Syntactic Exposures - A Lexically-Scoped Macro Facility for Extensible Compilers

by

Stephen Paul Carl, B.S.E.E.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Arts

The University of Texas at Austin

December 1996

Syntactic Exposures - A Lexically-Scoped Macro Facility for Extensible Compilers

Approved by
Supervising Committee:

____

____

This thesis is dedicated to the memory of my grandmother

Vivian Pauline Coday

June 29, 1917 - December 16, 1995

whose love and prayers have affected me more than I'll ever know.

Acknowledgments

I would like to thank Dr. Paul Wilson for his patience, for the long teaching sessions, and for taking the time to make this thesis possible. I have Donovan Kolbly to thank for a key implementation mechanism, for the metacircular interpreter I worked with, and for the RScheme compiler.

My thanks and love go to Christie and the little ones for letting me be away when work needed doing, and for spurring me on when necessary. Now I know what all those authors *really* mean when they thank their spouse and children for putting up with the late nights. For showing their approval and support, rather than let on how crazy they thought I was to try this, I thank my parents, Roger and Paula Carl.

To return to school and work toward an advanced degree had been a dream of mine for five years; without Don Chapman's encouragement and financial support it might never have happened. He has my deepest appreciation.

Stephen Paul Carl

The University of Texas at Austin
December 1996

Syntactic Exposures - A Lexically-Scoped Macro Facility for Extensible Compilers

Stephen Paul Carl, M.A.
The University of Texas at Austin, 1996

Supervisor: Paul R. Wilson

This thesis describes an implementation of *syntactic extensions* (or *macros*) which is novel in that it provides a new solution to the *name capture* problem and a foundation on which to build extensible compilers through *reflection on scope and context.* Syntactic extension is a powerful mechanism which can be used to extend the capabilities of a programming language by expressing new syntactic forms in terms of the base language. With the power comes a price—used carelessly, such extensions can introduce subtle errors into a program, because traditional implementations do not respect the lexical scope of identifiers. *Lexically-scoped* macro facilities were developed to ensure that lexical scope of identifiers is not violated; problems such as name capture, which can cause unexpected side effects, are thus avoided.

The approach described here, called *syntactic exposures*, performs syntactic extension without macro expansion; rather than replacing macro calls with macro body forms during a prepass, a macro's body expressions are compiled in place at the point of call. A refinement of normal scoping mechanisms prevents name capture from occurring, and compile-time information is used to determine how to treat identifiers. These features can be exploited to write macros which act as "hooks" into the compiler to modify scope rules, and to perform context-sensitive analyses and optimizations.

Contents

List of Figures

Chapter 1

Introduction

Syntactic extension is a programming language feature which can be used to write extensions to the syntax of the language; this is done by creating a transformation which maps new syntactic forms to expressions which are part of the base language. Such extensions, more generally known as *macros*, have been available for many years in various languages; a simple macro substitution facility is part of the C language [KR88], and a very powerful macro expansion system was incorporated into Lisp [SG93, Ste84, Que94] and its variants.[1] Macros written using the latter facility have been the basis for adding many important extensions to the language, such as control constructs, support for object-oriented programming [Pae93], and meta-object protocols [KdesRB91]. More generally, a macro facility can help in modularizing a language implementation into layers. Such an implementation might consist of a base compiler that understands a subset of the language and successive layers which implement the rest of the language forms as macros. During compilation, those forms in a program which are implemented as macros in the language implementation can be optimized and then transformed into the language subset understood by the base compiler.

[1] Lisp macros were invented by Timothy P. Hart in 1963, but their expressiveness is still not widely recognized. Similar facilities are reinvented on a regular basis, and often without recognition of the subtleties of this kind of macro system.

With such power comes a price—used carelessly, syntactic extensions can introduce subtle errors into a program. This problem is aggravated by the fact that most existing macro systems are poorly integrated into the base language and are hard to use correctly. We believe that this accounts for much of the resistance to macros outside the Lisp community. Because few of those who program in other languages have used a powerful macro facility, its utility is not widely recognized.

Traditional macro systems employ a substitution or *rewriting* step which transforms a macro call to the desired base-language expressions, usually in a pass through the program text before compilation; this is called *macro expansion.* Naive macro expansion causes the insertion of expressions defined in one place in a program into another where the expressions might have an entirely different meaning. The inserted expressions have no connection with the context in which the macro call occurs; this lack of contextual information can plague a programmer who wishes to use macros. In particular, it is the *lexical scope of identifiers*, which is not known when a macro is expanded, that is the important context that needs to be preserved. *Lexically-scoped macro systems* have been developed as an attempt to maintain scope information during macro expansion.

In this thesis we introduce a new lexically-scoped macro system which differs from systems based on preprocessing in a number of ways. First, this system provides syntactic extension without performing macro expansion during a separate phase of compilation. When a macro call is detected in a program text, our system compiles the body of the macro definition at the point of call; no separate parsing, analysis, or rewriting steps are performed as part of macro expansion. Second, lexical scope of identifiers is maintained by using binding environments which are extensions of the compiler's own data structures. As a result, the system has these advantages: it can be easy to understand and implement, it can be the basis for an extensible compiler, and it supports more flexible grammatical properties for extensible languages.

While this macro facility has been developed for an implementation of Scheme (a dialect of Lisp) [R4RS], we believe that the same mechanisms could be used for more conventional (Algol-like or C-like) block-structured, lexically-scoped extensible languages.

Before describing our system, we will first show that syntactic extensibility is an important programming language feature and that current facilities that support syntactic extensions are lacking. Given this motivation, we'll then see what it is that makes our system new and useful.

1.1 Why We Want Syntactic Extensions

Simple text-substitution macro facilities like that found in C allow a programmer to get the effect of defining constants or inlining procedures. For example, so-called "magic numbers" can be replaced with meaningful names (like PI for 3.14159) with a line of the form:

```
#define PI 3.14159
```

Also, small code fragments can be replaced with constructs, similar to procedure calls, that are inlined before compilation. While this can be convenient, it is in general a bad idea to use a macro facility for inline functions; it is cleaner and less error-prone for a language to provide facilities for defining constants and inlining.[2] On the other hand, a macro such as the familiar `swap` example [KR88], given below, is reasonable since it cannot be written in a language where arguments are passed by value.

```
#define swap(x, y) { int temp; temp = x; x = y; y = temp; }
```

[2] In fact, the RScheme compiler (designed by members of our research group) has an inline function facility implemented using a macro that generates macro definitions. An example of such a macro for R4RS Scheme is given in [DHB92].

More powerful macro expansion facilities such as that found in Lisp allow the user to construct *transcription functions* which perform the macro expansion by any number of means from simple template-filling functions to arbitrarily complex computations that produce optimized constructs tailored for specific situations. Syntactic extensions based on these more powerful facilities are useful in helping to write good programs by offering the following software engineering benefits:

- It is easier to write clear, understandable programs by substituting macro calls (which are formed by giving the macro name and zero or more argument expressions) for more complex expressions.
- Application-specific linguistic constructs can be specified and used in the program code that will be converted to the target language at compile-time. This makes it possible to write a program partially in the language of the problem domain.
- Compiler writers can convert some language-level syntax forms into a smaller set of forms, or convert much of the language-level syntax into a lower-level base syntax. In either case the benefit is that the compiler itself can be written to handle only a small number of well-understood forms, and to handle them very well [Ste79, Pit80].
- A language implementation can be structured in an open, layered way, so that lower-level details can be revealed and controlled as necessary to extend the system, retarget it to different platforms, or interface it cleanly and efficiently to other software systems such as database systems, software engineering support tools, and application domain-specific problem solvers[KdesRB91, Pae93, BTS94, WKC97].

There are those computer scientists who believe that syntactic extensibility is a feature that should be avoided. They argue that extensibility can lead to a proliferation of incompatible versions of languages, leading to a Tower of Babel situation where these languages are incomprehensible to all but those programmers who have worked to develop them. We would counter with the following points:

- A Babel-esque situation already exists precisely because of a *lack* of extensibility; many special-purpose "little languages" have been implemented which are completely incompatible with other languages and are often poorly designed. Also, language features are often built into applications (such as spreadsheets, word processors, and graphics programs) so that the applications end up looking like domain-specific programming environments. In situations where new languages are inevitable, variants of a good extensible language are preferable to a proliferation of entirely incompatible languages.

- By using an extensible system as a base for new language features, incompatibilities between code written in the new dialect of a language and legacy code can be minimized.

- Adding a small number of new syntax forms can greatly clarify code where the new forms present appropriate and useful abstractions. Many libraries extend a language with new application-specific control constructs and data definition facilities hidden behind a procedural interface. We believe that this makes the resulting programming language less clear and consistent without making the language any easier to learn or use.

- Rather than add new syntax, an "extension" might re-implement existing syntax. This is especially useful when the performance cost of an otherwise useful construct is unacceptable; by writing an extension that optimizes the construct, we ensure that its benefits are realizable.

For a more complete treatment of these issues, see [WKC97].

1.2 Why Our System Is Novel

The macro system described in this thesis works by incorporating a refinement of the normal scope mechanisms used in Scheme compilers (and many compilers for other block-structured languages such as Pascal and ML). As such, it is an intuitive extension of the ideas underlying most, if not all, Scheme implementations. This approach is applicable to other languages as well.

The macro system is integrated with the compiler; rather than build a separate subsystem that works, conceptually, as a preprocessor, our system is part of the compiler and uses information generated by the compiler to do its work. As a result, the lexical scope of identifiers, both those introduced by a macro and those passed as arguments to the macro call, can be maintained by extending the mechanisms which the compiler is already using to manage scope. These extensions are based on the *syntactic closures* of Bawden and Rees [BR88], described in Section 1.3.4.

A macro system without macro expansion may seem a very odd animal. Nevertheless, we continue to use the term "macros" for syntactic extensions which are transformed by our system, since this how they are used in the language-level sense, even though they are not implemented using macro expansion.

There are several advantages to our integrated approach:

- *Simplicity*—we avoid replicating compiler functionality in the macro system.
- *Extensibility*—the system will allow sophisticated users to extend the compiler in novel ways by exploiting compile-time information and data structures to do analyses which would be impossible using a normal macro facility.
- *Grammatical Flexibility*—we believe that our integrated approach to parsing, macro processing, and compilation has advantages for extensible languages with a more conventional surface syntax than Scheme's parenthesized prefix expressions.

In particular, our system makes possible *reflection on scope*, so that a sophisticated user can get access to and modify the implementation of scoping mechanisms without a detailed knowledge of the inner workings of the compiler. More generally, the system allows such a user to write macros which can perform *context-sensitive transformations*. We will describe these concepts and show examples of how they can be used in Section 4.

1.3 Motivation: Shortcomings of Macro Systems

Historically, programmers have had to pay an inordinate amount of attention to the construction and use of macros, for the reason mentioned above—lack of contextual information. Macro expansion involves two substitutions: (1), a call to a macro is replaced by the macro code, and (2), the argument expressions, if any, are substituted into the expansion in some way, usually replacing the formal arguments in the body of the macro. Both of these substitutions constitute an insertion of code defined in one context into another.

In Lisp, arbitrarily complex expressions may be generated and substituted for a macro call. Also, the programmer may define a *rewriting function* to perform any sort of computation from analysis of input forms to transformations that build the expressions to be substituted. But because of the contextual problem, the programmer using the system must be careful about the possibility of *name conflicts*. We give some examples of these in the coming sections.

Lexically-scoped macro systems were developed to ensure that the lexical scope of identifiers in a program is preserved during a macro call; the name capture problem of conventional macro systems is thus avoided. In the following sections we'll first see how name capture can occur in conventional Lisp macro systems. Then we'll look at two modern macro systems which were designed to avoid these problems, a lexically-scoped system based on *hygienic macro expansion* and a system based on *syntactic closures*, and point out the shortcomings of each.

1.3.1 Macro Definitions in Lisp and Scheme

In the sections that follow we will be giving a number of examples of macros written for conventional systems and macros written for our system. We differentiate between these by using *define-macro* to introduce a macro written for a conventional Lisp-like macro system and *define-syntax* for macros written for the new facility.

For clarity, all macros will be written in a simplified language. We use the standard template-filling idiom of Lisp macro systems [Ste84] in our examples of traditional macros; without going into detail, we note that an expression in a macro definition which is preceded by a backquote (‘) is textually substituted in the program text at the point of a call to that macro during a prepass, while expressions with a prepended comma in the same expression are substituted the other way by taking the argument expression from the call and substituting it into the macro body.

As an example, consider the following macro definition[3] which implements logical OR:

```
(define-macro (or a b)
  `(let ((temp ,a))
     (if temp
         temp
         ,b)))
```

The form `(let ((temp ,a)) (if temp temp ,b))` will be textually substituted for any call to the `or` macro in a program, and the argument expressions from the call will be substituted into the macro body for `,a` and `,b`.[4]

[3]Be aware that this is a simplified syntax; we'll deal exclusively with substitutional macros, rather than the more powerful procedural macros of Lisp, throughout. The identifier `define-macro` associates the body of the form with the name `or`. The symbols `a` and `b` are the formal arguments; the actual arguments passed to a macro call will be substituted for occurrences of these symbols in the macro body. For simplicity, assume that when a macro is called, its body is textually substituted into the program at the point of call.

[4]These substitutions are going both directions: the substitution of the macro body for the call

In our Scheme system, local macros are introduced using `let-syntax` and `letrec-syntax`. Both of these forms create a local macro definition that is visible only to the subexpressions of the forms, just as bindings created by `let` are only in scope during evaluation of the sub-forms. The `letrec-syntax` form allows the introduction of mutually recursive macros.[5]

1.3.2 Scoping Problems with Conventional Macros

In a conventional macro system like that found in Lisp, which has no way of maintaining the lexical scope of the identifiers a macro expansion introduces, name capture can occur in the following ways:

- A macro body may introduce local variable names which inadvertently capture names used in the macro's argument expressions, because the arguments are naively substituted into the context of the macro body.
- Free variable references in a macro body, such as to a global or top-level variable name, may be captured inadvertently by a symbol bound in the enclosing program text because the macro body is naively substituted into the context of the call site.

As a simple example of the first problem, consider again the Scheme macro definition which implements logical OR (Scheme already has this operation, but for the sake of the exposition we'll assume it does not). The logical OR operation is easy to write out and understand, and when written in the obvious way can cause name capture to occur:

```
(define-macro (or a b)
   `(let ((temp ,a))
```

expression gives the effect of the call itself; the substitution of the argument expressions into the (copy of) the macro body gives the effect of parameter passing.

[5] Note that while standard Scheme has optional macro-defining forms, we use a simplified syntax for clarity.

```
    (if temp temp ,b)))
```

The `let` construct is used here to associate the value obtained by evaluating the argument `a` with a temporary variable `temp`. This ensures that the argument will only be evaluated once during execution, since the argument expressions are evaluated each time they occur in a macro body (readers unfamiliar with the Scheme language can get an overview in Section 2.1).

Unfortunately, it is possible that a programmer unfamiliar with the inner workings of `or` could use it in the following way:

```
(let ((perm (member x permanent-employees))
      (temp (member x temporary-employees)))
  (if (or perm temp)
      (generate-pay x)
      (begin
         (display x)
         (display " is not an employee"))))
```

A traditional macro system that performs a simple rewrite would expand the macro call in this way:

```
(let ((perm (member x permanent-employees))
      (temp (member x temporary-employees)))
  (if (let ((temp perm))
        (if temp temp temp))    ; error
      (generate-pay x)
      (begin
         (display x)
         (display " is not an employee"))))
```

In the case where the binding of the actual argument `perm` contains the value `#f` and the binding of `temp` does not, the expression gives incorrect results. This happens because each use of the identifier `temp` in the `if` expression refers to the inner definition, but the third occurrence was supposed to refer to the outer `temp`. The name of the temporary variable conflicts with the variable names in existence at the point where the macro was substituted into the program.

As an example of the second problem, consider a macro defined as follows:

```
(define port (open-output-file ``trace.txt''))


(define-macro (emit-trace expr)
   `(begin
       (port-display port (quote ,expr))
       (port-display port ``has value:'')
       (port-display port  ,expr)))
```

This macro implements a trace facility. We want the expression `expr` to be evaluated and have both the expression and its value printed to the file attached to the variable `port`. To do this, we use the standard Lisp template-filling idiom described above. In this case, the `(quote ,expr)` form returns the internal representation of `expr` which `port-display` will convert to a printable representation, and the final `,expr` form returns the result of evaluating `expr`.

In this macro the body form includes a *free* reference to the identifier `port` and relies on the top-level definition given. That is, `port` is not defined in the macro body itself; the meaning of this identifier depends on the context. Name capture occurs when the macro is used in an expression where a different variable `port` has been defined:

```
(let ((port (open-output-file ``warning.log'')))
    ...
   (emit-trace (/ 4 2))
    ...)
```

Here the programmer has naively substituted the macro into a context where the definition of port has changed. The macro expansion therefore yields the following:

```
(let ((port (open-output-file ''warning.log'')))
    ...
   (begin
     (port-display port (quote (/ 4 2)))
     (port-display port ''has value:'')
     (port-display port 2))
    ...)
```

which will fail to act as expected when applied because the identifier `port` now refers to the port that writes to the file `warning.log` rather than to the original file, `trace.log`.

What is needed in these examples is a way to protect identifiers which are passed as arguments to the call from being naively substituted into the inserted code, and to protect free references in the body of the substituted macro code from being captured by names in the enclosing context of the call. Lexically-scoped facilities attempt to provide such protection.

1.3.3 Hygienic Macro Expansion is Limited

The purpose of a *lexically-scoped macro facility* is to ensure that lexical scope of identifiers is maintained properly. A number of systems have been proposed to do this. For example, the Revised^4 Report on the Algorithmic Language Scheme [R4RS] contains an appendix which describes a proposed lexically-scoped macro system for the language. Traditionally, syntactic extension in Scheme, as in Lisp, has been implemented using *macro expansion*, and suffers from the same problems; the system described in the R4RS appendix is intended to fix these problems. The proposed

Scheme macro system is divided into high-level and low-level facilities. The high-level facility allows macros to be written in a convenient pattern-matching language and is based on a technology called *hygienic macro expansion* [KFFD86]. This facility eliminates the name capture problem. The low-level facility (now in some disrepute [Ree93]) gives more power and flexibility, but at the cost of extra work for the programmer; it allows the hygienic facility to be bypassed when necessary.

The goal of systems based on hygienic macro expansion is to maintain the following *hygiene conditions* [CR91]:

- A hygienic macro cannot introduce a binding of a name that captures references to that name beyond those present in the macro definition. This condition prevents problems such as the binding introduced in the `or` example above which captured the reference to `temp`.

- A hygienic macro cannot insert a reference to a name that can be captured by a binding that is in existence at the macro call site. This condition prevents the captured reference to `port` in the `emit-trace` example.

Hygienic macro expansion works to ensure these conditions by automatically renaming identifiers introduced by the macro definition in some way (a "painting" step) and evaluating the new names in a new kind of *lexical environment* (defined in Section 2.1), which maps the new identifier names to the meanings of the original names. For example, in order to guard against inadvertent capture of the symbol `temp` in the above `or` example, a hygienic facility will automatically rename the introduced identifier `temp` so that capture is impossible. It will also rename the syntax forms `let` and `if` to guard against the possibility that these names were redefined somewhere between the definition and the use of the macro [Ree93].

This type of system is essentially an automated version of the standard way in which Lisp programmers have written macros over the years, which is to avoid name capture by explicitly writing or generating "unique" names. Here the renaming of identifiers is performed by the system, freeing the programmer from having to worry about name capture when writing macros. Hygienic macro expansion, then, adds a sort of "identifier painting" step to macro expansion.

Hygienic macro systems help to solve the name capture problem but suffer from the following problems:

- **Unintuitive.** Rather than being an intuitive extension of Scheme's internal mechanisms for handling scope, hygienic systems are an unintuitive internalization of the renaming workarounds (i.e., hacks) which are often used by Lisp macro programmers. One consequence of this is that since identifiers are renamed, the correspondence between source code and internal representations is lost or obscured, with the attendant negative impact on systems that rely on such information, such as debuggers.[6]

  Normally, lexical scope is maintained through environments. Hygienic macro expansion is an *indirect* way of getting environments to work right, by emulating environments by renaming in a specific way.

- **Not integrated with the compiler.** Hygienic systems must pre-parse their input and determine the syntactic role of identifiers, which means they replicate work which is more naturally done by the compiler proper. Being a separate pass which comes *too early in the compilation process*, that of analyzing the surface syntax of a program, macro preprocessors cannot take advantage of information that the compiler generates in subsequent phases, or control the compiler's later analysis and code generation strategies. Therefore, hygienic macroexpanders are really precompilers.

- **Lack of a natural representation of scope.** Encoding scope information by (in effect) "mangling" identifiers distributes that information (in an inscrutable way) throughout an expression. We would prefer a distinct data

[6] Auxiliary mechanisms can maintain a correspondence between source code and transformed code, but this is more difficult than with our system.

structure which encodes an expression's meaning such that it can be transformed in a controlled way. For example, we would like to have a powerful and expressive mechanism for adding features such as module scopes and private scopes internal to objects in object-oriented extensions so that certain variables and methods can be made private to instances of those objects.

We would like to support implementing such features as macros rather than by bolting them into a compiler in an unportable way. In order to perform the necessary kinds of context transformations in systems without an explicit environment structure encoding scope, we would have to essentially "demangle" expressions to recover the environment information implicit in painted identifiers, work our transformation on that environment, and then "remangle" the expressions according to the transformed environment structure.

1.3.4 Syntactic Closures

Syntactic closures [BR88] have also been advanced as a solution to the name capture problem. The underlying idea is that expressions defined in one context which are substituted into some other context can be *closed* in the definition context by packaging the context with the expression in a data structure. We consider this approach to be an elegant and intuitive solution to the name capture problem because it is based on environments and closures, the very building blocks of Scheme. As mentioned above, our system is based on syntactic closures.

Like a "normal" Scheme closure (defined in Section 2.1), a syntactic closure carries with it the context in which it was defined. Intuitively, this *fixes the meaning* of an expression according to the context where it is introduced, so that the meaning does not change whenever a code transformation moves it into a different context.

The data structure which represents a syntactic closure contains the expression that defines the macro body, the list of formal arguments, and a pointer to the *environment* (also defined in Section 2.1) where the macro is defined. Macro expansion is performed by packaging a macro's argument expressions into the syntactic closure (thereby preserving their context) and inserting them into the macro body, creating a syntactic closure for the whole macro expression, then replacing the macro call with the syntactic closure. At evaluation time, each component of the macro body will be evaluated in the proper context.

As originally presented, however, there are limitations on the types of macros that can be expressed with syntactic closures. In particular, macros are often written which introduce a new lexical scope as context in which argument expressions are evaluated (one example of this is the `for` loop, which we will study thoroughly later in the thesis). If the arguments to this kind of macro are closed in syntactic closures, it will not work properly, because arguments which are closed at the call site cannot be evaluated in scopes introduced in a macro body. We will show how the syntactic closure approach can be extended to handle such macros automatically. We will also show that integrating syntactic closures directly in the compiler leads to the ability to write *context-sensitive* macros, which are useful in constructing a framework for externalized compiler analyses and optimizations.

1.4 Motivation: Building an Extensible Compiler

In the macro facility described in this thesis, we have pulled the process of expanding a macro call into the compiler itself, in order to take advantage of the compiler's parsing mechanism, knowledge of binding environments, and information about the scope of each identifier used. One disadvantage of this is that existing compilers must be modified to take advantage of it, rather than simply replacing a prepass or front-end module. On the other hand, the changes required are fairly minor, as the implementation of our system fits in well with usual compilation mechanisms. More importantly, the system described here is intended to be the basis for the development of an extensible compiler based on an advanced meta-object protocol.

Using our macro system, an advanced user will be able to reprogram the compiler by writing macros which allow the system to *reflect* on scope, where *reflection* is defined in [Pae93] as "the ability to modify an implementation of a language without leaving the realm of the language"; by extending reflection to scope issues, we hope to provide information about identifiers and environments and allow programmers to conveniently code analyses which rely on such information, without having to understand the low-level internals of the compiler itself. The compiler can be organized into layers, where the lowest layer understands how to compile the base language very well, and an upper layer (the meta-object layer) provides a front-end for customizing the way the lower layers work. Such customizations can include such things as:

- re-implementation of oft-used language features to improve performance
- experimentation with new language constructs
- implementation of language extensions while maintaining backward compatibility

More generally, a macro system integrated with the compiler will make it easier to use macros which reflect on *context*, where scope effects are just one aspect of context. To show how the macro system described in this thesis might support the development of this type of compiler, a later section will describe language features and compiler optimizations that rely on scope information and that might be implemented using macros defined using our system.

Note that the macro system presented here is largely declarative to keep things simple; in our examples we present patterns to say what a macro matches, and then the template that gives the expressions that the macro transforms to. This makes examples easy to write and easy to follow.

However, the motivation for the mechanisms we use is to support more expressive procedural macros (roughly corresponding to the `syntax-case` form presented by Dybvig in [Dyb92]). With a procedural macro system it is possible to write essentially arbitrary analyses and optimizations to be performed at macro processing time (during compilation).

Our goal is to combine a convenient and intuitive representation of environments (as in syntactic closures) with the expressive power of `syntax-case` and to extend that expressiveness by providing a simple overall framework for the compilation process. An essential feature of such a framework is that macro processing steps must happen at well-defined times during compilation, giving a number of benefits described below.

(*Note to the reader—what follows is a description of macro processing in a compiler and contains terminology that may be unfamiliar.* Understanding the next two paragraphs is not a prerequisite for understanding the thesis. This description is meant to relate the issues of macro expansion in a compiler to more well-known language implementation concepts for those researchers who might be unfamiliar with these issues.)

We choose to process macros in a top-down manner, interleaved with the normal top-down processing of expressions in the front-end of the compiler; during processing, control flows from compiling expressions to compiling their subexpressions and back in a recursive way. This outside-in processing of expressions is structured as a top-down traversal of the abstract syntax tree as it is incrementally constructed from the surface syntax tree (i.e., the input S-expressions constructed by the Scheme reader; see [Wil96] for a gentle introduction). During this top-down analysis and transformation process the compiler keeps a context data structure known as the compile-time environment; this structure encodes the information about the runtime context in which the code being compiled will execute. Procedural macros can annotate this data structure and examine annotations left by earlier steps of macro processing, giving us the power of an attribute grammar system with inherited and synthesized attributes [ASU86]. Adding annotations to this data structure in ways that depend on information found upstream in the compile-time environment corresponds to inherited attributes; examining annotations left during the compilation of subexpressions and performing actions based on this information corresponds to synthesized attributes.

This system has two advantages over traditional attribute grammar-based compiler front ends:

- **The grammar isn't fixed.** We want an extensible context-sensitive grammar so that user-defined constructs can change the grammar used to parse constructs nested within them.

- **It presents a convenient user-interface to the grammar.** The language is extensible *from within the language itself*, in a convenient and portable way. Extending the language does not require modifying the compiler or understanding the particular compiler construction tools used to build it.

This work is being done in the context of a Scheme compiler, but eventually we intend to apply our ideas about the implementation of macros and extensible compilers to systems programming languages which have a more "traditional" syntax. We believe these facilities will work particularly well in the context of such programming languages because the integrated approach makes it easier to handle the inherent difficulties of combining syntactic extensibility with nontrivial syntax. In this approach, we expect that the parsing step will need to be performed incrementally, and that parsing must be integrated with macro processing.

1.5 Related Concepts

There are other techniques that are used to address issues of naming conflicts that may be familiar to some. We mention these to show how renaming issues are treated in other areas of programming language implementation. It is not necessary to understand these mechanisms to understand our facility, because our system solves the problem more directly.

- *α-conversion* [PJ87] is a process of renaming identifiers to ensure that different variables with the same name in different scopes end up having different names in later stages of processing in which scope information is unavailable. For simple α-conversion the only constraint is that each distinct variable in the original code have a distinct corresponding identifier in the resulting code even if originally different variables had the same name in different scopes.

- *Name-mangling* is a way of encoding scope information into the structure of a variable name that should be familiar to C++ programmers. C++ templates are instantiated by a process similar to macro expansion, and name conflicts are avoided by using name-mangling, which is a form of identifier renaming. As an oversimplified example, a reference to a static variable `foo` in method `bar` defined in class `baz` in C++ might result in the construction of the "mangled" name `baz_bar_foo` (with type information probably encoded as well). This name encodes the fact that the variable `foo` is scoped in the `bar` method which is in turn scoped in the `baz` class.

Identifier "painting" used in hygienic macro expansion is similar to name mangling but is extended to allow more sophisticated treatment of scope issues. In particular, the treatment of user-defined binding constructs relies on the structure of identifiers and its correspondence to environment structure. In some cases it is necessary to partially "demangle" an identifier (by un-painting, if you will) and then "remangle" it to, in effect, resolve the identifier in a different environment.

Another discussion of lexically-scoped macro systems that sketches an implementation based on the R4RS Appendix and constrasts syntactic closures with renaming systems can be found in [Ree93]. A system that extends hygienic macro expansion in a way that addresses some of the shortcomings of the R4RS proposal is the Syntactic Abstractions of Dybvig, Hieb, and Bruggeman [DHB92] Each approach performs a symbol renaming step that amounts to identifier painting. Our work pushes a step further in that our use of the environment structures in the compiler obviate the need for symbol renaming.

1.6 An Overview of the Thesis

This thesis is organized as follows: this introduction has described the motivation for designing a new kind of macro facility, such as addressing the shortcomings of existing macro facilities and driving the design of an extensible compiler. The second chapter introduces the background needed to understand the implementation, including a look at how Scheme systems work. The third chapter discusses the ideas behind our system and introduces the design through a series of examples. The fourth chapter describes support for future research into extensible compilers, and the final chapter summarizes the thesis.

Chapter 2

Background

The macro facility described in this thesis was developed as part of an interpreter for a subset of Scheme, based on a facility implemented for the RScheme compiler. This exposition is largely based on the interpreter implementation simply because it is easier to describe. It is useful, however, to understand how the Scheme compiler removes the interpretive overhead so that the facility becomes efficient—the macro processing in our system is trivial to factor out at compile time.

In this chapter we first review basics of the Scheme language, so that our examples will be comprehensible, then explain some of the underlying data structures and mechanisms used in a Scheme interpreter.[1] Next we examine two idioms which use extensions to these basic mechanisms, and show how they can be used to maintain lexical scope in a limited macro system. Then we touch on the notion of *compile-time environments* and how they are used by the compiler to factor out the processing overhead associated with these idioms.

Those familiar with the the Scheme programming language may want to skip the first section.

[1] For a gentle yet thorough introduction to Scheme, including a basic discussion of macros, the interested reader may wish to consult [Wil96]. This document is available in HTML format for easy browsing, making it a good reference for topics we can't cover thoroughly here.

2.1 Building Blocks of Scheme Programs

- **Expressions**:

  Expressions are the basic elements of Scheme programs. Complex expressions are built from *primitive* expressions, for example, numbers, identifiers, and operators [AS85]. Such expressions can be built by enclosing a list of other expressions (which may themselves be primitive or complex) in parentheses, using *prefix notation*. In prefix notation, the function or operator is always first in the list, as in the expression `(+ 1 2)`. So for example, `5` is an expression, as are `foo`, `(/ 10 5)`, and `(square-root (- (* b b) (* 4 a c)))`. The latter two expressions are *combinations*, which are procedure call expressions.

- **Evaluation**

  Combinations may be arbitrarily complex, but there is a simple model for evaluating them [AS85]. The leftmost element of a combination is the *operator* and the rest of the elements are the *operands*. To evaluate a combination, each of the operand expressions must be evaluated. Then the operator expression is evaluated (in the same way as the operand expressions) and the resulting value is applied to the values which result from the evaluation of the operands.

- **Special Forms**

  Some expressions, such as the procedure definition `(define foo (lambda (a b) (+ a b)))`, are not combinations (procedure call expressions), and are not evaluated in the manner described above. In Scheme there are a small number of *special forms* which have their own set of rules for evaluation. These include things like `define`, which associates a name with a value, binding constructs such as `let` (described below) and control constructs such as `if`. The fact that there are few of these forms, together with the simple general evaluation rule for combinations, means that Scheme has a simple syntax that can be readily understood after a short time of study.

- **Environments:**

  An *environment* is the set of all variable bindings which are visible at a given point in a program. Bindings associate names with locations in storage, which contain some value.[2] Note that a value can be any Scheme data object, such as a number, string, or even a procedure. Scheme has a top-level binding environment (TLE) where all variable bindings known at startup are stored. Any bindings defined at the top-level (globally) are also stored in the TLE.[3]

  Local variable bindings are created using one of Scheme's binding constructs, such as `let`. The `let` form has two parts, a set of name-value pairs to be bound, and a set of expressions to be executed. `let` creates a binding frame which holds the name-value pairs and adds it to the current environment; it then evaluates the body expressions in the new environment. The value of the last expression evaluated is returned as the value of the `let`, and the bindings introduced go out of scope.

  Binding constructs may be nested inside other binding constructs, with the frames created by each "added to the chain" of frames. Bindings above the current frame along the chain are visible to expressions executing in the environment; a reference to a variable causes a search in each frame along the chain starting with the most recently constructed, and the value from the first binding of that variable name found is returned. This is how lexical scope is maintained.

  Note that identifiers bound in nested `let` constructs may have the same names as enclosing bindings up the chain; we say that the new bindings *shadow* the

[2]Rather than naming a particular location in storage as in other languages, a `variable` in Scheme is a pointer that can refer to any given location, and any given `value` (which is any Scheme object) may be stored in a location. Types are associated with values, not variables.

[3]Most Scheme implementations, including our own RScheme compiler, allow for multiple top-level environments. Each is completely separate from the others, and can act as a *module*. We shall ignore multiple top-levels in this thesis, but it should be noted that our macro system works with multiple top levels and more general module facilities.

others, so that the new bindings are visible to an expression being evaluated, while the previous bindings are not. Note also that environment chains always correspond to lexical nesting and not to a call stack. In Scheme, procedures are evaluated in the environment where they are defined; when a procedure is called, the current environment will be changed to the procedure's environment of definition.

The interpreter is based on this simple model, explicitly building the structures that hold the bindings. (Note that in a compiler, environment representations can be optimized to do register allocation, stack-based allocation, etc. We ignore such complications because they do not affect the front end of the compiler.)

- **Closures**:

  Scheme procedures are first-class objects; any procedure can be passed to another procedure as an argument, returned as a value, or bound to a variable. Internally a procedure is represented as a *closure*, which is a data structure that holds a pointer to the environment where the procedure was defined and the procedure body expressions (in the interpreter, it also holds the procedure's formal arguments). We say that the free variable references in the procedure body are *closed* in the defining environment; intuitively, a closure *fixes the meaning of its body expressions* at the point where a procedure is created—the procedure "remembers" the bindings to which identifiers refer. This allows us to pass the procedure around and yet ensure that all references will be valid no matter the context in which the procedure is applied.[4]

[4]Environment frames are stored on the garbage-collected heap, and not stack allocated, because of the possibility of being captured by a closure. This implies that each variable binding in an environment is in allocated memory rather than in register, which in turn implies slower access to them. However, compilers can optimize references to variable bindings by determining that a particular environment frame is never captured by a closure; such a frame can be stack allocated, or its bindings may be stored in hardware registers. Closures in Scheme is a feature which is not shared by stack-based, block-structured languages such as Pascal. For such languages, our mechanism will still work fine, with no impact on stack allocation or register allocation. As will become clear

The Scheme special form `lambda` is used to create new procedures. `Lambda` is passed the procedure's arguments and body expressions and returns a closure.

- **Procedure Call**

  An application of a procedure to its arguments (which is sometimes called a *combination*) has the form

  ```
  (procedure arg1 arg2 ...)
  ```

  where there can be zero or more arguments.

  A procedure call performs the following steps:

  - evaluate the expressions that are passed as arguments in the current environment.
  - restore the environment where the procedure was defined.
  - create a new binding environment, scoped inside the procedure's definition environment, which associates the values of the actual arguments with the formal argument names.
  - execute the body of the procedure in that environment.

  Note that the lexical scope of the identifiers in the procedure body is preserved by scoping the binding environment for the argument values inside the environment *taken from the closure* rather than in the call site environment.

2.2 Basic Scheme Interpretation

A Scheme interpreter performs interleaved analysis and evaluation of Scheme expressions. This basically involves examining an expression to determine what it is, and then dispatching to code which knows how to evaluate it.

---

later, assuming that binding environments are heap-allocated simplifies our discussion of macro processing.

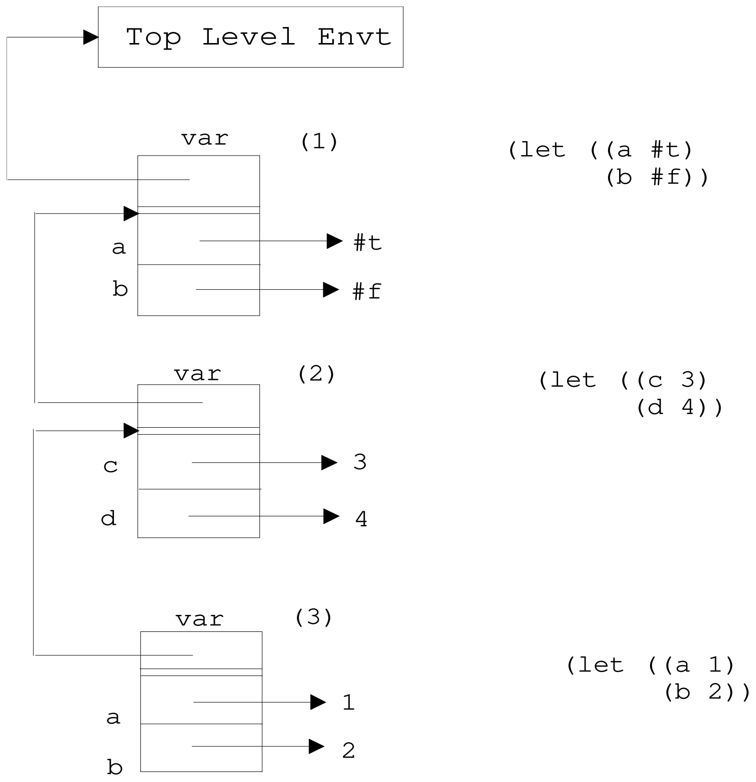

Figure 2.1: Interpretive Runtime Environment

The interpreter starts with a top-level environment and adds new frames to the environment chain as it encounters and evaluates binding constructs. At any given point in the evaluation process, the interpreter maintains a pointer to the *current environment*, which is the frame that the current code expression is evaluated in. The environment chain created by the interpreter in the course of evaluation mirrors the lexical nesting of binding constructs in the program. As mentioned above, this is how lexical scope is maintained.

As an example, consider the following code, which creates three nested binding environments and then evaluates a call to `list`. The interpretive runtime environment structure is illustrated in Figure 2.1.

```
(let ((a #t)                ; (1)
      (b #f))
```

```
(let ((c 3)            ; (2)
      (d 4))
  (let ((a 1)          ; (3), these bindings shadow the above
        (b 2))         ; definitions of a and b
    ...                ; intervening code
    (list a b c d))))  ; (4)
```

The first `let` construct (1) creates a binding frame for the identifiers `a` and `b`. In (2), another binding frame is created, nested inside the first, containing bindings for `c` and `d`. When (2) is evaluated, the symbol `let` is looked up in the binding created by (1) and since no binding for it is stored there, it is found in the top level. Similarly, when (3) is evaluated lookup for `let` is done in both intervening frames before it is found in the top level. The bindings created by the `let` in (3) *shadow* those in the first frame, which means that during the scope of the innermost `let`, references to `a` and `b` will return the values 1 and 2, respectively, rather than `#t` and `#f`.

(Note that binding frames in the Figure are represented in a simplified manner that is easier to draw and understand. In an interpretive environment, we store the slots of name/value pairs in an array or list of "boxes" as shown in Figure 2.2. Each pair of boxes is an object on the heap which holds the name and the associated value as data. Each slot can be referenced independently of the environment frame that contains it.)

Interpretation is slow because the interpreter must analyze each construct it sees to find out how to evaluate it, *each time* it sees it. Evaluation of a simple combination like `(func foo 2)` involves the following steps:

- noticing that it is a complex expression
- looking for the symbol `func` through the environment chain, to find out what sort of object it is

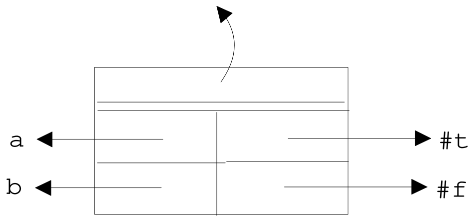

Figure 2.2: Environment Frame Data Structure Detail

- determining that `func` is a variable, not a special form or macro, and therefore that `(func foo 2)` is a procedure call
- evaluating, by recursive calls to `eval`, the symbol `foo` (another symbol lookup) and the number 2 (which evaluates to itself)
- restoring the environment stored in the closure associated with `func`
- creating a new binding environment which binds the argument values to the formal arguments of `func`
- executing the body forms of the procedure

If this combination occurred inside a loop that iterates ten times, the exact same procedure would have to occur ten times.

In addition, the time it takes the interpreter to find a symbol in the environment chain depends on the distance between the current environment and the frame which holds the symbol's binding, and the number of bindings in each frame. Every symbol must be looked up each time it occurs at runtime, which accounts for a great deal of the interpretive overhead.

2.3 Faking Pass-by-Name Procedures

In this section we describe the programming idiom of *thunks* and the details of *syntactic closures*, both of which are applications of normal Scheme closures to language design issues. Closely analogous techniques can be used to build a simple macro system which avoids the name capture problem for a particular class of macros, those that exhibit the behavior of *pass-by-name inline procedures.* Note that this class is just a subset of the possible macros that we wish to express, and don't give the full expressive power of a general macro system.

A *thunk* is a closure which has no arguments; it packages an expression in an environment so that the expression can be evaluated at a later time, but in the original environment, like any other procedure (recall that when a procedure is created using `lambda`, the meaning of the procedure's body expressions are fixed in the context of the environment in which it is created). Thunks are generally used to implement *delayed evaluation*, where an expression is not evaluated until its value is needed. In parameter-passing disciplines, thunks can be used to implement *pass-by-name*, in which the evaluation of an argument to a procedure call is delayed until it is used inside the procedure body [FWH92]. When an expression is packaged in a thunk it is said to be *thunkified.* Thunks can be implemented in Scheme by wrapping the expression to be thunkified in a lambda expression with no arguments. This creates a closure which will capture all the necessary information.

As an example of the use of thunks, we will illustrate a pass-by-name procedure that performs the logical OR operation, called `or-proc`:

```
(define or-proc
        (lambda (a b)
          (let ((temp (a)))
            (if temp
                temp
                (b)))))
```

```
; example of using or-proc
(let ((perm (member x permanent-employees))
      (temp (member x temporary-employees)))
  (if (or-proc (lambda () perm) (lambda () temp))
      (generate-pay x)
      (begin
         (display x)
         (display ``is not an employee''))))
```

Here we've defined `or-proc` as a procedure using `lambda`. Because we've thunkified the arguments passed to `or-proc` in the call, the procedure expects two closures in `a` and `b`, and must therefore make a procedure call to evaluate each argument. This has the effect of evaluating the bodies of the thunks in the environment where they were created.

Notice that as a consequence of delaying evaluation of the arguments, whenever the evaluation of `a` yields a true result, the value of `b` is not required and it is never evaluated. By contrast, in normal procedure call all arguments are evaluated up front; it is in fact this behavior that allows us to pass closures into the procedure. The procedure `or-proc` *does* evaluate its arguments, the lambda forms, which return the closures which are then associated with `a` and `b`.

The structure of the environments created by this definition and use of the procedure is shown in Figure 2.3.

The use of thunks to preserve the environment of call when passing arguments to macro calls was described in [Ste79]. We can use this idiom to avoid name capture in the `or` macro, but we have to be careful how we use it; just passing thunkified arguments as we did to `or-proc` won't work, because the lambda expressions are passed unevaluated to the macro, rather than evaluating to the closures that we really want to pass in.

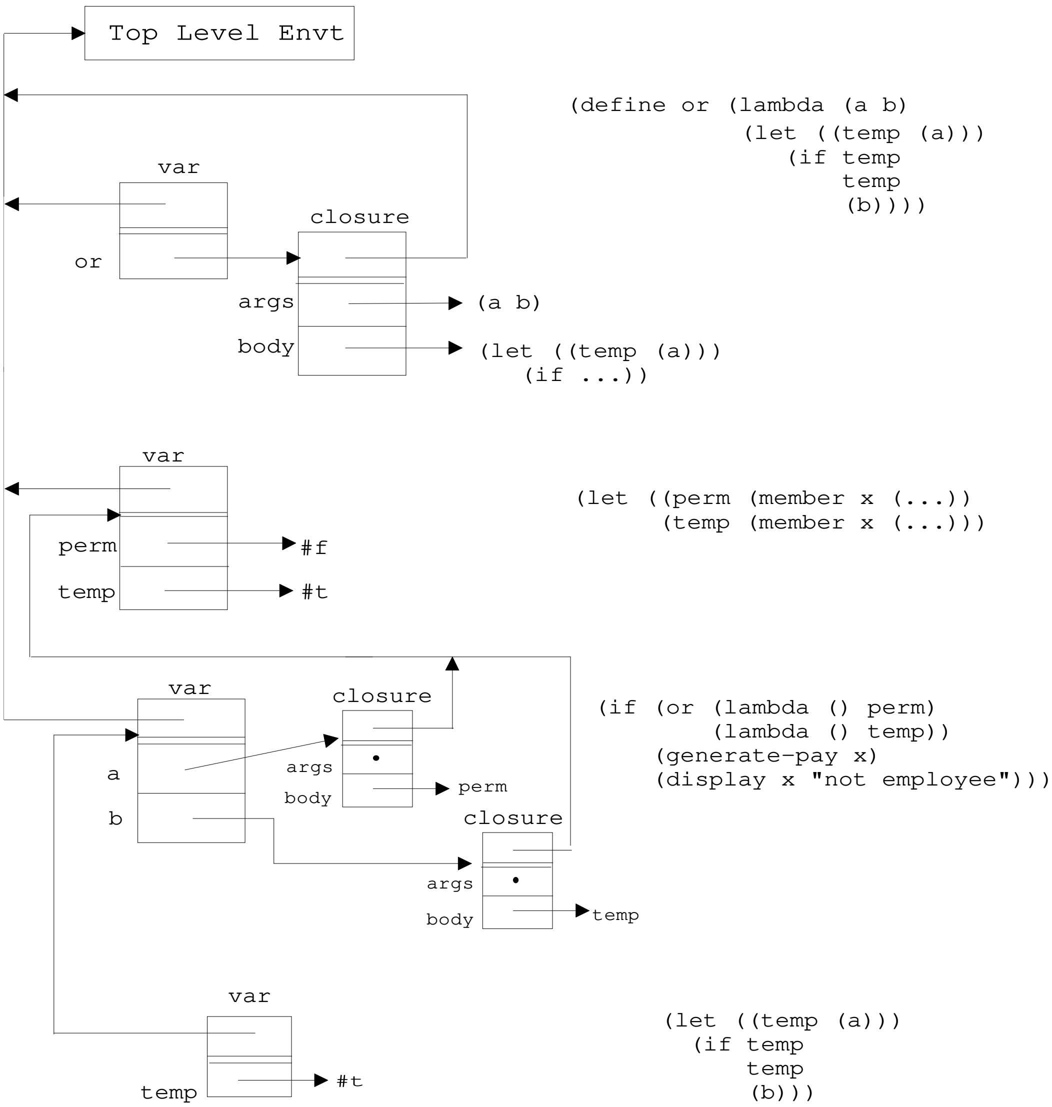

Figure 2.3: `or` Procedure Example

We could rewrite the macro itself to thunkify the appropriate argument just in time. Consider this example, adapted from [BR88]:

```
(define-macro (or a b)
  `(let ((a-thunk (lambda () ,a))
         (b-thunk (lambda () ,b)))
     (let ((temp (a-thunk)))
       (if temp
           temp
           (b-thunk)))))
```

Using the substitution model, the effect of using this macro in place of the procedure `or-proc` in the above code gives us

```
(let ((perm (member x permanent-employees))
      (temp (member x temporary-employees)))
  (if (let ((a-thunk (lambda () perm))
            (b-thunk (lambda () temp)))
        (let ((temp (a-thunk)))
          (if temp
              temp
              (b-thunk))))
      (generate-pay x)
      (begin
        (display x)
        (display ``is not an employee''))))
```

Now there is no chance of name capture, because the thunk has ensured that no matter what expression is passed in as argument `b`, it will return the value it has in the environment of the call.[5]

[5]This works because the `let` form evaluates the value expressions *before* creating an environment

This is obviously not the nicest way to get traditional macros to work; the notation in the above example is rather clumsy, and it would be tedious to analyze every argument in every macro and enclose those that introduced identifiers in a `lambda` form. Another disadvantage to using thunks in this way is the performance penalty; each argument thunk must be evaluated each time it is used in the macro body. A better mechanism would automate the use of thunks and get rid of the runtime overhead of creating and evaluating them. We now look at a way to solve both of these problems.

In Section 1.3.4 we briefly introduced the idea of *syntactic closures*. A syntactic closure is an extension of the notion of closure to a macro definition. Like a closure, it can be called, but syntactic closures are not first-class objects, and therefore are not built at runtime and cannot be stored in data structures or returned as the value of a procedure.

Syntactic closures treat arguments differently than regular closures; in fact, they essentially thunkify the argument expressions *automatically* and force evaluation of the thunks where they are used, giving us the same pass-by-name semantics as the explicit-thunking method. The meaning of the argument expressions are fixed at the point of call and lexical scope is preserved. Because this process happens automatically, the burden of determining which expressions must be thunkified and doing it by hand is off the programmer entirely.

Equally important, all of the scope resolution work necessary can be done at compile time, so that no runtime overhead is incurred. In our integrated macro system, we know exactly what code the closure will execute because syntactic closures are second class; therefore, we can compile the code for the macro at the call site as if it had occurred there. Also, we have enough scope information to analyze the identifiers it uses and pre-compute how they will be resolved in the runtime environment. We discuss how this works in more detail in Section 2.4.

---

frame. In this case, the `lambda` form will be evaluated in the environment of call, and the closure will contain a pointer to that environment.

Some languages allow procedures to be declared as *inline* for the sake of efficiency; each inlined procedure call is replaced by the procedure body, eliminating the procedure call overhead. In the usual substitution model of macro systems, the macro expansion step (which precedes or is interleaved with compilation) replaces a macro call with the body expressions from the macro definition, which has the effect of inlining the macro body at the call site. The integrated system, which compiles macro code stored in a syntactic closure at a call site, also has the effect of inline procedures. Unlike normal procedures, however, macro argument expressions are passed unevaluated.

Therefore, syntactic closures can give us exactly what we want for some purposes—a way to preserve lexical scope for the class of macros (such as `or`) that have the effect of pass-by-name inline procedures.

2.4 Compiling Scheme Programs

In Section 2.2 we saw that the main sources of inefficiency in an interpreter are the the analysis of each form (to see what to do with it) and identifier lookups to find variable bindings. In this section we'll see how a compiler implementation eliminates much of this overhead.

2.4.1 Expression Analysis at Compile Time

We saw above that an interpreter must analyze each Scheme expression input to it in order to determine how to evaluate it. At runtime, the interpreter must make the following determinations:

- What kind of expression is being evaluated?
- What kind of symbol, if any, is in the operator position of the expression?
- Which routine must be dispatched to evaluate the expression?

The compiler must make the same determinations, but can do so at compile-time, factoring out the interpretive overhead. The compile-time actions are as follows:

- Determine the kind of expression, simple or compound, based on its structure.

- If the expression is simple (a literal or a variable reference), then compile code for that simple operation.

- Otherwise, if it's a compound expression, analyze it further by analyzing the expression in the operator position to determine if it is a special-form or function application. Then go on to either process the special form in the first case or compile the function application in the second case.

The compiler avoids having to analyze expressions multiple times, by recording the actions that the interpreter would take to evaluate expressions. For example, during compilation of a procedure body, the compiler can do the expression analysis just once, recording the actions that the interpreter would take as intermediate code that specifies the runtime actions. The procedure body is only seen once by the compiler; at calls to the procedure, the compiler just generates code for the calling sequence that will invoke it at runtime.

The recursive structure of the compiler mirrors the recursive structure of the interpreter, but differs in two ways:

- at call sites, the compiler generates code for the call, not for the execution of the called procedure; only the code to make the call at runtime is generated.

- at conditionals, each branch that might be taken at runtime is traversed by the compiler, which generates a code fragment for each branch and puts the fragments together with conditional code which will take the appropriate branch at runtime.

2.4.2 Compile Time Environments

Another way that the compiler eliminates overhead is by mirroring at compile time the structure of the environment chain which the interpreter builds as it recursively evaluates nested expressions. Recall that the interpreter's environment chain maintains lexical scope by representing the static nesting of binding constructs in a program. In the same way, the compiler maintains this information using a structure called the *compile-time environment* (CTE).

The compiler creates environment frames similar to those created by the interpreter, except that only slots for each variable name in the frame are included. With each variable binding construct compiled, the compiler extends the compile-time environment with a frame that represents the runtime environment frame that will be created at that point in the code. This structure is used to calculate the location of each variable relative to the current environment at compile-time; at runtime, bindings in the environment structure that is created as the code executes will be referenced by location in the runtime chain rather than by name; in this way, the overhead of looking up symbols is removed. The slots in the runtime chain contain only values. It is as if the interpreter's environment is cut in half down the middle, with one half used at compile-time and the other half used at runtime. It's worth noting that there is a one-to-many relationship between compile-time environments and run time environments, as the bindings in the run time environment can change during execution (for example, in a loop).

The CTE is used for symbol lookup during compilation. The compiler takes advantage of the fact that the runtime environment chain will always have the same structure at a given point in the code, and pre-computes the location of a local variable relative to the current environment. It can do this because the compile-time environment abstracts over the runtime environments—a particular environment may hold different bindings each time through a particular point in the code, but its structure is always the same. When a symbol is found during a compile-time lookup, the system knows how many frames must be skipped relative to the current frame (the *lexical depth*), and at what offset within the frame the symbol resides in (the *slot offset*); this information is known as the symbol's *lexical address*. Based on this, the system can generate code to quickly reference the proper location in the runtime environment, using indirect addressing.[6]

Here is a simple example of how the CTE is used to find lexical addresses; a description of this code was given in Section 2.2:

```
(let ((a #t)                ; (1)
      (b #f))
  (let ((c 3)               ; (2)
        (d 4))
    (let ((a 1)             ; (3), these bindings shadow the above
          (b 2))            ; definitions of a and b
      ...                   ; intervening code
      (list a b c d))))     ; (4)
```

The interpretive runtime environment structure was previously illustrated in Figure 2.1; compare with Figure 2.4, which shows how the information contained in the interpretive environment is split between the CTE and the runtime environment structure.

When compiling the form `(list a b c d)` in (4), the compiler will look up the symbols in the CTE and compute the lexical addresses. Ignoring `list` for the moment (which is defined in the top-level environment), we find `a` and `b` in the current environment, so the lexical addresses are (0:0) for `a` and (0:1) for `b`. Identifiers `c` and `d` are one environment frame up, so their lexical addresses are (1:0) and (1:1), respectively.

[6]For simplicity, we ignore register allocation here, along with more sophisticated compile-time choices about runtime environment structure. These do not affect the basic scheme outlined here because representation selection can be done at a later stage in compilation than those we are focusing on—parsing, tranformation, and intermediate code generation.

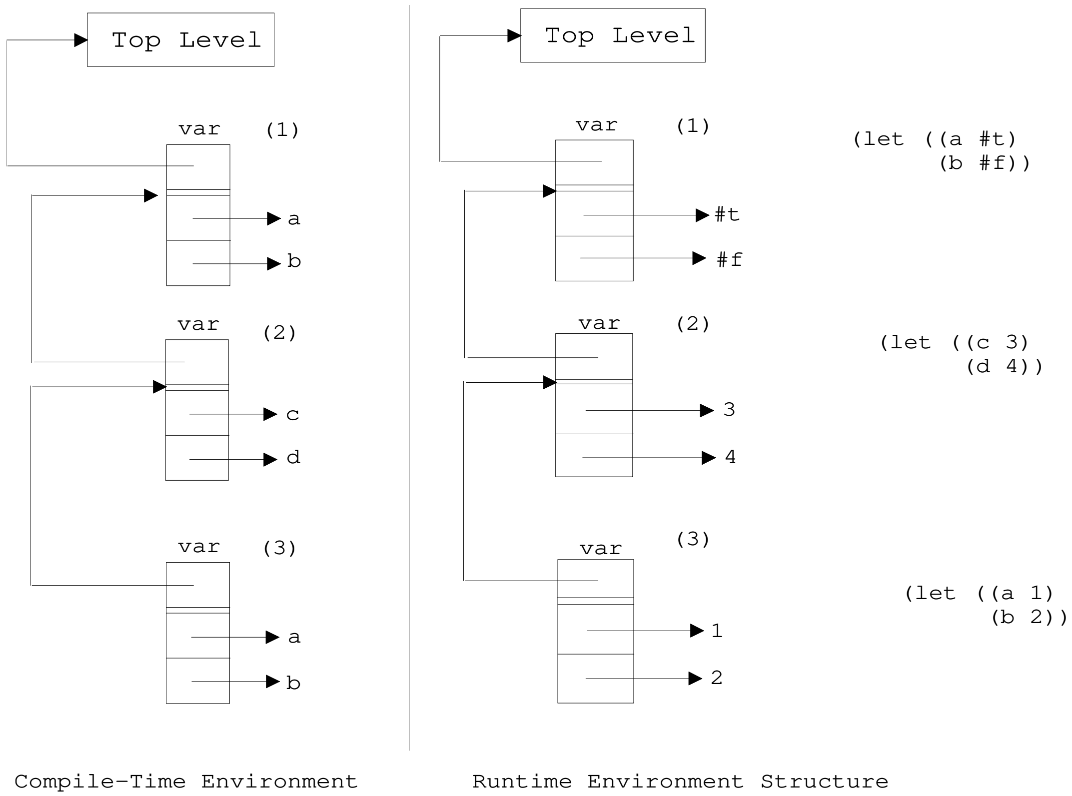

Figure 2.4: Compile Time Environment Example

Chapter 3

Design and Implementation

In this chapter we describe the design and implementation of our macro facility. To a first approximation, this system is based on the use of thunks to preserve lexical scope. Thunks can be created and evaluated at compile-time, so there is no runtime overhead involved in the representation of macros and argument expressions. We then need to extend the system to cover other classes of macros beyond pass-by-name inline procedures, such as control constructs which introduce bindings at the call site, and macros which require explicit name capture.

Our system has been implemented in version 0.6 of the RScheme compiler. For more information on RScheme, see the RScheme home page on the World Wide Web.[1]

We will first discuss the attributes of each class in detail, with examples. We then describe the mechanisms that handle each class in turn, with detailed figures showing the internal constructs that the system builds to evaluate the example macros. We then summarize the design.

[1] `http://www.rosette.com/~donovan/rs/rscheme.html`

3.1 Three Classes of Macros

The design of this system was driven by considering three classes of macros. The first class is made up of generally simple constructs which give the effect of an inlined procedure call. The second class contains control constructs which actually use controlled name capture to introduce new, lexically scoped binding constructs; we call this use of capture *advertent name capture*. Members of the last class work by explicitly capturing identifiers that are defined in the context where the macro is called. We give the name *explicit intentional capture* to macros which require this behavior. In this section we briefly describe each class.

3.1.1 Pass-by-name Inline Procedures

When we use macros such as the `or` macro defined in Section 1.3.2, we want the effect of a *pass-by-name, inlined procedure call* as described in Section 2.3. These are pass-by-name because the arguments are not evaluated before evaluating the procedure body, as they are in a normal procedure call, but only when used inside the procedure body; they are inlined because the procedure body is compiled into the program at the point of call rather than being stored in a closure.[2]

In a simple textual substitution model, this is exactly how most macros are evaluated. Most macro arguments are intended to be pass-by-name, and the textual substitution of a macro body for a macro call gives the same effect as inlining. Therefore, when the `or` macro described above is substituted into the example code we used for the `or` procedure, the *effect* is the same, except that name capture is still possible. We showed that one way to ensure lexical scope is to thunkify argument expressions and close a macro body in a syntactic closure. We also described how the overhead of building these structures and calling the thunks can be compiled out.

[2]Note that when we are specifically speaking of an interpreter implementation, there is no notion of inlining, which is a compile-time operation.

Our system handles this class by compiling macros at the point of call, using a second-class analogue of thunks to automatically close argument expressions in the call site environment, thereby preserving the semantics of both pass-by-name and inlining. This is the default mechanism for compiling any macro; if a macro is compiled that deviates from the default semantics, the system includes mechanisms for recognizing this and dealing with it properly.

We describe the mechanisms for handling this class of macros in detail in Section 3.2.

3.1.2 Inferring Advertent Capture

Some macros are designed intentionally to capture symbols that are passed in as arguments; in fact, many binding constructs implemented as macros rely on name capture to operate correctly. A good example of a macro which does this is the `for` loop. While Scheme has an iteration construct, it can be difficult to understand and use. Extending the language to handle a simple iteration construct (similar to Lisp's `do-times`) should be straightforward.

First we'll show that `for` is not a strict pass-by-name construct. We begin by studying the familiar Pascal `for` syntax.

```
for i := 1 to 10 do
    write(i);
```

The semantics of the for loop is to perform the following steps:

- evaluate the initial and limit value expressions (in this case 1 and 10, respectively, which being literals, trivially evaluate to themselves),
- bind the variable `i`, giving it the initial value 1,
- evaluate the body of the loop in the new environment that binds `i`, in this case the body is just `write(i)`

- increment the value of `i` to 2,
- repeat the above two steps until the final value of the variable `i` is 10.

We'd like to be able to provide a similar construct in Scheme, as shown:

```
(for (i 1 10)
    (write i))
```

Assuming a pattern-matching facility like that introduced in the Appendix to [R4RS], we can imagine that the following "flattened" version of the above is passed to the macro system:

```
(for i 1 10 (write i))
```

For purposes of explanation we will continue to use a sort of simplified Scheme syntax when defining macros, and ignore pattern-matching, because pattern-matching is essentially independent of the rest of macro processing.[3]

In starting to write a `for` macro for Scheme, we first notice that `for` is a *binding construct*. It is easy to see this in the Pascal example, where the loop variable `i` is bound to an initial value. The construct creates a new lexically-scoped binding which is visible to the body of the loop. Our macro must be able to bind the initial value to the loop variable and increment that value; this implies that the macro programmer does *not* want the loop variable closed in the call site by automatically thunkifying it, which would fix the value to be whatever binding is in scope in the environment of call. Further, the loop body refers to the loop variable, which implies that this expression must be evaluated inside the environment where the loop variable is bound, not in the call site environment.

[3]The RScheme compiler supports most of the [R4RS] Appendix pattern-matching syntax, while using the integrated macro processing system rather than hygienic macro expansion.

Here is a conventional macro which implements these semantics in terms of a hypothetical repeat-until construct[4] (which we use for ease of understanding), followed by an example of use:

```
(define-macro (for loopvar init limit body)
   (let ((loopvar init))
     (repeat-until (begin body
                          (set! loopvar (+ loopvar 1)))
                   (= loopvar limit))))
; example of use
(let ((sum 0))
  (for i 1 10 (set! sum (+ i sum))))
```

In the macro body, we bind `loopvar` to the initial value `init` and begin the repeat-until loop. The loop repeatedly evaluates the argument `body`, increments the value of `loopvar` by 1, and checks to see if the termination condition holds, which it will when `loopvar` and `limit` are equal.[5] We see that `loopvar` serves as the loop counter, but we often want to use the counter inside whatever expression is passed for `body`, so `loopvar` is a formal argument.

Notice that the argument substituted for `body` comes from the call site environment, and refers to `i` which is *not in scope* at that point.[6] In this example, that would imply that the use of `i` in `(set! sum (+ i sum))` would have to be in scope before macro expansion.

In our system, the actual argument `i` is interpreted not as a reference to an existing variable (in order to use its value), but as a name which the macro itself will

[4] Scheme does not actually have a repeat-until construct, but it does have a much more powerful iteration construct called "named `let`". We have chosen to implement `for` in terms of `repeat-until` because named `let` might be confusing to readers new to Scheme.

[5] This is not quite how a `for` loop would typically be defined in Scheme. Rather than side-effecting a single binding, it would rebind the loop variable at each iteration.

[6] Actually, there might be a binding for `i` somewhere along the environment chain, but the macro doesn't use it. The new binding introduced by the macro shadows the previous binding, just we've seen before in nested LET forms.

bind. We call such an identifier an *alias* because (in this example) it is just another name for the loop counter. We call this use of an alias *advertent capture* because we intend to capture the identifier `i` for the purpose of creating a binding for it inside the macro. The mechanism to handle this case must be invoked whenever both of the following two conditions hold:

- one (or more) of a macro's formal arguments are symbols that are bound in the macro body expressions (as `loopvar` is).
- another of the macro's arguments is substituted into the scope of the new binding(s) so that it can see and use the new binding(s) (as `body` does).

Our macro system is able to recognize these two conditions to infer that a macro is advertently capturing the aliasing symbol (e.g., the call site name `i` refers to the loop variable `loopvar`, bound during the execution of the macro body). This frees the macro writer from having to give the system explicit directions about how to handle this case. We describe how our system treats actual argument expressions so that `for` will work properly in Section 3.3.

3.1.3 Explicit Intentional Capture

In this section we give an example of a macro which must *explicitly capture* an identifier defined in the context of the macro call in order to work properly. In this example, we describe a macro that extends the language to provide support for exception handling via the *catch-throw* mechanism.

Some Scheme systems provide the special forms `catch` and `throw` to do controlled non-local exits from a block of code when an error condition is detected. A very simple example of how the *catch-throw* mechanism might work follows: when a programmer identifies a block of code that might require exception handling, the code is wrapped in a `catch` construct (which is a special form). If the code block executes normally, `catch` simply passes on the result of the execution; if an exception is detected during execution of the code, execution of that block is aborted and `catch` returns a value such as an error code. The error code is supplied by the `throw` construct, which the programmer adds at the point in the code where an exception condition is checked. What actually happens on detection of an exception is that the `throw` performs a non-local exit (with return value) from the currently executing block of code so that no further processing of the code is done. The `catch` expression provides the context into which the exit is made.

Consider the following example:

```
(catch
  (if (equal? foo 0)
       (throw 'infinity)
       (/ bar foo)))
```

Here we have wrapped a block of code which performs a division in a special form which will *catch* divide-by-zero error conditions. Execution of this form causes the argument expression (the wrapped up code) to be evaluated. If `foo` is non-zero, the division is performed and `catch` returns with the result. Otherwise, the expression `(throw 'infinity)` is executed. This expression performs a non-local exit, *throwing* execution back to the `catch`, which then returns the value `infinity`. `throw` is called the *escape procedure*.

The difficulty in writing `catch` as a macro is correctly scoping the symbol `throw`. As in the `for` macro, which introduces a loop iteration variable, the body expressions must be able to "see" the binding of `throw` as if it were defined at the call site. On the other hand, the identifier isn't passed to the `catch` macro at the call site, so we can't implement this using advertent capture. To do this without explicitly naming the escape procedure at the call site requires a slightly different mechanism, which we'll motivate here.

Here is a possible definition of `catch`:

```
(define-syntax (catch body)
   (call/cc (lambda (throw)
              body)))
```

The Scheme function `call/cc` is a natural choice for implementing `catch` because it captures the current state of the control stack before executing its argument (which must be a `lambda` expression). It also *implicitly* creates an escape procedure for jumping back to the captured point in the stack and gives it the name supplied as the argument to the `lambda` expression. So the effect of calling the `catch` macro is to capture the control stack state, then execute the argument expression `body` either to completion or until the escape procedure `throw` is called. In the latter case, control is returned to the captured state.

The function `call/cc` can be understood as being somewhat analogous to the C library function `setjmp`. A call to `setjmp` captures the current point in the C call stack, so that a succeeding reference to `longjmp` can perform a non-local exit at some point in the computation by skipping back over intervening activation records to the point saved by the `setjmp` call. The function `call/cc` is more powerful, however, because it makes a *virtual copy* of the call stack so that a return can be done through it any number of times (rather than just once), either non-locally or normally.[7] However, `call/cc` is difficult to understand because of its arcane syntax and because so much of its workings are implicit, which is a particularly good reason for wrapping it in a macro.

Here is the desired translation of the example use of `catch` given above when using `call/cc` to define the macro:

[7] `call/cc` captures the control state of the program, and the environment structure, but does *not* snapshot the states of variable bindings.

```
(call/cc (lambda (throw)
            (if (equal? foo 0)
                (throw 'infinity)
                (/ bar foo))))
```

The important point for our purposes is that `call/cc` creates the escape procedure and gives it the name `throw`. We want to allow the macro user to embed `throw` in the expression to be executed by `catch`, so what we want is a way to treat `throw` similarly to the loop variable in `for`—we want to introduce a binding that will be visible to the `body` argument. This is different from `for`, however, in that we have a literal name that was *not* given as an argument in the macro definition or call. This implies that we cannot infer capture—we need a mechanism that allows the macro writer to explicitly tell the system that we want to capture the name.

3.1.4 Compiling Inline Procedures Directly

In this section, we explain how normal, pass-by-reference procedures can be compiled inline in our system. This forms the basis for understanding how our system handles macro processing in the compiler. Eventually we'll combine these ideas with the descriptions of how each kind of macro we've discussed is handled in the interpreter.

In many implementations, procedure inlining is performed as a prepass, using a process similar to macro-expansion with renaming to avoid scope problems.[8] In our system, inline procedures are compiled *directly*, rather than via a preprocess step, and the normal scope resolution mechanisms are extended to handle the scoping issues introduced.

We saw how the compile-time environment is used to factor out scope work at compile-time in Section 2.4.2. We also would like to use the CTE to factor out procedure call overhead where possible so that procedures can be inlined. To see

[8]This process is similar to hygienic macro-expansion, but can be simpler because simple lexical scope is all that is needed, not advertent or explicit capture.

what is required to do this, we'll first talk about compiling normal procedure calls; then we'll discuss how our concept of the CTE needs to change to properly handle inlining.

Compiling procedures normally versus inlining. Recall that when a procedure call is evaluated in an interpreter, the environment where the procedure was defined is restored and used during evaluation of the procedure's body forms, so that symbol lookup is done relative to the definition rather than the call site of the procedure. During compilation, this same environment restoration is done when compiling a procedure body, so that the proper portion of the CTE is used for symbol lookup. Computation of lexical addresses of the symbols used in the procedure body is also done relative to the environment of definition. At runtime this environment will be loaded and used to access the proper values.

When inlining a procedure body at a point of call, the environment of definition of the procedure is *not* restored, in that the overhead which inlining avoids is loading and using the environment at runtime. Therefore we need to be able to access the symbols used by the procedure without needing runtime mechanisms to restore the definition environment, as a normal procedure call does.

The fact that the procedure is being called implies that its definition can be seen up the environment chain, which in turn implies that each symbol used in the procedure body is defined somewhere along the chain above that definition. Furthermore, there are a fixed number of environment frames between the call site and the definition site. The compiler can take advantage of these facts to eliminate the runtime overhead of making the call. To do this we need two environment pointers, one which reflects the runtime structure of the environment where the procedure is called (the *use* chain), and another which says where the procedure body is scoped, thereby telling the compiler where to start searching for symbols encountered in the procedure body; we call this the *lexical environment.*

The Compile-Time Environment. We use the information stored in the compile-time environment chain in our implementation of inlining, but notice that we must change our usual interpretation of this structure. In Section 2.4 we showed that essentially, the compile-time environment serves two purposes:

- it is a representation of the nesting of runtime environment frames, and
- it is a representation of the actual scope derived from that nesting.

These are the traditional roles of the CTE in a normal Scheme compiler. However, the abstraction reflected by the CTE is broken by inline procedures, because while the procedure body is compiled at the call site, it is *scoped* in the environment where it was defined. While the traditional CTE still reflects the nesting of environments at runtime, it no longer adequately represents the scope chain, since we want to change the visible scope when a procedure is inlined. Therefore we need to change the traditional view of the CTE by separating its two roles in this way: at any given point in the evaluation of a program, the CTE is a chain of every frame that is in the runtime context at that point. In other words, it is a superset of the scoping structure but with one piece missing, namely, where to start searches when compiling a procedure inline; this information is provided by the *lexical environment*.

In terms of scope, the lexical environment is the point in the chain where we *logically* compile the procedure body. It is only used for searching, not for computing lexical addresses. Given that the CTE represents a superset of the actual lexical scope, we can see this environment as a way of restricting lookups to things that are *currently* in scope.

In order to make this separation explicit, we call the structure embodied by the CTE the *use chain* (which we described above) and that portion of the chain which is in scope at any given time as marked by the lexical environment the *scope chain*.

Mechanism. Given this view of the compile-time environment structure, the system can compile inline procedures as follows:

- Lookups. Symbol lookups begin at the lexical environment. In effect, we skip past all the intervening frames between the environment of call and the point where the procedure is defined. At this point, lexical lookup proceeds normally.

- Computing Lexical Addresses. Once a symbol has been found, the lexical address must be computed. In this case, the mechanism must take into account the frames that were "skipped" by using the lexical environment as the starting point for lookups, since they will be part of the runtime environment. Therefore, lexical addresses are computed from the point of call.

Here is an example of how the CTE is used to compute lexical addresses during compilation of inline procedures, illustrated in Figure 3.1:

```
(let ((a 1)                  ; (1)
      (b 2))
  (let ((c 3)                ; (2)
        (d 4))
    (let ((foo (lambda ()    ; (3)
                 (+ a b))))
       ...                   ; intervening code
       (let ((a #t)          ; (4), these bindings shadow the above
             (b #f))         ; definitions of a and b
         (cons a b)          ; (5)
         (foo)))))           ; (6)
```

At the point where `foo` is called (6), any variable used in the procedure body is either in scope or shadowed by intervening bindings; in any case, it is somewhere

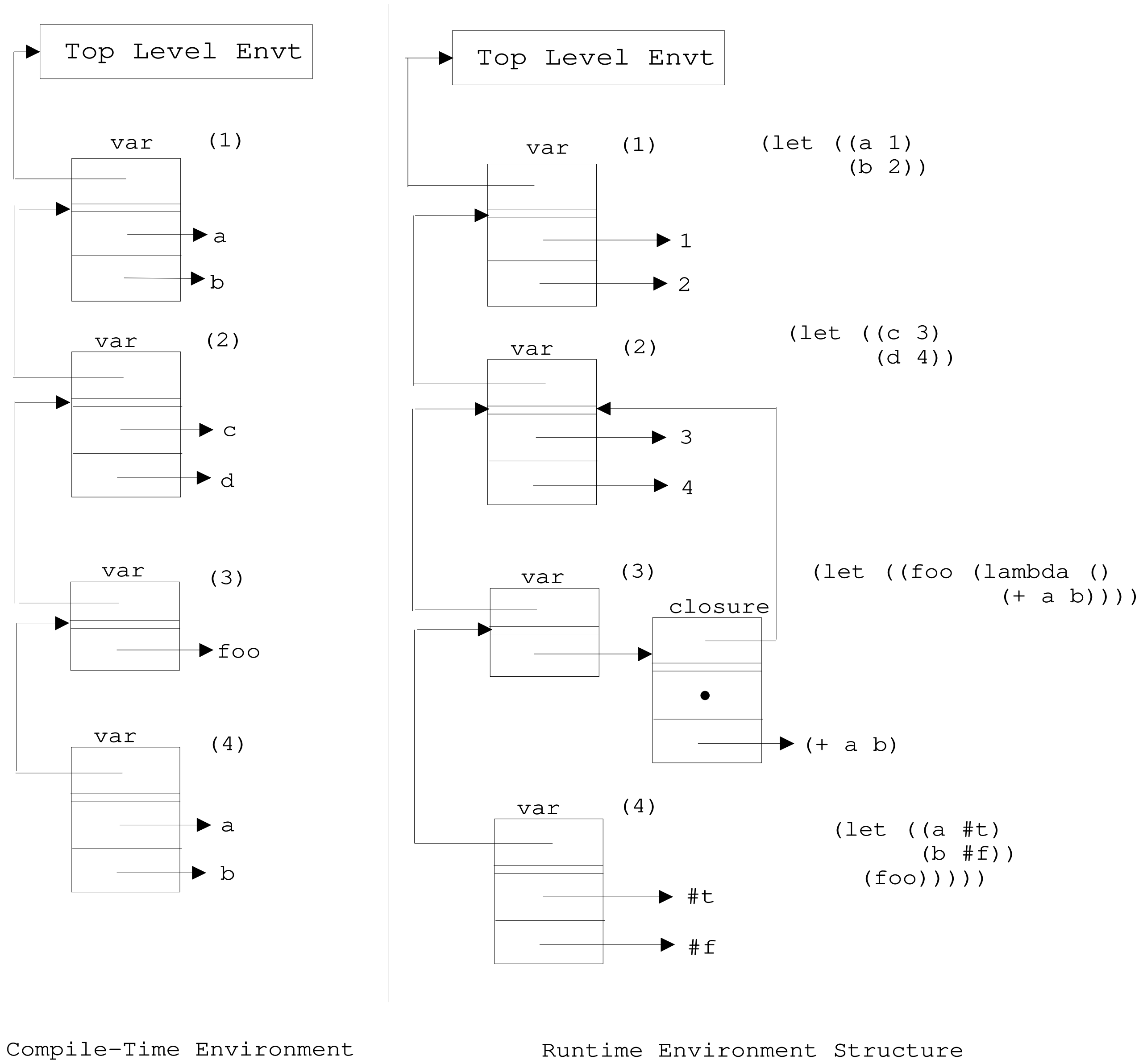

Figure 3.1: Inline Procedure CTE Example

up the runtime environment chain. The fact that we can find `foo` implies that we can find the correct bindings and generate lexical addresses for each symbol in the procedure body.

Because `foo` is itself up the chain at the contour created by the `let` in (3), the environments that it references are also up the chain *from that point*, at the contours created by the `let` forms in (1) and (2). At the call site, the lexical address of `foo` is (1;0), which means that the symbol `foo` can be found by moving back one environment frame relative to the current frame (the one created by the `let` in (4)) and looking in the 0th slot of the frame. We want `foo` to be inlined, so the lexical environment pointer is set to the frame created by the `let` at (3); all lookups for the variables referenced by `foo` start from there.

The lexical addresses of `a` and `b` used in `foo` are computed relative to the current environment (not the definition environment of `foo`), and are (3;0) and (3;1), respectively. If the call to `foo` had not been inlined, the lexical addresses would have been computed relative to the definition environment, and been (1:0) and (1:1).

(In contrast, consider the addresses computed when compiling form (5), where the current environment frame was created by the `let` in (4); here the lexical addresses of `a` and `b` are (0;0) and (0;1) respectively.)

In our integrated macro facility, which is based on this mechanism for compiling inline procedures directly, we can think of a macro definition as a compile-time analogue to a lambda expression. Just as `lambda` packages up code and an environment to make a closure, a macro definition packages up the macro body and environment of definition in a syntactic closure. What's different is that this work occurs at compile-time as opposed to runtime, as is the work done to compile the macro inline at the call site.

Just as in the case of inline procedures, we know that every symbol used in a macro body can be found up the environment chain above the macro definition due to lexical scope, since if the macro is in scope, it must be somewhere up the chain, and bindings it can see and use are somewhere up the chain from there. The use chain and lexical environment introduced above can therefore be used to factor out lookups at compile-time; when the macro's body expressions from the syntactic closure are compiled at the point of call, lexical addressing is used to record the locations of the symbols that are used by the macro relative to the current environment frame in the use chain.

3.2 Handling Pass-by-name Procedures

In this section we describe the mechanisms used in our system to implement macros which have the effect of pass-by-name procedures. We begin by illustrating how the interpreter evaluates a call to the `or` macro (presented in Section 1.3.2), showing the details of how pass-by-name is implemented in a step-by-step fashion. Then we explain how the effect of inlining is achieved efficiently by compilation.

3.2.1 Pass-by-name in the Interpreter

The first, simplified version of our system that we describe handles only macros which have the semantics of pass-by-name procedures, and does not inline them. In this simple interpretive model, syntactic closures and thunks are essentially identical to normal closures. Like closures, syntactic closures are created at runtime when a macro is defined. When a macro is called, the environment pointer is reset to the definition environment stored in that macro's syntactic closure, just as in a procedure call; the same occurs when a macro argument is evaluated and the syntactic thunk which closed it is called.

The following mechanisms are used to implement pass-by-name:

- Each macro definition returns a syntactic closure which packages the body expressions with a pointer to the environment where defined, preserving the context of the definition.

- Each argument expression in a macro call is implicitly wrapped in a syntactic thunk which packages the expression with a pointer to the call site environment, preserving the context of the call site.

- The macro body is interpreted where it is called, but in the scope of its definition as preserved in the syntactic closure.

- Argument expressions seen during interpretation of the macro body are evaluated in the call site environment as preserved in the syntactic thunks.

As we saw in Section 3.1.1, this mechanism ensures that evaluation of both the macro body and argument expressions occur in the appropriate environments, eliminating the possibility of name capture.

Macro definition in the interpreter. We now show an example of a macro defined using our system, in which the special form `let-syntax` is used to define the `or` macro locally. The special form `slambda` serves to introduce the macro and returns a syntactic closure which is bound to the identifier `or`. This binding is only in scope during evaluation of the let expression. We illustrate the data structures constructed during the evaluation of the example in Figure 3.2. Note that neither of our simplified versions of `let-syntax` or `slambda` are standard Scheme forms.

```
(let-syntax ((or (slambda (exp1 exp2)
                   (let ((temp exp1))
                     (if temp temp exp2)))))
  (let ((perm (member x permanent-employees))
        (temp (member x temporary-employees)))
    (if (or perm temp)
        (generate-pay x)
        (begin
           (display x)
```

```
(display ''is not an employee'')))))
```

The special form `let-syntax` defines a new scope (a *syntax* binding frame) in which the macro is visible, and executes body expressions in that scope. The `slambda` special form takes as arguments the macro argument list and macro body forms, and returns a syntactic closure which packages them along with a pointer to the environment where the macro is defined. The name `or` is associated with this syntactic closure in the `syntax` binding frame and is scoped inside the top-level environment.

Evaluation of the macro call form. The interpreter begins evaluation by first decoding the expression, noticing that the first identifier, `let-syntax`, is defined as a special form in the top-level environment. Evaluating this special form creates the `syntax` environment frame described above and binds the identifier `or`, as illustrated in the Figure. For the duration of this scope, `or` will be associated with the syntactic closure which represents the macro.

The next form begins with the symbol `let`. The interpreter looks up this symbol first in the `syntax` frame, and not finding it there, searches in the top-level environment, where it is found. The `let` form is a binding construct, and when evaluated it performs the following steps:

- Evaluates the initial value forms `(member x permanent-employees)` and `(member x temporary-employees)`; in this case, assume that the first expression returns "false" (represented in Scheme by `#f`), and that the second expression returns a sublist of `temporary-employees` beginning with the value refered to by `x`, such as `('perez 'pecore 'rhoads)`.
- Creates a frame in which `perm` and `temp` are bound to locations which contain the results of the computation.
- The resulting variable binding frame is scoped inside the `syntax` frame.

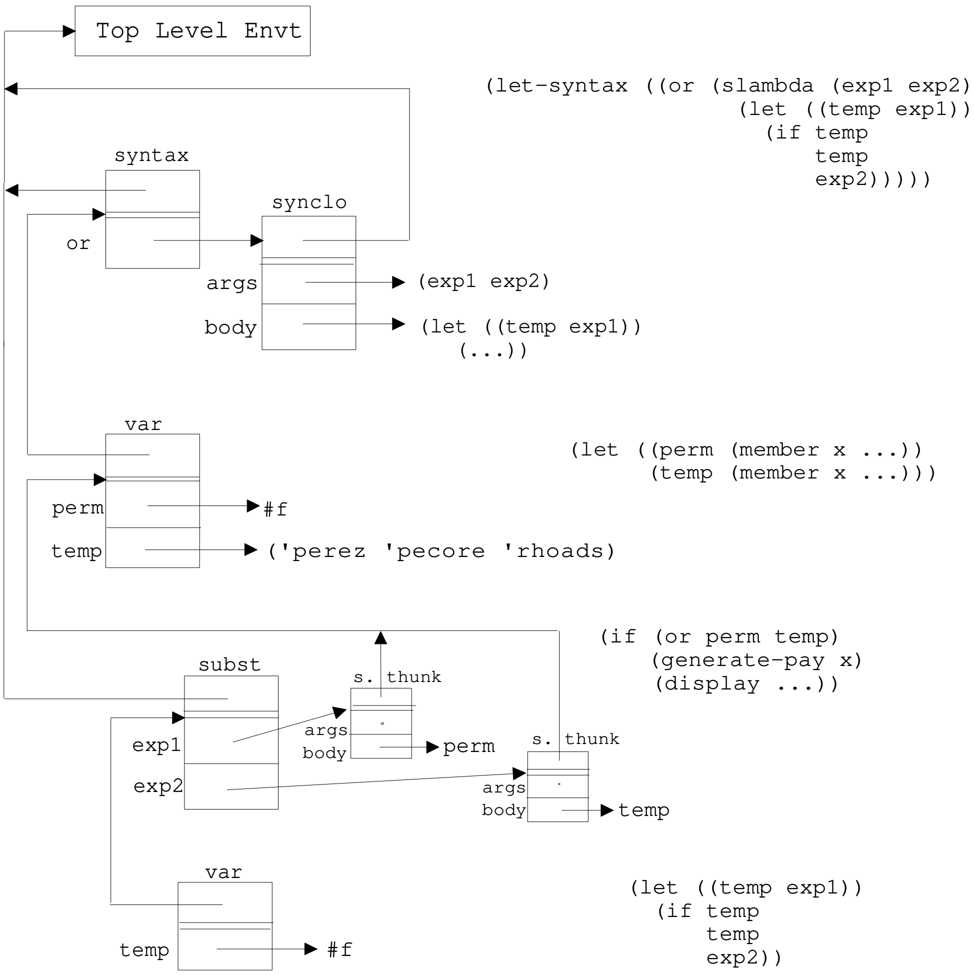

Figure 3.2: `or` Macro Example

The interpreter next evaluates the `if` special form in this new environment, after looking it up and finding it in the top-level environment. This form evaluates the first argument (the "condition clause") and if it is true, evaluates the second argument and returns the value. Otherwise, it evaluates and returns the value of the third argument—a classic if-then-else construct. In this case, the condition clause begins with the symbol `or`. The lookup for this symbol travels up each link to the `syntax` frame before it is found; evaluation of the name returns our syntactic closure. The system recognizes this as a macro and will recursively evaluate the forms in the macro body after switching to the macro context, as follows:

- A syntactic thunk is created for each of the argument expressions `exp1` and `exp2`, which contains the associated actual argument expression (the body of the syntactic thunk) and a scope link which points to the call site (the variable environment).

- A new kind of binding frame, the *substitution frame* (or `subst` frame) is created from the information in the syntactic closure. This contains a mapping of the formal argument names to the syntactic thunks which package the corresponding actual argument expressions, and a scope link which points to the macro's environment of definition, which becomes the current environment.

Now the macro body is evaluated in the environment represented by the substitution frame.

The first form in the macro body is a `let`; the interpreter searches for the identifier `let` in the `subst` frame, finds no binding for it, and moves up the scope link to the next binding frame, which is the top-level environment. As we've seen, `let` is defined there.

The `let` special form is evaluated in the usual way, except that when the initial value subform is evaluated, it turns out to be a reference to a macro argument variable (i.e., a slot in the substitution frame). This must be handled specially. We proceed as follows:

- The symbol `exp1` is first evaluated so that its value can be used in the binding frame that `let` will create. Lookup finds the symbol in the `subst` frame and returns the syntactic thunk associated with it.

- Evaluation of the syntactic thunk proceeds in the following way: the environment of definition (i.e., the call site of the macro where the syntactic thunk was created) is restored so that the expression in the syntactic thunk's second slot can be evaluated in the call site environment. The call site is the variable binding frame and the actual argument `perm` is found there; evaluation yields the value `#f`. The argument has been evaluated, so like a procedure call we return to the previous environment (inside the macro body), and continue.

- A binding frame is created that binds the identifier `temp` to a location which contains the value `#f`.

The body form of the `let` is just another `if`, evaluated in this new binding environment. (We still have to do the lookup, just in case the symbol `if` is redefined anywhere along the chain, but in this case it isn't, and after searching the new variable binding frame and the `subst` frame, we find the `if` special form in the top-level environment as before.)

The condition clause is next to be evaluated, and it is just the symbol `temp`—but which binding of `temp` is to be used? Thanks to lexical scope, we know that the binding of `temp` that we want is the first one found up the chain. In this case, the value returned is `#f`, and the interpreter knows to evaluate the third argument of the `if`, which is `exp2`.

Evaluation of `exp2` is similar to `exp1`. The interpreter searches back to the `subst` frame before finding a binding for this symbol, and returns the associated syntactic thunk. The interpreter evaluates the thunk as before, by restoring the call site environment saved in the thunk, and evaluating `temp` in that environment.

In this environment, `temp` is bound to the list returned by the `member` function, so that evalution of `exp2` returns this list, which is the value of the `if` expression, the enclosing `let` expression, and the `or` macro itself.

Given that the condition clause `(or perm temp)` returns a non-false value, the interpreter moves on to evaluate `(generate-pay x)` in the variable binding environment. All of the environments introduced by the macro evaluation are now out of scope, and no longer visible.

3.2.2 Inlining Pass-by-name Procedures

Now that we've seen how the interpreter works for this class of macros, we describe how macros are processed efficiently in the compiler. In Section 3.1.4 we showed how to compile inline procedures in the system. We also mentioned that this is similar to the way we compile the bodies of macros at the point where a macro is called; except for scoping considerations, we simply "pretend" that the code for the macro had occurred there. Macro body expressions are compiled at the point of call, similarly to how we interpret a macro at the call site, but with all expression analysis and symbol lookup done at compile-time. Essentially, our integrated macro system implements macros as inlined procedures.

Inlining Procedures. Recall the steps used to compile an inline procedure. The body of the procedure is compiled at the point of call, but the scope used to resolve the symbols in the body is given by the lexical environment, which points to the frame where the procedure is defined. All lookups are done from this point. Once symbols are found, the compiler computes lexical addresses for them.[9] Recall that the lexical address is the location of a variable binding relative to the current environment, expressed as the lexical depth (the number of frames up the chain from the

[9] This is an oversimplification. In RScheme, representation selection is done in a second pass so that certain optimizations can be done, such as merging binding frames. For this reason, lexical address computation cannot done at the point of compilation implied by the text.

current point) and the slot offset (the offset into the frame to the proper binding). The lexical depth of a variable used in an inlined procedure is calculated by taking the offset in frames from the lexical environment to the frame where the symbol binding is found, and adding it to the difference, in frames, between the current environment and the lexical environment. The slot offset stays the same.

As in the case of inline procedures, the abstraction reflected by the CTE is broken by the integrated macro system; a macro is compiled at the call site, but scoped in the environment where it was defined. The changes to the CTE described above adequately handle lookups and addressing for the case of pass-by-name macros. The lexical environment is the point in the chain where we *logically* perform searching for symbols referenced in a macro body.

So the same mechanism used to compile inline procedures applies when compiling a macro call, except that the environment structure is different given substitution frames and syntactic thunks. These pose no special difficulty, however, as we now show.

Optimizing away syntax and substitution frames. As it turns out, we can perform one easy optimization to our computation of lexical addresses. All the work of a macro call is done at compile time, so the macro representation doesn't actually need to exist at runtime—the macro body has been incorporated into the code at each call site. Therefore, neither the associated syntactic closure nor the syntax frame which contains the binding of the macro name are needed at runtime. Because they will not be part of the runtime environment chain, they must be ignored for purposes of computing lexical addresses.

Furthermore, the substitution environment frame which maps a macro's formal arguments to syntactic thunks is not necessary at runtime because the syntactic thunks themselves are inlined, so that the argument expressions are compiled as part of the macro body. None of these structures will be constructed at runtime, so they need not be part of the computation of lexical addresses.

Now we see that the use chain is not just a superset of the scope chain, but a superset of the runtime structure as well, because it includes frames which will not be created at runtime.

3.3 Handling Advertent Capture via Syntactic Exposures

In this section we describe the mechanisms used in our system to evaluate macros which create binding environments which are conceptually visible at the macro call site. We start by examining the evaluation of our example `for` macro using an oversimplified mechanism for clarity of exposition. We then describe the actual implementation in our system. We again will be discussing the interpretive version, for simplicity. The compiler version is almost identical, except for the differences explained above regarding symbol lookup and computing lexical addresses.

3.3.1 Handling the `for` Macro

Consider again the macro that defines the `for` construct, with an example of its use:

```
(define-macro (for loopvar init limit body)
   (let ((loopvar init))
     (repeat-until (begin body
                          (set! loopvar (+ loopvar 1)))
                   (= loopvar limit))))
; example use
(let ((sum 0))
  (for i 1 10 (set! sum (+ i sum))))
```

In Section 3.1.2 we showed that the semantics of this macro are not pass-by-name, because `for` is a binding construct; the macro writer is purposely creating a new environment that binds the loop variable `loopvar`, and evaluating the argument `body` inside it. To the user of the macro, it is the variable `i` that is bound by the `for` construct, creating a new lexical scope for the expression `(set! sum (+ i sum))` to execute in. Only the meaning of the loop variable name is changed; all other symbols in the argument expressions refer to variables in scope at the call site. Macro writers using conventional macro systems take advantage of uncontrolled name capture to write macros like this. We want to allow the same sort of programmer-controlled name capture in our system, without sacrificing lexical scope—no *other* name captures should occur accidentally.

Note that if this construct were implemented in a strict pass-by-name system, the `body` argument would be closed in the call site environment, before the loop variable is bound, so that the changing value of the loop variable would not be seen. Because of this, our macro system is not strictly pass-by-name; it must resolve scope issues like this, and does so *lazily* by waiting to see if the macro body expressions create new bindings that are supposed to be visible to arguments at the call site.

The interpreter (or compiler) cannot directly understand what is happening here; because `for` is not a primitive construct that the underlying system knows how to handle, it is not obvious that it is a binding construct that creates a new lexical scope. The macro system must infer this and let the interpreter know how to handle the situation.

Our solution is to provide a simple rule for determining when advertent name capture has occurred and a mechanism for exposing "captured" call site identifiers to other argument expressions of the syntactic closure. We therefore call the mechanism *syntactic exposure*.

Suppose we have an environment at the call site that can "magically" be extended with new bindings *retroactively*, bindings that appear at the call site inside the macro body, so that argument expressions can be evaluated inside them—and disappear when the binding forms are exited, so that they are no longer visible when arguments that are *not* scoped inside them are evaluated.

In other words, suppose the hypothetical environment could expose new bindings to some expressions, but not to those whose lexical scope we *don't* want to modify.

This is the sort of behavior we require to implement `for`. We want the binding of `loopvar` to be seen at the call site in the form of the argument `i`, but only when the argument `body` is evaluated. Our system needs an extra bit of machinery to perform the "magic" of defining a variable at the call site after we've already started evaluating the macro body in the context of the macro's environment of definition. We call this machinery the *environment virtualizer*, or EV. This is the crucial device that makes it possible to address capture issues without awkward renaming techniques.

The environment virtualizer. The EV is a data structure that makes advertently introduced binding environments visible at the macro call site. The data structure is created every time a macro is called, because we don't know whether there are any bindings introduced in any given macro until we evaluate its body expressions. Further, we don't know which, if any, of the macro arguments will be evaluated in the scope that is pushed back into the call site; we must wait to see how they are used in the body of the macro. Until these issues are resolved during evaluation, the EV acts simply as a place holder, marking the macro call site but not modifying scope in any way.[10]

We now require that the scope links in any syntactic thunks created at a macro call point to the EV rather than to the original environment of call, because when a macro creates a new lexical scope and evaluates an argument represented by one of the syntactic thunks inside that new scope, those bindings become visible to that argument expression by being "mirrored" within the EV. Therefore, the EV

[10]Our collaborator, Donovan Kolbly, was instrumental in clarifying the scoping issues here and devising this mechanism for properly managing scope.

is scoped inside the call site environment, as shown in Figure 3.3.

A slightly oversimplified mechanism. We can imagine the environment virtualizer being implemented as a pushdown stack of binding frames. Whenever a macro binds one of its formal argument symbols, making a binding that may need to be visible at the call site, we make note of the actual argument which acts as the alias. If another argument expression is used inside the new scope, the binding frame which introduces the new scope is pushed onto the stack. This makes the binding visible to the argument expression that the macro uses inside this scope as if the binding had come before the macro call, as well as to code inside the macro. When the scope is exited, the binding frame is popped off the stack. This works in the case where multiple scopes are introduced in a macro body as well. We call the frames on the stack *alias environment frames.*

We cannot actually implement the EV in quite this way, because of subtle difficulties that crop up in the interaction between destructively updating the stack and evaluating other expressions that may create closures. (We'll give an example and discuss the actual interpreter implementation later. It turns out to not be a problem in the compiler.)

The `for` example. We now describe how an interpreter uses the EV to handle scope by stepping through the evaluation of the `for` macro. The structure of the environment chain is illustrated in Figure 3.3. As before, the structure grows downward, where components at the top are created at the beginning of evaluation, and the later steps create structures toward the bottom. For convenience, we have numbered the data structures, starting with the top-level environment (0), to show the order in which they are created and to give reference points in the text.

The macro is defined using the special form `let-syntax` as in the `or` example in the previous section. Just as before, the `slambda` form creates and returns a syntactic closure (1), which packages the macro's formal arguments and body forms

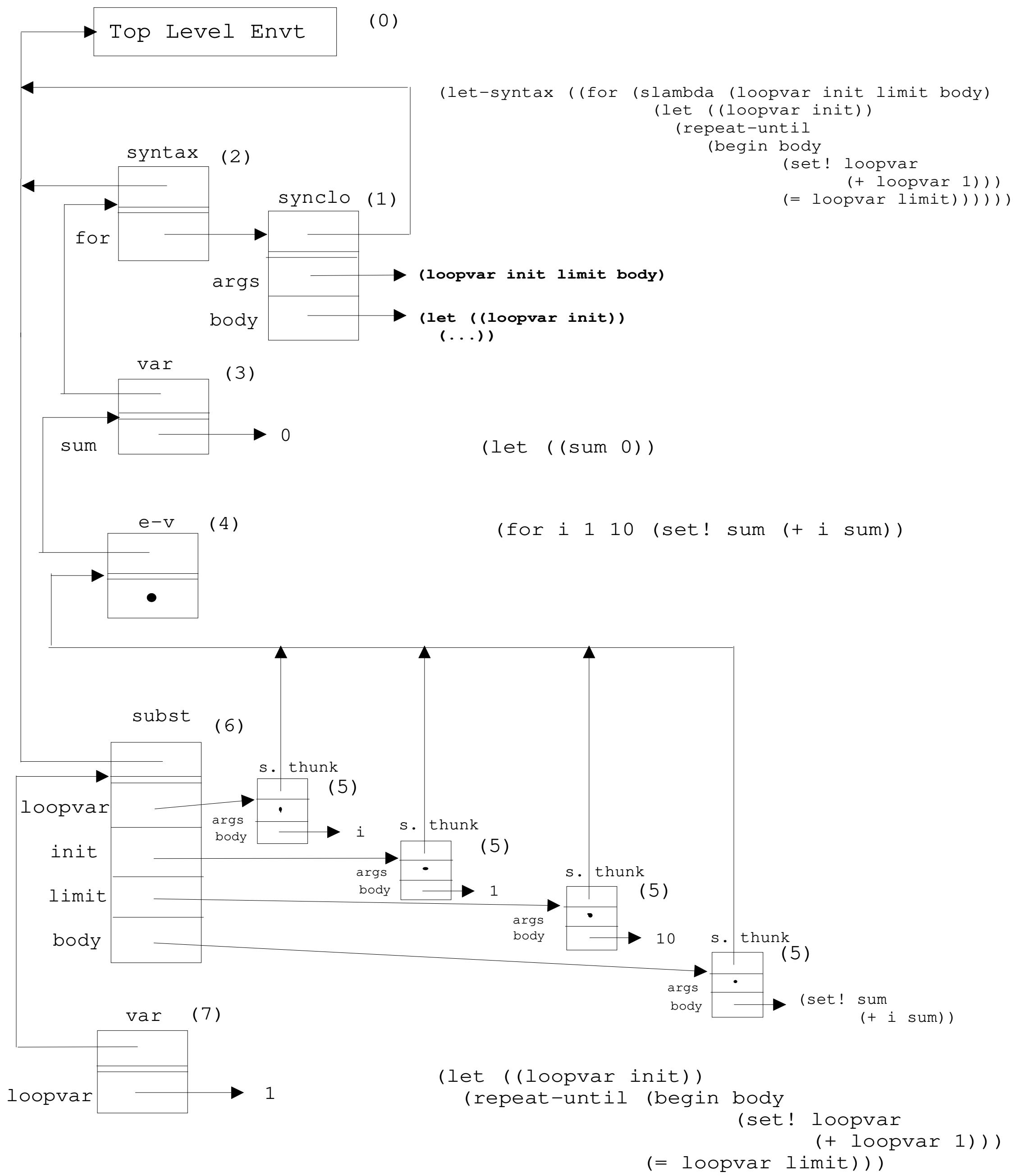

Figure 3.3: `for` Macro Example—Advertent Name Capture

with a pointer to the environment where the macro is defined. The name `for` is associated with this syntactic closure in the syntax binding frame (2), which is scoped inside the top-level environment. The rest of the expressions are evaluated as the body of the `let-syntax`.

The `let` form introduces a variable binding frame (3), which binds the name `sum` to a location containing the result of evaluating 0 (which evaluates to itself). This frame is scoped inside the `syntax` frame.

The interpreter next evaluates `(for i 1 10 (...))`. A lookup is done for the symbol `for`, and it is found in the `syntax` environment; evaluation of the name returns our syntactic closure (1). The system recognizes this as macro and will recursively evaluate the forms in the macro body; to set up properly to do this, the following actions are taken:

- the environment virtualizer is created (4), initially empty, and is scoped inside the call site environment (3). The current environment is now the EV.
- A syntactic thunk is created for each of the four argument expressions (5). The scope links must point to the environment virtualizer, so the definition environment slot in each syntactic thunk is filled with a pointer to the current environment. This reflects that the environments in which the arguments will be evaluated are not yet known.
- A new substitution frame is created that binds formal arguments to associated syntactic thunks (6). The scope link is copied from the syntactic closure so that this frame is scoped within the top-level environment, as do the argument names and body expressions.

With these elements in place, evaluation of the macro body can begin. At this point, the new mechanism first comes into play, so we'll break the evaluation of each form into its component steps and show more detail than usual.

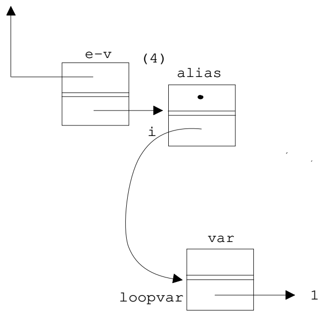

Figure 3.4: Pushdown Stack

The first form in the macro body is a `let`; the interpreter searches for the identifier `let` starting at the substitution frame, finds no binding for it, and moves up the scope link to the next binding frame, which is the top-level environment. As usual, the lookup finds that `let` is a special form and recursively evaluates it:

- The symbol `init` is first evaluated so its value can be used in the binding frame that `let` will create. The symbol is bound in the `subst` frame and lookup returns the associated syntactic thunk. Evaluation of this proceeds as in the previous section: the environment of definition is restored and the argument expression is evaluated in it. In this case, the argument evaluates to the number 1 trivially.

- A binding frame is created that binds the formal argument `loopvar` to a location which contains the value 1.

- At this point, we have satisfied the first condition of advertent capture, which is the binding of a formal argument inside the macro body. Our system checks this condition for each binding done during evaluation of a macro. The system records that `loopvar` has been aliased by a call site name so that if the second condition is met, the binding of `loopvar` can be referenced.

Now the body form of the `let` is evaluated in the new binding environment. Lookup of the first identifier of the form, `repeat-until`, returns our (fictional) special form, which performs iteration, from the top-level environment. This simply evaluates its body form, then the termination condition, iterating if the condition evaluates to false. We'll detail one iteration of the loop.

- The first identifier, `begin`, is looked up and found in the top-level environment; it is a special form which performs sequencing of its arguments, `body` and `(set! loopvar (+ loopvar 1)))`.
- The name `body` is looked up and found in the `subst` frame. Since it names a syntactic thunk, the interpreter restores the call site environment stored in the syntactic thunk and begins evaluating the argument expression `(set! sum (+ i sum))`.

Here the system notices that the second criterion for advertent capture has been met; the actual argument `body` is being used in the new environment created by the binding of the formal argument `loopvar`. We want to extend the call site environment with this binding, so the mechanism pushes an *alias frame* onto the pushdown stack maintained by the EV. This frame associates the actual argument `i` with the *slot* which holds `loopvar` (see Figure 3.4). It is important to understand that the alias refers to a particular slot (i.e., binding) and not the entire environment frame. In this way, if `i` is referenced in `body`, the value stored in the location bound to `loopvar` can be accessed. Here the alias frame ensures that the new scope created by the `for` construct is visible during the evaluation of the argument expression `(set! sum (+ i sum))`.

Continuing the evaluation:

- Beginning in the EV, the interpreter looks up `set!`. This form is not bound in any frame introduced by the evaluation, but it is bound in the top-level environment (it is a special form which performs assignment by updating the location bound to the first argument with the value returned from evaluating the second argument).

- The second argument is the combination `(+ i sum)`. First, the identifier `+` is looked up and found in the top-level environment; it returns the procedure that performs addition. Next, the symbol `i` is evaluated by looking it up. This symbol is now bound in the EV; its binding frame is on the top of the pushdown stack, so it is in scope here. The value 1 is returned by the lookup. Finally, `sum` itself is looked up. The symbol isn't bound in the EV, but it is found in the next frame up by taking the scope link; evaluation returns 0.

- The addition is performed, returning 1, which is assigned by `set!` to `sum`. The evaluation of `body` is now complete, and the current environment reverts to that in force before the syntactic thunk was evaluated, namely, the binding environment for `loopvar`.

- The second form in the sequence is `(set! loopvar (+ loopvar 1))`. Evaluation of `set!` proceeds as before; the only difference being that `loopvar` is visible in this scope, and the location bound to it is updated with the value 2. (Note that this new value would become immediately visible to argument thunks through the EV alias `i` as well, if any were being evaluated in the new environment at this time.)

- Since `(set! loopvar (+ loopvar 1))` is the last form in the sequence, evaluation of the body of the `repeat-until` is complete.

It only remains to evaluate the termination condition, the combination `(= loopvar limit)`. The symbol `=` is the relational operator for equality, found by looking up the scope links to the top-level environment. (The arguments of `=` are evaluated and "true" returned if they have equal values, "false" otherwise.) In this case, we are still evaluating in the binding frame for `loopvar`, so that lookup returns 2. For `limit`, we look up to the `subst` frame and return a syntactic thunk. Evaluation of `limit` is similar to that of `init` above; we restore the callsite environment only to trivially return the value 10. Since these values are not equal, the termination condition is not satisfied and the interpreter repeats the process with `begin` again.

Notice that when control passes out of the body of the `let` after the `repeat-until` completes, the binding of `loopvar` is no longer in scope and so the alias `i` is popped off the EV stack.

To summarize:

- Each time any macro is called, an environment virtualizer is created and treated as the call site environment. It acts as a proxy for environments that are actually created in the body of the macro, but are conceptually visible at the call site.

- The first criterion for advertent capture is met when, during evaluation of a macro body, the interpreter detects that a symbol passed in as a formal argument is being intentionally bound, rather than evaluated as a normal expression.

- When this occurs, the system pushes an alias environment frame onto the stack maintained by the EV, recording that the formal argument and the actual argument associated with it are the same thing. The actual (identifier) is an alias for the binding created inside the macro.

- Continued evaluation of the macro body expressions occurs in the environment where the formal argument is bound; if another macro argument is evaluated in this scope, it is evaluated in the environment captured by the syntactic thunk. The EV ensures that if it needs to see the new binding, it will, using the alias from the call site.

- When the new scope is exited, the corresponding alias frame is popped off the EV stack. Evaluation of argument thunks will no longer see the binding through the alias.

3.3.2 Implementation Details

In the previous section we introduced the environment virtualizer and detailed a macro evaluation using this structure to resolve the scope of call site identifiers bound inside the macro body. The description was oversimplified, however, because the EV cannot actually be implemented as a pushdown stack in the interpreter.

A stack implementation will not work because we sometimes need to capture the current environment in a closure, so the structure of the environment cannot be side-effected.[11] For example, a macro formal argument might be closed in a `lambda` expression in the macro body, which would require that the structure of its environment that is captured at the point where the lambda occurs to remain fixed thereafter. If a frame is pushed onto the EV stack after the lambda occurs, this requirement will be violated.

In this section we discuss an alternative interpreter implementation which maintains the effect of the pushdown stack, but works without side effects (thus allowing the environment to be captured by a lambda expression). The basic strategy is to move the alias frames (described above) into the environment chain at the point that the formal argument is bound, and change the search procedure so that aliases are looked up in the portion of the chain created by the macro evaluation.

Our implementation splits the pushdown stack into two parts, a shared part which says what actual arguments *can be* an alias and a (possibly) non-shared part which points out which symbols are being aliased in a given scope. These two parts are represented in the following way:

- Storing a list of those actual argument symbols which are potential aliases in the environment virtualizer itself.
- Adding the alias environment frames that were stored in the pushdown stack

[11] Note that it is the *bindings* of the variables that must not change; names must continue to refer to the same *storage*. Side effecting the variables is allowed, however—we must capture the environment, but the values of the variable bindings may still change due to assignments.

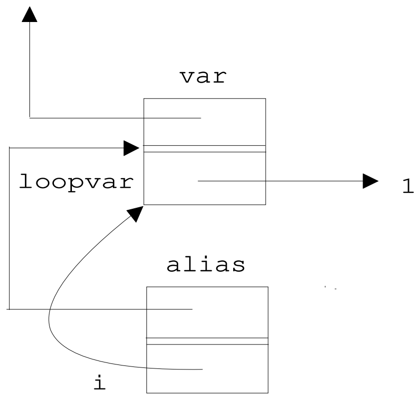

Figure 3.5: Alias Environment Frame

to the environment chain, scoped inside the environment created by adding the binding of the formal argument to the chain.

In the interpreter, when new scope is created during evaluation of a macro that binds a formal argument, the *alias* is found by searching for the formal argument up the environment chain and finding the syntactic thunk associated with that formal. At this point we recognize that the argument is simply an identifier, not a normal expression, so we strip away the environment that it was (unnecessarily) closed in. In the `for` example, the formal argument was `loopvar` and the alias was the actual argument `i`. In the case of recursive or nested macro calls, there may be aliases of aliases, so an entire alias list may have to be constructed, with potentially as many members as there are nested calls.

After the binding environment is constructed, the alias list is used to create an alias environment frame with slots for each alias. This frame consists of:

- a *key*, made up of the alias and a pointer to the environment virtualizer that contains that alias
- a pointer to the slot in the previous variable binding frame which holds the binding of the formal argument being aliased, so that the alias frame mirrors the binding of the formal argument.

The information in the key slot of the alias frame is conceptually an *annotation* of the formal argument's binding, marking it as having an alias. In the implementation, we found it convenient to have a separate frame scoped inside the formal argument's binding. The key is used not only to name the alias, but also to make sure that the reference to that name came from the same macro call that introduced the alias.

The interpreter maintains pointers to two frames in the environment chain, one to the call site and another to the frame most recently created during evaluation of the macro body. These pointers correspond to the compiler's lexical environment and current environment, respectively, and are used to control searching for aliased symbols.

Figure 3.5 shows the environment structure built during evaluation of the `for` macro using this implementation.

Given these elements, we can describe how evaluation of the `for` macro in the new implementation proceeds; refer to Figure 3.6. Generally, the system works as in the example given above. Searches for symbols detected during macro evaluation pass along the lexical scope of that evaluation, moving to the environment of call only when the symbol is a formal argument. Now, however, the environment virtualizer just marks those symbols which are acting as aliases. We will only come across an alias when evaluating an argument expression that uses it, such as `(set! sum (+ i sum))`. When evaluation reaches the addition and `i` is looked up, the first frame searched is the EV, where it will be found to be a potential alias. Now the system must perform a search determine if `i` is actually used as an alias, and if so, what name it is an alias for. The new search procedure is as follows:

- Create a search key made up of the alias name and a pointer to the current EV. In the example, the key pairs `i` with a pointer to EV.
- Search up the use chain starting at the current environment for an alias environment frame which has a key that matches the search key.

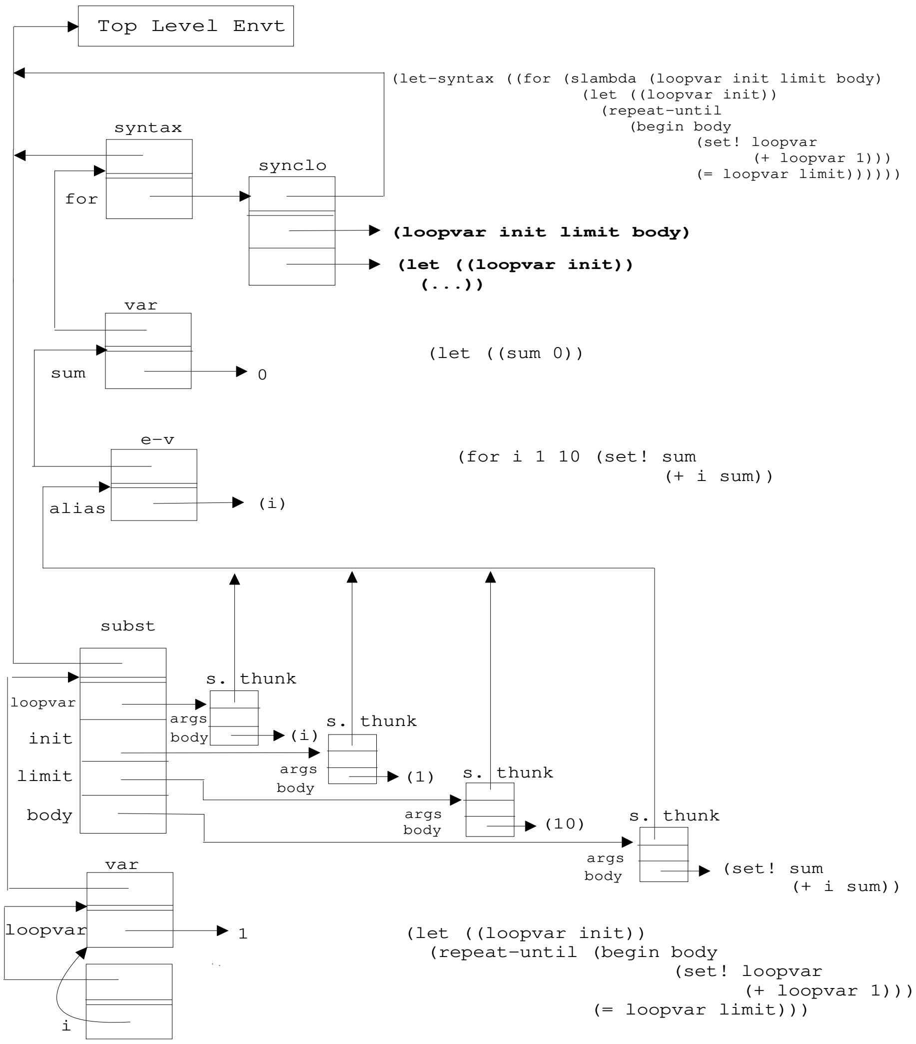

Figure 3.6: Evaluation of `for` Using Alias Environment Frame

- If found, return the proper value. This value will be equal to the current value in the binding of the aliased formal argument name, (`loopvar` in our example).
- If a frame with a matching key is not found, the search ends at the EV stored in the key; this doesn't happen in our example, but when it does happen it means the symbol being searched for is not used as an alias, and must therefore be a symbol in scope at the call site. In such cases, the system completes the lookup by searching in the normal way, starting with the frame referenced by the environment virtualizer's scope link.

Most of this machinery maps from the interpreter to the compiler; however, checking the list of aliases stored in the EV to see if a symbol used in the body expression is an alias does not. Because the compiler implements pattern matching, it cannot simply store a list of potential aliases because the identifiers in that list might expand later into some expression. Rather, the compiler searches for a matching alias frame on each symbol lookup during compilation of an argument expression. This slows compilation but not execution, whereas in the interpreter it would slow evaluation quite a bit.

Now that we've seen how the interpreter implementation uses the environment virtualizer and alias environment frames to model the effect of using a pushdown stack, for adding bindings retroactively at the call site without side-effecting the environment structure, we have completed our explanation of advertent capture.

3.4 Handling Explicit Intentional Capture

In this section we describe an extension to the existing implementation which would make it possible to write macros such as the `catch/throw` example described in Section 3.1.3.

We saw before that the idea of explicit intentional capture is similar to advertent capture, with the difference that this sort of capture cannot be inferred because the symbol to be captured is not given as an argument to a macro call. No new scoping mechanisms are required, because intentional capture is actually simpler than advertent capture. Whereas advertent capture requires that our system infer the intent of the programmer from the rebinding of macro argument variables, explicit intentional capture is easier to support because the programmer explicitly identifies the identifier to be aliased.

Here we propose the addition of a binding construct that can be used to explicitly inform the macro facility that we are introducing a binding that must be visible at the call site, in the same way that the binding of the loop variable in the `for` example is.

Consider the following implementation of `catch`:

```
(define-syntax (catch body)
   (call/cc (lambda (throw)
               (retro-let ((throw 0))
                  body))))
```

The `retro-let` expression informs the system that the binding created by the form is retroactive to the environment where the argument `body` came from. The system can add such bindings because the call site is marked with the environment-virtualizer. In this example, there is no initial value; the value stored in the location bound to `throw` will be a pointer to the exit procedure created during execution of `call/cc`. Once this is done, the system can treat references to `throw` in the argument expression `body` as if it had been advertently captured.

Note that while the machinery for handling this case is in place, we are not sure what the final policy on using it in writing macros is. The `retro-let` mechanism is a first stab at a way of presenting this machinery to the user, but is certainly not the final form.

Chapter 4

Reflection On Scope and Context-Sensitive Macros

We believe that a compiler which supports reflection on scope and context can be reprogrammed using our macro facility to implement various language features and optimizations. In the following, we briefly sketch how the facility might be used to implement Lisp-style "special" variables (often implemented using separate name spaces) and encapsulated slots in objects in the RScheme system. Then we show an example of *context-sensitive macros* and discuss a number of systems that could be built using these.

4.1 Special Variables

Common Lisp [Ste84] provides a special form for declaring variables which have *dynamic scope*, which contrasts with lexical scope in that such variables can be referenced anywhere in a program once bound, and whose value when referenced is the *most recent* binding of the variable, irrespective of intervening closures, etc. These "special" variables are said to have *indefinite scope* and *dynamic extent*.

Special variable bindings obey a stack discipline. If a binding construct associates the special variable `foo` with the value 9, then another binding associates `foo` with the value 90, any reference to `foo`, including in non-local procedures, will return 90. When evaluation of the second binding construct terminates, the first binding is restored.

Special variables can be added to a Scheme implementation by constructing a separate dynamic environment chain for the special bindings.[1] In effect, this implements a separate name space for special variables. To support this type of binding we could add three constructs to the language: `special-let` for creating new special bindings, `special-ref`, for accessing the value of a special variable, and `special-set!` for assigning to it.

Using our macro system it should be possible to implement special variables without creating a separate name space. We believe that macros can be written using our facility that extend the functionality of the normal lexical scope chain. We might create a new kind of binding environment frame that holds not only the special bindings but also "hooks" for telling the compiler how to access and assign to the new binding. In this case, special user-level macros for accessing and assigning to special variables would no longer be necessary; normal lexical lookup and the assignment operator `set!` can be used for both special and normal variables.

4.2 Encapsulated Object Slots

In object-oriented programming extensions to Lisp and Scheme (which are largely implemented using macros), classes can be defined as a set of *slots* (instance variables) which extend the definition of some base class. For example, the following simplified example of a class definition creates a class `point` for specifying a coordinate on a two dimensional surface, such as a computer's display screen:

[1] Special variables can be implemented one of two ways, either by *deep binding* or by *shallow binding*. Shallow binding involves associating a stack of bindings with each variable bound so that the top of the stack is the most recent binding; when a binding goes out of scope it can be popped off the stack so that the next most recent binding becomes extant. In deep binding, all special variables share a dynamic chain which acts as the stack. This discussion assumes deep binding.

```
(define-class <point> (<object>)
    (x 0)
    (y 0))
```

Here we define `point` to be a subclass of `object`, containing the slots `x` and `y` (each with initial value 0).

Unlike object systems in languages like Smalltalk, data slots of an object defined in this type of system are not opaque to methods defined on other classes. What we'd like to do is implement *encapsulated slots* for RScheme's object system and change the default scope so that methods defined on the class can access them. We believe this can be done in our macro system by combining the approach used in the fluid variable example with a meta-object protocol like that described in [KdesRB91]. The metaobject protocol gives us the ability to query the system about the particulars of any given class definition. Once again, the key trick is to implement a new kind of binding which contains hooks that the compiler uses to change scope rules.

In the example above, a method might be defined to draw a dot on the screen at the coordinates given by a particular instance of `point`. We do this by calling the macro `define-method` which adds the code that implements this functionality for this particular class to the *generic function* which holds the `draw` methods for each class which defines it. We should be able to compile a definition of the method `draw` so that it can access the slots of the object of class `point` which it operates on.

The new `define-method` will have to:

- Use a metaobject protocol function such as `all-slots` to get a list of slot descriptors for the object.

- Create a new scope frame containing a set of bindings for each slot. These aren't normal runtime bindings, but indexes into the object (which is currently in scope) which is the target of the method. This scope frame extends the environment of method definition.

- Compile the body of the method definition in the new scope that represents this method as just executing in the scope of the object.

4.3 Context-Sensitive Macros

One consequence of compiling macros directly is that macros can be written that do different things depending on the context in which they occur; such a macro is a *context-sensitive* macro. When the compiler begins translation of a macro call, a programmer-supplied routine can annotate the compiler's context data structure (the CTE) that describes what context the code will execute in. Then, when the compiler expands a macro call that occurs inside that context, the code for the new macro can look at the information about the context in which it will execute, and take advantage of that information to generate more efficient code in that context.

For example, suppose we've added macros `foo` and `bar` to produce synchronization constructs in a program. The `foo` construct might lock some resource for exclusive access, while the `bar` construct might also need to lock some resource for exclusive access. Assume that the semantics of nesting a `bar` in a `foo` construct means that `bar` has rights to the object `foo` locked; as a context specific optimization, if the compiler can infer that `foo` and `bar` are going to lock the same resource, then `bar` doesn't have to do another lock, because it can tell that `foo` already locked it and therefore it has rights to it.

If the procedural macro that produces `bar` can tell that it's operating on the same object as the other macro, then it may be able to determine how to produce an optimized `bar` construct; however, if it cannot it may translate `bar` in a very different way, because nesting a `bar` in a `foo` may introduce the possibility of deadlock which must be guarded against using a more expensive algorithm. Still other context sensitive translations are possible if the macros occur in sequence. In that case deadlock detection may be required but a simpler algorithm may suffice.

The following example of this is given in C because it's easiest to see how this might be done in a conventional systems programming language, plus we're able to emphasize that we believe this type of macro system is applicable to a wide range of programming languages.

```
for (i=1; i<100; i++)
{  /* intervening code for initialization */
   foo (some_condition)
   {  /* code... */
      for (j=1; j<1000; j++)
      {
         bar (some_condition)
         { /* bar code... */
         }
      }
   }
}
```

Here, we know the compiler will begin compiling the outer `for` before it hits the `foo` construct, and it will begin compiling `foo` before it sees the inner `for`, and only then will it process the `bar` construct. At each step, the compiler adds information to the context data structure that describes the runtime context in which the compiled code will execute.

When the compiler begins processing the `foo` construct, it calls whatever procedures we specify for processing a `foo` macro during compilation. Because we are using a procedural macro, we can decorate the compiler's context data structure with an annotation that the code scoped inside the `foo` construct will be executing inside that `foo`, which operates on a particular condition variable.

Then the compiler will do the basic processing of the inner `for` construct, adding context information about the environment in which the `for` code will execute. Then it begins processing the `bar` construct. Our procedural macro for this construct can examine the context chain to discover that it is executing inside the scope of a `foo` construct, and that the `foo` operates on a condition variable. Then the `bar` macro can choose how to translate itself, specialized to the context in which it occurs.

Given this basic low-level context-sensitive translation strategy, we can build a more declarative framework for such things as assertion checking, similar to the preconditions and postconditions of Eiffel, but operating at compile-time to perform statically checkable properties such as invariants with no runtime performance penalty. Such a facility might resemble the *transform* construct of Gries and Volpano [GV90], supporting the expression and rigorous checking of *coordinate transforms* (in the sense of Dijkstra [Dijk76]) to semi-automatically derive efficient implementations of programs expressed in a higher-level notation.

Similarly, our basic macro facility could be used as a substrate to implement *non-traditional parameterized types* in the sense of Batory, et. al. [BTS94], allowing the expression of consistent refinement of data structure implementations. Our ability to support context-sensitive analyses and transformations would also make it easier to implement self-optimizing, composable software components and automatic design-rule checkers which are integrated with the compiler as opposed to being separate, stand-alone tools.

4.4 Summary

We have described language features that might be implemented in an extensible compiler for Scheme using our macro system. Other, more compiler-oriented features which we believe could be implemented in the RScheme compiler or any compiler that incorporates the macro facility presented in this thesis include variable use analysis and conditional compilation based on external analyses. More examples with detailed descriptions of the macros used can be found in [WKC97].

Chapter 5

Summary

The new macro facility described in this thesis was developed to address the problems with regard to maintaining lexical scope in conventional macro systems, and the shortcomings of facilities that have been proposed to provide lexically scoped macros. It was also developed as a step toward building a metaobject protocol for an extensible compiler.

Our system provides lexically scoped macros by using syntactic closures and syntactic thunks to maintain the proper context of expressions that are substituted into new contexts, as macro argument expressions and body forms are when a macro is called in a program. We identified macros which act as pass-by-name inline procedures, macros which introduce new scope through advertent capture, and macros which perform intention name capture explicitly as three classes of macro usage and extended the basic mechanism of syntactic closures to handle each class.

We believe that as this facility is used and new compilation strategies built upon it, it will become the basis for extensible languages supporting flexibly layered and optimized software.

Bibliography

[AS85] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[BTS94] Don Batory, Jeff Thomas, and Marty Sirkin. Reengineering a Complex Application Using a Scalable Data Structure Compiler. In *Proceedings of the ACM SIGSOFT '94 Conference*, December 1994.

[BR88] Alan Bawden and Jonathan Rees. Syntactic Closures. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.* 1988, pp. 86–95.

[CR91] William Clinger and Jonathan Rees. Macros That Work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages.* 1991, pp. 155–162.

[Dijk76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[Dyb92] R. Kent Dybvig. Writing Hygienic Macros In Scheme With Syntax–Case. Technical Report 356, Indiana Computer Science Department. June, 1992.

[DHB92] R. Kent Dybvig, Robert Hieb and Carl Bruggeman. Syntactic Abstraction in Scheme. In *Lisp and Symbolic Computation: An International Journal*, Vol. 5, 4, 1992.

[FWH92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* The MIT Press, 1992.

[GV90] David Gries and Dennis Volpano. The Transform—A New Language Construct. In *Structured Programming*, No. 11, pp. 1–10, 1990.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, 1988.

[KdesRB91] Gregor Kiczales, Jim des Riveres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[KFFD86] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming.* 1986, pp. 151–159.

[Pae93] Andreas Paepcke, ed. *Object Oriented Programming: The CLOS Perspective.* The MIT Press, 1993.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Pit80] Kent Pitman. Special Forms in Lisp. In *Lisp Conference.* 1980, pp. 179–187.

[Que94] Christian Queinnec. *Lisp in Small Pieces.* Cambridge University Press, 1994.

[Ree93] Jonathan Rees. The Scheme of Things: Implementing Lexically Scoped Macros. *LISP Pointers*, Vol. 6, 1, 1993.

[R4RS] The Revised[4] Report on the Algorithmic Language Scheme. *LISP Pointers*, Vol. 4, 3, 1991.

[Ste79] Guy L. Steele Jr. Compiler Optimization Based on Viewing LAMBDA as Rename Plus GOTO. In *Artificial Intelligence: An MIT Perspective.* The MIT Press, 1979, pp. 397–431.

[Ste84] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, 1984.

[SG93] Guy L. Steele Jr. and Richard P. Gabriel. The Evolution of Lisp. Second ACM SIGPLAN History of Programming Languages Conference (HOPL-II), in *ACM SIGPLAN Notices.* Vol. 8, No. 3, 1993, pp. 231–270.

[Wil94] Paul R. Wilson. Computer Science 386L Class Notes. See `http://www.cs.utexas.edu/users/oops`.

[Wil96] Paul R. Wilson. An Introduction to Scheme and its Implementation. See `http://www.cs.utexas.edu/users/oops`.

[WKC97] Paul R. Wilson, Donovan Kolbly, and Stephen P. Carl. An Integrated, Lexically Scoped "Macro" System for Compiler Extensibility. In preparation.

Vita

Stephen Paul Carl was born on August 25, 1963 in San Antonio, Texas. He is the son of Paula Lea Carl and Roger Max Carl. He graduated Magna Cum Laude from John Foster Dulles High School in Stafford, Texas in 1981. He was awarded a National Merit Scholarship from Rice University, where he eventually earned a bachelor's degree in Electrical Engineering in 1986. It was there that he met Christie Athena Chapman, who became his wife in 1987. He has worked as a self-employed software developer in Houston, Texas, a contract programmer/analyst in Clinton, Iowa and Colorado Springs, Colorado, and a programmer in Aurora, Colorado. In 1992 he moved to Austin, Texas to begin fulfilling prerequisites for graduate work in Computer Science at the University of Texas at Austin. He was admitted to the Graduate School in January of 1994.

Permanent Address: 2705 Rae Dell
Austin, Texas 78704

This thesis was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.