for language extensions such as changing the parameter passing mechanism in applications.

Third, they carefully state that "each enhanced language should become the target language for the next set of macros." This allows for pyramiding of macro definitions, a common phenomenon in Lisp macro systems.

They feel, as we do, that the matching phase should involve as much syntax

checking as possible. Since they have COBOL in mind as a host language, they are not concerned with the list structure of macro calls. Nevertheless, they insist on verification of the delimiter structure for each macro call. Influenced by Leavenworth's syntax macros, they believe that the syntactic analysis should include a check on the "class of argument."

Following Cheatham and Leavenworth, Triance and Layzell assert that the context of each macro call must be specified with each macro declaration. Associated with each new macro is a description of the contexts in which a macro call is legal. If a Scheme macro system allowed for declaration as well as expression syntactic extensions, this would be an important aspect of its design.

Their sixth design principle is that macro calls should be made in free mode, without warning markers. We have already stated this in our discussion of Principle 5.

The next claim is that nesting of macro calls, allowing calls to appear as arguments to other macro calls, should be permitted. This would not even be an issue if it were not for some macro systems forbidding the use of a macro call as argument to another macro call. Doing so would run counter to the Scheme philosophy that any expression, including a syntactic extension expression, may appear anywhere an expression is deemed legal in the formal description of the language.

An interesting design principle is that macros should have the capability of switching other macros on and off. For example, Triance and Layzell believe that a macro A should have the ability to set some sort of expansion-time flag that enables calls to another macro B. However, no use of macros apart from pure replacement of the macro call by its expansion is sound. We want to be certain that, given a set of syntactic extensions, we can determine the semantics of each program that calls them. When the order of macro expansion of parallel branches of a syntax

tree cannot be guaranteed, there is no way of telling whether A has been expanded prior to expanding the call to B.

Triance and Layzell's tenth design principle relates to the declaration of templates of macro calls; patterns should be made by using data structure description facilities within the macro language. The macro language should be rich enough to include ways to describe structures needed within transcriptions. They cite two other methods, using a syntactic metalanguage and using a syntax diagram. Our work uses the syntactic metalanguage approach.

Within macro bodies, the generated text should be clearly set apart from the control part of the macro declaration, thus avoiding confusion for anyone who reads it. The integration of generated text and control within Lisp STFs causes a major problem in reading them. A source of misunderstanding in learning TEX is the mixing of control text and generated text within macro bodies. Triance and Layzell advocate a clearer system. In our work, we have carefully placed the template for the generated text in a specific location inside each transcription specification.

Twelfth, they feel that the macro language should support data items and procedures global to all macros, local to a set of macros, and local to specific macros. If the language of STFs is taken as the macro language, this is easily accomplished by closing each STF in the desired lexical environment.

Next, they wish their macro system to be able to generate data items and procedures with scopes global to the program, local to a set of macros, local to a single macro, and local to a single macro call. Again these happen by default in Lisp systems, for the STFs can generate whatever shared structures are desired.

Triance and Layzell want there to be a "comprehensive symbol table" for use in generating code. This principle seems largely based on the situation they are dealing with in building a macro processor system for COBOL.

Finally, they declare that an extended version of the host language should be used as the macro language. This issue was addressed in Section 4.8 with the resolution that the two languages should not be the same.

4.11. A Summary of Our Design Decisions

In this section we summarize the main design decisions we have made concerning a syntactic extension system for Scheme.

1. The syntactic extension mechanism is expression oriented. There is no means of declaring syntactic extensions to any other syntactic domains.
2. Macro calls take the form of lists with a keyword—the macro name—in the first position. Thus macros are called under free mode.
3. The macro writer should be cognizant of the syntax table but not necessarily of how it is implemented.
4. Macro calls are recognized by pattern matching against the declared patterns in the syntax table.
5. Macro calls have no fixed delimiter structure. The pattern variables of the syntax table patterns serve as formal parameters.
6. Macro calls are expanded by-name.
7. The expansion of a syntactic extension is not naïve. However, provision must be made for the capturing of key identifiers.
8. The set of keywords is disjoint from the set of identifiers.
9. The macro language does not coincide with the host language.
10. Macro declarations are made so that the macro writer does not have to compose an STF. Instead the patterns, transcriptions, and pattern variables already identified during the design of his syntactic extension are used.

Most of these design decisions are satisfied by the standard macro processors of traditional Lisp systems. In them, the macros are expression syntactic extensions,

the macro calls are made in free mode, macros have no fixed delimiter structure, and they are expanded by-name. We now concentrate on the declaration of syntactic extensions, the syntax table, and the expansion algorithm.

5. Declaring Syntactic Extensions

In this chapter we realize the decisions made concerning the declaration of syntactic extensions at the end of the last chapter. The first section is a user manual for our declaration tool `extend-syntax`. The second gives a formal description of the major feature of `extend-syntax`. The third discusses implementing the syntax table. The fourth describes the implementation of `extend-syntax` by presenting annotated Scheme code.

5.1. The `extend-syntax` Manual

The underlying goal of `extend-syntax` is to enable the declaration of syntactic extensions in a form similar to that used in describing the Lisp macros of Chapter 3. See, for example, Figures 3.6, 3.7, 3.8, 3.9, and 3.11. The format of an `extend-syntax` declaration is

```
(extend-syntax ⟨keywords-list⟩ ⟨key-identifiers-list⟩
  ⟨production-rule⟩
  ⟨production-rule⟩ ...).
```

The *keywords-list* is a list of all keywords specified in the new special form. Since our syntactic extensions include a macro name as the first component of the expression, this initial keyword always appears in the keywords-list. There might

```
(extend-syntax (ite then else) ()
  [(ite p1 p2 ... then e1 e2 ... else f1 f2 ...)
   (if (begin p1 p2 ...)
       (begin e1 e2 ...)
       (begin f1 f2 ...))])
```

Figure 5.1. *An if-then-else macro.*

also be auxiliary keywords, as in a conditional form with explicit use of `then` and `else` (Figure 5.1).

It is possible to specify more than one production rule in an `extend-syntax` declaration. All productions associated with a given macro name must be declared at the same time. As an example, consider the form `and`:

```
(extend-syntax (and) ()
   [(and) true]
   [(and e) e]
   [(and e1 e2 ...) (if e1 (and e2 ...) false)]).
```

The ordering of the production rules within a declaration is significant. In the example, the second rule is a special case of the third. Both are necessary to implement the standard semantics of `and` expressions.

The ordering of production rules is similar to the ordering of alternatives in simple recursive function definitions. The boundary conditions are placed first. They serve to guard the recursion in the same way the first two production rules guard the recursive macro call in the third rule of `and`.

As in the notation of other chapters, the ellipses indicate zero or more occurrences of the terms they immediately follow. Thus, in the `ite` example there are one or more sub-expressions in each of the predicate, consequent, and alternate portions of the expression.

Also in that example, auxiliary keywords appear in the production rule. In general, they may appear in some rules and not others, or they may not appear at all. A variant on `ite` with an optional `else` part illustrates this:

```
(extend-syntax (ite then else) ()
   [(ite p1 p2 ... then e1 e2 ... else f1 f2 ...)
    (if (begin p1 p2 ...)
        (begin e1 e2 ...)
        (begin f1 f2 ...))]
  [(ite p1 p2 ... then e1 e2 ...)
   (if (begin p1 p2 ...)
       (begin e1 e2 ...)
      false)]).
```

Here, the first rule is more restrictive than the second; therefore it comes first.

The generated identifiers can be partitioned into two disjoint sets: those that are of no consequence to the user of the syntactic extension and those that do matter. As observed in Section 1.3.2 there are times when specific generated identifiers are of consequence to the user. These identifiers are declared by the syntactic extension designer in the key-identifiers-list. The infinite loop macro `inf-loop` (Figure 1.6) is declared by writing

```
(extend-syntax (inf-loop) (exit-with-value)
   [(inf-loop e1 e2 ...)
    (call/cc
      (lambda (exit-with-value)
        (while true e1 e2 ...)))]).
```

The expansion of syntactic extensions can rely upon the key-identifiers-list so as to avoid the inadvertent capture of free identifiers; it specifies which identifiers are eligible for capture by binding instances in the generated text. The question the macro writer must ask is, "Are there any special identifiers that the user must know in order to employ my syntactic extension?" If so, these identifiers, which are part of the syntactic extension's protocol, must be declared in the key-identifiers-list.

An example serves to clarify the concept of key identifiers. We generalize the special form (or *exp1 exp2*) as

```
(extend-syntax (gor) ()
   [(gor exp1 exp2)
    (let ([first-result exp1])
       (if (test? first-result)
           first-result
           exp2))]).
```

When a call to gor is expanded and run, sometimes the value of exp1 is returned and sometimes that of exp2. A macro expander can be designed so that the identifier `first-result` will be renamed, if necessary, to avoid capturing any occurrences of the identifier `first-result` in exp2. How this is done is explained in detail in Chapter 6.

Equally plausible is

```
(extend-syntax (gor/c) (first-result)
   [(gor/c exp1 exp2)
    (let ([first-result exp1])
       (if (test? first-result)
           first-result
           exp2))])
```

in which the expression exp2 is evaluated in an environment in which `first-result` is bound to the value of exp1. The identifier `first-result` is not renamed because of its presence in the key-identifiers-list.

As additional illustrations of `extend-syntax`, the declarations of `rec`, `recur`, and `letrec` are given in Figure 5.2.

5.1.1. PRODUCTION RULES. A grammar for `extend-syntax` production rules is given in Figure 5.3. The rest of this section is devoted to an explanation of the three components of production rules: the pattern specifications, or left-hand sides; the transcription specifications, or right-hand sides; and the fenders, or attributes.

```
(extend-syntax (rec) ()
   [(rec name exp)
    ((lambda (name)
       (set! name exp)
       name)
     '*)])

(extend-syntax (recur) ()
   [(recur name ([i e] ...) b ...)
    ((rec name (lambda (i ...) b ...))
     e ...)])

(extend-syntax (letrec) ()
   [(letrec ([i e] ...) b ...)
    (let ([i '*] ...)
      (set! i e) ...
      b ...)])
```

Figure 5.2. `extend-syntax` *declarations of some common Scheme macros.*

⟨production rule⟩ ::=
(⟨pattern specification⟩ ⟨transcription specification⟩) |
(⟨pattern specification⟩ ⟨fender⟩ ⟨transcription specification⟩)

Figure 5.3. `extend-syntax` *production rules.*

In a pattern specification, fender, or transcription specification the term that immediately precedes an ellipsis is a *prototype*. Every use of an ellipsis must be preceded by a prototype. Intuitively, a list containing an ellipsis and a prototype—an *ellipsis-list*—represents a syntactic structure formed from zero or more occurrences

```
(extend-syntax (cond else) ()
   [(cond) false]
   [(cond [else exp1 exp2 ...]) (begin exp1 exp2 ...)]
   [(cond [exp] clause ...) (or exp (cond clause ...))]
   [(cond [exp1 exp2 exp3 ...] [exp ...] ...)
    (if exp1
       (begin exp2 exp3 ...)
       (cond [exp ...] ...))])
```

Figure 5.4. *A declaration of* cond.

of terms similar in form to the prototype. Lists containing ellipses may be nested provided each ellipsis has a prototype. As an example, consider the version of the special form cond in Figure 5.4.

Arbitrary nestings of ellipses within pattern and transcription specifications have ambiguous semantics. It is not always clear what expansion should be produced from a declaration containing nested ellipses. The semantics of the next section gives a precise formulation of what ellipses mean in simple cases. The user of `extend-syntax` is cautioned that arbitrarily nested ellipses lists may not produce the effect he intends.

In a pattern specification, an ellipsis must either be the last term in a list, be followed immediately by a keyword, or be followed immediately by another list containing a keyword as its left-most atom. The pattern

```
(ite p1 p2 ... then e1 e2 ... else f1 f2 ...)
```

is legal because in the `extend-syntax` expression for `ite` we declared `then` and `else` to be keywords. Auxiliary keywords used to signal the end of an ellipsis-list segment cannot appear more than once in each pattern specification. Ellipsis-list segments that are followed by a keyword or list containing a left-most keyword are

```
(extend-syntax (case else) ()
   [(case tag [t1 e1 e2 ...] ... [else f1 f2 ...])
    (let ([v tag])
       (cond [(eqv? v 't1) e1 e2 ...] ...
             [else f1 f2 ...]))]
   [(case tag [t1 e1 e2 ...] ...)
    (let ([v tag])
       (cond [(eqv? v 't1) e1 e2 ...] ...))])
```

Figure 5.5. *A declaration of* `case`.

called *terminated.*

The special form `case` can be declared in terms of `cond` expressions (Figure 5.5). The two production rules demonstrate the optional use of an `else` clause. In the first rule, the top-level ellipsis part is terminated by a list containing the keyword `else` as its first component.

The restriction to terminate ellipsis-lists by either a keyword or by a list containing a keyword as its left-most atom is an arbitrary one. Nevertheless, some decision must be made about how to end ellipsis-lists that do not extend to the end of the list in which they are found. This particular rule provides a convenient way to express the standard Scheme special forms in terms of our core language.

For ellipses appearing in transcription specifications there are only two requirements and no special restrictions on terminating ellipsis-list segments. The first requirement is that each use of an ellipsis has an immediately preceding prototype. The second deals with what pattern variables may be included in that prototype. If a transcription specification prototype contains pattern variables extracted from more than one pattern prototype, the lists described by the various pattern components must be the same length in all calls to the macro being declared. For example,

this specification of `pairwise-sum`

```
(extend-syntax (pairwise-sum) ()
   [(pairwise-sum (a ...) (b ...))
    (list (+ a b) ...)])
```

describes a syntactic extension that must be used with two lists of equal length. A formal description of the meaning of ellipses with pattern-transcription specifications is given in Section 5.2.

Often it is desirable to apply further tests than a simple syntactic match in selecting a production. For this reason it is possible to include a *fender* within each production. If the left-hand side of a production matches the source language expression and if a fender is present, then the production will only be chosen if, when evaluated, the fender is true. Consider, as an example,

```
(extend-syntax (fendex) ()
   [(fendex a) (number? 'a) (+ a 5)]
   [(fendex a) (symbol? 'a) (list a 'symbol)]
   [(fendex a) (list? 'a) (car a)]).
```

The transcription of `(fendex 6)` is `(+ 6 5)`, that of `(fendex x)` is

```
(list x 'symbol),
```

and that of

```
(fendex (quote (x y z)))
```

is

```
(car (quote (x y z))).
```

As is evident from this example, pattern variables may be used within fenders. In the third sample call to `fendex`, the pattern variable a is a name for the structure `(quote (x y z))`. The fender actually performs the test

```
(list? '(quote (x y z))).
```

The `quote` which remains represented with the single quote mark can be thought of as the `quote` from the fender.

When using fenders, it is necessary to keep in mind that they are transcription-time checks on the structure of syntactic extension expressions. They are operational descriptions of conditions that macro calls must meet in order for the corresponding transcription to be made. Suppose a `fendex` expression is used in the context

```
(let ([x 3]) (fendex x)).
```

The value of this expression is not 8, rather it is `(3 symbol)`. The first fender asks the question whether the value of the pattern variable `a` is a number. The value of the pattern variable is the symbol `x`. It fails the number test but passes the symbol test. Hence, the transcription is `(list x 'symbol)`.

The text of the fender is processed in two ways corresponding to what lies inside and what lies outside of any `quote` expressions. This difference is referred to as *partial-processing*. The syntactic extension writer thinks of the parts inside `quote` expressions as instructions for the selection and organization of components of program text from the unexpanded expression. The parts outside `quote` expressions describe the conditional tests to be performed on the components. As illustrated above with `fendex`, it is easy to determine a fender test for a specific macro call by substituting the values of the pattern variables for the variables inside quotations. A pattern variable in a fender is not recognized as such unless it appears within a quoted expression. Because of the distinction between quoted and non-quoted parts of a fender, a fender is referred to as a "partial-processing rule (pp-rule) expression."

Identifiers outside of quoted expressions are dereferenced in the lexical environment in which the macro declaration appears. For example, the entire `extend-syntax` expression may appear as the body of a `let` expression that binds a set of

special purpose identifiers for use in its fenders.

The ellipsis may also be used within fenders. For example, the fender in

```
(extend-syntax (pairwise-sum) ()
   [(pairwise-sum (a ...) (b ...))
    (for-all number-pair? '((a b) ...))      ; fender
    (list (+ a b) ...)])
```

checks that both lists `(a ...)` and `(b ...)` are made up of numbers by pairing them and asking whether the pairs are composed of numbers. During the transcription of the expression `(pairwise-sum (1 2 3) (4 5 6))`, the fender test

```
(for-all number-pair? '((1 4) (2 5) (3 6)))
```

is made. The ellipsis—always occurring within quoted parts of the fender—is used as it is in transcription specifications.

Fenders provide important, expansion-time checks on the syntax of macro calls. For example, within a call to the `rec` macro, the second component must be a symbol. A fender should be used to verify this during macro transcription. An improved version of the declaration given in Figure 5.2 uses a fender:

```
(extend-syntax (rec) ()
   [(rec name exp)
    (symbol? 'name)
    ((lambda (name)
       (set! name exp)
       name)
     '*)]).
```

5.1.2. NEW VARIABLES. On occasion, it is useful to bind new variables, other than pattern variables, to pieces of program text. We might, for example, want the symbol `x` to represent a piece of program text not found within the pattern. There are two forms that allow this; they are called `with` and `withrec`. Their syntax within transcription specifications is described in Figure 5.6.

```
⟨transcription specification⟩ ::=
        ⟨basic transcription⟩ |
        (with (⟨with spec⟩ ⟨with spec⟩ ... ) ⟨transcription specification⟩) |
        (withrec (⟨with spec⟩ ⟨with spec⟩ ... ) ⟨transcription specification⟩)

    ⟨with spec⟩ ::=
        [⟨variable⟩ ⟨pp-rule expression⟩] |
        [(⟨variable⟩ ⟨ellipsis⟩) ⟨pp-rule expression⟩]
```

Figure 5.6. *Transcription specifications.*

A ***basic transcription*** is a transcription specification that does not use either of these two new forms. A pp-rule expression is a piece of text, possibly using pattern variables, ellipses, and `quote`, whose expansion-time significance is given by the partial-processing rule (as described for fenders).

Two kinds of `with` specifications are allowed. We discuss the first, distinguished by a ⟨variable⟩ as the first component, in this section. We describe the second, distinguished by the variable and ellipsis pair, in the next section.

The forms `with` and `withrec` enable the definition of new variables that may appear in the enclosed transcription specification. Within that transcription specification, they are used in the same way as pattern variables. During the transcription of a given syntactic extension, the pp-rule expression is evaluated—like the evaluation of a fender—and the result is associated with the `with` or `withrec` variable. Subsequently, that value is included within the transcribed text in place of the variable.

The distinction between `with` and `withrec` concerns the scope of the variables

```
(extend-syntax (fibon) ()
   [(fibon n)
    (and (integer? 'n) (zero? 'n))    ;  fender
    0]
   [(fibon n)
    (and (integer? 'n) (zero? (sub1 'n)))    ;  fender
    1]
   [(fibon n)
    (and (integer? 'n) (positive? 'n))    ;  fender
    (with ([next (sub1 'n)])
          (with ([nextnext (sub1 'next)])
                (+ (fibon next) (fibon nextnext))))])
```

Figure 5.7. *A macro for Fibonacci numbers.*

being defined. The new variables from a `with` specification are known only in the enclosed transcription specification. Those from a `withrec` specification are known in the transcription specification and in the pp-rule expressions at the same level. The lexical scoping rules for the special forms `let` and `letrec` follow the same pattern. Thus, a `withrec` declaration can be used to associate a recursive function or data-structure with the new variable.

To illustrate most of this, we consider the syntactic extension `(fibon` n`)` that is equivalent to an arithmetic expression that computes the n^{th} Fibonacci number (Figure 5.7). The complete expansion of `(fibon 5)` is

```
(+ (+ (+ (+ 1 0) 1) (+ 1 0)) (+ (+ 1 0) 1)).
```

The transcription occurs only when the pattern variable n is associated with a non-negative integer. It does not work when the pattern variable is associated with any other kind of expression because the fender tests and the expansion are made during the text-processing phase of compilation.

```
(extend-syntax (writeln) ()
   [(writeln exp1 exp2 ...)
    (withrec ([print-list
               (lambda (l)
                  (if (null? l)
                      (newline)
                      (begin (print (car l))
                             (print " ")
                             (print-list (cdr l)))))])
      (print-list (list exp1 exp2 ...)))])
```

Figure 5.8. *An illustration of* `withrec`.

In some Scheme implementations, functions appearing within code are self-evaluating objects. This gives rise to code that includes functions as part of the source language. The means of producing such code is by `with` or `withrec`. As an example, consider the syntactic extension of Figure 5.8 that is equivalent to an expression that prints the values of `exp1 ...` without an intermediary line feed. It utilizes the local, recursive function `print-list`. The code produced by transcribing `(writeln 1 2 3)` contains the `print-list` function

(*function* `(list 1 2 3)`).

An alternative way of declaring `writeln` is to make the the `print-list` function local to the declaration, as in Figure 5.9. This method still requires the use of `with` to incorporate the declaration-time value of `print-list` into the STF. It works because the created STF is closed in the environment of the `extend-syntax` call. If the `with` specification were omitted, as in

```
(letrec ([print-list function])
   (extend-syntax (writeln) ()
      [(writeln exp1 exp2 ...)
       (print-list (list exp1 exp2 ...))])),
```

```
(letrec ([print-list
          (lambda (l)
            (if (null? l)
                (newline)
                (begin (print (car l))
                       (print " ")
                       (print-list (cdr l)))))])
  (extend-syntax (writeln) ()
    [(writeln exp1 exp2 ...)
     (with ([pl print-list])
           (pl (list exp1 exp2 ...)))]))
```

Figure 5.9. *A second declaration of* `writeln`.

the local definition of `print-list` in expansions of calls to `writeln` would not be used. Whatever value of `print-list` is known where the call appears would be employed instead.

This is what distinguishes the pattern and new variables from other symbols in the transcription specification. The values of variables are always put in the replacement text; the other symbols are always literals. The code generating portion of the STF produced by

```
(extend-syntax (goo) ()
   [(goo x)
    (with ([y z])
          (w x y z))])
```

is

```
(let ([y z])
   (list 'w (cadr pattern) y 'z)).
```

A second example of a Scheme structure embedded in code is found in Figure 5.10. In it two special forms which share the same cons cell are declared. We

```
(let ([box (cons '() '())])
  (extend-syntax (peek) ()
    [(peek) (with ([cell box])
                  (car 'cell))])
  (extend-syntax (poke) ()
    [(poke v) (with ([cell box])
                    (set-car! 'cell v))]))
```

Figure 5.10. *Two special forms that share a cons cell.*

pay special attention to quoting the cell in the transcription because cons cells that appear in code are not self-evaluating.

On rare occasion there are expansion structures that cannot be described as a mere rearrangement of pieces of the pattern, or at least it is not obvious how to do so. The partial processing feature of `with` and `withrec` specifications allow any transformation of the original expression to be associated with a new variable. This new variable is replaced, during transcription, by the result of the computation described in the partial-processing rule expression. For example, one may write

```
(extend-syntax (hairy) ()
  [(hairy exp ...)
   (with ([ans (arbitrary-fn '(exp ...))])
         ans)]).
```

5.1.3. REPEATED SYMBOLS. At times, we need to introduce a set of new identifiers into the transcription of a syntactic extension expression, not knowing at declaration time how many are needed. Consider the syntactic extension

(parset! (*i e*) ...),

a form that evaluates the expressions *e*, ... and then makes assignments to the

```
(extend-syntax (letrec) ()
   [(letrec ([i e] ...) b ...)
    (for-all symbol? '(i ...))
    (with ([(star ...) (map (lambda (x) '*) '(i ...))])
          ((lambda (i ...)
              (set! i e) ...
              b ...)
           star ...))])
```

Figure 5.11. *A declaration of* `letrec` *without using* `let`.

corresponding identifiers. It accomplishes a parallel assignment:

```
(extend-syntax (parset!) ()
   [(parset! (i e) ...)
    (for-all symbol? '(i ...))
    (with ([(k ...) (map (lambda (x) (gensym)) '(i ...))])
          (let ([k e] ...)
             (set! i k) ...))]).
```

The simple ellipsis-list (a single variable followed directly by the ellipsis) appearing as the first component of the `with` specification defines a new pattern variable that serves as the prototype of an ellipsis-list. The actual list is described by the second component, a pp-rule expression, of the `with` specification. In the example, a list of generated symbols corresponding to the assignment identifiers is created.

It is interesting to note that without this capability `letrec` cannot be declared in terms of the application of a `lambda` expression. With its use, it can (Figure 5.11).

5.1.4. NESTED `extend-syntax` CALLS. Sometimes we wish to declare other macro-declaring special forms using `extend-syntax` for both the declaration and

the transcription. For example, the macro `alias` can be declared as

```
(extend-syntax (alias) ()
   [(alias new old)
    (extend-syntax (new) ()
       [(new x ...) (old x ...)])]).
```

The difficulty with this is that once the nested `extend-syntax` call is included in the transcription specification, the ellipsis means something different. It is no longer used to indicate the repetition of pattern variable prototypes from the pattern specification. Instead it appears as a literal in the clause

```
[(new x ...) (old x ...)].
```

Consider the syntactic extension

```
(extend-syntax (foo) ()
   [(foo a b ...)
    (extend-syntax (a) ()
       [(a x ...) (list x ... b ...)])]).
```

The macro call `(foo bar 1 2 3)` will not cause the creation of a macro `bar` that appends its arguments to the list `(1 2 3)`. In fact, the call to `foo` results in an error; there is no pattern ellipsis-list corresponding to the transcription prototype `b`.

Nesting is detected within a transcription specification by the presence of one of the three *extend-syntax names*. They are

`extend-syntax`,
`extend-syntax/code`, and
`syntactic-transform-function`.

The last two of these are discussed in the next section.

A simpler version of `extend-syntax` prohibiting multiple production rules and fenders, requiring that the macro name be the only keyword, and lacking the dec-

laration of key-identifiers can be declared as

```
(extend-syntax (syntax extend-syntax extend-syntax/code
                       syntactic-transform-function
                       ... with withrec)
  ()
  [(syntax (name x ...) transcription)
   (extend-syntax (name) ()
     [(name x ...) transcription])]).
```

The presence of the ellipsis in the keywords list indicates that the ellipsis is itself a keyword in `syntax` calls.

There is a set of *extend-syntax keywords* that the macro writer must know about in order to use `extend-syntax` to construct other macro-declaring macros. They are

`extend-syntax`,
`extend-syntax/code`,
`syntactic-transform-function`,
`...` (the ellipsis),
`with`, and
`withrec`.

If the extend-syntax names are not included in the keywords list of macro-declaring macros, the nesting of macro declarations cannot be detected.

5.1.5. RELATED SPECIAL FORMS. There are two other special forms similar to `extend-syntax`. One, `extend-syntax/code`, displays the code of an STF generated by `extend-syntax`. The other, `syntactic-transform-function` returns the STF. Neither form effects a macro declaration. The only difference in the syntax of their calls is that "`extend-syntax`" is replaced by either of the two other names. For example, the result of writing

```
(extend-syntax/code (let) ()
   [(let ([i e] ...) b ...)
    (for-all symbol? '(i ...))
    ((lambda (i ...) b ...) e ...)])
```

```
(lambda (G07)
   (if (and (and (pair? (cdr G07))
                 (or (null? (cadr G07))
                     (and (pair? (cadr G07))
                          (for-all
                             (lambda (G08)
                                (and (pair? G08)
                                     (and (pair? (cdr G08))
                                          (null? (cddr G08)))))
                             (cadr G07))))
            (or (null? (cddr G07)) (pair? (cddr G07))))
       (let ((G07 (unstamp G07)))
          (for-all symbol? (map car (cadr G07))))
       (cons (cons 'lambda
                   (cons (map car (cadr G07)) (cddr G07)))
             (map cadr (cadr G07)))
       (macro-use-error G07)))
```

Figure 5.12. *The generated* STF *for the macro* `let`.

can be seen in Figure 5.12.

Most of the STF is dedicated to error checking; the part that actually generates code is in the consequent part of the `if` expression. The function `macro-use-error` is a system-specific error routine designed to inform the macro user of a fault in his macro call. The error is signaled when the call does not match any of the declared patterns for expressions with a given macro name. Another example, helpful in understanding what happens when the specification includes `with` appears in Figure 5.13.

The special form `syntactic-transform-function` is useful when one wishes

```
(extend-syntax/code (foo) ()
   [(foo a ...)
    (with ([b anything]
           [(x ...) (map (lambda (z) (gensym)) '(a ...))])
          (list b x ...))])

produces

(lambda (G01)
   (if (or (null? (cdr G01)) (pair? (cdr G01)))
       (let ([b anything]
             [x (map (lambda (z) (gensym))
                     (cdr G01))])
         (cons 'list (cons b x)))
       (macro-use-error G01)))
```

Figure 5.13. *Another illustration of* `extend-syntax` *produced code.*

to create an STF that is a first-class object in Scheme. Employed as

```
(define let-stf
   (syntactic-transform-function (let) ()
      [(let ([i e] ...) b ...)
       (for-all symbol '(i ...))
       ((lambda (i ...) b ...) e ...)])),
```

it has the same effect as writing the STF by hand. The new function `let-stf` can be used to generate the replacement text for any `let` expression.

5.2. A Formal Semantics of Ellipsis

In this section we give a formal semantics for the behavior of ellipses within pattern and transcription specifications. We have simplified our task by not considering terminated ellipsis lists and not allowing a special case involving nested ellipsis lists. This special case is explained at the end of the section. We also are not concerned here with fenders or new pattern variables.

Special Symbol:

Δ the ellipsis character.

Syntactic Domains:

$p \in pat$ pattern specifications,
$t \in trans$ transcription specifications,
$a \in atom$ atomic symbols,
$c \in call$ macro calls.

We also refer to

$v \in atom$ pattern variables.

Syntax:

$$p ::= a \mid (p\,\Delta) \mid (p_1 \ldots p_n) \text{ for } n \geq 1,$$
$$t ::= a \mid (t\,\Delta) \mid (t_1 \ldots t_n) \text{ for } n \geq 1,$$
$$c ::= a \mid (c_1 \ldots c_n) \text{ for } n \geq 0.$$

Semantic Domain:

$$\rho \in ENV = atom \xrightarrow{o} (N \times call).$$

Figure 5.14. *Formal semantics of ellipsis* (1).

Figures 5.14, 5.15, and 5.16 contain a denotational style description of what it means to transcribe a macro call for which a given pair of pattern and transcription specifications exists. Since the language of denotational style descriptions uses ellipses "..." in the mathematical sense, we have chosen the character Δ to represent the `extend-syntax` ellipsis within the formal pattern and transcription language specifications.

Other miscellaneous notations are:

- The symbol N represents the domain of natural numbers.
- The bold left and right parentheses are used around list structures in the syntax

domains of pattern specifications, transcription specifications, and macro calls. The non-bold face parentheses are used as part of the meta-language or are used to represent lists of environments.

- Angle brackets enclose pairs from the product domain $N \times call$.
- The symbol nil represents the empty list. It is used in the definitions of the functions $\mathcal{L}$ and *decompose* to indicate an empty list of environments. The list-building function *cons* creates other lists.
- The function *hd* selects the first component, the head, of a syntactic structure from one of the Syntactic Domains. The function *tl* returns the structure minus the head.
- The vertical bar | denotes function restriction.
- The syntactic domains of pattern and transcription specifications are described separately for two reasons. First, while in this formal description they appear the same, in full `extend-syntax` notation they are different. And second, the atomic components of the two domains have different roles. All atoms in a pattern specification are pattern variables. Some of the atoms in a transcription specification may be literals.

The top-level semantic function is $\mathcal{E}$. It can be thought of as an interpreter for macros given three inputs: a pattern specification, a transcription specification, and a macro call. The answer it produces is the expanded call. For example,

$$\mathcal{E}[\![(\mathbf{foo\ a\ (b\ c)}\ \Delta)]\!]\ (\mathbf{list\ a\ b}\ \Delta)\ (\mathbf{foo\ u\ (w\ x)\ (y\ z)})$$

produces the transcription

$$(\mathbf{list\ u\ w\ y}).$$

The function $\mathcal{D}$ builds an environment based on the pattern specification and the macro call. The environment associates each atom from the pattern with a pair

Semantic Functions:

$$\mathcal{E}: pat \to trans \to call \to ANS,$$
$$\mathcal{D}: pat \to ENV \to call \to ENV,$$
$$\mathcal{T}: trans \to ENV \to ANS,$$
$$\mathcal{L}: call \to pat \to ENV^*;$$

$$\mathcal{E}[\![p]\!] = \lambda t\, c.\, \mathcal{T}[\![t]\!](\mathcal{D}[\![p]\!]\ \textit{null-env}\ c);$$

$$\begin{aligned}
\mathcal{D}[\![a]\!] &= \lambda \rho\, c.\, \rho[\langle 0, c\rangle / a], \\
\mathcal{D}[\![()]\!] &= \lambda \rho\, c.\, (c \stackrel{?}{=} ()) \to \rho, \bot, \\
\mathcal{D}[\![(p\ \Delta)]\!] &= \lambda \rho\, c.\, \textit{combine}\, \rho\, (\mathcal{L}[\![c]\!]p), \\
\mathcal{D}[\![(p_1 \ldots p_n)]\!] &= \lambda \rho\, c.\, \mathcal{D}[\![(p_2 \ldots p_n)]\!](\mathcal{D}[\![p_1]\!]\rho(hd\, c))(tl\, c);
\end{aligned}$$

$$\begin{aligned}
\mathcal{L}[\![a]\!] &= \lambda p.\, \text{nil}, \\
\mathcal{L}[\![(c_1 \ldots c_n)]\!] &= \lambda p.\, (\mathcal{D}[\![p]\!]\textit{null-env}\ c_1 \ldots \mathcal{D}[\![p]\!]\textit{null-env}\ c_n);
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}[\![a]\!] &= \lambda \rho.\, a \in \text{Dom}(\rho) \to ((\rho a \downarrow 1) \stackrel{?}{=} 0 \to (\rho a \downarrow 2), \bot), a, \\
\mathcal{T}[\![()]\!] &= \lambda \rho.\, (), \\
\mathcal{T}[\![(t\ \Delta)]\!] &= \lambda \rho.\, \exists v \in \text{Dom}(\rho|_{\textit{Vars-of}\, t}) \text{ such that } (\rho v \downarrow 1) \geq 1 \\
&\qquad \to (\mathcal{T}[\![t]\!]\rho_1 \ldots \mathcal{T}[\![t]\!]\rho_n), \\
&\qquad\ \bot \\
&\qquad \text{where } (\rho_1 \ldots \rho_n) = \textit{decompose}\, (\rho|_{\textit{Vars-of}\, t}), \\
\mathcal{T}[\![(t_1 \ldots t_n)]\!] &= \lambda \rho.\, (\mathcal{T}[\![t_1]\!]\rho\ .\ (\mathcal{T}[\![t_2 \ldots t_n]\!]\rho)).
\end{aligned}$$

Figure 5.15. *Formal semantics of ellipsis* (2).

comprised of a natural number and a piece of the macro call. The number indicates at which level of ellipsis nesting each pattern variable appears. For example, using Scheme code and an association-list representation for the environment, the pattern

```
(foo a (b c) ...)
```

and macro call

```
(foo u (w x) (y z))
```

are combined to form the environment

```
((a . (0 u)) (foo . (0 foo)) (c . (1 (x z))) (b . (1 (w y)))).
```

Also, from the pattern

```
(goo (a ...)  ...)
```

and the call

```
(goo (r s t) (u v w) (x y z))
```

the environment

```
((goo . (0 goo)) (a . (2 ((r s t) (u v w) (x y z)))))
```

is made.

In the $\mathcal{D}[\![()]\!]$ line, bottom ($\perp$) signals an error if the macro call does not match the pattern specification. This error occurs when the pattern indicates a list structure should be present in the macro call and it is not there. For example, the macro call

```
(bad a)
```

matched against the pattern specification

```
(bad (x y))
```

is illegal. The pattern describes the second component of every call as consisting of a proper list with exactly two elements.

The natural numbers in the pairs from $N \times \mathit{call}$ measure the ellipsis-depth of the pattern variables. The portions of the macro calls associated with the pattern variables fall into two categories. First, if the ellipsis-depth is zero—the pattern

Auxiliary Functions:

$$
\begin{aligned}
&\mathit{combine}\colon \mathit{ENV} \to \mathit{ENV}^* \to \mathit{ENV},\\
&\mathit{decompose}\colon \mathit{ENV} \to \mathit{ENV}^*,\\
&\mathit{splita}\colon (N \times \mathit{call}) \to (N \times \mathit{call}),\\
&\mathit{splitb}\colon (N \times \mathit{call}) \to (N \times \mathit{call});
\end{aligned}
$$

$$
\begin{aligned}
\mathit{combine}\,\rho\,(\rho_1 \ldots \rho_n) = \lambda v.\, v \in \mathrm{Dom}(\rho_1)\\
&\to \langle (\rho_1 v \downarrow 1) + 1, ((\rho_1 v \downarrow 2) \ldots (\rho_n v \downarrow 2)) \rangle,\\
&\quad \rho v;
\end{aligned}
$$

$$
\begin{aligned}
\mathit{decompose}\,\rho = \exists\, v \in \mathrm{Dom}(\rho) \text{ such that } (\rho v \downarrow 2) = (\,)\\
&\to \text{ nil},\\
&\quad \mathit{cons}\,\{(v_1,\ \mathit{splita}\,(\rho v_1)), \ldots, (v_k,\ \mathit{splita}\,(\rho v_k))\}\\
&\qquad \mathit{decompose}\,\{(v_1,\ \mathit{splitb}\,(\rho v_1)), \ldots, (v_k,\ \mathit{splitb}\,(\rho v_k))\}\\
&\qquad \text{where } \{v_1, \ldots, v_k\} = \mathrm{Dom}(\rho);
\end{aligned}
$$

$$
\mathit{splita}\,\langle n, c\rangle = (n \stackrel{?}{=} 0) \to \langle 0, c\rangle, \langle n - 1, (\mathit{hd}\ c)\rangle;
$$

$$
\mathit{splitb}\,\langle n, c\rangle = \langle n, (n \stackrel{?}{=} 0) \to c, (\mathit{tl}\ c)\rangle.
$$

Figure 5.16. *Formal semantics of ellipsis* (3).

variable is not part of an ellipsis list prototype—then the associated value is the part of the call that corresponds directly, in terms of list structure, to the pattern variable. Second, if the ellipsis-depth is greater than zero, the associated value is the list corresponding to the outermost prototype and ellipsis containing the pattern variable.

The auxiliary function *combine* requires that

$$
\mathrm{Dom}(\rho_1) = \cdots = \mathrm{Dom}(\rho_n)
$$

and that for all $v \in \mathrm{Dom}(\rho_1)$ and $i, j = 1, \ldots, n$,

$$(\rho_i v \downarrow 1) = (\rho_j v \downarrow 1).$$

The function $\mathcal{T}$ uses the environment created by $\mathcal{D}$ and the transcription specification to create a transcribed macro call. When it encounters an ellipsis in the transcription specification, it determines the pattern variables occurring within that ellipsis' prototype and restricts the environment to a smaller environment containing only the prototype variables. The restricted environment is split by the function *decompose* so that the ellipsis-depth count can be reduced.

For example, *decompose* applied to the restricted environment

```
((a . (2 ((r s t) (u v w) (x y z)))))
```

returns the list of environments

```
(((a . (1 (r s t)))) ((a . (1 (u v w)))) ((a . (1 (x y z))))).
```

When $\mathcal{T}$ encounters an atomic symbol, it first determines whether that symbol is a pattern variable by checking the domain of the environment. If it is not, then the symbol is treated as a literal and included within the answer. If it is, the symbol is looked up in the environment, resulting in a depth-value pair. If the depth is zero that value is inserted in the answer. If the depth is not zero an error results.

The help function *Vars-of* returns a set comprised of the atoms in the transcription specification provided as its argument. Its functionality is

$$\mathit{Vars\text{-}of} : \mathit{trans} \rightarrow 2^{\mathit{atom}}.$$

Each prototype variable from the pattern has a pattern-specification depth d_p. This number is included in the environment returned by $\mathcal{D}$. Each variable from the transcription can also be associated with an ellipsis-depth d_e. If for at least

one pattern variable $d_p = d_e$ the transcription can proceed. If they are unequal for all variables in a transcription prototype an error results. Thus not every pattern variable need be nested to the appropriate depth. For example, in the declaration

```
(extend-syntax (frob) ()
   [(frob a b ...)
    (grob (a b) ...)])
```

the value of d_p for a is 0 while that of d_e is 1. The semantics allows this behavior by never changing (within *splita*) a pair of the form $\langle 0, c \rangle$. The environment returned by $\mathcal{D}$ for the pattern `(frob a b ...)` and call `(frob x 1 2)` is

```
((a . (0 x)) (frob . (0 frob)) (b . (1 (1 2)))).
```

The function *decompose*, if given the prototype `(a b)` from the transcription specification `(grob (a b) ...)`, decomposes this environment into the sequence of environments

```
(((a . (0 x)) (b . (0 1))) ((a . (0 x)) (b . (0 2)))).
```

If $d_p < d_e$ for all variables the algorithm indicates the error by returning $\perp$ from the $\mathcal{T}[\![(t\,\Delta)]\!]$ line of Figure 5.15. This case occurs when the ellipsis-depth of the transcription is greater than that of the pattern. For example, the macro declaration

```
(extend-syntax (fot) ()
   [(fot a ...) (bar (a ...) ...)])
```

is ill-formed.

If $d_p > d_e$ for any variable the algorithm indicates the error by returning $\perp$ from the $\mathcal{T}[\![a]\!]$ line of Figure 5.15. This is the restricted situation we mentioned at the beginning of this section. The `extend-syntax` facility allows such macros as

```
(extend-syntax (fob) ()
   [(fob a ...) (bar a)])
```

or

```
(extend-syntax (fog) ()
   [(fog (a ...) ...)
    (bar a ...)])
```

where the ellipsis-depths of the transcription occurrences of a are smaller than the depths of its pattern occurrences. Such cases are not described by this formal semantics. But extend-syntax permits it, treating the pattern variables as names of the first components of the ellipsis-lists. Thus

(fob 1 2 3 4) ⇒ (bar 1), and

(fog (1 2 3) (4 5 6) (7 8 9)) ⇒ (bar 1 2 3).

Moreover, an equally plausible expansion of

(fog (1 2 3) (4 5 6) (7 8 9))

is

(bar 1 4 7).

Earlier versions of extend-syntax produced this result. Both versions have their merits; we settled for one of them. Exactly what the semantics of nested ellipses specifications should be is an open question.

5.3. An Implementation of a Syntax Table

The syntax table's organization and use are described in Section 4.5. Here, its implementation is outlined.

We assume that the table is organized as some sort of associative memory, keyed off macro names. Each macro name has three items associated with it: a list of keywords (one of which is the macro name), a list of key identifiers, and an STF. The STF is responsible for implementing the pattern matching and error detection involved with the selection of an appropriate production rule.

There are two requisite functions for use of the syntax table. One, the function `add-to-syntax-table`, is a low-level system function designed to enter new macros in the table. It takes three arguments: the keywords, the key identifiers, and the STF. The other function, `syntax-table-lookup`, takes a macro call as an argument and returns the three items associated with the macro name. A third function, to remove macros from the table, is helpful.

The keywords and key identifiers are included in the table so that the macro expander can take advantage of that information.

5.4. The Implementation of `extend-syntax`

Like other programs that have existed for a long period of time, the code for `extend-syntax` has undergone a lot of revision, tending towards greater complexity as new features have been incorporated (and sometimes lost). The code from a running implementation was excerpted and revised for ease of exposition. The full version is found in Appendix B.

In this section, the identifier `ellipsis` represents "..." and the identifier `*pattern*` is bound to a unique symbol used as the formal parameter of the resultant STF. Since Common Lisp does not allow "..." as an identifier, those wishing to implement `extend-syntax` in that language must choose another symbol. One possibility is "___".

The top-level functions for

```
extend-syntax,
extend-syntax/code, and
syntactic-transform-function
```

are the same except for what each returns. The first declares a macro by adding an STF to the syntax table; it returns the macro name. The second does not declare a macro. Instead it pretty-prints the code for the STF described in the call. The third does not declare a macro, but it returns the STF as a Scheme function.

```
(add-to-syntax-table
   '(extend-syntax extend-syntax/code
       syntactic-transform-function with withrec ...)
   '()
   (lambda (l)
      (let ([keywords (cadr l)]
            [keyids (caddr l)]
            [production-rules (cdddr l)])
         `(add-to-syntax-table ',keywords ',keyids
             ,(main-body keywords production-rules)))))
```

Figure 5.17. *Top-level* `extend-syntax`.

The code for `extend-syntax` is given in Figure 5.17. Since it is itself a macro, it is declared using the low-level macro primitive `add-to-syntax-table`. There are no key identifiers, so that list is empty.

The function `main-body` (Figure 5.18)[15] performs three tasks. First, it assures that all identifiers in the `extend-syntax` expression are in an un-stamped form. Stamping takes place as part of our hygienic expansion algorithm that avoids the capturing problem. Since `extend-syntax` cannot cause this problem, it is safe to remove any stamps that may appear at this stage on the identifiers in an `extend-syntax` call. We explain this more fully in Chapter 6.[16]

Second, `main-body` sets the global flag `stop-gen` to `false`. This flag relates

[15] Unless we state otherwise, we present the code for all auxiliary functions by writing a definition clause from a single `letrec` expression. All these functions lie within the scope of one another, and the calls to `main-body` occur within this same scope.

[16] There are two unstamping functions: `unstamp` which makes a copy of the expression in addition to removing stamps and `unstamp-no-copy` which removes stamps and returns the same cons cells. The same cons cells are necessary in code containing embedded Scheme structures.

```
[main-body
   (let
      ([do-main-body
          (lambda (prod-rules keywords)
             (set! stop-gen false)
             `(lambda (,*pattern*)
                 ,(mk-match-exp
                     prod-rules
                     keywords
                     '()
                     '()))))])
      (lambda (keywords prod-rules)
         (do-main-body
            (unstamp-no-copy prod-rules)
            (unstamp-no-copy keywords))))]

[mk-match-exp
   (lambda (prod-rules keywords specials geners)
      (let ([do-prod-rule prod-rule-function])
         (ifify
            (append
               (map (lambda (prod-rule)
                       (if (fender-present? prod-rule)
                           (do-prod-rule
                              (car prod-rule)
                              (cadr prod-rule)
                              (caddr prod-rule))
                           (do-prod-rule
                              (car prod-rule)
                              '()
                              (cadr prod-rule))))
                    prod-rules)
               `((true (macro-use-error ,*pattern*)))))))]
```

Figure 5.18. *Two more preparatory functions.*

to the problems of terms following ellipses in pattern specifications, as in `ite` (Fig-

ure 5.1). There must, of course, be some way of determining when the partial list indicated by a prototype and ellipsis ends. In cases where the indicated list extends to the right parenthesis, there is no problem. But, in situations like

```
(a ... b c ...),
```

there must be some way of determining where the a's end. There are two permissible ways of doing this. The ellipsis list is either terminated by a keyword or by a list containing a keyword as its first component (Section 5.1.1). If either of these two phenomena occur, some special code must be emitted within the STF to take care of the difficulty. The flag `stop-gen` indicates whether this special code must be included. At the outset, the flag is set to false since the necessity of emitting the special code has not been determined.

And third, the function `main-body` begins the generation of an STF by setting up a `lambda` expression with whatever formal parameter is associated with the global symbol `*pattern*`. It calls the function `mk-match-exp` to build the body of the generated STF.

The bodies of the generated STFs—the pieces of code returned by `main-body`—always take the form of conditional expressions. Each STF is designed to perform a series of tests on the macro call. We set up the syntax table so that each STF contains code to generate expansions of all legal macro calls with a given macro name. The STF for `and` must cover all three possible transcriptions.

The conditional tests involve the matching of the call against the pattern specification and any fender tests stipulated in the macro declaration. Both components are incorporated into the test. For a transcription generator to be selected, a macro call must pass both parts of the test.

The function `mk-match-exp` sets up the basic conditional structure of an STF. The inner call to append returns a list of `cond` clause style pairs, a predicate and a

consequent. The predicate is the conditional which combines pattern matching and fender tests. The consequent is the transcription generator—a Scheme expression with free variable `*pattern*` which computes the replacement text.

The function `ifify` turns these clauses into a giant `if` expression. Except for the last, the error condition, each pair is generated by applying the function *prod-rule-function* to a production rule from the call to `extend-syntax`.

The *prod-rule-function*, closed within the indicated scope, is defined in Figure 5.19. It returns a pair. The first element of the pair is a predicate expression, combining the generated syntactic test (by `mk-matcher`) and the test represented by a fender, if it is present. The second element is the transcribing expression that is part of the actual code generation of the STF being created.

If there is a fender, code is emitted to copy it and unstamp any time-stamped identifiers. This time-stamping is part of the macro expansion algorithm. If `extend-syntax` were implemented separately, say in a naïve expansion system, the unstamping would be unnecessary. However, it is needed in our system because time-stamping alters the `eq?`-equality of identifiers; a time-stamped identifier is not `eq?` to an unstamped one. Because the fender may perform tests based on the presence of certain symbols, we insure that the test is applied to unstamped versions of the identifiers.

There are four directions to go from here: discussion of each of the three functions `mk-expander`, `mk-matcher`, and `partial-process` and consideration of the handling of new pattern variables via `with` and `withrec`. We do the last now and devote a special section to each of the three functions.

The help function `new-variables?` detects the presence of a `with` or `withrec` declaration within the transcription specification. The two lists `specials` and `geners`, which are both initially empty, are used to keep track of the two sorts of

```
(lambda (pat fender trans)
  `(,(if fender
         `(and ,(mk-matcher keywords pat)
               (let ([,*pattern* (unstamp ,*pattern*)])
                 ,(partial-process fender pat '() '())))
         (mk-matcher keywords pat))
    ,(if (new-variables? trans)
         (let* ([sprec (regroup-withs
                         trans
                         pat
                         specials
                         geners)]
                [newgeners (cadr sprec)])
           (letrecify (car sprec)
             (mk-expander pat (cddr sprec)
               (filter-geners
                 (append (getspecs (car sprec)) specials)
                 newgeners)
               newgeners true pat)))
         (mk-expander pat trans
           specials
           geners true pat))))
```

Figure 5.19. *The prod-rule-function.*

new variables: respectively, those declared as a single variable and those declared as a variable-ellipsis pair (Section 5.1.2).

The function `regroup-withs` returns a list called `sprec`. This list has three components. The third component is the basic transcription from the deepest part of the `with`. The second component is a list of "geners"—those symbols that are part of prototype-ellipsis `with` specifications. The first component contains the `with` or `withrec` specifications structure serially instead of nested. The first

component is easiest to describe by example. If the transcription is given as

```
(with ([a *] [b **])
      (withrec ([(c ...) ***])
               (with ([d ****])
                     basic-transcription))),
```

then the first component appears as

```
((with ([a *] [b **]))
 (withrec ([(c ...) ***]))
 (with ([d ****]))).
```

The purpose of this is to enable mapping down the new variable declarations.

The pp-rule expressions enclosed in the declarations are themselves processed by the function `partial-process`. The results of partial processing depend on the "specials" and "geners" known when the partial processing takes place. Thus, `regroup-withs` must keep track of the specials and geners at each stage in its recursive descent through the transcription. This creates the scoping of new variables, so that they may be used in enclosed pp-rule expressions. The code for `fibon` (Figure 5.7) contains an example. The definition of `regroup-withs` is in Figure 5.20.

The function `get-geners` extracts the prototype-ellipsis style declarations from a `with` list. The function `filter-geners` removes the geners from the list of specials.

In the STF, `with`s correspond to `let` expressions and `withrec`s to `letrec` expressions. The function `letrecify` (called in the *prod-rule-function*) takes the sequence of `with` and `withrec` specifications returned by `regroup-withs` and changes them into nested `let` and `letrec` expressions. An identifier that is used in either style `with` specification is also used as a bound identifier in the `let` or

```
[regroup-withs
 (lambda (trans pat specials geners)
   (if (new-variables? trans)
       (let* ([newgeners (get-geners (cadr trans))]
              [allspecials
                 (filter-geners
                    (append (map car (cadr trans)) specials)
                    newgeners)]
              [allgeners (append newgeners geners)]
              [gsp (regroup-withs (caddr trans) pat
                                  allspecials
                                  allgeners)]
              [weaver
                 (let ([g (if (eq? (car trans) 'withrec)
                              allgeners
                              geners)]
                       [s (if (eq? (car trans) 'withrec)
                              allspecials
                              specials)])
                   (lambda (pair)
                     (cond [(atom? (car pair))
                            (cons (car pair)
                                  (partial-process
                                     (cdr pair) pat s g))]
                           [(and (pair? (car pair))
                                 (eq? (cadr (car pair))
                                      ellipsis))
                            `(,(caar pair)
                              ,(partial-process
                                  (cadr pair) pat s g))]))))])
         (cons (cons (cons (car trans)
                           (map weaver (cadr trans)))
                     (car gsp))
               (cdr gsp)))
       (cons '() (cons geners trans))))]
```

Figure 5.20. *Regrouping and partial processing of new variable declarations.*

`letrec` expression. The sample declaration is transformed to

```
(let ([a -] [b --])
  (letrec ([c ---])
    (let ([d ----])
      -----))).
```

The `let` and `letrec` identifiers appear as free identifiers in the enclosed body.

5.4.1. THE PRINCIPAL FUNCTION. The overall structure of `mk-expander` has not changed since the very first version. It returns a Scheme expression in which the identifier associated with `*pattern*` is free. This expression is the one that actually generates the replacement text of a macro call.

Whenever `mk-expander` processes a transcription that does not contain an ellipsis, its action is straightforward. A simple transcription specification can be built out of various list building primitives, quoted constants, and car-cdr chain references to pieces of the pattern specification. However, when `mk-expander` works on a transcription with one or more ellipsis-lists, it recursively calls itself. To generate the code specified by an ellipsis-list, some sort of looping or mapping construction must be emitted. We have chosen to produce calls to `map`. The recursive invocation produces the body of the function that is to be mapped.

For an illustration, consider a few simple specifications and their generated STFs. First, there is the simple

```
(extend-syntax (foo) ()
   [(foo a b) (bar b a)]).
```

If the predicates involved in pattern matching are ignored, a suitable STF for this macro is

```
(lambda (gensym)
   (list 'bar (caddr gensym) (cadr gensym))).
```

It is easy to see that we need some method for producing the car-cdr chains which

locate the a and b in the pattern, and hence also in any call to this macro. The function which does this is called car-cdr-chain.

Next, consider a simple macro declaration involving the ellipsis:

```
(extend-syntax (foo) ()
   [(foo (a b) ...)
    (bar a ...)]).
```

The ellipsis in the transcription specification corresponds to a map of some function down the macro call. The desired code might be

```
(lambda (gensym)
   (cons 'bar (map car (cdr gensym)))).
```

The code generator, mk-expander, must be capable of determining appropriate calls to map.

A mapping expression requires both the list down which to map and the function that is to be mapped. The most involved part of mk-expander, a help function named mk-ellipsis-body, is concerned with computing the two significant components of a call to map. Suppose, for example, that there is a pattern specification

```
(foo (a b) ...)
```

and a transcription specification

```
(list 'a ...).
```

The desired generated code is then something like

```
(cons 'list
      (map (lambda (*) body)
           (cdr gensym)))
```

where *gensym* is the free, formal parameter of the STF body. The expression for *body* is computed by first locating the prototype from the pattern, (a b), and the

```
[mk-expander
   (lambda (abrv trans specials geners not-nesting fullabrv)
      (when abrv
         (letrec
            (more definitions
             [mke (lambda (trans)
                    (if (atom? trans)
                        (mk-atom trans)
                        (begin
                           (when (memq (car trans) mknames)
                                 (set! not-nesting false))
                           (mk-one-liner trans)]))])
            (mke trans))))]
```

Figure 5.21. *The principal function* `mk-expander`.

prototype from the transcription, `'a`. We want an expression that converts the first to the second. It is obtained by recursively calling `mk-expander` with new pattern specification `(a b)` and new transcription specification `'a`. The result is the expression

```
(list 'quote (car *)).
```

There is some cleaning to do so that the formal parameter of the mapping `lambda` expression—represented by the star—corresponds to the free, formal parameter in the expression returned by the recursive call to `mk-expander`. The final product is

```
(lambda (gensym1)
   (cons 'list
         (map (lambda (gensym2)
                (list 'quote (car gensym2)))
              (cdr gensym1)))).
```

The function `mk-expander` (Figure 5.21) is called with six arguments; they are documented in Figure 5.22. A large number of help functions, closed upon each

`abrv`	The current pattern specification
`trans`	The current transcription specification
`specials`	New variables
`geners`	Repeated new variables
`not-nesting`	Flag indicating a nested declaration
`fullabrv`	The initial pattern

Figure 5.22. *The formal parameters of* `mk-expander`.

invocation, are created. These functions have one or more of the formal parameters to `mk-expander` as free identifiers within their bodies. When we set forth their definitions in this section, we assume that they are closed within the appropriate lexical scope.

The `mknames` are the names of the three `extend-syntax` style special forms (Section 5.1.4).

The two help functions called by `mke` are relatively tractable. First, there is `mk-atom` (Figure 5.23), for processing atomic symbols.

If the symbol represented by `atm` is neither a Scheme constant, a special, nor a gener and appears in the pattern `abrv`, a car-cdr chain is produced that locates the symbol in the pattern. Now, it may not appear in the pattern `abrv`, especially if the call to `mk-atom` occurs during a recursive invocation of `mk-expander`. In such a case, `mk-expander` is attempting to produce the body of the function for mapping down a list. The target atom may not be in the mapping prototype, so an effort is made to locate it in the original pattern, `fullabrv`. If the atom is nowhere to be found, the code to include it as a literal is emitted.

The other function, `mk-one-liner` (Figure 5.24), is also easy. It recursively calls itself, generating the body of a function that can produce a replacement text.

```
[mk-atom
   (lambda (atm)
      (cond [(scheme-const? atm) atm]
            [(memq atm specials) atm]
            [(memq atm geners) atm]
            [(member* atm abrv)
             (car-cdr-chain atm abrv *pattern*)]
            [(member* atm fullabrv)
             (car-cdr-chain atm fullabrv '*temp*)]
            [else `(quote ,atm)]))]
```

Figure 5.23. *The atom processing function* mk-atom.

```
[mk-one-liner
   (lambda (trans)
      (cond [(atom? trans) (mk-atom trans)]
            [(atom? (cdr trans))
             `(cons ,(mk-one-liner (car trans))
                    ,(mk-atom (cdr trans)))]
            [(memq ellipsis trans) (mk-ellipsis-body trans)]
            [else (if (member* trans abrv)
                      (car-cdr-chain trans abrv)
                      (let ([ans (mk-regular-body trans)])
                         (if (all-quote? ans)
                             `(quote ,(remquote ans))
                             `(list . ,ans))))]))]
```

Figure 5.24. *The function* mk-one-liner.

Its `else` clause contains two efficiency routines. If a `trans` can be located (by `member*`) within the current pattern, the car-cdr chain to that location is returned. And if the body is a list of constants, it is transformed into a quoted list.

```
[mk-regular-body
   (lambda (trans)
      (when trans
         (let ([firstex (car trans)])
            (if (atom? firstex)
                `(,(mk-atom firstex)
                   . ,(mk-regular-body (cdr trans)))
                `(,(mke firstex)
                   . ,(mk-regular-body (cdr trans)))))))])
```

Figure 5.25. *Processing transcriptions without ellipses in* `mk-regular-body`.

The next two functions, `mk-regular-body` and `mk-ellipsis-body`, deal with generating the code for replacement texts for transcription specifications without and with ellipses, respectively. The two functions differ because of the nature of the generated code. If the transcription specification includes an ellipsis, a call to the function `map`, as explained above, is to be produced. Each ellipsis corresponds to a call to `map`.[17] If the specification contains no ellipses, then the generated code consists of calls to `cons`, `list`, `append`, and car-cdr chains. The ellipsis case is handled by `mk-ellipsis-body` and the other by `mk-regular-body`.

The easier of the two is `mk-regular-body` (Figure 5.25). It recursively cdr's down the transcription specification, using `mk-atom` and `mke` to handle the separate situations of atomic and non-atomic car's. It is straightforward; the backquotes produce the appropriate code.

The creation of code generators for transcription specifications involving ellipses occurs in stages. As with the main function `mk-expander`, we present first

[17] This is not true all of the time. If a transcription ellipsis list duplicates a pattern ellipsis list and if this duplication is detected, then a car-cdr chain is emitted instead of a call to `map`.

```
[mk-ellipsis-body
  (lambda (el-trans)
    (let ([prototype (let ([temp (pt&mo el-trans)])
                       (if temp (car temp) '()))])
      (if (equal? (car el-trans) prototype)
          (let*
            ([used-geners (geners-in prototype geners)]
             [dotted-lists (find-dls prototype abrv)]
             [pt&mo-abrvs (map pt&mo dotted-lists)]
             [pt&mostops (map mostopsfun pt&mo-abrvs)]
             [mapfunbod value]
             [elocs value]
             [newid (gensym)]
             [mapfun value]
             [el-part value])
            (if (cddr el-trans)
                `(append ,el-part ,(mke (cddr el-trans)))
                el-part))
          (let
            ([exd
               `(cons
                    ,(mke (car el-trans))
                    ,(mk-ellipsis-body (cdr el-trans)))])
            (if (idfun? exd)
                (cadr (cadr exd))
                exd)))))]
```

Figure 5.26. *A skeleton of the function* `mk-ellipsis-body`.

the skeleton for `mk-ellipsis-body` (Figure 5.26) and then proceed to discuss the supporting routines. A description of each of the nine values computed by the `let*` expression is given in Figure 5.27.

The first step is finding the prototype in the transcription specification. The function `pt&mo` recurs down `el-trans` until it finds an ellipsis in the cadr. It is like

`used-geners`	The geners used in the prototype
`dotted-lists`	The ellipsis lists from the pattern
`pt&mo-abrvs`	Prototype-ellipsis list tails of the ellipsis lists
`pt&mostops`	Terminating keywords of prototype-ellipsis lists
`mapfunbod`	The body of the function of the `map` expression
`elocs`	The car-cdr chain locations of the ellipsis-lists
`newid`	The formal parameter of the mapped function
`mapfun`	The generated, mapped function
`el-part`	The complete `map` expression

Figure 5.27. *Values computed during* `mk-ellipsis-body`.

a version of `memq` that looks ahead one term. The car of the result is the prototype.

We next make sure that the prototype heads the transcription specification. We are getting ready to emit a `map` expression and want to take care of any terms that may precede the actual prototype and ellipsis. The code is inefficient; it computes the prototype on each recurrence while cdr-ing down the `el-trans`. In practice this inefficiency is negligible since the lists involved in transcription specifications are usually quite short.

Because the geners are defined as prototype-ellipsis combinations, they must be taken into account. The prototype may contain identifiers that were declared as new, repeated pattern variables. The function `geners-in` locates any declared geners that appear in the prototype. It returns a list of all the used gener names.

An ellipsis in the transcription specification corresponds, in general, to a call to the function `map` in the STF being produced. During macro transcription, the mapping occurs simultaneously down all components of the macro call that correspond to the pattern variables from the ellipsis prototype. The routine `find-dls` locates all lists in the current pattern `abrv` that contain an ellipsis and correspond

```
(find-dls 'a '(foo a ...))
⇒ ((foo a ...))

(find-dls 'a '(foo ((a b) ...)))
⇒ (((a b) ...))

(find-dls 'b '(foo a ... bar b ...))
⇒ ((bar b ...))

(find-dls '(a b) '(foo (a ...) (b ...)))
⇒ ((a ...) (b ...))

(find-dls '((a ...) b) '(foo (a ...) (b ...)))
⇒ ((b ...))

(find-dls 'a '(foo (a ...) ...))
⇒ ((a ...))

(find-dls '(a ...) '(foo (a ...) ...))
⇒ ((foo (a ...) ...))
```

Figure 5.28. *The specification of* `find-dls`.

to that prototype. We omit the code in this section, since it is not very illuminating, and provide a "specification by example" for all the cases with which it deals (Figure 5.28). It always returns a list of ellipsis-lists.

The semantics of nested occurrences of ellipses are not well-defined. Many nested expressions have more than one plausible interpretation. Consider the macro declaration

```
(extend-syntax (foo) ()
   [(foo (a (b ...)) ...)
    (bar (a b) ...)]).
```

Should the macro call `(foo (x (1 2)) (y (3 4)))` transcribe to either

```
(bar (x 1) (y 2)),
(bar (x 1) (y 3)),
```

or something else? The formal semantics of the last section gives a semantics for simple cases of nested ellipses, but it does not specify what should happen in situations like this. The current implementation of `extend-syntax` does not resolve all such cases either. Designing a satisfactory solution to the problem of nested ellipses is an area for further research. Additional illustrations of the problem are given at the end of the preceding section and in Section 7.3.1.

After finding the dotted-lists, the next step is to find the prototype-and-more's from the ellipsis-lists just found. Since the `dotted-lists` are the entire lists corresponding to the prototype, we locate the actual prototypes of these lists with `pt&mo`. The result is a list of the tails of the `dotted-lists` beginning with the prototypes.

Since some ellipsis-lists in the pattern specification are terminated by keywords or lists with keywords as their first components, we also need to figure out these terminating conditions. The function `mostopsfun` does this. If any terminated lists are found, the flag `stop-gen` is set to true. The list of terminators is returned. For example, if the transcription prototype is `a` and the pattern specification is

```
(foo a ...  bar b ...)
```

where `bar` is a keyword, then

```
dotted-lists    is    ((foo a ... bar b ...)),
pt&mo-abrvs     is    ((a ... bar b ...)), and
pt&mostops      is    (bar).
```

The recursive call to `mk-expander` (Figure 5.29) described above is made. The transcription `prototype` becomes the new transcription argument. Specials are

```
[mapfunbod
   (mk-expander
      (append-geners
         (if (null? pt&mo-abrvs) '()
             (if (cdr pt&mo-abrvs)
                 (car (transpose
                        (map (lambda (pta) (list (car pta)))
                             pt&mo-abrvs)))
                 (caar pt&mo-abrvs)))
         used-geners)
      prototype specials '() not-nesting fullabrv)]
```

Figure 5.29. *The recursive call in* `mk-expander`.

carried over as is. Geners are taken care of in the `used-geners` list, so a null argument is passed for them. The nesting flag and the full pattern `fullabrv` are passed as is. The only complicated part is the pattern argument.

If there is only one element in the list `pt&mo-abrvs`, the first element of the first element is a prototype. It becomes the new pattern if there are no used geners. If there is more than one pattern prototype, the map needs to take place down two or more lists. Since we are using the version of `map` that traverses only one list, we transpose the prototypes into one list. Appending the `used-geners` with the `pt&mo-abrvs` finishes the task of computing a new pattern. The special function `append-geners` takes care of this:

```
[append-geners
   (lambda (ptabrvs geners)
      (if (pair? geners)
          (if (null? ptabrvs)
              (if (null? (cdr geners)) (car geners) geners)
              (cons ptabrvs geners))
          ptabrvs))].
```

```
[elocs
   (append
      (dotted-list-locs dotted-lists pt&mo-abrvs
         (map
            (lambda (dl) (car-cdr-chain dl abrv))
            dotted-lists))
      used-geners)]
[dotted-list-locs
   (lambda (dotted-lists pt&mo-abrvs chains)
      (letrec ([dlls
                  (lambda (l pt chain)
                     (if (eq? l pt)
                         chain
                         (dlls (cdr l) pt (ad 'd `(,chain)))))])
         (map
            (lambda (ls) (dlls (car ls) (cadr ls) (caddr ls)))
            (transpose
               (list dotted-lists pt&mo-abrvs chains)))))]
```

Figure 5.30. *Computing the locations of the* `dotted-lists`.

The car-cdr chain locations of the `dotted-lists` (Figure 5.30) are computed and appended to the list of `used-geners`. Geners, of course, have no car-cdr location within the pattern. The special and gener symbols are used as the `let` or `letrec` variables representing the value. As before, `transpose` is used because the mapped function takes only one argument.

At this point the body of the mapped function, `mapfunbod`, has been computed. The locations of the various components, `elocs`, that the function maps down have been determined. A new formal parameter, `newid`, for the mapped function is obtained, and it is substituted for the old parameter in the body by calling `*pattern*-subst`. Before doing so, some checks for possible optimization with `idfun?` are made. This function checks for two simple special cases: when

mapfunbod is a single atom or when it cons's the car of a list onto its cdr. Both situations represent the identity function

```
(lambda (x) x).
```

The line which does this work is

```
[mapfun
   (if (idfun? mapfunbod)
       newid
       (*pattern*-subst newid mapfunbod))].
```

Further tests of this sort—to detect the identity function in more of its guises—can be done; we do not feel it is worth the effort.

The entire map expression is put together by computing el-part (Figure 5.31). The initial cond test makes sure we have located the desired lists. If not, we determine whether we are trying to process a nested specification (Section 5.1.4). If there is nesting, the transcription el-trans is incorporated as is.

Under some circumstances, the recursive call to mk-expander cannot compute the body of the mapped function. This happens when the prototype contains an ellipsis list whose prototype is not found in the dotted-lists, for example,

```
(extend-syntax (foo) ()
   [(foo (a ...) (b ...))
    (bar ((a ...) b) ...)]).
```

When determining the mapped function corresponding to the outer transcription prototype ((a ...) b), the ellipsis list (b ...) is found by find-dls, but not (a ...). So the recursive call to mk-expander tries to determine the transcribing expression with pattern (b ...) and transcription ((a ...) b). Thus no ellipsis-list for (a ...) can be found in the new pattern. This is the case handled by the "not-nesting" cond clause. If a transcribing expression based on the original pattern can be computed, we do so, taking care to avoid conflicting formal parameter names.

```
[el-part
   (cond
      [(and elocs (car elocs))
       (if (and (atom? mapfun)
                (not (and stop-gen (car pt&mostops))))
           (if (cdr elocs)
               `(transpose (list . ,elocs))
               (car elocs))
           (let*
              ([map-l value]
               [map-exp value])
              map-exp))]
      [not-nesting
         (let ([try (subst '*temp* *pattern*
                           (mk-expander fullabrv el-trans
                              specials geners not-nesting '()))])
           (if try try
              (mkmac-prototype-error prototype abrv)))]
      [true `',el-trans])]
```

Figure 5.31. *Computing the* `map` *expression.*

There are three cases within the first `cond` clause: (1) whether the body of the mapped function, `mapfun`, is a single symbol and whether there is no terminating keywords, (2) if the first case is true, whether there is more than one list to map down, and otherwise (3) the general situation of a non-identity function mapped function, more than one list, or the presence of terminated lists.

In the general situation, we first compute the code for the list or lists we plan to map down (Figure 5.32). If there is more than one, a call to `transpose` must be omitted. No matter how many there are, if any of them are keyword terminated lists, they must be dealt with also.

The function `trim` chops off the terminated lists. It is invoked while transcribing

```
[map-l
   (if (cdr elocs)
       `(transpose
           (list .
              ,(if (and stop-gen (car pt&mostops))
                   (map
                      (lambda (el) `(,trim ,el ',pt&mostops))
                      elocs)
                   elocs)))
       (if (and stop-gen (car pt&mostops))
           `(,trim ,(car elocs) ',pt&mostops)
           (car elocs)))]
```

Figure 5.32. *Computing the code for the second argument in a call to* `map`.

a macro call. It returns the beginning of a list, stopping immediately before the terminator. We omit its code, noting that the only slightly tricky part is recognizing lone keyword and keyword-as-first-component terminators.

The entire `map` expression is put together in `map-exp` (Figure 5.33). If `mapfun` is an atom, the identity function is to be mapped down the list. So, the list itself is returned instead of emitting a call to `map`. If the body of the mapped function is of the form (*f newid*), then in place of a `lambda` expression, we emit the function *f*.

That finishes the discussion of `mk-ellipsis-body`. We should, however, say something about `car-cdr-chain`. It is relatively straightforward except for two things. First, it not only finds the car-cdr chain of atomic symbols within lists, but it also finds them for sub-lists. Second, terminated lists cause difficulties because of locating the terminator first and taking car's and cdr's from that point. For example,

```
(car-cdr-chain 'b '(foo a ... bar b ...))
⇒ (cadr (stopq 'bar pattern)).
```

```
[map-exp
   (if (atom? mapfun)
       map-l
       `(map
           ,(cond
               [(and (atom? (cadr mapfun))
                     (null? (cddr mapfun)))
                (car mapfun)]
               [true `(lambda (,newid) ,mapfun)])
           ,map-l))]
```

Figure 5.33. *Putting together the entire call to* `map`.

Keywords cannot, in general, be repeated within a single pattern specification, for the auxiliary keywords are used to locate the end of ellipsis-list segments. If the keyword is repeated, it is ambiguous which one is desired.

The function `stopq` does an `eq?` test on successive terms in the list *pattern* or, if the term is itself a list, on the left-most atom of the term:

```
(define stopq
   (lambda (a b)
      (let ([key (if (pair? a) (leftmostat a) a)])
         (recur loop ([b b])
            (cond [(null? b) '()]
                  [(or (eq? key (car b))
                       (and (pair? (car b))
                            (eq? key (caar b))))
                   b]
                  [else (loop (cdr b))])))))).
```

5.4.2. THE GENERATION OF PATTERN MATCHERS. The pattern matchers are responsible for comparing macro calls with declared pattern specifications. Recall that the *prod-rule-function* called during `mk-match-exp` invokes the function `mk-matcher` in order to produce the predicate corresponding to a given pattern

specification. Its two arguments are the keywords list k and the pattern specification p:

```
(define mk-matcher
   (lambda (k p)
      (mm '(cdr ,*pattern*) k (cdr p)))).
```

Work begins with the cdr of the pattern because of the assumption that the macro system has matched the macro name in the process of choosing an STF from the syntax-table. The first argument to the help function mm (Figure 5.34) is the answer. It is initially code that takes the cdr of the macro call.

The pattern matchers have two goals: first, to check the position of keywords (except the macro name) within macro calls and second, to be sure that the list length of all components of the macro call corresponds to what is specified in the pattern. Both goals are complicated by the presence of ellipses.

An example of the code generated by mk-matcher and mm is found in the STF for the macro let in Figure 5.12.

The portion of the cond clause of Figure 5.34 that deals with ellipsis lists appears in Figure 5.35.

The function for-all maps a predicate down a list, seeking whether the predicate is true of all terms in the list:

```
(define for-all
   (lambda (f l)
      (if (null? l)
          true
          (and (f (car l)) (for-all f (cdr l)))))).
```

The function ad takes a car/cdr invocation and composes it with another car or cdr, depending on whether the first argument is a or d. It allows the customary

```
(define mm
   (lambda (a k p)
      (cond
         [(null? p) `(null? ,a)]
         [(memq p k) `(eq? ,a (quote ,p))]
         [(not (pair? p)) true]
         [(and (pair? (cdr p)) (eq? (cadr p) ellipsis))
          matching an ellipsis list]
         [else
            (let ([ans (remq true
                             `(and (pair? ,a)
                                   ,(mm (ad 'a `(,a)) k (car p))
                                   ,(mm (ad 'd `(,a)) k (cdr p))))])
               (if (= (length ans) 2)
                   (cadr ans)
                   ans))])))
```

Figure 5.34. *The function* `mm`*, which helps generate pattern matchers.*

Lisp car/cdr compositions of up to four combinations of a's and d's, as

```
(ad 'a '(cdr x)) ⇒ (cadr x),
(ad 'a 'x) ⇒ (car x),
(ad 'a '(cdddr x)) ⇒ (cadddr x), and
(ad 'd '(cadddr x)) ⇒ (cdr (cadddr x)).
```

5.4.3. PARTIAL PROCESSING. During the processing of fenders and `with` specifications, parts of an `extend-syntax` declaration are partial-processed (Figure 5.36). A pp-rule expression consists of quoted and unquoted parts. The unquoted parts are incorporated essentially as is into the predicate or `let/letrec` portions. The quoted parts are processed by `mk-expander`. This is illustrated in Figure 5.13.

5.4.4. CONCLUDING REMARKS ABOUT THE IMPLEMENTATION. Code without ellipses would be notably simpler. And even if there were ellipses but not terminated

```
(cond
   [(and (pair? (cddr p)) (memq (caddr p) k))
    (let ([temp (gensym)])
       `(and (pair? ,a)
             (let ([,temp (stopq (quote ,(caddr p)) ,a)])
                (and ,temp ,(mm temp k (cddr p))))))]
   [(and (pair? (cddr p))
         (pair? (caddr p))
         (memq (car (caddr p)) k))
    (let ([temp (gensym)])
       `(and
          (pair? ,a)
          (let ([,temp (stopq (quote ,(car (caddr p))) ,a)])
             (and ,temp ,(mm temp k (cddr p))))))]
   [(null? (cddr p))
    (let ([ans (let* ([arg (gensym)]
                      [ans (mm arg k (car p))])
                 (if (eq? ans true)
                     true
                     `(for-all
                         (lambda (,arg) ,ans)
                         ,a)))])
       `(or (null? ,a)
            ,(if (eq? ans true)
                 `(pair? ,a)
                 `(and (pair? ,a) ,ans)))))]
   [else (pattern-error p)])
```

Figure 5.35. *The* `cond` *clause from the function* `mm`.

ellipsis-lists, the program would be much easier. We cannot, however, do without either if we are to be able to conveniently express all of the special forms from the Revised[3] Report on Scheme in terms of the core language.

What "conveniently" means is largely an aesthetic judgment. Any macro that could be declared in the old style—by writing an STF—can be declared with `extend-`

```
(define partial-process
   (lambda (p abrv specials geners)
      (letrec
         ([pp (lambda (p)
                (cond
                   [(atom? p) p]
                   [(null? (cdr p)) (list (pp (car p)))]
                   [(eq? (car p) 'quote)
                    (mk-expander abrv (cadr p) specials geners
                       true abrv)]
                   [else (cons (pp (car p)) (pp (cdr p)))]))])
         (pp p))))
```

Figure 5.36. *The definition of partial-processing.*

syntax in the fashion of the macro hairy (Section 5.1.2).

The best suggestion on understanding how extend-syntax behaves is to follow the path we take in working with it. We begin with "regular" pattern and transcription specifications—no ellipses. We then write an STF which accomplishes our desired transformation. The question then becomes one of how to generate the STF. Since it is a list structure, we of course use the list-building primitives cons and list. Each atomic symbol is either a Scheme constant, a pattern variable, or a literal; the appropriate generating code is emitted in each case.

After grasping how regular lists are processed, we move on to ellipsis lists and consider how the requisite map expressions are produced. Then we think about the other details, such as new pattern variables, fenders, and the pattern matching. Each of these four aspects is independent from the others.

We talk about the impact of extend-syntax on our Scheme programming and its influences on our thoughts about macros in Chapter 7.

6. Expanding Syntactic Extensions

In this chapter we take up the issue of expanding syntactic extensions. Formal descriptions of the common naïve expansion algorithm and of the *hygienic expansion algorithm* are given. The difference between the two is that the hygienic algorithm takes into account the capturing problem and prevents it.

The first section of this chapter presents a formal description of the naïve expansion algorithm. It includes a discussion of some of the techniques that have been used by Lisp programmers to avoid the capturing problem. The next section presents a formal description of the hygienic expansion algorithm for a host language that is a subset of the Scheme host language. It contains a theorem that describes the difference between naïve and hygienic expansion. The third section deals with modifying the expansion algorithm to take into account key identifiers. The fourth section discusses extending the formal host language to that of the Scheme core. The chapter concludes with a section of miscellaneous details related to the algorithm's implementation.

6.1. Naïve Macro Expansion

An expression syntactic extension expander maps expressions from an extended programming language to a host language. Hence, we begin the presentation of the naïve expansion algorithm with a discussion of these two languages. Everything

said about them in this part of this section also applies to the languages the hygienic expansion algorithm uses.

The most important aspect of the host language with respect to the capturing problem is its binding mechanism. We have chosen to use the λ-calculus as it is the prototype of a programming language in which identifiers occur free and bound. It is syntactically simple, yet contains all the required elements to make the case interesting; it has the right level of complexity. Furthermore, it is a trivial task to generalize the algorithms for the λ-calculus to Scheme core.

The variant of the λ-calculus used as the host language is defined by the grammar

$$\begin{aligned} \lambda term ::= \; & var \mid \\ & const \mid \\ & (\textbf{lambda}\ var\ \lambda term) \mid \\ & (\lambda term\ \lambda term). \end{aligned}$$

The tokens (,), and **lambda** are terminal symbols. The symbol **lambda** is the only element in the set of core tokens: $coretok = \{\textbf{lambda}\}$. The set *const* includes all constants commonly found in Scheme such as strings, vectors, numbers, closures, and so forth. The set *var* is composed of Lisp symbols that are used as identifier names; it is disjoint from the set of core tokens.

As is customary, identifier and constant expressions denote arbitrary values. Abstractions, or **lambda** expressions denote procedures of one formal parameter. The occurrence of an identifier in the declaration part of a **lambda** expression is its *binding instance*. Applications correspond to function invocations.

The source language for the macro processor is an extension of the host language. It must allow for one kind of expression—the syntactic extension—that is specified in a rather general way. We refer to the source language as the language *Syntax-tree*

(Section 2.3.2) and define it inductively by

$$\begin{aligned} \mathit{stree} ::= {} & \mathit{var} \mid \\ & \mathit{const} \mid \\ & \mathit{mstree} \mid \\ & (\textbf{lambda}\ \mathit{var}\ \mathit{stree}) \mid \\ & (\mathit{stree}\ \mathit{stree}). \end{aligned}$$

The set *mstree* is the sublanguage of *syntactic extensions*.

Macro calls are recognized by the presence of macro names, that is, elements of a distinguished set *mactok*. A macro name is always the first component of a macro call, which is itself an arbitrarily long syntax tree:

$$\mathit{mstree} ::= (\mathit{mactok}\ \mathit{stree}_1 \ \ldots\ \mathit{stree}_n) \text{ for all } n \geq 0.$$

6.1.1. A FORMAL DESCRIPTION OF THE NAÏVE EXPANSION ALGORITHM. Equipped with these definitions, we can define an expansion function $\mathcal{E}_{naive}$ which effects naïve expansion. It is given in Figure 6.1. Being a standard, recursive descent algorithm, it fully expands each macro call. The Scheme-coded version of it would be a suitable definition of the function expand that is called by the parser, provided, of course, that the Scheme version handled the full host language.

The domain *keyvar* is the domain of key identifiers. They are ignored until Section 6.4.

The first semantic domain STF consists of the STFs. The second, ST, is the domain of syntax tables. As previously described (Section 5.3), each syntax table maps a macro call to a pair consisting of key identifiers (*keyvar*) and an STF.

The value $(\vartheta e \downarrow 2)$ is the STF corresponding to the macro call e; the value $(\vartheta e \downarrow 2)e$ is its transcription.

6.1.2. SOME WAYS OF AVOIDING THE CAPTURING PROBLEM. One of the common solutions to the capturing problem uses bizarre or freshly created identi-

Syntactic Domains:

$c \in const$ constant names,
$v \in var$ identifier names,
$m \in mactok$ macro tokens,
$k \in keyvar$ key identifiers,
$e \in mstree$ macro expressions,
$s \in stree$ expressions.

Syntax:

$$s ::= c \mid v \mid e \mid (\textbf{lambda}\ v\ s) \mid (s_1\ s_2),\ s_1 \notin mactok,$$
$$e ::= (m\, s_1 \ldots s_n) \text{ for } n \geq 0 \text{ with the above restriction.}$$

Semantic Domains:

$$K = \{k_1, \ldots, k_n\} \text{ for } n \geq 0,$$
$$STF = mstree \to stree,$$
$$\vartheta \in ST = mstree \to (K, STF),$$
$$\lambda term.$$

Semantic Functions:

$\mathcal{E}_{naive} : stree \to ST \to \lambda term;$

$$\mathcal{E}_{naive}[\![c]\!] = \lambda\vartheta.\, c,$$
$$\mathcal{E}_{naive}[\![v]\!] = \lambda\vartheta.\, v,$$
$$\mathcal{E}_{naive}[\![e]\!] = \lambda\vartheta.\, \mathcal{E}_{naive}[\![(\vartheta e \downarrow 2)e]\!]\vartheta,$$
$$\mathcal{E}_{naive}[\![(\textbf{lambda}\ v\ s)]\!] = \lambda\vartheta.\, (\textbf{lambda}\ v\ \mathcal{E}_{naive}[\![s]\!]\vartheta),$$
$$\mathcal{E}_{naive}[\![(s_1\ s_2)]\!] = \lambda\vartheta.\, (\mathcal{E}_{naive}[\![s_1]\!]\vartheta\ \mathcal{E}_{naive}[\![s_2]\!]\vartheta).$$

Figure 6.1. *Naïve macro expansion.*

fier names for macro-generated bindings. Another solution involves the freezing—closing—of user-code at the right time in the correct environment (Steele and Sussman, [71]). In Figure 6.2, the macro `(or` *exp1 exp2*`)` is declared using each of these three techniques.

It is clear that bizarre names only lower the probability of the problem occur-

```
                                        ; bizarre identifier
(extend-syntax (or) ()
   [(or exp1 exp2)
    (let ([***bizarre-or-id*** exp1])
       (if ***bizarre-or-id***
           ***bizarre-or-id***
           exp2))])

                                        ; freshly created identifier
(extend-syntax (or) ()
   [(or exp1 exp2)
    (with ([newsym (gensym)])
          (let ([newsym exp1])
             (if newsym newsym exp2)))])

                                        ; freezing and thawing
(extend-syntax (or) ()
   [(or exp1 exp2)
    (let ([v exp1]
          [thunk (lambda () exp2)])
       (if v v (thunk)))])
```

Figure 6.2. *Three specifications of the macro* (or *exp1 exp2*).

rence, but do not eliminate it. The freshly created identifier approach works only if the macro writer always correctly specifies which identifiers are to be so considered. Freezing and thawing user-code is even more complicated since it has to be done in the right environment. These solutions suffer from the same drawback; the macro writer is responsible for their realization.

6.2. Hygienic Macro Expansion

Naïve macro expansion violates Principle 12 because of the unexpected capturing of free identifiers by generated, binding instances of the same identifiers. In this section, we present an expansion algorithm that does not have this problem.

The capturing problem of the naïve expansion algorithm is analogous to the

substitution problem in the λ-calculus (Barendregt, [3]; Church, [17]). When an expression M with free identifiers is to be substituted into an expression N, the binding identifiers of N must be different from the free ones in M. Bindings in N must not capture free identifiers in M. Barendregt calls this a "hygiene condition" [4].

We want to impose something similar to this hygiene condition on macro expansion. With few exceptions—the key identifiers—we do not want generated, binding instances to capture apparent identifiers from a macro call. Reviewing the naïve expansion sequence

```
(or exp1 exp2)

⇒ (let ([v exp1])
       (if v v exp2))

⇒ ((lambda (v) (if v v exp2))
   exp1),
```

we make the observations:

1. Full expansion of a macro call consists of a tree of transcription steps. The tree branches when, on recursive descent, embedded macro calls are found.
2. In the source language the only binding form known to the expander is the core form `lambda`. Thus, we cannot determine which instances cause bindings until the expression has been expanded down to the level of `lambda` expressions.
3. When a macro writer constructs a macro that creates a binding instance of an identifier, he intends to bind only those instances that are generated during the transcription step that introduces the binding. For example, the transcription from the `or` expression to the `let` expression creates a binding instance of the identifier `v`, even though we cannot yet tell that it is. The only instances of `v` that we can allow to be captured by this binding instance are those that are

generated during the same transcription, the ones inside the `if` expression. If the expression *exp2* contains a macro call that generates a free occurrence of `v`, this `v` cannot be captured by the `or-to-let` expression `v`.

The whole point of the last observation is that the generated, binding identifiers of a macro transcription are secret; they are hidden from the user of the macro. So, a free instance of `v`, whether it appears free in *exp2* or whether it occurs free in the fully expanded version of *exp2*, must not be captured by the generated, binding `v`.

This condition is the *Hygiene Condition for Macro Expansion*:

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

It does not take into account the presence of key identifiers. After discussing the algorithm that establishes the Hygiene Condition, we present modified versions of both the condition and the algorithm. The changes will allow designated identifiers to be captured during expansion.

In the λ-calculus, if the hygiene condition is not satisfied, it can be established by an appropriate number of α-conversions. That is the basis for our expansion algorithm. Ideally, α-conversions should be applied with every transcription step. However, it cannot be known in advance which generated identifiers will wind up in binding positions, that is, among the formal parameters of some `lambda` expression, renaming at every step is impossible; an identifier intended to occur free could be renamed. For example, the Scheme macro call (`foo` *exp*) might transcribe to the expression (`bar a` *exp*), another macro call. Without completely expanding the call to `bar`, it is impossible to determine whether the generated `a` is intended to occur free or bound in the full expansion.

The Hygiene Condition leads to the requirement that, during expansion, we keep track of the origin of each identifier. We need to know whether the identifiers in a

replacement text were apparent in the macro call or whether they were generated by the particular transcription step. Furthermore, since generated identifiers from one transcription may appear in macro calls in the replacement text, we need to distinguish which transcription step generated each identifier. This is accomplished with a tracking mechanism.

Tracking is accomplished with a time-stamping scheme. The time-stamps are non-negative integers. Each apparent identifier in a top-level macro call is stamped with a zero. Subsequently, each time a transcription is made, the generated identifiers are stamped with a number one larger than that used immediately before. Actually, this is more than is needed. If we formulate the idea of an expansion tree similar to the parser tree of Section 4.7, then when transcribing calls by recursive descent, some calls will be independent of each other, lying on different branches of the expansion tree. The generated identifiers in these parallel calls may be stamped with the same numbers. Time-stamping is reminiscent of Leavenworth's block numbering scheme.

For example, in parsing and expanding

```
(if z (or u v) (or w y)),
```

two same-level macro calls are encountered in the consequent and alternate portions of the `if` expression. The identifiers in both calls are stamped with zeros, to obtain

```
(if z (or u:0 v:0) (or w:0 y:0)).
```

Next, transcribing each macro call separately by applying the appropriate STFs produces

```
(if z
    (let ([v u:0])
         (if v v v:0))
    (let ([v w:0])
         (if v v y:0))).
```

The last stamp used on each branch was zero, so we stamp each generated identifier with a one. This yields

```
(if z
    (let ([v:1 u:0])
         (if v:1 v:1 v:0))
    (let ([v:1 w:0])
         (if v:1 v:1 y:0))).
```

Technically, if we are working with the syntactic domains of identifiers *var* and natural numbers N, the domain of time-stamped identifiers *tsvar* is isomorphic to the product domain $tsvar = var \times N$.

What is yet to be done? We continue with the recursive descent tracking and transcribing until we encounter a binding form—a `lambda` expression. In the example, only one more step on each branch is needed to obtain

```
(if z
    ((lambda (v:1) (if v:1 v:1 v:0))
     u:0)
    ((lambda (v:1) (if v:1 v:1 y:0))
     w:0)).
```

When we find a binding form, we are ready to perform the α-conversions. All identical time-stamped identifiers within the `lambda` expression are renamed with the same new symbol. The new symbol is unique in the sense that it is not used inside the `lambda` expression prior to the renaming. The example is transformed into something like

```
(if z
   ((lambda (a) (if a a v:0))
    u:0)
   ((lambda (a) (if a a y:0))
    w:0)).
```

We continue in this fashion, recursively descending through the expression looking for either macro calls or `lambda` expressions. When a macro call is found, the

appropriate STFs is applied and the newly generated identifiers are time-stamped. When a `lambda` expression is found, all the stamped identifiers from the formals declaration are α-converted.

Once an expression is fully expanded in the sense that no more macro calls remain, the last step is to remove any time-stamps remaining on the free identifiers, no matter what stamp is on them. The example has only time-stamp zeros, but it is possible that other numbers can appear. The result is

```
(if z
    ((lambda (a) (if a a v))
     u)
    ((lambda (a) (if a a y))
     w)).
```

6.3. A Formal Description of the Hygienic Expansion Algorithm

Time-stamped syntax trees are defined like syntax trees but instead of identifiers they include elements from the union of identifiers and time-stamped identifiers. The formal definition of all of this is found in Figure 6.3.

Figure 6.4 contains functions which connect time-stamped domains with pure ones. The function $\mathcal{S}$ takes a time-stamp as an argument and returns a function which injects identifiers into *tsvar* with the given time-stamp. $\mathcal{S}_0$ is the function which stamps an identifier with a 0. It will play a role in the treatment of intended capturings. The function $[v/w]t$ acts like a substitution: given a time-stamped w, a v, and a time-stamped $\lambda term$, it substitutes all free occurrences of w by v. We omit the formal definition of the domain of time-stamped $\lambda term$'s; it is the subset of *tsstrees* which do not contain syntactic extensions.

Our hygienic expansion algorithm consists of four major phases (Figure 6.5). It starts out by transforming the user-supplied *stree* into a time-stamped syntax tree. This is accomplished by the function $\mathcal{T}$ which scans the *stree* and stamps all

Syntactic Domains:

$c \in const$	constant names,
$v \in var$, $w \in tsvar$	(time-stamped) identifier names,
$m \in mactok$,	macro tokens,
$k \in keyvar$,	key identifiers,
$e \in mstree$, $f \in tsmstree$	(time-stamped) macro expressions,
$s \in stree$, $t \in tsstree$	(time-stamped) expressions.

We also refer to

$x \in const \cup var \cup mactok \cup coretok$,
$y \in const \cup tsvar \cup mactok \cup coretok$,
$z \in coretok \cup tsstree$.

Syntax:

$$s ::= c \mid v \mid e \mid (\textbf{lambda}\ v\ s) \mid (s_1\ s_2),\ s_1 \notin mactok,$$
$$e ::= (m\, s_1 \ldots s_n) \text{ for } n \geq 0,$$
$$t ::= c \mid v \mid w \mid f \mid (\textbf{lambda}\ v\ t) \mid (t_1\ t_2),\ t_1 \notin mactok,$$
$$f ::= (m\, t_1 \ldots t_n) \text{ for } n \geq 0 \text{ with the above restriction.}$$

Semantic Domains:

$$K = \{k_1, \ldots, k_n\} \text{ for } n \geq 0,$$
$$STF = mstree \rightarrow stree,$$
$$\vartheta \in ST = mstree \rightarrow (K, STF),$$
$$\langle \lambda term \rangle.$$

Figure 6.3. *Hygienic macro expansion* (1).

identifier leaves with the function τ. For the initial pass, τ is the function $\mathcal{S}_0 = (\mathcal{S}0)$. Then the real expansion process begins and the stamp is increased to 1.

The function $\mathcal{E}$ recursively descends through *tsstree*-expressions that are also $\lambda term$-expressions. When it discovers a syntactic extension, it looks up the STFs in the syntax table and applies it to the macro call. This produces a transcription. The algorithm then time-stamps all the generated identifiers. Again, this time-

Auxiliary Functions:

$\mathcal{S}: N \to var \to tsvar$; $\mathcal{S}\, i\, v = v{:}i$.
$\mathcal{S}_0: var \to tsvar$; $\mathcal{S}_0\, v = v{:}0$.
$[\ /\ ]: var \times tsvar \to tsstree \to tsstree$
where the *tsstrees* are restricted to time-stamped λ*terms*;

$$[v/w_1]w_2 = (w_1 \stackrel{?}{=} w_2) \to v, w_2,$$
$$[v/w_1](\textbf{lambda}\ w_2\ t) = (w_1 \stackrel{?}{=} w_2) \to (\textbf{lambda}\ w_2\ t), (\textbf{lambda}\ w_2\ [v/w_1]t),$$
$$[v/w](t_1\ t_2) = ([v/w]t_1\ [v/w]t_2).$$

Figure 6.4. *Hygienic macro expansion* (2).

stamping process is performed by the function $\mathcal{T}$ in co-operation with the function $(\mathcal{S}\, j)$, where j is the current stamp. The stamp is incremented, and the expansion continues.

The result of the function $\mathcal{E}$ is a time-stamped λ*term*. It differs from the result of the naïve expansion algorithm. Wherever the naïve result contains an identifier, the hygienic result contains a corresponding time-stamped identifier. For example, if the naïve algorithm returns the tree

$$(\textbf{lambda}\ x\ (\textbf{lambda}\ x\ ((f\ x)\ x))),$$

the hygienic result might look like

$$(\textbf{lambda}\ x{:}0\ (\textbf{lambda}\ x{:}1\ ((f{:}1\ x{:}0)\ x{:}1))).$$

This indicates that according to the Hygiene Condition, the naïve algorithm would be getting the bindings wrong.

The third phase of the hygienic algorithm replaces all bound, time-stamped identifiers with non-conflicting, unstamped identifiers. It is important to be able

Semantic Functions:

$\mathcal{E}_{hyg}$: $stree \to ST \to \lambda term$,
$\mathcal{T}$: $tsstree \to (var \to tsvar) \to tsstree$,
$\mathcal{E}$: $tsstree \to ST \to N \to \lambda term$,
$\mathcal{A}$: $tsstree \to tsstree$,
with the domain restricted to time-stamped $\lambda terms$,
$\mathcal{U}$: $tsstree \to stree$;

$$\mathcal{E}_{hyg}[\![s]\!] = \lambda\vartheta.\, \mathcal{U}[\![\mathcal{A}[\![\mathcal{E}[\![\mathcal{T}[\![s]\!]\mathcal{S}_0]\!]\vartheta j_0]\!]\!] \text{ where } j_0 = 1;$$

$$\begin{aligned}
\mathcal{T}[\![y]\!] &= \lambda\tau.\, y,\\
\mathcal{T}[\![v]\!] &= \lambda\tau.\, \tau v,\\
\mathcal{T}[\![(z_1 \ldots z_n)]\!] &= \lambda\tau.\, (\mathcal{T}[\![z_1]\!]\tau \ldots \mathcal{T}[\![z_n]\!]\tau);
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\![c]\!] &= \lambda\vartheta j.\, c,\\
\mathcal{E}[\![w]\!] &= \lambda\vartheta j.\, w,\\
\mathcal{E}[\![f]\!] &= \lambda\vartheta j.\, \mathcal{E}[\![\mathcal{T}[\![(\vartheta f \downarrow 2)f]\!](\mathcal{S}\, j)]\!]\vartheta(j+1),\\
\mathcal{E}[\![(\textbf{lambda}\ w\ t)]\!] &= \lambda\vartheta j.\, (\textbf{lambda}\ w\ (\mathcal{E}[\![t]\!]\vartheta j)),\\
\mathcal{E}[\![(t_1\ t_2)]\!] &= \lambda\vartheta j.\, (\mathcal{E}[\![t_1]\!] j\vartheta\ \mathcal{E}[\![t_2]\!]\vartheta j);
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}[\![v]\!] &= v,\\
\mathcal{A}[\![y]\!] &= y,\\
\mathcal{A}[\![(\textbf{lambda}\ w\ t)]\!] &= (\textbf{lambda}\ v\ \mathcal{A}[\![[v/w]t]\!])\\
&\qquad \text{where } v \text{ is a fresh variable,}\\
\mathcal{A}[\![(t_1\ t_2)]\!] &= (\mathcal{A}[\![t_1]\!]\ \mathcal{A}[\![t_2]\!]);
\end{aligned}$$

$$\begin{aligned}
\mathcal{U}[\![x]\!] &= x,\\
\mathcal{U}[\![v{:}i]\!] &= v,\\
\mathcal{U}[\![(z_1 \ldots z_n)]\!] &= (\mathcal{U}[\![z_1]\!] \ldots \mathcal{U}[\![z_n]\!]).
\end{aligned}$$

Figure 6.5. *Hygienic macro expansion* (3).

to tell when a stamped identifier is generated as different time-stamps indicate

```
(extend-syntax (or) ()
   [(or) false]
   [(or exp) exp]
   [(or exp1 exp2 ...)
    (let ([v exp1])
       (if v v (or exp2 ...)))])
```

Figure 6.6. *An* `extend-syntax` *declaration of the Scheme standard macro* `or`.

different transcription steps. Since the expression is now a time-stamped *λterm*, α-conversions easily achieve the desired effect. The function $\mathcal{A}$ recursively descends through the term and applies the appropriate substitution function to any abstractions. The example becomes something similar to

(lambda a **(lambda** b **((**f:1 a**)** b**)))**.

The **lambda** bindings are now correct. The only remaining time-stamped identifiers correspond to the free identifiers. Their significations are given by their identifier components; they must be unstamped. This is the task of the function $\mathcal{U}$. It recursively descends through the tree and removes all time-stamps. The result of this fourth and last phase is a pure *λterm*:

(lambda a **(lambda** b **((**f a**)** b**)))**.

A trace of the expansion of `(or (zero? v) v)` using the hygienic expansion algorithm $\mathcal{E}_{hyg}$ and the declaration of `or` of Figure 6.6 is found in Figure 6.7. The result is exactly what it should be; there is no inadvertent capturing of the apparent `v`'s in the macro call.[18]

[18] The double arrow indicates a transcription step as well as the kind of stamping being performed. The fresh identifier created in the α-converting step is represented by $*$. Even though we reproduce the entire expression in each step, the functions designated above the arrows operate on only parts of the whole.

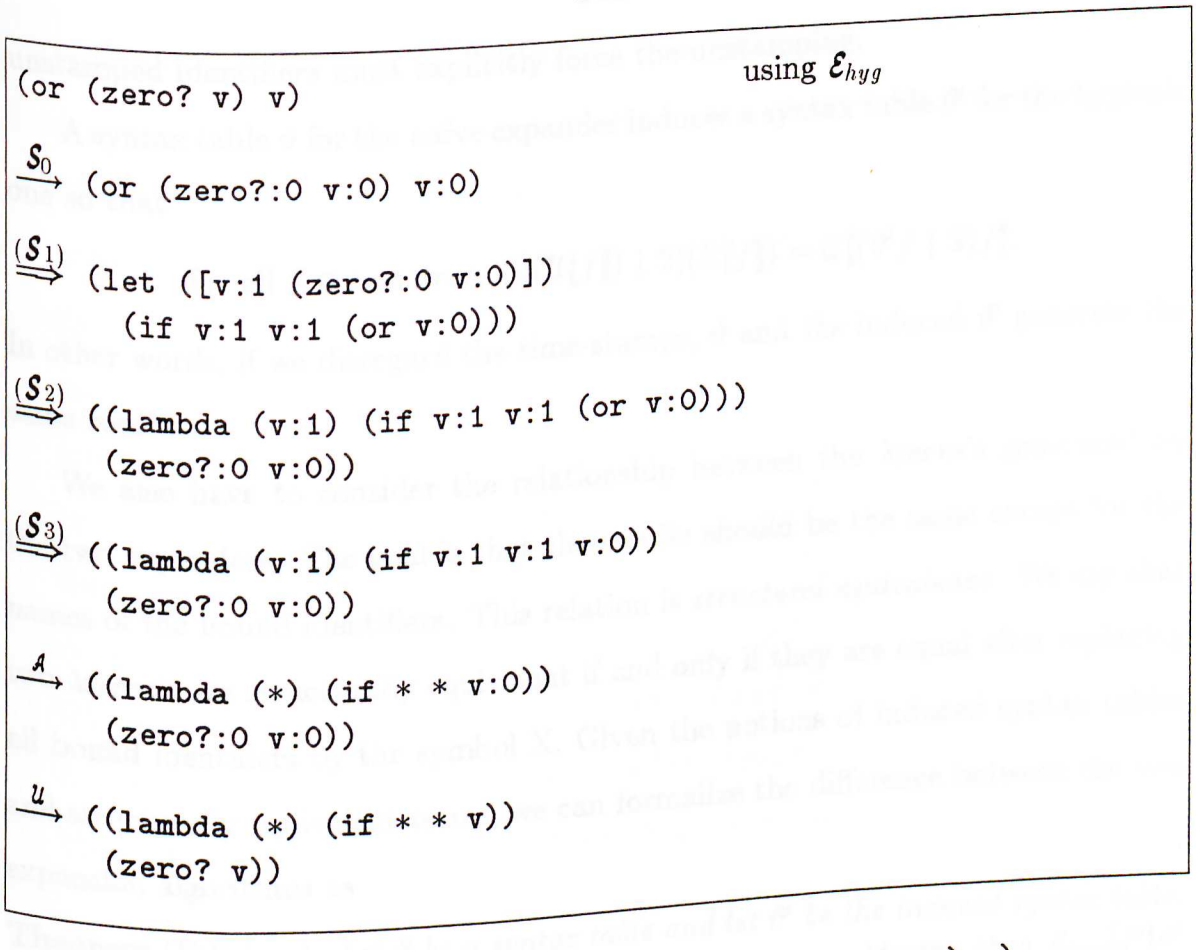


Figure 6.7. *Tracing the expansion of* `(or (zero? v) v)`.

Before we compare the results of the naïve and hygienic expansion algorithms, we discuss the implications that hygienic expansion has upon the STFs. Because the source and target languages for STFs are different for the two algorithms, we might expect the STFs themselves to be different. However, the change of languages is really a minor one. For, if we consider time-stamped identifiers as a special kind of identifier, the STFs are no different, except that functions which need to know or compare identifier names must unstamp the appropriate symbol with the function U (or an appropriate restriction thereof). This is the reason for the presence of the call to `unstamp` in the fender processing *prod-rule-function* of Section 5.4. It also explains `unstamp-no-copy` in the function `main-body`. An STF that works with

unstamped identifiers must explicitly force the unstamping.

A syntax table ϑ for the naïve expander induces a syntax table ϑ' for the hygienic one so that

$$\text{for all } f \in tsmstree,\ (\vartheta(\mathcal{U}[\![f]\!]) \downarrow 2)(\mathcal{U}[\![f]\!]) = \mathcal{U}[\![(\vartheta' f \downarrow 2)f]\!].$$

In other words, if we disregard the time-stamps, ϑ and the induced ϑ' generate the same results.

We also have to consider the relationship between the $\lambda term$'s generated by the two expanders. The goal is that the results should be the same except for the names of the bound identifiers. This relation is *structural equivalence*. We say that two $\lambda term$'s are structurally equivalent if and only if they are equal after replacing all bound identifiers by the symbol X. Given the notions of induced syntax tables and structurally equivalent terms, we can formalize the difference between the two expansion algorithms as

Theorem (Felleisen). *Let ϑ be a syntax table and let ϑ' be the induced syntax table. Then, for all stree's P, if $\mathcal{E}_{naive}[\![P]\!]\vartheta$ expands into a $\lambda term$, then $\mathcal{E}_{hyg}[\![P]\!]\vartheta'$ expands into a structurally equivalent term which satisfies the Hygiene Condition for macro expansion.*

PROOF. The proof is structured according to the four phases of the function $\mathcal{E}_{hyg}$.

Step 1. The result of $\mathcal{T}[\![P]\!]\mathcal{S}_0$ is structurally equivalent to P; all identifiers have the 0 time-stamp. This claim can be verified by an induction on the structure of P.

Step 2. Call the output of *Step 1*, P_0. Then we can prove two statements about the relationship of $\mathcal{E}_{naive}[\![P]\!]\vartheta$ to $\mathcal{E}[\![P_0]\!]\vartheta'$:

1) The two results are equal except for the time-stamps, that is,

$$\mathcal{E}_{naive}[\![P]\!]\vartheta = \mathcal{U}[\![\mathcal{E}[\![P_0]\!]\vartheta']\!].$$

This proposition depends on the fact that ϑ' is induced by ϑ. It implies that the two results are structurally equivalent.

2) Moreover, all variables of a transcription step receive the same time-stamp which is unique with respect to the path from the root of the term to the occurrence of the respective syntactic extension. This follows from the fact that all transcriptions previous to the current one were time-stamped with a number less than j, the current value. This statement is true for all syntactic extensions occurring in P_0. It is re-established by the time-stamping that immediately follows a transcription step. The identifiers in $(\vartheta' f \downarrow 2)f$ are either pure identifiers or time-stamped ones that already occurred in f. The pure ones are stamped with j and are thus distinguishable from all the previously generated identifiers. Afterwards the stamp value is incremented, and all following expansions receive time-stamps at a higher level. As for applications, syntactic extensions in the function and argument part cannot overlap. Hence, it is justified to continue the expansion process on both paths with the same time-stamp.

Step 3. From the previous step, the result of $\mathcal{E}$ is structurally equivalent to the result of $\mathcal{E}_{naive}$ and that all identifiers of the hygienic result have a unique time-stamp reflecting their origin. Hence, α-converting all λ-expressions such that each time-stamped parameter is replaced by a fresh identifier results in an expression satisfying the Hygiene Condition. That expression is also structurally equivalent to the input of $\mathcal{A}$. This can easily be verified by showing that $[v/w]z$ is a substitution function and that $\mathcal{A}$ otherwise preserves the tree structure.

Step 4. The input to the last step is a term which satisfies the Hygiene Condition and is structurally equivalent to the naïvely expanded program except for time-stamps of free identifiers. It is a routine matter to prove by induction that the function $\mathcal{U}$ removes these time-stamps and leaves all other properties intact. □

Before concluding this section we want to comment on isolated and mixed expansion methods (Section 4.7.3). The hygienic expansion algorithm is described

for isolated expansion. Once a macro call in encountered in the parser's recursive descent, that call is fully expanded before the result is given back to the parser. However this need not be the case. If the parser is designed so that it recognizes time-stamped symbols as identifiers, then it can parse transcribed text as it is generated with the precaution that the time-stamping is not reset.

6.4. The Modified Hygienic Expansion Algorithm

The hygienic algorithm of the last section does not permit the capturing of some identifiers. The `inf-loop` macro of Section 1.3.2 must cause the capture of the identifier `exit-with-value` when it is present in a macro call. The naïve expansion algorithm has no trouble with this; every apparent identifier is eligible for capture by a generated binding. However, the hygienic algorithm, as expressed, never allows for the possibility of capturing. In this one aspect, the two algorithms have exactly the opposite behavior. The theorem asserts that in all other aspects the algorithms produce the same results.

So, we must reintroduce a way of obtaining the capturing behavior in those circumstances when it is desired, but we must be careful about the type of capturing we want. The capture of a generated, free identifier by some binding instance outside of the macro call that generates it is not hindered by the hygienic algorithm. This is the "simulating dynamic variables" technique (Section 3.3.10). It still works because the hygienic algorithm never renames the free identifiers in a macro call's fully expanded replacement text.

It is clear from `inf-loop` that we wish to allow the capturing of apparent identifiers, those that in the hygienic algorithm are time-stamped with a zero. Taking advantage of each macro's key identifier list, $(\vartheta f \downarrow 1)$, we can insure that the key identifiers get stamped with zeros. They will then seem as if they are apparent identifiers.

The Hygiene Condition is modified to reflect this change. It becomes the *Modified Hygiene Condition for Macro Expansion*:

> *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.*

To realize this, a new top-level expander $\mathcal{E}_{modhyg}$ is defined and the definition of the function $\mathcal{T}$ is changed:

$$\mathcal{E}_{modhyg}: stree \to ST \to \lambda term;$$
$$\mathcal{T}: tsstree \to (var \to tsvar) \to K \to tsstree;$$

$$\mathcal{E}_{modhyg}[\![s]\!] = \lambda\vartheta.\mathcal{U}[\![\mathcal{A}[\![\mathcal{E}[\![\mathcal{T}[\![s]\!]\mathcal{S}_0\emptyset]\!]\vartheta j_0]\!]\!] \text{ where } j_0 = 1;$$

$$\begin{aligned}\mathcal{T}[\![y]\!] &= \lambda\tau\kappa.y,\\ \mathcal{T}[\![v]\!] &= \lambda\tau\kappa.\tau\kappa v,\\ \mathcal{T}[\![(z_1 \ldots z_n)]\!] &= \lambda\tau\kappa.(\mathcal{T}[\![z_1]\!]\tau\kappa \ldots \mathcal{T}[\![z_n]\!]\tau\kappa);\end{aligned}$$

$$\mathcal{E}[\![f]\!] = \lambda\vartheta j.\mathcal{E}[\![\mathcal{T}[\![(\vartheta f \downarrow 2)f]\!](\mathcal{S}\, j)((\vartheta f) \downarrow 1)]\!]\vartheta(j+1);$$

and the definitions of two of the auxiliary functions:

$$\mathcal{S}: N \to var \to K \to tsvar;$$
$$\mathcal{S}_0: var \to K \to tsvar;$$
$$\mathcal{S}\, i\, v\, \kappa = v \in \kappa \to v{:}0,\ v{:}i;$$
$$\mathcal{S}_0\, v\, \kappa = v{:}0.$$

Capturing can be made by a binding instance generated at any level within the expansion. For example, tracing the modified hygienic expansion of `(one a)` with the pyramided macro declarations

```
(extend-syntax (one) ()
   [(one exp) (two exp)])

(extend-syntax (two) (a)
   [(two exp) (lambda (a) exp)])
```

verifies that the apparent instance of a is indeed caught. By looking at the declarations, we infer that a macro call of the form (one *exp*) produces an abstraction of one argument with *exp* as its body. Furthermore, any free occurrences of a within *exp* become bound:

```
(one a)                                          using ℰ_modhyg
  ──S0──▶ (one a:0)
  ═(S1)═▶ (two a:0)
  ═(S2)═▶ (lambda (a:0) a:0)
  ──A──▶ (lambda (*) *)
  ──u──▶ (lambda (*) *).
```

We get the predicted result. The call (one a) should produce the identity function.

The situation is not so simple if a generated symbol from one transcription is a key-identifier in a pyramided declaration. Consider the pair of macros (start *exp*) and (pass-it-on *exp1 exp2*):

```
(extend-syntax (start) ()
   [(start exp) (pass-it-on exp a)])

(extend-syntax (pass-it-on) (a)
   [(pass-it-on exp1 exp2) (lambda (a) (exp1 exp2))]).
```

Here, pass-it-on has the key identifier a. This means that any free occurrence of a within whatever expressions exp1 and exp2 represent is to be ultimately captured by the binding instance of a in the lambda expression.

The pyramided combination of start and pass-it-on produces an abstraction of one argument. The abstraction always applies the expression *exp* from the call (start *exp*) to an identifier (the same one as the formal parameter) with the proviso that free occurrences of that identifier within *exp* become bound.

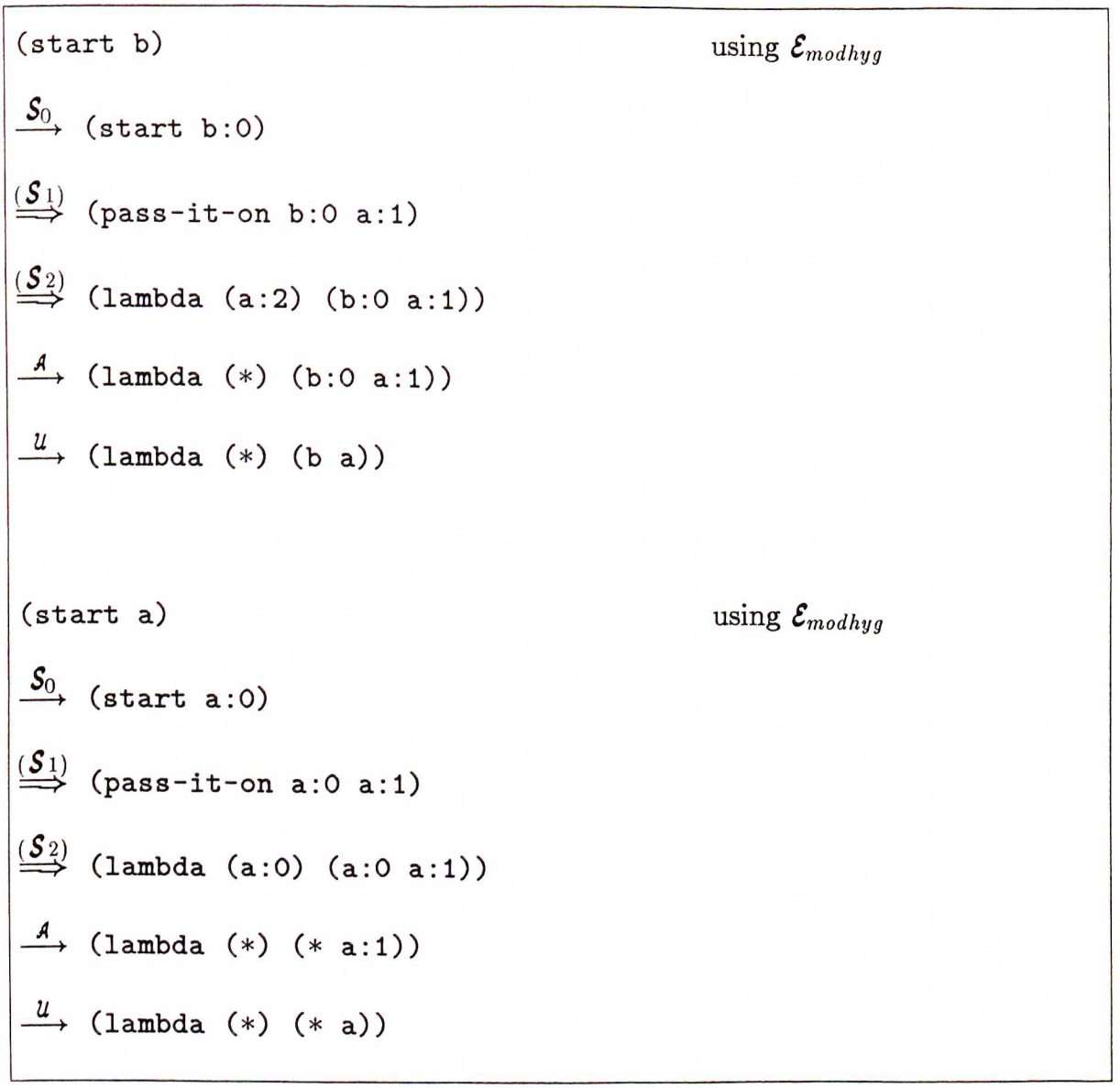


Figure 6.8. *Tracing the expansion of* `(start b)` *and* `(start a)`.

A trace through the modified hygienic expansion algorithm demonstrates the expansion behavior for two expressions (Figure 6.8).

Neither result conforms to our expectations. The flaw lies in not declaring `a` as a key identifier for `start`. If we amend the declaration of `start` to be

```
(extend-syntax (start) (a)
   [(start exp) (pass-it-on exp a)])
```

then the modified hygienic expansion of `(start b)` is

```
(lambda (*) (b *))
```

and that of `(start a)` is

```
(lambda (*) (* *)).
```

The effect of having a be a key identifier for `start` and not `pass-it-on` is seen in

`(start b)` $\overset{\mathcal{E}_{modhyg}}{\Longrightarrow}$ `(lambda (*) (b a))` and

`(start a)` $\overset{\mathcal{E}_{modhyg}}{\Longrightarrow}$ `(lambda (*) (a a))`.

These are not correct, either. The rule is, for one macro to generate a key identifier situated within another macro call, the outer macro must recognize the identifier as a key identifier. If we write a macro `gil` that generates an `inf-loop` expression containing, as is most likely, the identifier `exit-with-value`, that identifier must be a key identifier for `gil`. After all, since `inf-loop` creates a binding, we should think of any macro that generates a call to `inf-loop` as causing that same binding.

There is a question as to whether other binding schemes might be useful. For example, what if macros were allowed to capture generated identifiers as well as user-supplied apparent ones? That is, what if the original versions of `start` and `pass-it-on` cause apparent `a`'s not to be caught, and instead catch the one generated by `start`? Then `(start a)` would expand to

```
(lambda (*) (a *)).
```

To do this, the expander would have to recognize that `pass-it-on` always occurs at a stamp-level one greater than the level of any call to `start`.

In a macro system in which the syntax table is unscoped, as is ours with every macro global to every other one, this position yields unpredictable results. However,

if the syntax table is set up so that help-macros remain local to a specific outer macro call, this position has some merit. The relationship between stamping levels of our macro calls could always be statically deduced. In the global macro world, this cannot be done.

6.5. Enlarging the Host Language

The toy language of *strees* and the modified expansion algorithm both need to be extended to handle the full Scheme core (Figure 3.2). The language lacks assignment expressions, conditional expressions, applications and abstractions of more than one argument, and quoted structures.

We made a slight change in the Scheme code as used in the last section; the `lambda` expression formal parameter appeared in a list. We continue with this change, not bothering to rewrite the lines in the definitions of $\mathcal{E}$ and $\mathcal{A}$ that deal with **lambda** expressions unless we need to for another reason.

First of all, it is a simple matter to adapt our algorithm so that **lambda** expressions can have more than one formal parameter. We redefine the application-line of $\mathcal{E}$ to become

$$\mathcal{E}[\![(t_1 \ldots t_n)]\!] = \lambda\vartheta j.(\mathcal{E}[\![t_1]\!]j\vartheta \ldots \mathcal{E}[\![t_n]\!]\vartheta j)$$

for all $n \geq 1$. The line of $\mathcal{A}$ that makes the substitution becomes

$$\mathcal{A}[\![(\textbf{lambda } (w_1 \ldots w_n)\ t)]\!] = (\textbf{lambda } (v_1 \ldots v_n)\ \mathcal{A}[\![[v_1/w_1]\ldots[v_n/w_n]t]\!])$$

where $n \geq 0$ and $\{v_1, \ldots v_n\}$ are fresh identifiers. The substitution function [/] must also be changed in a way analogous to $\mathcal{A}$.

To add more core forms, the set *coretok* is extended to include whatever symbols chosen to designate them. To achieve the full core, the set becomes

$$\textit{coretok} = \{\textbf{lambda, set!, if, quote}\}.$$

Assignments and conditionals cause no problems at all because they are not binding constructs requiring special treatment by $\mathcal{E}$. Hence, they are treated like applications with a slightly more elaborate syntactic structure, the keyword.

Quoted atoms or lists need special treatment. The expander can only recognize syntactic forms not occurring within other calls (Section 4.7.2). Special forms which seem to occur inside macro calls may get rearranged during the expansion process. What appears to be a quoted structure because of the presence of the symbol "**quote**" may not be an actual quoted structure. Thus its components must be time-stamped. However, when $\mathcal{E}$, $\mathcal{A}$, and [/] encounter an expression of the type **(quote** β**)**, they must inhibit their activity. Expressions of that type are constant expressions of the host language. The respective additional lines in these three functions are

$$\mathcal{A}[\![\textbf{(quote } \beta\textbf{)}]\!] = \textbf{(quote } \beta\textbf{)},$$

$$\mathcal{E}[\![\textbf{(quote } \beta\textbf{)}]\!] = \textbf{(quote } \beta\textbf{)}, \text{ and}$$

$$[v/w]\textbf{(quote } \beta\textbf{)} = \textbf{(quote } \beta\textbf{)}.$$

The time-stamps in β are ultimately removed when the unstamp function $\mathcal{U}$ is applied to the entire program.

An example, using the Scheme macros `pseudo-case` and `fake`:

```
(extend-syntax (pseudo-case) ()
   [(pseudo-case exp1 (tag exp2))
    ((lambda (a)
       (if (eq? a (quote tag))
           exp2
           error))
     exp1)])

(extend-syntax (fake) ()
   [(fake x) (quote x)])
```

illustrates this. A trace of the expansion of

```
(lambda (a) (pseudo-case (fake a) (quote a)))
```

```
(pseudo-case (fake a) (quote a))                    using ℰ_modhyg

 S0
--> (pseudo-case (fake a:0) (quote a:0))

 (S1)
==>
((lambda (a:1)
    (if (eq?:1 a:1 (quote quote))
        (quote a:0)
        error:1))
 (fake a:0))

 (S2)
==>
((lambda (a:1)
    (if (eq?:1 a:1 (quote quote))
        (quote a:0)
        error:1))
 (quote a:0))

 A
-->
((lambda (*)
    (if (eq?:1 * (quote quote))
        (quote a:0)
        error:1))
 (quote a:0))

 U
-->
((lambda (*)
    (if (eq? * (quote quote))
        (quote a)
        error))
 (quote a))
```

Figure 6.9. *A trace of the extended, modified hygienic algorithm involving* quote.

using these definitions is in Figure 6.9.

6.6. Scheme Implementation Details

Creating a running Scheme version of $\mathcal{E}_{modhyg}$ is an easy task. This is principally because Scheme readily lends itself to expressing algorithms described in the style of denotational semantics. Our 1986 Lisp Conference paper [39] contains an implementation.

The only detail not described in the algorithm is how to actually create the time-stamped identifiers. Time-stamped identifiers have two essential properties.

First, they must be distinguishable from every other term in the program. Each must be unique in the sense that it is distinguishable from every other symbol used in the replacement text, either as a temporary in some of the transcription steps or as a part of the full expansion. Of course, there may be multiple instances of the same time-stamped identifier. Also, it must be possible to tell whether a given symbol is time-stamped.

Second, they must contain a component which indicates the original name. Hence, we can use gensym'd atoms with a property `original-name`. If these new symbols turn out to be bound identifiers, the gensym's can remain in place. If they are free, they can be replaced by the original name. The function $\mathcal{A}$ is then implemented as the identity function.

The function $\mathcal{U}$ must be made available to the macro writer. We pointed out before that we need it in two forms, `unstamp` and `unstamp-no-copy`, to implement `extend-syntax`. In general, it is `unstamp` that we need. However, for the processing of `with` specifications that create code with embedded Scheme structures, we need the non-copying version.

All of this has ramifications for the STFs. A transform function must satisfy two new conditions:

1. In a situation where the name of an identifier is needed by an STF, the STF

must map the time-stamped identifier to its identifier name with the function `unstamp`. An example of this occurs in the processing of fenders in an `extend-syntax` declaration (Section 5.1.1).

2. Any identifier in the output of an STF must either be unstamped or time-stamp-equal to the apparent identifiers in the macro call. An STF cannot be permitted to create its own time-stamped identifiers. It must be done under the sole control of the expansion algorithm.

There is one last item to discuss: keywords that are not macro names. We were careful in the definition of $\mathcal{T}$ to insure that symbols which are not identifiers are not time-stamped. In doing so, we are conforming to Principle 7. However, no provision has been made for the auxiliary keywords a macro might possess. In terms of an implementation, all that is required is that `add-to-syntax-table` (Section 5.3) must maintain a system-wide list of keywords. The function used to implement $\mathcal{T}$ checks this list in determining whether a given symbol is a keyword.

7. Conclusion

Our goal of presenting a rationalized syntactic extension mechanism has three components: first, the proposal of a syntactic extension mechanism for a high level language; second, the discussion of the design and implementation of a declaration mechanism for Scheme and other Lisp-like languages; and third, the presentation of a new macro expansion algorithm.

While the first aim serves as a unifying "philosophy of macro system design," the products of the second and third, `extend-syntax` and the modified hygienic macro expansion algorithm, are independent of one another. Those wishing to incorporate one or the other into a macro processor may do so. Yet their combined effect has greatly influenced Scheme programming at Indiana University.

In this chapter we examine separately the implications of the declaration and expansion components of our macro system. Then we give some suggestions for possible related work that still needs to be done on macro processors, in particular on those in Lisp systems. Some final remarks on the influence of our entire macro processing system conclude the chapter.

7.1. The Ramifications of `extend-syntax`

What began as a project on a generator for creating Scheme macros took on a life of its own as more and more people began wanting to use `mkmac`, as it was called in those days, to write their macros. Freed from having to compose STFs by hand,

users did not shy away from writing macros. Large projects concerned with the embedding of other languages within Scheme were made possible.

In a sense, `extend-syntax` makes the writing of macros too easy. Those who do not understand, or chose not to think about, the fact that they are changing their base language every time they declare a new macro, soon run into the trouble that they feel restricted in their choices for identifier names because of the large number of reserved words. Furthermore, those who neglect the fact that macros are not first class programming objects and should not be used in place of first class objects, as in aliasing a macro to a function invocation, curtail the expressive power they would have if they stuck to first-class objects.

Even though `extend-syntax` makes macro writing easier, it gives the macro writer no additional programming power. Anything that can be done by writing an STF can be done with `extend-syntax` and *vice versa*. The perceived power comes from two sources. First, `extend-syntax` declarations permit macro writers to concentrate on the intended semantics of their new special forms. Second, the idea of code with embedded Scheme structures, while it was always possible before, was never emphasized in Lisp programming. It is a little tricky the first time to see how to build an STF that incorporates a shared data structure or a function. The uses for such code have barely been explored.

We perceive five problems with the customary Lisp macro tools for the declaration of macros; the special form `extend-syntax` serves as a remedy for each of them. Perhaps the advantage most people immediately see is the ease of writing and reading macro declarations made using it. Of equal importance is the implicit syntax check provided by the pattern matching that so many macro writers fail to include in their macros. Value lies in reinforcing the ideas that macros are organized in a syntax table and that they are best used only as syntactic extensions.

The existence of `extend-syntax`, particularly in its early forms, led to thinking about what syntactic constructions are preferable in special forms. For example, a previous version did not allow for a pattern ellipsis-list to be terminated by a list containing a keyword as its left-most atom. A keyword could be used only at the same list-level as the ellipsis. Bowing to pressure to make the declaration of the macro `case` easier, we introduced the left-most atom facility. However, before doing that we believed that `case` itself was poorly designed, that `else` clauses should appear as

(`case` *tag-exp* [*tag exp1 exp2* ...] ... `else` *expa expb* ...).

Even though we have enhanced `extend-syntax` to handle the standard `case`, we still feel this alternate form has a better syntax.

We believe in our tool, letting its powers shape what we write. One area in which `extend-syntax` is lacking is identified below. Other than that, we hold that if a macro writer is having trouble expressing his patterns and transcriptions in the language we have provided, he should give more thought to the syntax he is designing.

The form `extend-syntax` is essentially a rewrite-rule system. The user specifies what each rule should do, and `extend-syntax` implements the rule. It can be viewed as a code-generator generator. The paradigm of describing a set of production rules and having the system implement the corresponding transformations is valuable whenever one is studying rewrite rules. The suitability of the ellipsis notation for expressing transformations of arbitrarily long sequences of terms is evident.

7.2. The Ramifications of Hygienic Macro Expansion

While the only controversies `extend-syntax` ever generated was over whether macro writing should be so easy for the general programmer and over what fea-

tures it should possess, the hygienic expansion algorithm elicited several arguments against its use. In this section some of those arguments and our response to each of them are given.

The strongest argument in favor of hygienic expansion is that it is correct in the sense of upholding Principle 12. It is not enough to claim that each macro expands correctly; we must also assert that the semantics of a program is not changed by expanding macro calls within it. Violations of this principle occur under the naïve expansion algorithm because it does not take into account the context of macro calls.

7.2.1. THE ARGUMENT "THE DEFAULT IS NOT BACKWARDS." Among people who have been aware of the capturing problem and have programmed around it, some feel that to switch the default from "everything is captured" to "nothing is captured" is a mistake. In essence, that is all the hygienic expander does. It switches the default, providing a means of effecting capture just as capture can be guarded against in naïve expansion with carefully declared macros.

It has been our experience, both in writing macros for the special forms in Scheme and in our own programming, that we want to avoid the capture of apparent identifiers by generated binding instances more often than we want such capturing. Not particularly liking the fresh identifier approach to the problem, we have struggled with getting the freezing and thawing right so that identifiers are bound as we wish. The macro writer should not have to bother with this detail. He should be freed from it just as `extend-syntax` frees him from having to compose an STF.

Thus, we worked on an expansion algorithm that assumes the burden of avoiding the unwanted captures. It has always been a tenet of ours that the language designer has a greater responsibility than the language user. In the role of language designer

and implementor, we should take the responsibility for making programming easier.

7.2.2. THE ARGUMENT "THE ALGORITHM IS TOO COMPLICATED." The attraction of the naïve expansion is its great simplicity. It can be easily implemented. The macro writer knows that the identifiers he includes in any transcription specification will be the identifiers that wind up in the replacement text. Using the hygienic expansion algorithm, he knows that the names of generated bound identifiers will be different. This is disconcerting to some people. They understand the naïve algorithm operationally, not having the same grasp on the hygienic algorithm.

Understanding how each component in a language processor works should not be important to a user of the language. Full comprehension of how a compiler allocates registers does not make better programmers. We may be able to use our knowledge in some carefully encoded algorithms, but then we are writing programs in the language as it is implemented on one machine.

Purely operational understandings reduce some of the power of abstraction gained from the language. In a sense, those who hold them do not understand the semantics of the language they are using, just its implementation. If they can give up the operational understanding of macro expansion and realize that their generated, non-key identifiers which effect bindings will be renamed, they can still regard macro expansion as the replacement of a macro call by its transcription.

In addition, it has been our experience that the effort to adapt to the hygienic algorithm is minimal. There are many incorrect programs in which the errors occurred because the macro writer forgot to prevent the capturing problem. This kind of error is very hard to find. The reason why a program produces incorrect results when this mistake is made cannot be discovered without examing the fully expanded text of the program. The hygienic expansion algorithm alleviates this situation.

With the hygienic algorithm, the corresponding error is neglecting to declare an identifier as a key identifier. The result is often easily detected as an unbound identifier error informing the user that an identifier he expected to be bound in the fully expanded program is not.

7.2.3. THE ARGUMENT "THE ALGORITHM IS TOO EXPENSIVE." The (modified) hygienic expansion algorithm appears to require four phases, each phase consisting of at least one recursive descent through the entire program. The initial time-stamping with $\mathcal{S}_0$ is one pass. The transcribing and time-stamping phase $\mathcal{E}$ makes up the second pass, but there is no bound on how many transcriptions, each with its own time-stamping, may occur. This number is controlled by the pyramided and recursive nature of the macros themselves. The α-converting phase $\mathcal{A}$ makes another complete recursive descent, as does the final unstamping phase $\mathcal{U}$.

The $\mathcal{A}$ phase was dealt with in Section 6.6. As pointed out there, the $\mathcal{A}$ phase can be omitted with the choice of representing time-stamped identifiers with gensym's. Furthermore, the $\mathcal{U}$ phase can be incorporated into $\mathcal{E}$; each time an identifier is encountered we choose whether to unstamp it or not. We maintain a lexical environment structure as we recursively descend through the expression. Then, the identifiers to be left as is (the bound ones) can be distinguished from those to be unstamped (the unbound ones).

Thus the situation is not as bad as it appears from the formal description. The difficulty is that no matter what expansion algorithm is used, the macro writer always plays a large role in determining how long expansions will take. The more he uses pyramided macro declarations, the longer they will take to expand.

The hygienic algorithm is more expensive than the naïve one, but the cost is reasonable and justified. Hygienic expansion is based partly on the philosophy that the system should be responsible for providing useful defaults. Our experience has

shown that macro writers who rely on hygienic expansion can encode their macros faster and with greater accuracy. Too many people get caught by the capturing problem. If any beneficial component can be designed to work with a reasonable cost, it should be built into the system.

7.2.4. THE ARGUMENT "THE RENAMING OF IDENTIFIERS IS BAD." Some people have been concerned that the renaming of identifiers ruins the debugging tools, first-class environments, and other language features that depend on identifier names remaining consistent. It is true that the hygienic expansion algorithm changes the names of all bound identifiers in the replacement text of a macro call. The argument against hygienic expansion is that arbitrary identifier names appear in the code that is handed to the actual compiler. Therefore the user will have little chance of finding the cause of an error reported in terms of these unknown identifiers.

However, problems of this sort occur whenever there is a macro processor that supplies non-user written programs to the compiler. Macros can generate new symbols of which the person using the macro has no notion. An error about an unexpected occurrence of the identifier x has no more meaning than one about some gensym-named identifier if the user cannot tell from where it came. As McIlroy observed, created symbols are of no concern to the programmer. Moreover, the $\mathcal{A}$ phase of the hygienic expansion algorithm can be designed not to rename most instances of identifiers. The ones that can be preserved are precisely those involved in situations when the naïve algorithm does not cause any undesirable captures.

When macros are documented for use by those other than the macro writer, they are frequently specified operationally by telling what transcription text goes with each macro call. As part of the documentation, the macro writer must list the keywords, the key identifiers, and any other calling protocols. He also has to insure

that the user knows that any generated, binding identifiers will have system-created names.

The problem of knowing the identifiers from the original source code plagues all compilers. Whatever means are used to preserve the names across code generation should be extended to preserve them across macro expansion as well.

7.3. Possibilities for Further Research

There are several areas in which our macro system is deficient. We examine some of them in this section and make a few suggestions for what needs to be done.

7.3.1. SEMANTICS OF MULTIPLE ELLIPSES. There are still some unresolved questions concerning the semantics of multiple uses of the ellipsis within macro declarations. A pair of plausible expansions of the macro call

```
(fog (1 2 3) (4 5 6) (7 8 9))
```

was given at the end of Section 5.2. In the situation described there, the nesting of ellipses in the declaration

```
(extend-syntax (fog) ()
   [(fog (a ...) ...)
    (bar a ...)])
```

could mean either `(bar 1 2 3)` or `(bar 1 4 7)`. The formal semantics does not specify which. It reports the situation as an error.

A related question arises concerning the intent of the ellipses in the declaration

```
(extend-syntax (fop) ()
   [(fop (a ...) ((b ...) ...))
    (bop ((a b) ...) ...)]).
```

Should the call `(fop (1 2) ((w x) (y z)))` expand to either

```
(bop ((1 w) (1 x)) ((2 y) (2 z)))
```

or

```
(bop ((1 w) (2 x)) ((1 y) (2 z)))?
```

The formal semantics of Section 5.2 specifies the former. The implementation of `extend-syntax` given in the Section 5.4 and Appendix B does not handle this situation.

The problems of interpretation occur when ellipses are nested to varying depths in the pattern and transcription specifications. They need to be resolved. A resolution might take the form of extending the formal semantics so that only one meaning for each type of ellipsis nesting is possible. On the other hand, it might be useful to develop a new notation, so that the macro writer can pick whichever version he wants for a particular macro.

7.3.2. NEW NOTATIONS FOR DECLARATIONS. On the whole, the `extend-syntax` notation for macro declarations is a good one. At one time we claimed that if we wished to declare a new macro whose syntax could not be describe by `extend-syntax`, that the new macro's calling syntax was poorly designed. Even though we still largely believe this claim, there are common Lisp macros that do not lend themselves to `extend-syntax` declarations without changing their calling syntax. The chief example is `prog`. The ellipsis notation does not permit the prototype to be more than the immediately preceding term. One cannot write

```
(extend-syntax (prog) ()
   [(prog (id ...) exp0 ... label1 exp1 ...)
    whatever])
```

and expect the labels and expressions to alternate at random.

There needs to be an `extend-syntax` style notation for expressing this kind of repetition.

7.3.3. SYNTAX MACROS AND ENHANCED DECLARATIONS. We agree with Leavenworth's proposals for syntax macros. Instead of merely declaring pattern vari-

ables, there should also be some sort of syntactic categorization of the components of a new special form. Work should be done on finding a suitable notation for this type of macro declaration and implementing it. The `extend-syntax` mechanism declares "untyped" macros. For the same reasons that other forms of type-checking are incorporated in programming languages there should be a macro-type structure facility.

It might be useful for Scheme to have a declaration syntactic extension mechanism in addition to the expression one.

7.3.4. SCOPED MACRO DECLARATIONS. There are two different kinds of local macros. First, there are the macros which are declared local to a particular region of program text. Second, there are the help-macros that are local to the expansion of another macro call. We discuss each, in turn.

Common Lisp provides a means of declaring macros which are local to a region of program text. It is called `macrolet` ([70], p. 113). We have experimented with a special form `local-extend-syntax` that allows `extend-syntax` style declarations of local macros. In both cases, the declaration takes a form similar to

(`macrolet` *declarations body*...).

In Common Lisp, because of the explicit use of STFs in the declarations, care was taken to accurately express the bindings of identifiers used within the STF. Steele says:

> Macros often must be expanded at "compile time" (more generally, at a time before the program itself is executed), and so the run-time values of variables are not available to macros defined by `macrolet`. The precise rule is that the macro-expansion functions defined by `macrolet` are defined in the *global* environment; lexically scoped entities that would ordinarily be lexically apparent are not visible within the expansion functions. However, lexically scoped

> entities *are* visible within the body of the `macrolet` form and *are* visible to the code that is the expansion of a macro call. (p. 114)

In the form `local-declare-syntax`, the STFs were closed in the same lexical scope that the declaration appeared in because of Scheme structures embedded within code. Macros could not share functions and data structures unless the STFs were closed in the right scope.

In either case, we are not fond of this sort of local declaration. A Scheme expression is something that is first macro expanded and later executed. The one-expression form for local macros mixes this ordering. The macro declarations are executed first, next the bodies are macro expanded, and finally the bodies are executed. Using the Common Lisp rule, the situation is better, for one can think of the declaration/expansion pass as being completed before any execution is begun. However, with the approach we took, there was always switching between declaration, expansion, and execution phases.

In spite of our reservations about the way this kind of local declaration is made, macros local to specific regions of program text are valuable. Since the programming language is, in effect, a different, larger language within those regions, the local declarations should be made with the user cognizant that he is changing his language. One method that appeals to us is installing and removing sets of local macros by means of expressions completely separate from those in which the macro calls appear. We usually term such sets *packages*. The syntax table may utilize a lexically-based structure, with its environment contours introduced by the order packages are installed and removed.

The source of this attitude is the incremental compilation and execution of Scheme programs. Whenever macro declarations are enclosed within another expression, the purpose of the entire expression should be to make the macro declara-

tions. We distinguish those expressions that declare special forms from those which use them.

The other kind of local macros—help macros local to a particular macro—is also important. When the host language is extended with a macro that requires other, new macros, the entire language should not be burdened with the help macros. They appear only in the transcriptions of calls to the main macros; users never write calls to them. Work needs to be done on the design of a form for the declaration of help macros and on their organization within the syntax table.

7.3.5. ERROR REPORTING. A major weakness of macro processors is their destruction of source code so that compilation or run-time errors are reported in terms of the generated code rather than what the user wrote. Work needs to be done on the integration of macro processors with type-checkers, compilers, and interpreters so that source text is maintained across expansions.

7.4. Final Remarks

We have been fortunate to have both components of our syntactic extension system in use within Scheme 84 for several years. Their utility in supporting classroom and research work has been demonstrated. The ability to form textual abstractions in a convenient manner has contributed to the productivity of many people.

A thoughtfully laid out textual abstraction system has great benefit. Well-designed syntactic extension processors should be an integral part of all programming languages. With care, they can be added to existing languages so that they are not abused by programmers. The goal of a syntactic abstraction system should be to allow users to create new syntactic constructions. The restriction to permit macros only to effect syntactic transformations and not to perform other operations does not inconvenience those who are using macros for their original purpose. Assuming that programmers adopt this attitude, the language implementors can

direct their efforts toward providing a cleanly designed macro system.

We have made significant contributions toward a better macro system in one language, Lisp. Much of what we have to say about the design of syntactic extension facilities applies to all programming languages. We hope that our work motivates others to include good macro processors in other languages.

Appendix A: Twelve Design Principles

This appendix lists the twelve design principles described in Chapter 4.

Principle 1. *Since syntactic extension declarations always involve the specification of calling forms, transcription structures, and the pattern variables used in both, they should be made in such a way that the writer uses these three components as directly as possible.* (Section 4.1)

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.* (Section 4.2)

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.* (Section 4.3)

Principle 4. *A syntactic extension from domain X should expand to a complete term from domain X.* (Section 4.3)

Principle 5. *Syntactic extensions to domain X should be called in the same syntactic style as terms in domain X.* (Section 4.4)

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.* (Section 4.5)

Principle 7. *The set of keywords should be disjoint from the set of identifiers.* (Section 4.5.1)

Principle 8. *During the pre-execution processing of any program text, the set of keywords and the contents of the syntax table should remain constant.* (Section 4.5.1)

Principle 9. *Macro calls are matched against all declared patterns in the syntax table in some specified order.* (Section 4.5.2)

Principle 10. *In order to preserve program semantics, the complete expansion of syntactic extensions should take place prior to program interpretation or code generation.* (Section 4.7.1)

Principle 11. *The global environment of the macro language and the host language should not coincide.* (Section 4.8)

Principle 12. *The expansion semantics of a program P should be mathematically equivalent to the primary semantics of P.* (Section 4.9)

Appendix B: Scheme Source Code for `extend-syntax`

This appendix contains the complete code for `extend-syntax` as implemented in Chez Scheme, version 1.0 [20]. Section 5.4 contains an annotated description of the major components of this code.

```
(letrec
 ([debug false]
  [ellipsis '...]
  [mknames '(extend-syntax/code extend-syntax
             syntactic-transform-function)]
  [mkmackeywords '(extend-syntax extend-syntax/code
                   syntactic-transform-function
                   ... with withrec)]
  [*pattern* (gensym)]
  [stop-gen false]
  [with-list '(with withrec)]
  [withrec-list '(withrec)]
  [ad
   (let ([l '((car caar . cdar) (cdr cadr . cddr)
              (caar caaar . cdaar) (cadr caadr . cdadr)
              (cdar cadar . cddar) (cddr caddr . cdddr)
              (caaar caaaar . cdaaar) (caadr caaadr . cdaadr)
              (cadar caadar . cdadar) (caddr caaddr . cdaddr)
              (cdaar cadaar . cddaar) (cdadr cadadr . cddadr)
              (cddar caddar . cdddar) (cdddr cadddr . cddddr))])
     (lambda (new old)
      (if (atom? (car old))
          (cons (if (eq? new 'a) 'car 'cdr) old)
          (let ([fun (caar old)])
           (let ([funlist (assq fun l)])
            (if funlist
             (cons (if (eq? new 'a)
                       (cadr funlist)
                       (cddr funlist))
                   (cdar old))
             (cons (if (eq? new 'a) 'car 'cdr) old)))))))])
```

```
[all-quote?
 (lambda (l)
  (cond [(null? l) true]
        [(scheme-const? (car l)) (all-quote? (cdr l))]
        [(atom? (car l)) '()]
        [(eq? (caar l) 'quote) (all-quote? (cdr l))]
        [else '()]))]
[for-all (lambda (f l)
           (if (null? l) true
               (and (f (car l)) (for-all f (cdr l)))))]
[atomsin
 (lambda (l)
  (cond [(null? l) '()]
        [(atom? l) (list l)]
        [(atom? (car l))
         (cons (car l) (atomsin (cdr l)))]
        [else
         (append (atomsin (car l)) (atomsin (cdr l)))]))]
[leftmostat (lambda (l)
              (cond [(null? l) '()]
                    [(atom? (car l)) (car l)]
                    [else (leftmostat (car l))]))]
[stopq (lambda (a b)
        (let ([key (if (pair? a) (leftmostat a) a)])
         (recur loop ([b b])
          (cond [(null? b) '()]
                [(or (eq? key (car b))
                     (and (pair? (car b))
                          (eq? key (caar b))))
                 b]
                [else (loop (cdr b))]))))]
```

```
[car-cdr-chain
 (letrec
  ([car-cdr-chain-help
    (lambda (a l ans)
     (when l
      (let ([firstchain (car l)])
       (cond
        [(equal? a firstchain) (ad 'a ans)]
        [(equal? a (cdr l)) (ad 'd ans)]
        [(member* a firstchain)
         (let ([parts (memq-bar firstchain '())])
          (if (member* a (car parts))
              (car-cdr-chain-help a firstchain
               `(,(ad 'a ans)))
              (dottails a (cdr parts) `(,(ad 'a ans)))))]
        [else (car-cdr-chain-help a (cdr l)
               `(,(ad 'd ans)))])))))]
   [dottails
    (lambda (a l ans)
     (when l
      (if (equal? a l)
          `(,stopq ',(car l) . ,ans)
          (let ([elparts (memq-bar l '())])
           (if (member* a (car elparts))
               (car-cdr-chain-help a (car elparts)
                `((,stopq ',(caar elparts) . ,ans)))
               (dottails a (cdr elparts) ans)))))))]
   [memq-bar
    (lambda (l ans)
     (cond [(null? l) (cons ans l)]
           [(eq? (car l) ellipsis)
            (cons (append ans (cons ellipsis '()))
                  (cdr l))]
           [else
            (memq-bar (cdr l)
             (append ans (cons (car l) '())))]))])
```

```
  (lambda (a l seed)
   (if (equal? a l)
       seed
       (let ([parts (memq-bar l '())])
        (if (member* a (car parts))
            (car-cdr-chain-help a l (cons seed '()))
            (dottails a (cdr parts) (cons seed '()))))))))]
[dotted-list-locs
 (lambda (l1s l2s anss)
  (letrec
   ([dlls
     (lambda (l1 l2 ans)
      (if (eq? l1 l2)
          ans
          (dlls (cdr l1) l2 (ad 'd `(,ans)))))])
   (map
    (lambda (ls) (dlls (car ls) (cadr ls) (caddr ls)))
    (transpose (list l1s l2s anss)))))]
[error (lambda (msg)
        (writeln "[extend-syntax error: " msg "]")
        (reset))]
[find-dls
 (lambda (ptype l)
  (let ([targets (mk-set (pterms ptype))]
        [els (sort member* (mk-set (nest-el l)))])
   (flatmap-mk-set
    (lambda (target)
     (recur loop ([l els])
      (cond [(null? l) '()]
            [(pmem target (car l)) (car l)]
            [else (loop (cdr l))])))
    targets)))]
[pmem (lambda (target el)
       (let ([n (recur loop ([l target])
                  (if (atom? l) (cons 1 1)
                      (let ([ans (loop (car l))])
                       (cons (car ans)
                             (add1 (cdr ans))))))])
        (find-el-n-deep (car n) (cdr n) el)))]
```

```
[find-el-n-deep
 (lambda (tar n el)
  (recur loop ([l el] [m n])
   (cond [(null? l) false]
         [(negative? m) false]
         [(atom? l) (and (eq? tar l) (zero? m))]
         [(null? (cdr l)) (loop (car l) m)]
         [(eq? (cadr l) ellipsis)
          (or (loop (car l) (sub1 m))
              (loop (cddr l) m))]
         [else (or (loop (car l) m)
                   (loop (cdr l) m))]))))]
[pterms
 (lambda (pt)
  (cond
   [(null? pt) '()]
   [(atom? pt) (list pt)]
   [(null? (cdr pt))
    (if (atom? (car pt))
        pt
        (pterms (car pt)))]
   [(eq? (cadr pt) ellipsis)
    (let ([ps (pterms (car pt))])
     (append
      (map (lambda (p) (list p ellipsis)) ps)
      (pterms (cddr pt))))]
   [else (append (pterms (car pt))
          (pterms (cdr pt)))]))]
[nest-el
 (lambda (l) (append (nest-el1 l) (nest-el2 l)))]
[nest-el1
 (lambda (l)
  (cond [(null? l) '()]
        [(atom? l) '()]
        [(memq ellipsis l)
         (cons l (nest-el1 (cdr (memq ellipsis l))))]))]
```

```
    [nest-el2
     (lambda (l)
      (cond [(atom? l) '()]
            [(null? l) '()]
            [(atom? (car l)) (nest-el2 (cdr l))]
            [else
             (append (nest-el (car l))
              (nest-el2 (cdr l)))]))]
   [flatmap-mk-set
    (lambda (f l)
     (if (null? l)
         '()
        (let ([a (f (car l))])
         (if a
          (let ([rest (flatmap-mk-set f (cdr l))])
           (if (memq a rest)
               rest
               (cons a rest)))
          (flatmap-mk-set f (cdr l))))))]
[funbodfun (lambda (pt) `(,(car pt)))]
[idfun?
 (lambda (b)
  (or (atom? b)
      (and (eq? (car b) 'cons)
           (not (atom? (cadr b)))
           (eq? (caadr b) 'car)
           (not (atom? (caddr b)))
           (eq? (car (caddr b)) 'cdr)
           (equal? (cdr (cadr b)) (cdr (caddr b))))))]
```

```
[ifify
 (lambda (l)
  (cond [(null? l) '()]
        [(eq? (caar l) true) (cadar l)]
        [else
         `(if ,(caar l)
              ,(cadar l)
              ,(ifify (cdr l)))]))]
[member*
 (lambda (a l)
  (cond [(equal? a l) l]
        [(null? l) '()]
        [(atom? l) (eq? a l)]
        [else (or (member* a (car l))
                  (member* a (cdr l)))]))]
[mk-matcher
 (lambda (k p)
  (mm? `(cdr ,*pattern*) k (cdr p)))]
[mm?
 (lambda (a k p)
  (cond
   [(null? p) `(null? ,a)]
   [(memq p k) `(eq? ,a (quote ,p))]
   [(not (pair? p)) true]
```

```
[(and (pair? (cdr p)) (eq? (cadr p) ellipsis))
 (cond
  [(and (pair? (cddr p)) (memq (caddr p) k))
   (let ([temp (gensym)])
    `(and (pair? ,a)
          (let ([,temp (,stopq (quote ,(caddr p)) ,a)])
           (and ,temp ,(mm? temp k (cddr p))))))]
  [(and (pair? (cddr p))
        (pair? (caddr p))
        (memq (car (caddr p)) k))
   (let ([temp (gensym)])
    `(and
      (pair? ,a)
      (let ([,temp (,stopq (quote ,(car (caddr p))) ,a)])
       (and ,temp ,(mm? temp k (cddr p))))))]
  [(null? (cddr p))
   (let ([ans (let* ([arg (gensym)]
                     [ans (mm? arg k (car p))])
                (if (eq? ans true)
                    true
                    `(,for-all
                      (lambda (,arg) ,ans)
                      ,a)))])
    `(or (null? ,a)
         ,(if (eq? ans true)
              `(pair? ,a)
              `(and (pair? ,a) ,ans))))]
  [else (error "bad use of ellipsis in pattern")])]
[else
 (let ([ans (remq true
                  `(and (pair? ,a)
                        ,(mm? (ad 'a `(,a)) k (car p))
                        ,(mm? (ad 'd `(,a)) k (cdr p))))])
  (if (= (length ans) 2)
      (cadr ans)
      ans))]))]
```

```
[mk-set
 (lambda (l)
  (cond [(null? l) '()]
        [(member (car l) (cdr l)) (mk-set (cdr l))]
        [else (cons (car l) (mk-set (cdr l)))]))]
[mostopsfun (lambda (dl) (when (cddr dl)
                               (set! stop-gen true)
                               (caddr dl)))]
[ormap (lambda (f l)
        (and l (or (f (car l)) (ormap f (cdr l)))))]
[partial-process
 (lambda (p abrv specials geners)
  (letrec
   ([pp (lambda (p)
         (cond [(atom? p) p]
               [(null? (cdr p)) (list (pp (car p)))]
               [(eq? (car p) 'quote)
                (mk-expander abrv (cadr p) specials geners
                 true abrv)]
               [else (cons (pp (car p)) (pp (cdr p)))]))])
   (pp p)))]
[pt&mo
 (lambda (l) (cond [(atom? l) '()]
                   [(atom? (cdr l)) '()]
                   [(eq? (cadr l) ellipsis) l]
                   [else (pt&mo (cdr l))]))]
[remall (lambda (l1 l2)
         (if l1
          (remall (cdr l1) (remq! (car l1) l2))
          l2))]
[remquote
   (lambda (l)
      (when l
         (cons (if (scheme-const? (car l))
                   (car l)
                   (cadr (car l)))
               (remquote (cdr l)))))]
```

```
[scheme-const? (lambda (x) (or (null? x) (eq? x true)
                               (number? x) (string? x)))]

[*pattern*-subst
 (lambda (newid mapfunbod)
  (subst *pattern* '*temp* (subst newid *pattern* mapfunbod)))]
[mk-expander
 (lambda (abrv trans specials geners nesting fullabrv)
  (letrec (
[mk-one-line-macro
 (lambda (trans)
  (cond [(atom? trans) (mk-atom trans)]
        [(atom? (cdr trans))
         `(cons ,(mk-one-line-macro (car trans))
                ,(mk-atom (cdr trans)))]
        [(memq ellipsis trans) (mk-ellipsis-body trans)]
        [else (if (member* trans abrv)
                  (car-cdr-chain trans abrv *pattern*)
                  (let ([exd (mk-regular-body trans)])
                   (if (all-quote? exd)
                       `(quote ,(remquote exd))
                       `(list . ,exd))))]))]

[geners-in
 (lambda (ptype geners)
  (let ([apt (atomsin ptype)])
   (recur loop ([glist geners])
    (cond [(null? glist) '()]
          [(memq (caar glist) apt)
           (cons (caar glist) (loop (cdr glist)))]
          [else (loop (cdr glist))]))))]
[append-geners
 (lambda (ptabrvs geners)
  (if (pair? geners)
      (if (null? ptabrvs)
          (if (null? (cdr geners)) (car geners) geners)
          (cons ptabrvs geners))
      ptabrvs))]
```

```
[mk-ellipsis-body
 (lambda (el-trans)
  (if abrv
   (let ([prototype (let ([temp (pt&mo el-trans)])
                      (if temp (car temp) '()))])
    (if
     (equal? (car el-trans) prototype)
     (let*
      ([used-geners (geners-in prototype geners)]
       [dotted-lists (find-dls prototype abrv)]
       [pt&mo-abrvs (map pt&mo dotted-lists)]
       [pt&mostops (map mostopsfun pt&mo-abrvs)]
       [mapfunbod
        (mk-expander
         (append-geners
          (if (null? pt&mo-abrvs) '()
              (if (cdr pt&mo-abrvs)
                  (car (transpose
                        (map funbodfun pt&mo-abrvs)))
                  (caar pt&mo-abrvs)))
          used-geners)
         prototype specials '() nesting fullabrv)]
       [elocs
        (append
         (dotted-list-locs dotted-lists pt&mo-abrvs
          (map
           (lambda (dl)
            (car-cdr-chain dl abrv *pattern*))
           dotted-lists))
         used-geners)]
       [newid (gensym)]
       [mapfun (if (idfun? mapfunbod)
                   newid
                   (*pattern*-subst newid mapfunbod))]
```

```
[el-part
 (cond
  [(and elocs (car elocs))
   (if (and (atom? mapfun)
            (not (and stop-gen (car pt&mostops))))
       (if (cdr elocs)
           `(transpose (list . ,elocs))
           (car elocs))
       (let*
        ([map-l
          (if (cdr elocs)
              `(transpose
                  (list .
                   ,(if (and stop-gen
                             (car pt&mostops))
                        (map
                         (lambda (el)
                          `(,trim ,el
                            ',pt&mostops))
                         elocs)
                        elocs)))
              (if (and stop-gen (car pt&mostops))
                  `(,trim ,(car elocs)
                    ',pt&mostops)
                  (car elocs)))]
         [map-exp
          (if (atom? mapfun)
              map-l
              `(map
                ,(cond
                  [(and (atom? (cadr mapfun))
                        (null? (cddr mapfun)))
                   (car mapfun)]
                  [else `(lambda (,newid)
                           ,mapfun)])
                ,map-l))])
       map-exp))]
```

```
        [nesting
         (let ([try
                (subst '*temp* *pattern*
                 (mk-expander fullabrv el-trans
                  specials geners nesting '()))])
          (if try try
              (begin
               (when debug
                (begin
                 (newline)
                 (writeln "abrv: " abrv)
                 (writeln "trans: " trans)
                 (writeln "el-trans: " el-trans)
                 (writeln "fullabrv: " fullabrv)
                 (writeln "prototype: " prototype)
                 (writeln "used-geners: " used-geners)
                 (writeln "dotted-lists: " dotted-lists)
                 (writeln "pt&mo-abrvs: " pt&mo-abrvs)
                 (writeln "pt&mostops: " pt&mostops)
                 (writeln "mapfunbod: " mapfunbod)
                 (writeln "elocs: " elocs)
                 (writeln "mapfun: " mapfun)
                 (newline)))
(writeln "[expansion prototype: " prototype "]")
(writeln "[pattern: " abrv "]")
(error "expansion prototype has no pattern prototype match")))])
          [else `',el-trans])])
      (if (cddr el-trans)
          `(append ,el-part ,(mke (cddr el-trans)))
          el-part))
    (let ([exd `(cons ,(mke (car el-trans))
                      ,(mk-ellipsis-body (cdr el-trans)))])
      (if (idfun? exd)
          (cadr (cadr exd))
          exd))))
  '()))]
```

```
[mk-atom (lambda (atm)
           (cond [(scheme-const? atm) atm]
                 [(memq atm specials) atm]
                 [(assq atm geners) atm]
                 [(member* atm abrv)
                  (car-cdr-chain atm abrv *pattern*)]
                 [(member* atm fullabrv)
                  (car-cdr-chain atm fullabrv '*temp*)]
                 [else `(quote ,atm)]))]
[mk-regular-body
 (lambda (trans)
  (when trans
   (let ([firstex (car trans)])
    (if
     (atom? firstex)
     `(,(mk-atom firstex) . ,(mk-regular-body (cdr trans)))
     (if (closure? firstex)
         `(,firstex . ,(mk-regular-body (cdr trans)))
         `(,(mke firstex) . ,(mk-regular-body (cdr trans))))))))]
[mke (lambda (trans)
      (cond [(atom? trans) (mk-atom trans)]
            [else (when (memq (car trans) mknames)
                        (set! nesting false))
                  (mk-one-line-macro trans)]))])

(mke trans)))]
[trim (lambda (l s)
        ((rec ts (lambda (l)
                   (if (or (null? l) (stopq (car l) s))
                       '()
                       (cons (car l) (ts (cdr l))))))

         l))]
```

```
[get-geners
 (lambda (with-pairs)
  (cond [(null? with-pairs) '()]
        [(and (pair? (caar with-pairs))
              (eq? (cadr (caar with-pairs)) ellipsis))
         (cons (cons (caaar with-pairs) (cdar with-pairs))
               (get-geners (cdr with-pairs)))]
        [else (get-geners (cdr with-pairs))]))]
[regroup-withs
 (lambda (e abrv specials geners)
  (if (new-variables? e)
      (let*
       ([newgeners (get-geners (cadr e))]
        [allspecials
         (filter-geners
          (append (map car (cadr e)) specials)
          newgeners)]
        [allgeners (append newgeners geners)]
        [gsp (regroup-withs (caddr e) abrv
              allspecials
              allgeners)]
        [weaver
         (let
          ([g (if (eq? (car e) 'withrec)
                  allgeners
                  geners)]
           [s (if (eq? (car e) 'withrec)
                  allspecials
                  specials)])
```

```
          (lambda (defpair)
           (cond [(atom? (car defpair))
                  (let ([pp-def (partial-process
                                 (cdr defpair) abrv
                                 s g)])
                    (cons (car defpair) pp-def))]
                 [(and (pair? (car defpair))
                       (eq? (cadr (car defpair)) ellipsis))
                  `(,(caar defpair)
                    ,(partial-process (cadr defpair) abrv
                      s g))]))])
       (cons (cons (cons (car e) (map weaver (cadr e)))
                   (car gsp))
             (cdr gsp)))
      (cons '() (cons geners e))))]
[letrecify
 (lambda (sprec e)
  (cond [(null? sprec) e]
        [else
         `(,(if (memq (caar sprec) withrec-list)
                'letrec
                'let)
           ,(cdar sprec) ,(letrecify (cdr sprec) e))]))]
[getspecs (lambda (sprec)
           (cond [(null? sprec) '()]
                 [else (append (map car (cdar sprec))
                               (getspecs (cdr sprec)))]))]
[filter-geners
 (lambda (specials geners)
  (cond [(null? specials) '()]
        [(assq (car specials) geners)
         (filter-geners (cdr specials) geners)]
        [else
         (cons (car specials)
               (filter-geners (cdr specials) geners))]))]
```

```
[new-variables?
 (lambda (expan) (and (pair? expan)
                      (memq (car expan) with-list)))]

[fender-present?
 (lambda (patpair) (not (null? (cddr patpair))))]
[mk-match-exp
 (lambda (patpairs keywords specials geners)
  (if patpairs
   (ifify
    (append
     (map
      (let
       ([doit
         (lambda (pat preds expan)
          `(,(if preds
                `(and ,(mk-matcher keywords pat)
                      (let ([,*pattern*
                             (unstamp
                              ,*pattern*)])
                        ,(partial-process preds pat
                          '() '())))
                (mk-matcher keywords pat))
              ,(if (new-variables? expan)
                   (let* ([sprec (regroup-withs expan
                                  pat specials geners)]
                          [newgeners (cadr sprec)])
                     (letrecify (car sprec)
                      (mk-expander pat
                       (cddr sprec)
                       (filter-geners
                        (append (getspecs (car sprec))
                         specials)
                        newgeners)
                       newgeners true pat)))
                   (mk-expander pat expan
                    specials geners true pat))))])
```

```
      (lambda (patpair)
       (if (fender-present? patpair)
           (doit (car patpair) (cadr patpair)
                 (caddr patpair))
           (doit (car patpair) '() (cadr patpair)))))
     patpairs)
    `((else
       (begin
        (writeln "[syntactic extension error: ")
        (writeln "syntactic extension use "
         (unstamp ,*pattern*))
        (writeln
         "does not match any pattern in its defintion]"))))))
  (error "You need at least one pairing"))))]
[main-body
 (let
  ([do-main-body
    (lambda (pair-list keywords-list)
     (set! stop-gen false)
     `(lambda (,*pattern*)
       ,(mk-match-exp pair-list
         keywords-list '() '())))])
  (lambda (keywords pairs)
   (do-main-body
    (unstamp-no-copy pairs)
    (unstamp-no-copy keywords))))])
(set! mkmac-debug (lambda (x) (set! debug x)))
(add-to-syntax-table
 (cons 'syntactic-transform-function mkmackeywords)
 (lambda (l)
  (let ([name (car (car (cadddr l)))])
   (let ([keywords
          (if (eq? (and (cadr l) (car (cadr l))) name)
              (cadr l)
              (if (memq name (cadr l))
                  (cons name (remq! name (cadr l)))
                  (cons name (cadr l))))])
    (main-body keywords (cdddr l))))))
```

```
(add-to-syntax-table
 (cons 'extend-syntax/code mkmackeywords)
 (lambda (l)
  (let ([name (car (car (cadddr l)))])
   (let ([keywords
          (if (eq? (and (cadr l) (car (cadr l))) name)
              (cadr l)
              (if (memq name (cadr l))
                  (cons name (remq! name (cadr l)))
                  (cons name (cadr l))))])
    `(begin (newline)
            (pretty-print ',(main-body keywords (cdddr l)))
            (newline)
            ',name)))))
(add-to-syntax-table (cons 'extend-syntax mkmackeywords)
 (lambda (l)
  (let ([name (car (car (cadddr l)))])
   (let ([keywords
          (if (eq? (and (cadr l) (car (cadr l))) name)
              (cadr l)
              (if (memq name (cadr l))
                  (cons name (remq! name (cadr l)))
                  (cons name (cadr l))))])
    `(eval-when (compile load eval)
      (add-to-syntax-table ',(append keywords (caddr l))
       ,(main-body keywords (cdddr l))))))))
```

Bibliography and References

1. ALBERGA, CYRIL N. LISP Assembler Program: Reference Manual. IBM Research Report RA 172 (#51268), IBM Thomas J. Watson Research Center, Yorktown Heights NY, 23 September 1985.

2. ARDEN, B. W., B. A. GALLER, AND R. M. GRAHAM. The MAD definition facility. *Commun. ACM 12*, 8 (August 1969), 432–439.

3. BARENDREGT, H. P. *The Lambda Calculus: its Syntax and Semantics.* Revised edition, *Studies in Logic and the Foundations of Mathematics 103*, North-Holland, Amsterdam, 1984.

4. BARENDREGT, H. P. Introduction to the lambda calculus. *Nieuw Archief voor Wisenkunde 2*, 4 (1984), 337–373.

5. BOWLDEN, HENRY J. Macros in higher-level languages. *SIGPLAN Notices 6*, 12 (December 1971), 39–44.

6. BROOKS, RODNEY A. *Programming in Common Lisp.* John Wiley & Sons, Inc., New York, 1985.

7. BROWN, P. J. The ML/I macro processor. *Commun. ACM 10*, 10 (October 1967), 618–623.

8. BROWN, P. J. *Macro Processors and Techniques for Portable Software*, John Wiley & Sons, London, 1974.

9. BROWN, P. J. Macros without tears. *Software—Practice and Experience 9*, 6 (June 1979), 433–437.

10. BROWN, P. J. SUPERMAC—a macro facility that can be added to existing compilers. *Software—Practice and Experience 10*, 6 (June 1980), 431–434.

11. Brown, P. J. and J. A. Ogden. The SUPERMAC macro processor in Pascal. *Software—Practice and Experience 13*, 4 (April 1983), 295–304.

12. Calingaert, Peter. *Assemblers, Compilers, and Program Translation*. Computer Science Press, Inc., Potomac MD, 1979.

13. Campbell-Kelly, M. *An Introduction to Macros*. Macdonald, London and American Elsevier Inc., New York, 1973.

14. Cheatham, T. E., Jr. The introduction of definitional facilities into higher level programming languages. In *AFIPS Proceedings—Fall Joint Computer Conference 29*, (1966), 623–637.

15. Chen, Pee-Hong and Daniel P. Friedman. Prototyping Data Flow by translation into Scheme. Technical Report No. 147, Computer Science Department, Indiana University, Bloomington, (August 1983).

16. Christensen, Carlos and Christopher J. Shaw, Eds. *Proceedings of the Extensible Languages Symposium. SIGPLAN Notices 4*, 8 (August 1969).

17. Church, Alonzo. *The Calculi of Lambda Conversion*. Annals of Mathematics Studies No. 6. Princeton University Press, Princeton NJ (1941).

18. Cole, A. J. *Macro Processors*. Cambridge University Press, Cambridge, 1976.

19. Comer, Douglas. MAP: a Pascal macro preprocessor for large program development. *Software—Practice and Experience 9*, 3 (March 1979), 203–209.

20. Dybvig, R. Kent. *The SCHEME Programming Language*. Prentice-Hall, Inc., Englewood Cliffs NJ, to appear.

21. Felleisen, Matthias. Transliterating Prolog into Scheme. Technical Report No. 182, Computer Science Department, Indiana University, Bloomington IN (October 1985).

22. Felleisen, Matthias and Daniel P. Friedman. A closer look at export and import statements. *Computer Languages 11*, 1 (January 1986), 29–37.

23. Flores, Ivan, Paul Kaminsky, and Daniel Ryan. List and execute forms of macros. *Computer Languages 1*, 1 (January 1975), 45–60.

24. Flores, Ivan. Making macrospace effective. *Computer Languages 3*, 2 (1978), 95–113.

25. Foderaro, John K., Keith L. Sklower, and Kevin Layer. *The FRANZ LISP Manual*. University of California, Berkeley CA, June 1983.

26. Friedman, Daniel P., Christopher T. Haynes, Eugene Kohlbecker, and Mitchell Wand. Scheme 84 interim reference manual, version 0.7. Computer Science Department, Indiana University, Bloomington IN (June 1985).

27. Galler, B. A. and A. J. Perlis. A proposal for definitions in ALGOL. *Commun. ACM 10*, 4 (April 1967), 204–219.

28. Galler, B. A. and A. J. Perlis. *A View of Programming Languages*. Addison-Wesley Publishing Company, Reading MA, 1970.

29. Halpern, Charles D. An implementation of 2-Lisp. Technical Report No. 160, Computer Science Department, Indiana University, Bloomington IN (June 1984).

30. Hammer, Michael. An alternative approach to macro processing. *SIGPLAN Notices 6*, 12 (December 1971), 58–64.

31. Harland, David M. *Polymorphic Programming Languages, design and implementation*. Ellis Horwood Limited, Chichester, 1984.

32. Harrison, Malcolm C. BALM—an extendable list-processing language. In *AFIPS Proceedings—Spring Joint Computer Conference 36*, (1970), 507–511.

33. Henderson, Peter and Roger B. Gimson. Modularization of large programs. *Software—Practice and Experience 11*, 5 (May 1981), 497–520.

34. IBM Corporation. *Student Text: An Introduction to the Compile-time Facilities of PL/I*. Form C20-1689-0, IBM Corporation, White Plains NY, 1968.

35. Irons, Edgar T. Experience with an extensible language. *Commun. ACM 13*, 1 (January 1970), 31–40.

36. Jensen, Kathleen and Niklaus Wirth. *Pascal User Manual and Report*. Springer-Verlag, New York, 1974.

37. Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs NJ, 1978.

38. Knuth, Donald E. *The TEXbook*. Addison-Wesley Publishing Company, Reading MA, 1984.

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

40. Landin, Peter J. An abstract machine for designers of computing languages. *Proceedings of the IFIP Congress 65*, 438–439.

41. Layzell, P. J. The history of macro processors in programming language extensibility. *The Computer Journal 28*, 1 (January 1985), 29–33.

42. Leavenworth, B. M. Syntax macros and extended translation. *Commun. ACM 9*, 11 (November 1966), 790–793.

43. Lindstrom, Gary. Control extension in a recursive language. *BIT 13*, (1973), 50–70.

44. Liskov, B., A. Snyder, R. R. Atkinson, and J. C. Schaffert. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (August 1977), 564–576.

45. MacLaren, M. Donald. Macro processing in EPS. *SIGPLAN 4*, 8 (August 1969), 32–36.

46. Macleod, J. A. MP/1—a FORTRAN macroprocessor. *The Computer Journal 14*, 3 (August 1971), 229–231.

47. McCarthy, John, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. Second edition. MIT Press, Cambridge MA, 1965.

48. M. Douglas McIlroy. Macro instruction extensions of compiler languages. *Commun. ACM 3*, 4 (April 1960), 214–220.

49. Meehan, James R., Ed. *The New UCI Lisp Manual*. Lawrence Erlbaum Associates, Inc., Hillsdale NJ, 1979.

50. Metzner, John R. A graded bibliography on macro systems and extensible languages. *SIGPLAN Notices 14*, 1 (January 1979), 57–68.

51. Mooers, C. N. TRAC, procedure describing language for the reactive typewriter. *Commun. ACM 9*, 3 (March 1966), 215–219.

52. Mooers, C. N. How some fundamental problems are treated in the design of the TRAC language. In *Symbol Manipulation Languages and Techniques*. D. Bobrow, Ed., North-Holland, Amsterdam, 1968, 178–190.

53. Mooers, C. N. and L. P. Deutsch. TRAC, a text handling language. In *Proceedings of the 20th ACM National Conference*, (1965), 229–246.

54. Moon, David, Richard M. Stallman, and Daniel Weinreb. *Lisp Machine Manual*, fifth edition. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge MA, January 1983.

55. Munn, R. J. and J. M. Stewart. RATMAC: a preprocessor for writing portable scientfic software. *Software—Practice and Experience 10*, 9 (September 1980), 743–749.

56. Nagata, Hiroyasu. FORMAL: a language with a macro oriented extension facility. *Computer Languages 5*, 2 (1980), 65–76.

57. Naur, Peter, Ed. Revised report on the algorithmic language ALGOL 60. *Commun. ACM 6*, 1 (January 1963), 1–17.

58. Pitman, Kent M. Special forms in Lisp. In *Conf. Record of the 1980 LISP Conference*, Standford (August 1980), 179–187.

59. Quine, Willard Van Orman. *Mathematical Logic*. Harvard University Press, Cambridge MA, 1951.

60. Rees, Jonathan and William Clinger, Eds. The revised[3] report on the algorithmic language Scheme. Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA (July 1986).

61. Revesz, Gyorgy. A note on macro generation. Technical Report 83-112, Department of Computer Science, Tulane University, New Orleans LA (October 1983).

62. Sammet, Jean E. Why Ada is not just another programming language. *Commun. ACM 29*, 8 (August 1986), 722–732.

63. Sassa, Masataka. A pattern matching macro processor. *Software—Practice and Experience 9*, 6 (June 1979), 439–456.

64. Schooler, Richard. *Partial Evaluation as a Means of Language Extensibility*. Master of Science thesis, Massachusetts Institute of Technology Laboratory for Computer Science, 1984.

65. Schuman, Stephen A., Ed. *Proceedings of the International Symposium on Extensible Languages. SIGPLAN Notices 6*, 12 (December 1971).

66. Smith, Brian Cantwell. *Reflection and Semantics in a Procedural Language.* Ph.D. dissertation, Laboratory for Computer Science Report MIT-TR-272, Massachusetts Institute of Technology, Cambridge MA, January 1982.

67. Smith, Brian Cantwell. Reflection and semantics in Lisp. In *Conf. Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, Salt Lake City (January 1984), 23–35.

68. Solntseff, N. A classification of extensible programming languages. *Information Processing Letters 1*, 3 (February 1972), 91–96.

69. Standish, Thomas A. Extensibility in programming language design. *SIGPLAN Notices 10*, 7 (July 1975), 18–21.

70. Steele, Guy L., Jr. *Common Lisp: the Language.* Digital Press/Digital Equipment Corporation, Bedford MA, 1984.

71. Steele, Guy Lewis, Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. AI Memo No. 452, Massachusetts Institute of Technology Artificial Intelligenece Laboratory (January 1978).

72. Stoy, Joseph E. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge MA, 1977.

73. Strachey, C. A general purpose macrogenerator. *The Computer Journal 8*, 3 (October 1965), 225–241.

74. Triance, J. M. and P. J. Layzell. Macro processors for enhancing high-level languages—some design principles. *The Computer Journal 28*, 1 (January 1985), 34–43.

75. Waite, W. M. The mobile programming system: STAGE2. *Commun. ACM 13*, 7 (July 1970), 415–421.

76. Wegbreit, Ben. *Studies in Extensible Programming Languages.* Ph.D. dissertation, Center for Research in Computing Technology, Harvard University, Cambridge MA, 1972.

77. Wegner, Peter. *Programming Languages, Information Structures, and Machine Organization.* McGraw-Hill Book Company, New York, 1968.

78. Wilensky, Robert. *LISPcraft.* W. W. Norton & Company, New York, 1984.

79. Wirth, N. The programming language Pascal. *Acta Inf. 1* (1971), 35–63.

80. WIRTH, NIKLAUS. *Programming in Modula-2*, (2nd edition). Springer-Verlag, Berlin, 1982.

81. WITTGENSTEIN, LUDWIG. *Tractatus Logico-Philosophicus*. Routledge and Kegan Paul, London, 1922.

Vita

Eugene Edmund Kohlbecker, Jr. received a Bachelor of Science degree in Mathematics and Physics from MacMurray College, Jacksonville, Illinois, in 1975. He completed a Master of Science degree in Mathematics from the University of Illinois at Urbana-Champaign in 1977. From 1977 to 1981 he was an Instructor of Mathematics, Computer Science, and Physics at MacMurray College, where he also served as Director of the Computer Facility from 1979 to 1981.

Since 1981 he has been an Associate Instructor and Research Assistant at Indiana University, Bloomington. In 1984 he participated in the Engineering Summer Development Program at the Computer Science Laboratory of Texas Instruments, Inc. in Dallas. During 1985 he was an IBM Graduate Fellow. He has been a member of the Association for Computing Machinery since 1979.