Extensible Syntax in the Presence of Static Analysis

or

Scheme macros meet ML types

by

Ravi A. Nanavati

S.B., Massachusetts Institute of Technology (2000)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2000

Author .....
Department of Electrical Engineering and Computer Science
August 14, 2000

Certified by .....
Olin Shivers
Research Scientist, Massachusetts Institute of Technology
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Extensible Syntax in the Presence of Static Analysis

or

Scheme macros meet ML types

by

Ravi A. Nanavati

Submitted to the Department of Electrical Engineering and Computer Science
on August 14, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Extensible syntax systems can be powerful programming tools. They allow programmers to create domain-specific languages, capture common patterns of code generation and implement complex user languages in terms of simple kernel languages. Unfortunately, traditional extensible syntax systems, like the C preprocessor, suffer from severe usability problems. Previous work on hygienic macro systems solved the usability problems associated with the use of identifiers by syntax extensions. This work develops extensions to hygienic macro technology that solve the problems associated with the static analysis of programs that use syntax extensions. These extensions have been used to build a macro system for a Scheme-like language that statically analyzes programs using Hindley-Milner type inference. The distinctive feature of this macro system is that macros can report type errors about their uses in terms of original source code instead of macro-expanded code.

Thesis Supervisor: Olin Shivers
Title: Research Scientist, Massachusetts Institute of Technology

Thesis Co-supervisor: Alan Bawden
Title: Research Scientist, Boston University

Acknowledgements

First and foremost, I would like to thank Olin Shivers and Alan Bawden, my thesis supervisors. They were an essential driving force behind this project and without their guidance, insight, suggestions and support this system would never have come as far as it has. Their patience and feedback while I struggled with writing this document were also invaluable. Second, I would like to thank the many teachers who have inspired me over the years. Though I cannot do justice to them individually here, I cannot let my debt go unacknowledged. Without their inspiration, I would never have found the dreams that I have chosen to achieve, including this one. Third, I would like to thank Anne Hunter for all of the help she has given me with the bureaucratic side of preparing this thesis. Last but not least, I would like to thank my parents. I could not imagine my life without their love and unwavering support.

Contents

List of Figures

Chapter 1

Introduction

Extensible syntax systems can be powerful programming tools. They allow programmers to create domain-specific languages, capture common patterns of code generation and implement complex user languages in terms of simple kernel languages. Unfortunately, traditional extensible syntax systems, like the C preprocessor, suffer from severe usability problems. Previous work on hygienic macro systems solved the usability problems associated with the use of identifiers by syntax extensions. This work develops extensions to hygienic macro technology that solve the problems associated with the static analysis of programs that use syntax extensions. These extensions have been used to build a macro system for a Scheme-like language that statically analyzes programs using Hindley-Milner type inference. The distinctive feature of this macro system is that macros can report type errors about their uses in terms of original source code instead of macro-expanded code.

1.1 Extensible syntax

1.1.1 Motivation

Extensible syntax systems have a long history. They began with the macro systems of early assemblers. These macro systems allowed assembly-language programmers to define new instructions, or macros, that expanded into sequences of more primitive machine instructions. Assembly-language programmers used macros to structure their programs and to make them easier to understand. Macro assemblers were eventually marginalized by high-level languages, but high-level language programmers found their own uses for extensible syntax [17].

The most widely used extensible syntax system is the C preprocessor, cpp. C programmers typically use cpp for a variety of straightforward tasks: controlling compilation, defining constants, hiding platform details, abbreviating frequently used code sequences, and so on. However, the preprocessor is a much more powerful tool. A book on scientific computing, *Numerical Recipes in C*, uses the preprocessor to create domain-specific extensions to C that are used in its example programs [16]. The C preprocessor has also served as a vehicle for language experimentation. For

instance, early versions of "C with Classes" (which became C++) were implemented using cpp [20].

1.1.2 Problems

The power of the C preprocessor reflects the power of extensible syntax. Unfortunately, the problems of the C preprocessor also reflect the problems faced by extensible syntax systems. cpp is known as a source of subtle and hard-to-find bugs. In fact, the usability problems of cpp are so severe that one of the influences on the design of C++ has been a desire to eliminate the need for a preprocessor [20]. One class of usability problems faced by cpp macros involves the identifiers used in a macro. A cpp macro that swaps two integers (defined in Figure 1-1 and used in Figure 1-2) illustrates the problem.

```
#define swap(a,b) { \
  int temp = (a); \
  (a) = (b); \
  (b) = temp; \
}
```

Figure 1-1: A C preprocessor **swap** macro

The intended meaning of the macro use in Figure 1-2 is straightforward. The code should swap the temperatures in the variables **temp** and **high_temp** if today's temperature (**temp**) exceeds the high temperature (**high_temp**). If the definition of the **swap** macro is not taken into account, the code looks correct.

```
int temp = today.weather.temperature;
if (temp > high_temp) swap(high_temp, temp);
```

Figure 1-2: An erroneous use of the C preprocessor **swap** macro

Unfortunately, when the definition is taken into account, the preprocessed version of the code (Figure 1-3) reveals a problem. There is a conflict between the variable **temp** used to store today's temperature and the internal variable **temp** used by the **swap** macro. This conflict prevents the temperatures from being swapped. This is a problem with the **swap** macro itself because changes in programs that use the **swap** macro (using or not using a variable named **temp**) can unexpectedly change the macro's behavior. Users of the C preprocessor try to avoid this class of problems by choosing obscure names for the identifiers that their macros introduce. This is a brittle solution that makes their macros harder to understand and maintain.

As mentioned earlier, one approach to the problems of the C preprocessor is to eliminate the need for a preprocessor altogether. Along those lines, C++ has introduced constants, templates, inline functions, and namespaces as new language features that replace common uses of the preprocessor. One problem with this approach is that it results in a complex, kitchen-sink language that is difficult to specify and implement. Another problem is that the other benefits of syntax-extension are lost:

```
int temp = today.weather.temperature;
if (temp > high_temp) {
  int temp = (high_temp);
  (high_temp) = (temp);
  (temp) = temp;
};
```

Figure 1-3: The preprocessed C code corresponding to Figure 1-2

language experimentation becomes much more difficult and domain-specific languages cannot be created by users.

A better approach to these problems is to create improved syntax-extension technology to solve them. This is the approach taken by *hygienic macro systems* [2, 4, 5, 7, 8, 12]. A hygienic macro system gives macro writers some control over the resolution of the identifiers that their macros use. This control enables the creation of syntax extensions whose behavior does not depend on the programs in which they are used[1]. For example, a hygienic version of the **swap** macro would manipulate the identifier for its internal temporary variable so that it would never conflict with any program variable.

1.2 Static analysis

1.2.1 Motivation

Static program analyses are programs that statically deduce information about other programs. They can be powerful programming aids. Perhaps the most familiar form of static program analysis is type-checking. From one point of view, a type-checker is a program that verifies that a programmer's mental model of a program (expressed through the types that program uses) is consistent. If the model is not consistent, a type-checker explains the inconsistencies. Type-inference is an advanced form of type-checking, that deduces the types of program expressions whenever possible, relieving programmers of the burden of explicitly specifying their model. Other static analyses enhance compilation. For instance, data-flow analyses gather information that compilers can use to generate optimized output code.

1.2.2 Problems

Unfortunately, static analyses do not interact well with extensible syntax systems. The fundamental problem is that traditional static analyses do not understand the extended syntax a program may use. This means that they must operate on the expanded version of a program, where all the uses of syntax extensions have been

[1]Hygienic macro systems can also enable the creation of syntax extensions that interact with program identifiers in controlled ways. For example, a syntax extension could create a new binding for some specific variable.

replaced by the core language constructs that implement those extensions. Unfortunately, the expanded version of a program is usually larger, more complicated and less precise than the original program. This means that it is more difficult for a static analysis to recover information from the expanded version. In some instances, information that is easy to recover in the original program might even be impossible to recover from the expanded program. Even worse, because the expanded program is less precise, any information that is recovered tends to be less useful than the information that can be recovered from the source program. For an example of this class of problem, look at the use of the cpp **swap** macro in Figure 1-4.

```
float c_temp = today.weather.temperature;
if (c_temp > high_temp) swap(high_temp, c_temp);
```

```
test.cpp:32:  warning:  initialization to 'int' from 'float'
```

Figure 1-4: A use of the **swap** macro that generates a type warning

This use of the **swap** macro has changed a program identifier to avoid identifier conflicts, but there is a new problem. When this code is compiled, it generates a confusing type warning about a conversion between an integer and a float. This warning is confusing because it is not in terms of the source code that was written. That source code does not directly contain a conversion from an integer to a float. The underlying problem is that the temporary variable used by **swap** has integer type, so the **swap** macro should not be used to swap floating-point numbers[2]. Unfortunately, the information that would help a programmer understand this problem, that the **swap** macro only supports swapping integers, is not available in the macro-expanded version of the program because there are no references to **swap** at all in that version. This means the only way for a programmer to understand this problem, as in the case of the identifier conflicts discussed earlier, is for the programmer to study the source code for **swap** directly, destroying the abstraction **swap** is meant to provide.

This particular example might seem trivial, but this kind of problem quickly becomes severe as the macros involved become more complex. It might be reasonable to expect programmers to study the **swap** macro in order to understand problems related to it, but programmers do not want to study the macros that implement a powerful set of complex syntax extensions to understand problems that arise when those extensions are used. Instead, as with the identifier conflicts described earlier, programmers limit their use of syntax extensions to manage this class of usability problems. Again, a better solution would be to extend syntax-extension technology. In this case, improved extension technology should include extensible static analyses. Extensible static analyses are analyses that can be extended to analyze syntax extensions directly. This means an extensible static analysis can analyze a program in terms of its original source code, so its results are more precise and more useful than an analysis of the expanded version of that program.

[2]If **swap** is used to swap two floating-point numbers, the results will depend on how floating-point numbers are converted to integers and back again.

In the particular case of type-checking, an extensible type-checker should enable the creation of type-aware macros. A *type-aware* macro is a macro that is aware of the constraints a type system imposes on its uses and can control the errors those constraints generate. A minimally type-aware version of the **swap** macro above would report an error whenever it was used to swap two non-integers. This is a clearer and more useful error message than the type warning above, but ultimately unsatisfying because swapping is a concept that can be applied to more than integers. A more advanced type-aware **swap** macro could use type information about the variables being swapped to control the type of its internal variable, and only report an error when the variables swapped are of incompatible types.

1.3 Thesis

My thesis is that extensible static analyses can be successfully combined with hygienic macro systems. This combination makes it possible to bring the full benefits of syntax extension (including language experimentation, modular language design and domain-specific languages) to programming languages with significant static analyses. This is important because many programming languages have a significant static analysis, usually some form of type-checking. For those programming languages, this technology can be used to create type-aware macros, macros that report type errors in terms of original source code rather than macro-expanded code.

I will prove my thesis by describing the syntax-extension framework that I have built and showing how I can use this framework to build a hygienic, type-aware macro system for a statically-typed programming language. My demonstration language will be a variant of Scheme typed using Hindley-Milner type-inference. Since Hindley-Milner type-inference is a complex static analysis and my framework is not analysis-specific, this demonstration will also show that my framework supports arbitrary extensible static analyses.

1.3.1 do

The macro I have written that best illustrates the promise of extensible type-checking (and, by implication, extensible static analyses in general) is a variant of the Scheme `do` macro [9]. I have used my type-aware macro system for a statically-typed dialect of Scheme to write this type-aware version of `do`. The implementation is given in Appendix A. The exciting thing about this macro is that it reports specific type errors in terms of the `do` expression itself. For example, in Figure 1-5 (where the variable `x` is bound to an integer list), the macro precisely finds the mistake in a loop meant to sum the elements of a list. This is a dramatic improvement over a version of `do` that is not type-aware, as in Figure 1-6.

```
Expression: (do ((x x (car x)) ;; BUG: car should be cdr
                 (sum 0 (+ sum (car x))))
                ((null? x) sum))

Type inference failed
do : types of variable and step expression incompatible
Type clash between:
(listof int)
int
In form: (x x (car x))
```

Figure 1-5: A type-aware do macro generates precise and useful type-error messages

```
Expression: (do ((x x (car x))
                 (sum 0 (+ sum (car x))))
                ((null? x) sum))

Type inference failed
Inconsistent variable types in letrec bindings
Type clash between:
(-> (int int) int)
(-> ((listof int) int) int)
In form:  (letrec ((do-loop.294
                    (lambda (x.297 sum.297)
                      (if (null? x.297)
                          sum.297
                          (do-loop.294 (car x.297)
                                       (+ sum.297
                                          (car x.297)))))))
            (do-loop.294 x.293 0))
```

Figure 1-6: A non-type-aware do macro generates vague and unhelpful type-error messages

1.3.2 Structure of thesis

The first step towards proving this thesis is building a framework capable of supporting syntax extensions and extensible static analyses. There are three primary design requirements I have for this framework:

1. It must be a complete syntax-extension framework, including support for hygienic syntax extensions, so that it is not a step backward in syntax-extension technology.
2. It must be a general system, capable of supporting any reasonable static analysis, so that the framework can be reused even as analyses change.
3. It must support languages with many different language elements, so it can be used with languages more complex than dialects of Scheme.

Chapter 2 describes the design and implementation of a framework that meets these design requirements. The following chapter describes how I use the framework to implement a demonstration language and its associated static analysis. Chapter 4 builds three different syntax-extension interfaces on top of that implementation and discusses the strengths and weaknesses of each interface. The remaining chapters describe related work and potential directions for future research based on this syntax-extension framework and its associated language implementation and syntax-extension interfaces.

Chapter 2

Syntax-Extension Framework

This chapter discusses the design and implementation of my syntax-extension framework. As discussed in the introduction, this framework is the infrastructure I use to build syntax-extension interfaces that solve the problems faced by syntax extensions in the presence of static analysis. The first section of this chapter discusses a simplified version of my framework. By focusing on the simpler version, my framework's essential features are easier to describe. The second section of this chapter uses this simplified framework to implement an extension system that is used with a toy static analysis of Scheme. The following section adds the features necessary to resolve hygiene problems to the simple framework. Without these features, the framework would be a step backwards from existing Scheme macro technology. The next section generalizes the framework to support languages with arbitrary kinds of language elements. The final section discusses some additional features of the framework that do not fit in the main line of discussion.

2.1 Simple framework

This section describes a simplified syntax-extension framework that is the core of the complete framework. It is simplified in two different respects:

- It has no mechanisms for controlling conflicts between identifiers in the source program and identifiers that might be introduced by syntax extensions. This means that it is impossible to write hygienic syntax extensions. These mechanisms will be added to the system in section 2.3.
- The framework can only represent languages that have two kinds of language elements — keywords and expressions. The framework will be generalized to support arbitrary kinds of language elements in section 2.4.

This simple version of my syntax-extension framework was developed by studying an existing Scheme macro system written by Bawden [1]. Like that system, my framework is organized as a compiler written in a dialect of Scheme (Scheme 48) [11]. The compiler has two phases. In the first phase, an input program, in some source language

(represented using S-expressions), is processed to create an abstract representation of that program. This abstract representation includes semantic information that is used by static analyses that are performed on the program. In the second phase, the abstract representation is used to create an output version of that program, for some target machine. To understand the features of this framework, imagine that it is being used to implement a syntax-extension system for Scheme (as it will be in the next section).

2.1.1 Nodes

The abstract representation of a program is too complicated to construct directly. Instead, input programs are recursively decomposed into *forms*, the distinct syntactic entities whose meanings will determine the meaning of a program. The details of the recursive decomposition process will be explained later. For Scheme, examples of forms include:

- primitive expressions like `5` and `car`,
- syntactic keywords like `define` and `if`,
- and compound expressions like`(+ x 1)` and `(if (> y 0) y (- y))`.

Nodes are the objects that are used to abstractly represent forms. Nodes are represented as structures that contain three fields: a kind field, a converter field and a data field. The roles of these fields are explained below:

kind The kind field of a node contains a symbol that distinguishes the kind of form the node represents. In the simplified framework, only two kinds of forms are distinguished — keywords (which have the kind `keyword` and can represent Scheme's syntactic keywords) and expressions (which have the kind `exp` and can represent all the other elements of a Scheme program). The procedure `node/kind` is used to extract the kind of a node.

converter The converter field of a node contains a conversion procedure. A node's conversion procedure is used to create a "lower-level", or *converted*, representation of that node. Conversion procedures are used to implement the second phase of the compiler described above. In particular, the converted representation of the node that represents the entire program is a target-machine version of that program. Nodes are converted using the procedure `node/convert`. `node/convert` extracts the conversion procedure from the node to be converted and uses it to create the converted representation. The first argument to `node/convert` is the node to be converted. The rest of the conversion protocol is determined by the kind of the node. For Scheme, the conversion procedure for an expression node would generate output code that implements that expression. The conversion procedure for a keyword node is a keyword procedure that will be explained in the next subsection.

attributes The *attributes* of a node organize the semantic information associated with that node. Each attribute of a node is named by a symbol. This symbol is used to access (or modify) the attribute it names through the procedures `node/attribute-get` and `node/attribute-set!`. For expression nodes, a useful attribute might be a `variety` attribute that indicates the particular variety of expression a node represents (variable, constant, conditional, sequence, etc.). Some attributes of a node might be maintained to help conversion. In general, however, most of the attributes of a node will be defined by the static analyses that are implemented for a particular language. For instance, a type-checking static analysis could define a `type` attribute to store the deduced type of an expression[1]. The kind of a node will determine at least a minimal set of attributes that the node provides[2].

The specific features of nodes were chosen to so that nodes provide a reusable framework for extensible syntax systems. The possible set of node attributes is left unspecified so that no particular static analysis is built into the framework. Instead, as will be described later in the chapter, authors of static analyses can define their own attributes and use them to create interfaces between their static analyses and syntax extensions. Nodes construct their converted representations by using conversion procedures because semantic or contextual information that is not available when a node is constructed might be required to convert that node. For example, some conversion procedures could use information about variable references (gathered through a static analysis) to eliminate dead code in the converted representations they return. Conversion procedures enable nodes to "promise" that they can be converted in the future (during the second phase of compilation), without having to construct their converted representation in the present (during the first phase). This means that compilation strategies that use the results of static analyses are possible. Also, since the protocols implemented by node conversion procedures are left unspecified[3], the framework can be used with different conversion strategies, source languages and target machines.

2.1.2 Process

The `process` procedure is the procedure that recursively decomposes programs into forms. It implements the recursive decomposition so that it can construct nodes that

[1] In some type systems, including Hindley-Milner type inference, it is not convenient to deduce the type of an expression when constructing its corresponding node. In that case, a type analysis could define an attribute that will compute the type of an expression at a later time.

[2] At this point, it should be clear the kind field of a node is actually a type that determines the organization of the rest of the node object. Given that, it might seem natural that this framework should be implemented in an object-oriented style where different kinds of nodes become different types of objects. The primary reason the system was not built in a more directly object-oriented style is that Scheme 48 (my implementation language) does not come with a good object system. Also, it is not clear whether or not an inheritance hierachy could be constructed that would correctly capture all of the important relationships between different kinds of nodes.

[3] Except for the keyword procedures described in the next subsection, which are used to implement extensible syntax.

represent the meanings of these forms. It will be revised extensively as the extension framework is improved. Figure 2-1 contains the source code for the simplest version of process.

```
(define (process form env)
  (cond ((symbol? form)
         (environment/lookup env form
                             (lambda (var)
                               (syntax-fail "Unbound variable"
                                            var))))
        ((constant? form)
         (make-constant-node form))
        ((list? form)
         (let ((op-node (process (car form) env)))
           (if (eq? (node/kind op-node) 'keyword)
               (node/expand op-node form env)
               (make-combination-node op-node form env))))
        (else (syntax-fail "Unknown syntax" form))))
```

Figure 2-1: The initial version of process

This version of process takes two arguments — a form and a syntactic environment. It returns a node that represents the meaning of the input form. process is also responsible for checking the syntax of the input form. For process itself, this just means signaling a syntax error (using the procedure syntax-fail) when it encounters an input form it does not recognize. The syntactic-environment argument is used to determine the meanings of forms that are symbols (see section 2.1.3). For constant forms, the meaning is constructed using the procedure make-constant-node. For Scheme, this procedure would construct a node whose conversion procedure returns the code for a constant expression in the target language (in some manner determined by that target language).

The part of process that enables syntax extension is the part that determines how to construct the meaning of forms that are lists. The first step in constructing the meaning of a list form is using process recursively to determine the meaning of the first element of that list[4]. If the node representing the meaning of the first element is not a keyword node, the meaning of the input form is constructed by make-combination-node. For Scheme, this procedure would construct a node that can be converted into target-language code that implements the appropriate procedure call. The interesting case is when the first element's node is a keyword node. In that case, the conversion procedure of the keyword node (the keyword procedure mentioned earlier) is used as a "virtual subroutine" of process. It is called with the arguments of process (the form and the syntactic environment), and has all responsibilities of process — returning a node that represents the meaning of the form and checking the syntax of the input form. This special behavior of process enables syntax extension because when process "finds" new keyword nodes, the syntax of the language has been extended.

[4]The empty list is considered a constant, so the list must have a first element.

Figure 2-2 contains skeleton code that illustrates what constructing a keyword node to implement Scheme `if` expressions might look like.

```
(define $if
  (make-keyword-node
   (lambda (form env)
     (if (not (= (length form) 4))
         (syntax-fail "If expression incorrect size" form))
     (let ((test-node (process (cadr form) env))
           (then-node (process (caddr form) env))
           (else-node (process (cadddr form) env)))
       (make-exp-node
        ...
        )))))
```

Figure 2-2: Skeleton code to implement a Scheme `if` expression

The procedures `make-keyword-node` and `make-exp-node` (used in the skeleton code) are shorthand for creating keyword nodes and expression nodes, respectively. The only argument to `make-keyword-node` is the associated keyword procedure. The arguments to `make-exp-node` depend on the target language and any static analyses that may be performed, so the call is left unspecified here. The first part of the `if` keyword procedure performs syntax checking (lines 4–5). The next part of the keyword procedure recursively constructs the nodes for the parts of the `if` expression. In a complete keyword procedure, these nodes would be used to construct the expression node that is returned. These calls are performed before the call to `make-exp-node` because they also perform necessary syntax checking. These calls use `process` recursively, so they check the syntax of the subexpressions of the `if`.

2.1.3 Syntactic environments

The only part of `process` that has not been completely explained is how meanings are determined for forms that are symbols. As mentioned above, the syntactic-environment argument to `process` is used to determine the meanings of symbols that are processed. Specifically, syntactic environments bind symbols to nodes that represent the compile-time meaning of those symbols. These nodes are returned when those symbols are looked up by `process`. Syntactic environments are represented as structures that contain three fields: a parent field, a frame field and an envid field. The fields of an environment are used as follows:

parent This field stores the parent environment of the current syntactic environment. If a symbol is unbound in the frame of the current syntactic environment, it is looked up in the parent environment. This allows syntactic environments to implement nested scopes. For Scheme, these scopes would be lexical scopes. If the parent environment is `#f`, then the environment has no parent (it would be a global syntactic environment).

frame The frame field of a syntactic environment contains a table that stores the bindings of the innermost frame of that syntactic environment. If a symbol is bound in the frame, that binding overrides any binding that might be found by looking in the parent syntactic environment.

envid The envid field of a syntactic environment contains a number that is different for each syntactic environment. This number can be used to generate distinct output code for the same symbol in different syntactic environments. One reason this might be necessary is that the target language might not have the same kind of scoping as the source language, so symbols which do not conflict in the source might conflict in the output without this ability to distinguish them. This field is also used in implementing support for hygienic syntax extensions, discussed in section 2.3.

Syntactic environments are accessed and manipulated through the procedures `environment/lookup`, `environment/bind!`, and `environment/extend`. The procedure `environment/lookup` was used in the first version of `process`. It takes three arguments: the syntactic environment, the symbol to look up, and a procedure to call if that symbol is unbound. In the case of `process` that third argument is used to signal a syntax error (see Figure 2-1). `environment/bind!` takes four arguments: the environment, the symbol to bind, the node to bind it to, and a procedure to call if the symbol is already bound in the innermost frame of that environment. `environment/extend` takes one argument, an existing syntactic environment, and returns a new syntactic environment with an empty innermost frame. The parent of the new syntactic environment is the environment supplied.

At this point, it is possible to see the role syntactic environments play in syntax extension. In the previous discussion of `process`, syntax extensions became available when `process` "found" new keyword nodes while processing. `process` can "find" new keyword nodes if they are added to an existing syntactic environment by parts of a language implementation. Specifically, language implementations provide syntactic environments that bind symbols to keyword nodes that implement the core syntax of a particular language. For example, implementations of Scheme's core syntax would provide a syntactic environment that binds `lambda`, `if`, `define`, `set!`, and `quote`. Other syntactic environments could implement Scheme's derived expressions in terms of the core syntax. Implementations of `define-syntax`, `let-syntax`, and `letrec-syntax` would use `environment/bind!` (on the appropriate syntactic environment) to bind new keyword nodes for `process` to "find."

Finally, it is important to note that syntactic environments store more than syntactic keywords. They also store the meaning of non-keyword symbols. These non-keyword symbols follow the same scoping rules that syntactic keywords do. This is probably what is desired for most language implementations because multiple sets of scoping rules are confusing for programmers. In the implementation described so far, syntactic keywords and non-keyword symbols share the same namespace. This is exactly what is desired when implementing Scheme because Scheme keywords and Scheme variables (the non-keyword symbols that will be bound for Scheme) are sup-

posed to share a single namespace. This may not be desired when implementing other languages. A mechanism for avoiding such conflicts is described in section 2.4.2.

2.2 Example: Analyzing Scheme code

This section describes how to use the simplified framework to implement a syntax-extension system that is used with a toy static analysis of Scheme. To motivate my example static analysis, consider implementing a source-code debugger for compiled Scheme programs. When compiling a debugging version of a program, the target language would connect object code to its associated source code. For this to be possible, however, the code generators must have access to the source form of an expression. In the presence of syntax extensions, this may not always be easy to guarantee, but a static analysis could gather that information. This is a trivial static analysis, and an improved version of `process` does this automatically (see section 2.5.2), but its implementation will serve to illustrate how my framework supports implementing and extending static analyses. These same techniques will be used in Chapter 3 to implement Hindley-Milner type inference for my demonstration language.

The following steps describe how to use the framework to implement a static analysis for some source language (these steps will generalize to the full framework):

1. You should decide how to represent the various elements of your language using nodes, attributes and syntactic environments.

2. You should write keyword procedures that implement the core syntax of your language, including generating any attributes that your language's static analyses need. You should also construct the syntactic environments that your language needs and bind your keyword procedures (and any other necessary nodes) appropriately in those environments.

3. At this point, if your goal is create an extended user syntax implemented in terms of your core syntax, then all you need to do is write the keyword procedures for the extended syntax and bind them appropriately. If, on the other hand, you want to export a syntax-extension interface, you need to decide on an interface for creating and binding keyword procedures. The simplest interface to implement is to allow users to create keyword procedures by writing Scheme 48 code that directly manipulates nodes and attributes. However, for most language representations, this is a cumbersome and hard-to-use interface. You should try to develop simpler interfaces for syntax extension[5].

[5]This is easier said than done. See Chapter 4 for a discussion of the challeges in creating an easy-to-use syntax extension interface that allows users to improve static analyses.

2.2.1 Representing Scheme

To review, these are the key ways the framework of the previous section can be used to represent Scheme[6]:

- The meaning of Scheme language expressions can be represented using expression nodes. As mentioned before, expression nodes are constructed with the procedure `make-exp-node`. For this language and static analysis, `make-exp-node` takes three arguments:

  1. An expression variety — a symbol describing the variety of Scheme expression (conditional, assignment, constant, etc.) that the expression node represents. This is stored in the `variety` attribute of the expression node.
  2. A conversion procedure that would generate target-machine code that corresponds to the Scheme expression. To simplify the presentation, the conversion procedures will be omitted from calls to `make-exp-node`[7].
  3. An S-expression that is the source language form corresponding to the expression that the node represents. This will be stored in the `form` attribute, and the goal of the static analysis is to put the correct S-expression in the `form` attribute for each expression node.

- The nested scopes provided by syntactic environments can be used to represent Scheme's lexical scoping.

- Nodes that implement program variables and syntactic keywords are bound in these syntactic environments. Keywords and variables are bound in the same syntactic environment, so they share the same space of identifiers, as they should (for Scheme).

2.2.2 Implementing Scheme's core syntax

At this point, I have completed step 1 of implementing a static analysis for Scheme. The next step is to implement the core syntax of my source language (Scheme), omitting the conversion procedures (as discussed previously). Two different things must be done to implement the core syntax of Scheme:

- The procedures `make-constant-node` and `make-combination-node` must be written[8].

[6]The syntactic restrictions on definitions required by Scheme cannot be naturally implemented using the simplified framework. Implementing those restrictions using the complete framework will be discussed in section 2.4.4.

[7]Including the converters would involve discussing the debugger that would use the source code information that the analysis collects — an unnecessary distraction from discussing the framework itself.

[8]In the simplified system, these procedures are hardwired into `process`, so it is not clear why an implementation of Scheme's core syntax should provide them. They are being discussed because, in the full system, these procedures will not be hardwired into `process`, so any core syntax implementation will be required to provide them.

- The keyword procedures, which implement the rest of Scheme's core syntax must be written and bound in an appropriate syntactic environment.

The implementation of `make-constant-node` will be omitted because it trivial without a conversion procedure. Figure 2-3 contains the implementation of `make-combination-node`.

```
(define (make-combination-node op-node form env)
  (let ((operand-nodes (map (lambda (op-form)
                              (process op-form env))
                            (cdr form))))
    (make-exp-node
     'combination
     ;;; Omitted conversion procedure
     (map get-form (cons op-node operand-nodes)))))
```

Figure 2-3: Implementation of the `make-combination-node` procedure for the example static analysis

The first thing to notice about `make-combination-node` is that it is not a keyword procedure. Instead, it is a procedure hardwired into `process` for processing combinations whose operator does not represent a keyword. It takes three arguments: the operator node, the form that represents the combination and the syntactic environment in which to process that combination. The operator-node argument allows `make-combination-node` to take advantage of the work that `process` has already done in processing the operator of the combination. The procedure uses its `form` argument to construct the nodes that represent the operands of the combination, as shown on lines 2–4. The source-code form corresponding to the combination is constructed on line 8. The `get-form` procedure is convenient shorthand for a call to `node/attribute-get` that extracts the `form` attribute of a node. By mapping `get-form` over a list that contains the operator node and the operand nodes of the combination, the source code of the combination is reconstructed. It would be simpler to use the argument `form` directly, instead of reconstructing the source code. This implementation, however, demonstrates how nodes that represent parts of a form can be used to construct attributes for the entire form. This will be important, in Chapter 3, when the static analysis (Hindley-Milner type inference) is more difficult than source-code reconstruction.

The other part of implementing Scheme's core syntax is writing the keyword procedures that implement Scheme's primitive special forms. Figure 2-4 contains one example of a keyword implementation. The source code in that figure creates a keyword node for `lambda` that reconstructs the source form of a `lambda` expression.

The first thing to notice in that source code is the call to `make-keyword-node` on line 2. As discussed earlier, this call creates the keyword node from the supplied keyword procedure. On lines 4–8 there is code that checks the syntax of the `lambda` expression. This code only checks the surface syntax of the `lambda` expression. As in the skeleton keyword procedure for `if` (Figure 2-2), syntax-checking on subexpressions of the `lambda` will be performed by recursive calls to `process`.

```
(define $lambda
  (make-keyword-node
   (lambda (form env)
     (if (or (< (length form) 3)
             (not (var-list? (cadr form))))
         (syntax-fail
          "Illegally formatted lambda expression"
          form))
     (let* ((vars (cadr form))
            (body (cddr form))
            (new-env (environment/extend env))
            (var-nodes (map
                        (lambda (var)
                          (make-variable var new-env))
                        vars)))
       (for-each
        (lambda (var node)
          (environment/bind!
           new-env var node
           (lambda (var)
             (syntax-fail
              "Duplicate variable use in parameter list"
              form))))
        vars
        var-nodes)
       (let* ((body-node (make-sequence-node
                          (process-body body new-env)
                          form))
              (body-sequence (exp-node-sequence body-node)))
         (make-exp-node
          'abstraction
          ;; Omitted code generator that uses body-node
          `(lambda ,(map get-form var-nodes)
             ,@(map get-form body-sequence)))))))))
```

Figure 2-4: Implementation of the `lambda` keyword for the example static analysis

The `let*` bindings on lines 9–15 bind variables that are used to create the `lambda` node itself. The variable `vars` stores the symbols that name the parameters of the procedure that will be constructed. The variable `body` holds the list of forms that form the body of the `lambda` expression. The procedure `environment/extend` is used to create an extended syntactic environment for the body of the `lambda` expression, implementing lexical scoping for the `lambda`. That syntactic environment is stored in the variable `new-env`. Finally, the list `var-nodes` holds the list of variable nodes that the variable identifiers will be bound to in the environment `new-env`. These variable nodes are ordinary expression nodes of the expression variety `variable`. They are created using the procedure `make-variable`, which can create a variable node given a symbol and a syntactic environment. The syntactic environment should be the environment in which the new variable node will be bound. It is used by that node to generate distinct code even for variables with the same symbol, as discussed in section 2.1.3.

The `for-each` statement (lines 16–25) following the bindings completes the process of setting up the syntactic environment for the body of the `lambda` expression. Each of the variable identifiers is bound to the variable node that was created from it using the procedure `environment/bind!`. If the same identifier is bound more than once, it is detected by `environment/bind!` and a syntax error is signaled. Otherwise, the new syntactic environment is ready to be used to create the body of the `lambda` expression.

The final part of the keyword procedure constructs the output expression node. Lines 26–29 bind the variables needed. First, they bind the node that represents the body of the `lambda` to the variable `body-node`. This node is created using the procedures `process-body` and `make-sequence-node`. The procedure `process-body` uses `process` to process a list of forms that represent a list of expressions that will be executed sequentially. It is the procedure that detects and flattens nested expression sequences, and it returns the flattened list of expression nodes. The call to `make-sequence-node` creates a sequence node from that list of expression nodes. Then, the procedure `exp-node-sequence` is used on the sequence node to recover the list of nodes that represents the body of the `lambda` expression. That list is stored in the variable `body-sequence`. Finally, on lines 30–34, `make-exp-node` is called. As in the implementation of `make-combination-node` (Figure 2-3), the S-expression source code is reconstructed from the nodes that represent parts of the `lambda` expression.

2.2.3 Adding syntax extensions

Assuming keyword procedures are written for the other parts of the core syntax of Scheme, step 2 of the plan from the beginning of this section is also complete. The final step is to develop an interface for defining syntax extensions. As discussed in section 2.1.3, all syntax extensions ultimately end up binding new keyword nodes in some syntactic environment. Once the new keyword nodes have been bound, code processed in that syntactic environment can use the extended syntax. Keyword nodes implement syntax extensions through the keyword procedures that they contain. Therefore, to understand the features and limitations shared by all syntax-extension interfaces built

```
(define-syntax 1+
  (lambda (form env)
    (if (not (= (length form) 2))
        (syntax-fail "1+ form incorrect size" form))
    (let* ((arg (cadr form))
           (output-node (process `(+ ,arg 1) env))
           (real-form `(1+ ,arg)))
      (set-form! output-node real-form)
      output-node)))
```

Figure 2-5: Syntax definition for 1+, first version

using this framework, it is only necessary to study a "raw" interface where the user directly writes the keyword procedures that are used to construct keyword nodes. Any other interface would end up transforming user input into a keyword procedure to be bound, so it will share the features and limitations of the "raw" interface.

To make this discussion concrete, for the rest of the chapter, syntax extensions will be defined with the following `define-syntax` form: `(define-syntax <keyword> <keyword-proc>)`, which binds the identifier `<keyword>` to a keyword node whose keyword procedure is the *Scheme 48* procedure created by evaluating `<keyword-proc>`. Although both the source language being represented and the language that keyword procedures are written are dialects of Scheme, it is important to remember that they are distinct languages. For this framework, the source language could be any language with S-expression syntax, not just Scheme. For the simplified system, `<keyword-proc>` is evaluated in a Scheme 48 environment that contains special bindings for syntax-extension keyword procedures (as opposed to core-syntax keyword procedures).

The Scheme 48 environment in which syntax-extension keyword procedures are evaluated is closely related to the environment in which core-syntax keyword procedures are evaluated. The most important difference between the two environments is that the procedures that create particular kinds of nodes (like `make-exp-node`) are not available in the syntax-extension environment. This means syntax extensions must use `process` to create the node that represents the meaning of their extended syntax. This output expression node will be converted based on the form that is processed, so syntax extensions will not directly depend on the details of conversion (code generation). To create that output node, `process` requires a form (as well as a syntactic environment). This requirement means that syntax extensions are constrained to implement themselves in terms of existing syntax, as they should be.

A simple syntax definition that defines a macro 1+ (that adds one to a single expression) is given in Figure 2-5. This 1+ macro, if used legally, builds a node that represents adding one to the body of a 1+ expression. This can be seen from the call to `process` on line 6 of the syntax definition. The call to `set-form!` on line 8 is where the macro semantically improves its output node. The procedure `set-form!`, like `get-form`, is convenient shorthand for node manipulation. In this case, `set-form!` is shorthand for a call to `node/attribute-set!` to set the `form` attribute of an expression node.

The source form used to improve the output node is constructed on line 7 of Figure 2-5. It gets the source code of the argument form of the 1+ macro directly from the variable `arg`. The variable `arg` gets its source code from the input variable `form` (on line 5 of Figure 2-5). This is not the right way for a macro to construct semantic information for a static analysis. The only reason it works is because the static analysis under consideration is source-code reconstruction — and the source form of an expression is easy to access. It will not generalize to other static analyses, such as type-checking, where semantic information is harder to construct. The following subsection explains how to fix this problem with the 1+ macro, using the technique of node insertion.

2.2.4 Fixing the 1+ macro: Node insertion

The problem with the 1+ macro from the previous subsection is that it is not constructing its semantic information recursively. What the 1+ macro should be doing is extracting a source form from a node that represents the argument expression of the 1+ macro, the way `make-combination-node` and the keyword procedure for `lambda` extract source code from nodes that represent their parts (Figures 2-3 and 2-4). The reason this is difficult is because the node the 1+ macro needs to access is a subnode of the output node that represents the macro-expanded code, and that output node does not provide access to its subnodes[9]. The desired subnode could be constructed by processing the argument form separately, but, in the system described so far, there is no way to connect that node to the output node of the keyword procedure. One solution would be to process the body form twice: once to extract the semantic information and once (as part of a larger form) to create the output node. This is not a good idea for two reasons:

1. Equivalent body nodes should be created both times, so it is inefficient to process it more than once (especially as the body form might use other macros that could also be redundantly processing their subforms).

2. The semantic information from the first body node might not be consistent with the semantic information with the second body node. For example, constructing the semantic information for the body node might involve side effects[10]. In that case, incorrect semantic information might break (or at least fail to improve) a static analysis.

As suggested earlier, a better solution is to provide a way to connect the body node with the output node of the keyword procedure. The second version of `process`, given in Figure 2-6, does that. The only difference between the two versions of `process` is

[9]In fact, other than sequence nodes, no nodes provide access to their subnodes. This is a feature to consider adding in a future version of the framework.

[10]One example of a static analysis that uses side effects to construct its semantic information is the implementation of Hindley-Milner type inference in Chapter 3. Side effects are used to keep track of the constraints on the types of source-code variables.

```
(define (process form env)
  (cond ((node? form) form)
        ((symbol? form)
         (environment/lookup
          env
          form
          (lambda (var)
            (syntax-fail "Unbound variable" var))))
        ((constant? form)
         (make-constant-node form))
        ((list? form)
         (let ((op-node (process (car form) env)))
           (if (eq? (node/kind op-node) 'keyword)
               (node/expand op-node form env)
               (make-combination-node op-node form env))))
        (else (syntax-fail "Unknown syntax" form))))
```

Figure 2-6: A version of process that supports node insertion

```
(define-syntax 1+
  (lambda (form env)
    (if (not (= (length form) 2))
        (syntax-fail "1+ form incorrect size" form))
    (let* ((arg-node (process (cadr form) env))
           (output-node (process `(+ ,arg-node 1) env))
           (real-form `(1+ ,(get-form arg-node))))
      (set-form! output-node real-form)
      output-node)))
```

Figure 2-7: Syntax definition for 1+, second version

the clause on line 2 of the second version. This line says that if process discovers that the form it is processing is already a node, then process should just return that node. Since the goal of process is to transform forms into nodes, no further work needs to be done. This is a simple idea, but it has powerful consequences. It means that syntax extensions can transform their input forms into output nodes in multiple steps. Consider a second version of the 1+ macro (Figure 2-7).

The differences between the two versions are found on lines 5–7 of the second version. Now the output node is constructed in two steps. First, a node representing the argument form of the 1+ expression is constructed. Then, using that node, the node that represents the entire expression is constructed. On line 7, the source code corresponding to the argument form is extracted directly from the argument node and used to construct the source code for the entire 1+ expression. This source code is used to improve the output node before it is returned.

Node insertion is the technique illustrated by the second version of the 1+ macro. However, node insertion can be used for more than just extracting semantic information that corresponds to part of a macro. As will be seen in the next section, node insertion is one of the techniques that makes it possible to write hygienic macros.

2.3 Hygienic macros

Experienced users of Scheme macro systems will notice that the 1+ macros discussed in the previous section have one serious flaw — they are not hygienic [2, 4, 5, 7, 8, 12]. Specifically, if either version of the macro is used in an environment where the symbol + is not bound to the system's addition procedure, they will behave in unexpected ways. Consider the following example code:

```
(let ((+ string-append))
  ... ;; Omitted code that does string processing
  (1+ (- (string-length result-string)
         (string-length prefix-string))))
```

The example code locally binds + to the procedure `string-append` so it can simplify the appearance of some (omitted) string processing code. However, it wants to return the difference in length between the strings `result-string` and `prefix-string` plus one. It tries to use the 1+ macro to do this, but this will not work because + is bound to an unexpected procedure. The problem is that the symbol + inserted by the 1+ macro ends up being looked up in the currently active environment.

2.3.1 Hygienic macros through node insertion

Ideally, in any expansion of the 1+ macro, the symbol + would always be looked up in the global environment so the macro's behavior would not depend on the environment in which it is used. This would make the 1+ macro hygienic. At first glance, it is not obvious how to guarantee that the + inserted by 1+ always refers to the global binding for +. After all, the procedures that process a particular form decide the syntactic environments in which parts of that form are processed. The `if` keyword procedure discussed earlier (Figure 2-2) decides that the parts of an `if` expression are processed in the same syntactic environment as the `if` itself. On the other hand, the `lambda` keyword procedure (Figure 2-4) decides that the body of a `lambda` is processed in a new syntactic environment that the keyword procedure constructs. In the current case, `make-combination-node` (from `process`) decides that the operator of the expression, the symbol +, will be processed in the current syntactic environment.

By using node insertion, the 1+ keyword procedure can overrule `make-combination-node`. Figure 2-8 contains a hygienic version of the 1+ macro. This version of 1+ uses node insertion to guarantee that the expression nodes it constructs find the correct version of +.

This version of 1+ uses the variable `global-env` to access the global syntactic environment, so that the symbol + can be processed there. The node, `%+`, that 1+ inserts is processed in the current syntactic environment by `make-combination-node`, but `%+` is already a node so the syntactic environment in which it is processed does not affect the node `process` returns[11].

[11] A simpler way to make the 1+ macro from Figure 2-7 hygienic would be to create its output node

```
(define-syntax 1+
  (lambda (form env)
    (if (not (= (length form) 2))
        (syntax-fail "1+ form incorrect size" form))
    (let* ((arg-node (process (cadr form) env))
           (%+ (process '+ global-env))
           (output-node (process `(,%+ ,arg-node 1) env))
           (real-form `(1+ ,(get-form arg-node))))
      (set-form! output-node real-form)
      output-node)))
```

Figure 2-8: Syntax definition for a hygienic version of the 1+ macro.

2.3.2 The procedure `capture-env`

As described so far, node insertion allows macro writers to write hygienic macros. A macro can control what symbols in its expansion mean by processing those symbols (or forms that contain them) in an appropriate syntactic environment. The problem with this solution is that it is inflexible — the only environments to which a macro writer has access are the current syntactic environment (which is usually the source of an undesirable conflict) and a limited set of "well-known" environments, like the global syntactic environment. In particular, there are times when a macro writer wants to process a form in the syntactic environment created for the body of a `lambda`, `let` or other local binding expression. This allows the symbols in the form processed to refer to some locally bound identifiers, but not others. In order to allow macro writers to capture such environments (so they can be used for processing), the procedure `capture-env` is provided[12]. This procedure is essentially the same as the procedure `capture-syntactic-environment` in the syntactic closures macro system proposed by Hanson [6].

The procedure `capture-env` takes a single argument, a procedure. `capture-env` uses its argument procedure to construct a special sort of form, which `capture-env` returns. This argument procedure takes one argument, a syntactic environment, and it returns a form. When the special form created by `capture-env` is processed, the argument procedure that was used to construct it is called with the current syntactic environment. This means the argument procedure can use the syntactic environment in which the special form was processed (a syntactic environment to which the macro

---

by processing its expansion in the global syntactic environment. This would work because the only part of that expansion which needs access to local bindings is the previously processed argument form. The benefit of the strategy described in this section is that it is easier to generalize to macros more complex than 1+.

[12]Another solution that might occur to some readers is for a macro writer to use `environment/extend` and `environment/bind!` to create a desired local environment. The problem with this solution is that macros should not arbitrarily modify syntactic environments. Language implementations should be able to enforce invariants about bindings that macros cannot destroy. In the next version of the system (section 2.4), syntactic environments will be extended so that language implementations can control access to them — so macros will not be able to construct the local environments they may need.

would otherwise not have access) to construct its output form. The form returned by that procedure is then processed (in the same syntactic environment) and the resulting node is returned. Implementing capture-env does not require any changes to process. The special form that capture-env creates is a list form with a single element — a keyword node. The keyword procedure of the keyword node is what invokes the argument procedure and processes its result.

```
(define-syntax swap
  (lambda (form env)
    (if (not (= (length form) 3))
        (syntax-fail "swap expression incorrect size" form))
    (if (not (and (symbol? (cadr form))
                  (symbol? (caddr form))))
        (syntax-fail "argument of swap not a variable" form))
    (let* ((a (cadr form))
           (b (caddr form))
           (%let   (process 'let   global-env))
           (%set!  (process 'set!  global-env))
           (%begin (process 'begin global-env))
           (swap-form
            `(,%let ((temp ,a))
                 ,(capture-env
                   (lambda (temp-env)
                     (let ((%temp (process 'temp temp-env)))
                       `(,%begin
                         ,(process `(,%set! ,a ,b) env)
                         ,(process `(,%set! ,b ,%temp) env)))))))))
      (process swap-form env))))
```

Figure 2-9: A hygienic swap macro, using capture-env

The macro defined in Figure 2-9 is a hygienic macro that swaps the values contained in two variables. It uses capture-env to create a temporary variable that cannot be accessed by code outside of the macro. To keep the focus of this example on the use of capture-env, this macro does not include any semantic improvements.

Through node insertion, this swap macro creates a form that binds a variable whose body is processed in a syntactic environment without that binding. Instead, capture-env is used to capture the syntactic environment that contains the temporary variable's binding in the variable temp-env. Only forms that the swap macro specifically processes in temp-env can see the binding, so the macro is hygienic. Though capture-env has other uses, it is a cumbersome way to hygienically create a temporary variable. A simpler way to create temporary variables is described in the next subsection.

2.3.3 Generating identifiers

The problem with macros that want to bind temporary, internal variables is that the symbols that name those variables can conflict with symbols in user programs, making those macros unhygienic. The procedure genid can solve that problem by

constructing a fresh identifier (that can be bound in a syntactic environment) that is not a symbol. It is called with one argument, a symbol that is used as the basis of the generated identifier. The major implementation overhead of having identifiers is in changing the environment operations (see section 2.1.3) so that they understand identifiers as well as symbols. Figure 2-10 contains source code for a hygienic `swap` macro that uses generated identifiers instead of `capture-env`.

```
(define-syntax swap
  (lambda (form env)
    (if (not (= (length form) 3))
        (syntax-fail "swap expression incorrect size" form))
    (if (not (and (identifier? (cadr form))
                  (identifier? (caddr form))))
        (syntax-fail "argument of swap not a variable" form))
    (let ((%temp (genid 'temp))
          (a (cadr form))
          (b (caddr form))
          (%let (process 'let global-env))
          (%set! (process 'set! global-env)))
      (process
       `(,%let ((,%temp ,a))
          (,%set! ,a ,b)
          (,%set! ,b ,%temp))
       env))))
```

Figure 2-10: A hygienic `swap` macro, using generated identifiers

Besides the direct use of generated identifiers, one other thing to notice in Figure 2-10 is the `identifier?` procedure. It is used in the place of the `symbol?` procedure used by the first version of `swap` to check if the arguments to swap are legal variable names[13]. The change is necessary because, in the presence of generated identifiers, symbols are not the only legitimate variable names.

2.3.4 The envid field revisited

The conversion procedures that generate target-machine code use syntactic environments to prevent identifier conflicts in that target-machine code. The techniques described in this section (which can be used to write hygienic macros), all involve manipulating syntactic environments in new ways. If the conflict-resolution mechanisms provided by syntactic environments fail to work (because of the way a macro manipulates them), then incorrect target-machine code will be generated. Fortunately, the conflict-resolution mechanisms will not fail. This is because syntactic

[13]The testing strategy used by both versions of the `swap` macro is actually incomplete. It does not signal a problem if a variable is unbound or bound to a keyword node instead of a variable node. A better test would process the arguments to `swap` in the current syntactic environment and verify that the resulting nodes are expression nodes that represent variables. Such a test was not used here because it would not have changed between the two versions of the `swap` macro, eliminating the opportunity to explain the `identifier?` procedure.

environments use their envid field to create distinct output identifiers for symbols bound in different syntactic environments. Since each syntactic environment has a distinct envid, these output identifiers will not conflict, no matter how syntactic environments are manipulated. The only remaining question is whether there will be conflicts caused by generated identifiers. In order to ensure that there are not, each generated identifier also has its own envid, that is used to generate target machine code for that identifier. This envid is separate from the envid of any other syntactic environment (including any syntactic environment in which the generated identifier might be bound), and separate from that of any other generated identifier. This separation ensures that generated identifiers also cannot cause conflicts in target-machine code.

2.4 Additional language elements

Now that the simplified version of the framework has been discussed, it is possible to explain the generalizations that allow the complete framework to support language elements that are not keywords or expressions. This support can be used in representing languages that are more complex (have additional language elements such as types, patterns, or declarations) than Scheme, or it can be used to support embedding other languages inside Scheme (or another S-expression language). The framework will be generalized to support additional language elements in three steps:

1. The limitations that the current version of `process` imposes on the use of different language elements are discussed and the points at which `process` needs to be generalized are identified.

2. Syntactic environments are extended with *namespaces*, tokens that allow multiple languages to coexist without their bindings conflicting.

3. The extensions are organized into *contexts*, objects that encapsulate the knowledge about how to process different parts of a program.

To motivate these generalizations, consider using the complete framework to add an extensible notation for describing regular expressions to Scheme. An example of such a regular-expression notation (that, as currently implemented, is not extensible) is the Symbolic Regular Expression (SRE) notation developed by Shivers [19]. In addition to extensibility, there is a second advantage to re-implementing SREs using this framework. SREs distinguish between regular expressions that represent character classes and ones that do not by performing a simpleminded "type" analysis[14]. In the current SRE implementation, this analysis complicates regular-expression parsing. Using this framework, this analysis could be implemented naturally.

The final part of this section will explain another use for the extended framework: enforcing the syntactic restrictions Scheme imposes on definitions.

[14]This distinction is made because some SRE operations are only allowed on character classes.

2.4.1 Generalizing `process`

The current version of the `process` procedure is the key bottleneck that makes it difficult to use the simplified framework for languages that have elements other than keywords or expressions. Several different aspects of the `process` procedure limit the ways in which it can be used to construct non-expression nodes (that represent non-expression language elements). First, `process` uses the procedures `make-constant-node` and `make-combination-node` to process constants and lists, respectively. Both of these procedures return expression nodes. This means that non-expression nodes cannot be constructed by `process` from constants or from lists that do not begin with a keyword. In addition, `process` generates a syntax error when it encounters unbound identifiers. This means non-expression nodes that are described by unbound identifiers cannot be constructed using `process`, either.

These limitations mean that language designers who want additional language elements have two choices in the simplified framework:

1. Avoid using the `process` procedure to construct the nodes corresponding to non-expression forms.

2. Accept the limitations that the current version of `process` imposes on how non-expression nodes can be described.

Both of these choices are bad. The first choice means that the bulk of the language implementation would not take advantage of the syntax-extension framework. Non-expression forms would not allow syntax extensions unless the designer of the non-expression language implemented his own syntax-extension mechanism. Implementing such a mechanism would be largely redundant work because any syntax-extension mechanism would have to resolve the same problems the core framework resolves. On the other hand, the second choice would mean that non-expression nodes could not be described by full-fledged languages of their own. In the case of keywords, these limitations are not severe because keywords are *intended* to be accessed only as identifiers. For more complex languages, like the SRE regular-expression notation, these limitations would be severe. For example, the SRE notation uses constants (characters and strings, in particular) to describe primitive regular expressions. If those constants had to be surrounded by keywords, the SRE notation would be much harder to use.

To summarize, different kinds of language elements need to be able to specify three things:

- how `process` constructs nodes from constant forms,
- how `process` constructs nodes from lists (that do not begin with a keyword),
- and how `process` handles unbound symbols,

in order for the framework to support specifying new elements for different, extensible languages.

2.4.2 Namespaces

The generalization of the `process` procedure described above is intended to allow the framework to support new language elements, including elements of embedded programming languages. The next question to consider is the relationship between the identifiers used by the different embedded programming languages that could coexist in the framework[15]. If the syntactic-environment system were not changed, all of the embedded languages would share a common space of identifiers, the way Scheme keywords and variables do. The advantage of this choice is that programmers reading source code only have to consider one thing when trying to figure out the meaning of an identifier: the scope in which that particular identifier is enclosed.

The disadvantage of this choice is that different embedded languages do not have the flexibility that they might desire in choosing the identifiers they use. For example, in the SRE notation the identifier * is a regular-expression keyword. It means "construct a regular expression that matches zero or more occurrences of the rest of the SRE form." If the SRE notation were embedded inside Scheme, there would be an identifier conflict because Scheme uses * to refer to the built-in multiplication procedure. A programmer implementing an embedded regular-expression language might try to resolve the conflict by processing regular expressions in a special syntactic environment. In that environment, the identifiers that are regular-expression keywords would be bound to the appropriate keyword nodes. However, this solution means that embedded languages break the nested scopes that syntactic environments otherwise provide. This would mean that the identifiers used in embedded languages would not be scoped in the way programmers would expect them to be. Even worse, if the embedded language permitted embedding expressions inside of it (as the SRE notation does), the identifiers used in those expression would also not be scoped correctly[16]. With these scoping problems, embedded languages would be hard to use, so creating special syntactic environments for embedded languages is not a reasonable solution.

A better way to allow embedded languages to control the identifiers they use is to extend the syntactic-environment system with namespaces. A namespace is a unique token created by a particular invocation of the procedure `namespace/create`. These tokens are combined with identifiers to turn syntactic environments into two-dimensional structures. Specifically, each binding in a syntactic environment becomes a mapping from an identifier and a namespace to a node. When syntactic envi-

[15]This seems to omit consideration of the top-level language in which the other languages are embedded. However, there is no reason why the apparent top-level language cannot be embedded in some of the other languages, so the choice of a top-level language is a matter of perspective. In the following subsection, I will explain how to choose different top-level languages.

[16]One might think that this could be solved by keeping track of two syntactic environments: one for processing the embedded language itself and one for processing expressions inside of the embedded language. However, there are three problems with this solution. First, it is difficult to use `process` to communicate more than one syntactic environment at a time. Second, more syntactic environments must be kept track of if more embedded languages are used. Third, it does not solve the problems faced by multiple levels of embedding (what are the appropriate environments in that case?).

ronments are extended with namespaces, the procedures `environment/bind!` and `environment/lookup` both take an additional namespace argument that completes the binding that they are creating or indexing. However, no additional argument is added to the procedure `environment/extend`, so all of the different namespaces of identifiers share the same nested scopes.

To resolve identifier conflicts, each embedded language is allowed to specify the namespace that is used to look up identifiers when forms in that language are being processed. One advantage of this solution is that it resolves identifier conflicts without breaking the scoping of embedded languages. This is because, as mentioned earlier, all of the different namespaces of identifiers share the same nested scopes. A second advantage of this solution is that the extended framework can be used to represent languages that have multiple namespaces, like Common Lisp. A third advantage of this solution is that languages with a private namespace can use the token that controls access to that namespace to control how identifiers that affect their language are bound. For example, an implementation of Scheme could ensure that all Scheme identifiers are bound though either `lambda`, `define`, or `define-syntax` — making it possible to maintain invariants about those bindings. One disadvantage of this solution is that programmers have to consider two things when figuring out the meaning of an identifier: the scope in which that identifier is enclosed and the embedded language the identifier is being used in (including the particular element of that embedded language the identifier is being used to describe). Another disadvantage of this solution is that making syntax-extension interfaces that support hygienic syntax extensions can become more complicated. To minimize this disadvantage the `environment/forward!` procedure is provided.

`environment/forward!`

The problem with creating syntax-extension interfaces after namespaces have been introduced is subtle. Syntax-extension interfaces that support hygienic syntax extensions must provide some method for managing identifier conflicts. This generally involves providing ways for syntax extensions to decide the syntactic environment in which particular identifiers should be resolved. The problem is that when a syntax-extension interface manipulates an identifier it will often not know the role the identifier plays. That is, the interface will not know whether the identifier is being bound or resolved, and the interface will not know the namespace associated with that identifier use. This means the interface cannot use node insertion to implement its environment-control methods (because the identifier might be being bound). Creating new bindings to implement environment-control is more promising (new bindings can be shadowed if an identifier is being bound), but a syntax-extension interface does not know the namespace in which to create a new binding. To get around this, the `environment/forward!` procedure makes it possible to create a new binding for an identifier that forwards all references to that identifier to another environment, regardless of namespace. This means syntax-extension interfaces can use `environment/forward!` without worrying about namespaces directly.

`environment/forward!` takes five arguments. The first argument is the syntactic

environment in which to create the new binding and the second is the identifier to be bound. The third argument is the environment to which references should be forwarded to and the fourth is the particular identifier within that environment to reference (it does not need to be the same as the identifier bound). The fifth argument to `environment/forward!` is a procedure to call if the identifier to be bound is already bound in the first environment. In that case, since identifiers cannot normally be rebound in syntactic environments, no new binding is made.

There is one other thing that is special about the bindings created by `environment/forward!`. Unlike other bindings, they *can* be rebound. This allows syntax-extension interfaces to use `environment/forward!` even if there is a chance that the identifier being forwarded will be bound in the same environment. This makes using `environment/forward!` easier for them because, as mentioned earlier, they often will not know whether or not an identifier is being bound.

The macro-by-renaming and macro-by-attributes syntax-extension interfaces discussed in Chapter 4 use `environment/forward!` to implement their identifier management, so they can coexist with multiple namespaces.

2.4.3 Contexts

At this point, I have described several different ways in which embedded languages should be able to control the behavior of the `process` procedure. The next step is to understand the new version of `process` that allows embedded languages this control. This new version of `process` takes three arguments. The first two arguments are the form and syntactic environment, as in the old version. The third argument is a *context*, an object that organizes the information used to control the behavior of `process`. Four fields of a context have already been described. They are:

- a procedure that constructs nodes from constant forms,
- a procedure that constructs nodes from list forms that do not begin with a keyword,
- a namespace that specifies how identifiers are looked up,
- and a procedure that is called when `process` encounters an unbound symbol.

These different fields are extracted from a context by using the procedures `context/constant-handler`, `context/list-handler`, `context/namespace`, and `context/unbound-handler`. These fields of a context are immutable so that users of a context cannot damage that context. Figure 2-11 contains a first attempt at a context-sensitive version of `process`.

This attempt at a new version the `process` procedure makes most of its decisions by extracting information from its context argument, as expected. However, line 11 of this procedure exposes a latent decision that a context-sensitive version of `process` needs to make. This decision is how to process the operator of a list. In the previous version of `process`, there was only one way to process a form, so no decision needed

```
(define (process form env context)
  (cond ((node? form) node)
        ((identifier? form)
         (environment/lookup env
                             (context/namespace context)
                             form
                             (context/unbound-handler context)))
        ((constant? form)
         ((context/constant-handler context) form))
        ((list? form)
         (let ((op-node (process (car form) env context)))
           (if (eq? (node/kind op-node) 'keyword)
               (node/expand op-node form env context)
               ((context/list-handler context) op-node form
                                               env
                                               context))))
        (else (syntax-fail "Unknown syntax" form))))
```

Figure 2-11: First attempt at a context-sensitive version of the `process` procedure

to be made. In the new version of `process`, forms are processed differently depending on the contexts in which they are processed, so some part of the framework needs to decide how to process the operator of a list form. The listing above assumes that the operator of a list form is processed in the same context in which the entire form is processed. However, there are languages where the operator of a list form and the operands of a list form should not be processed in the same way because they represent different kinds of language elements. One example of such a language is Common Lisp, which uses symbols in the operator position to refer to functions (among other things) and uses symbols in an operand position to refer to variables. To allow such languages to be represented by this framework, contexts include another field: an operator context for processing the operators of list forms. These operator contexts will be used in the next subsection, as part of enforcing Scheme's syntactic restrictions on definitions.

The operator context of a context can be extracted using the procedure `context/op-context`. There is also a procedure `context/init-op-context!` that can be used to set the operator context of an existing context. This procedure is necessary because some contexts are recursive (or mutually recursive with other contexts) in their operator contexts. However, `context/init-op-context!` could allow clients of an embedded language to damage that language's contexts. To prevent this, `context/init-op-context!` is written so that it can only be used if a context does not contain a valid operator context. If `context/init-op-context!` is called on a context that contains a valid operator context, then `context/init-op-context!` signals a system error.

Figure 2-12 contains a second attempt at a context-sensitive version of the `process` procedure. The only differences between this attempt and the previous one are found on lines 11–12 of Figure 2-12. On those lines, the operator context contained in the context argument of `process` is extracted and used to process the operator of a list

```
(define (process form env context)
  (cond ((node? form) node)
        ((identifier? form)
         (environment/lookup env
                             (context/namespace context)
                             form
                             (context/unbound-handler context)))
        ((constant? form)
         ((context/constant-handler context) form))
        ((list? form)
         (let* ((op-context (context/op-context context))
                (op-node (process (car form) env op-context)))
           (if (eq? (node/kind op-node) 'keyword)
               (node/expand op-node form env context)
               ((context/list-handler context) op-node form
                                               env
                                               context))))
        (else (syntax-fail "Unknown syntax" form))))
```

Figure 2-12: Second attempt at a context-sensitive version of process

```
(define (process form env context)
  (define (process-int form env context)
    ;; Internal subroutine - body omitted
    )
  (let ((processed (process-int form env context)))
    (if (not (eq? (node/kind processed)
                  (context/kind context)))
        (syntax-fail
         "Kind of form does not agree with enclosing context"
         form)
        processed)))
```

Figure 2-13: A kind-checking version of the process procedure

form.

There is one final aspect of contexts to discuss. In the version of process in Figure 2-12, there are no constraints on the kinds of nodes that a call to process could return. This makes writing robust clients of process difficult, because the client would not know what kinds of nodes to expect from process. To make matters worse, even the context does not have any control over the kind of node that is returned — the context could be overruled by an inserted node. To control this complexity, contexts are required to fix the kind of node that they return. They accomplish this through a kind field, that can be extracted with the procedure context/kind.

The kind field of a context contains a symbol that must match the kind of every node that is returned from a call to process using that context. The node kind is checked by a wrapper procedure that is the true public interface to process. If the kind of node and the kind of context do not match, process signals a syntax error. The process procedure that has been previously discussed is really an internal

subroutine of the public `process`. This internal subroutine calls itself recursively and does not check the kinds of nodes it is creating[17]. Figure 2-13 contains a kind-checking version of the `process` procedure. It includes the wrapper procedure that performs the kind checking, but omits the details of the internal subroutine that has already been explained.

Kind checking might appear to limit the ways in which programmers can create embedded languages. In fact, as discussed earlier, kind checking a limited form of "syntactic type-checking" that makes it easier to write robust embedded languages. For instance, in the complete framework, kind checking is what signals a syntax error if a keyword (like `lambda`) is used where an expression is expected. If kind checking is too restrictive for a particular embedded language, it can always be avoided by making the elements of the language have the same kind and using attributes to distinguish between language elements.

2.4.4 Implementing definitions

The implementation of Scheme's core syntax discussed in section 2.2.2 was incomplete in one important respect: it did not implement the syntactic restrictions that Scheme imposes on definitions. This section will use contexts and the new version of `process` to implement those restrictions. At the level of core syntax, Scheme only allows definitions in two places:

- at the top level of a program,
- and in the body of a `lambda` expression.

In addition, `begin` expressions are supposed to be transparent to definitions. This means if a definition would be allowed where a `begin` expression occurs, definitions are allowed in the sequence of expressions that form the body of that `begin`.

Two different contexts will be used to implement these syntactic restrictions. The context `exp-context` will be used wherever a definition is not allowed, and the context `def-context` will be used wherever a definition is allowed. Among other things, this means that forms at the top level of a program will be processed using `def-context`. The next question to consider is the kind of node each context will accept. It might seem natural for the expression context (`exp-context`) to return expression nodes and for the definition context (`def-context`) to return definition nodes. However, expression nodes and definition nodes should share the same conversion procedure

[17]At first glance, it is not obvious why the internal version of `process` does not check the kind of the operator node it creates. The problem is checking the operator node when it is a keyword node. It is not enough to know that the operator node is a keyword node, the internal version of `process` should also check that the keyword procedure of that keyword node will return the expected kind of node when called. However, it is not possible to check the kind of node a keyword procedure returns without calling it, so the internal version of `process` does not try to perform incomplete checks on the operator node. The internal part of `process` could check keyword nodes by checking an attribute of the keyword node that indicates the kind of node the keyword procedure promises to return, but this has not been implemented yet.

```
(define scheme-namespace (namespace/create))

(define exp-context (context/create 'scheme #f
                                    scheme-unbound-identifier
                                    make-constant-exp
                                    make-combination-exp
                                    scheme-namespace))

(context/init-op-context! exp-context exp-context)

(define def-context (context/create 'scheme exp-context
                                    scheme-unbound-identifier
                                    make-constant-exp
                                    make-combination-exp
                                    scheme-namespace))

(define (definition-context? context)
  (eq? context def-context))
```

Figure 2-14: Source code that creates the contexts `exp-context` and `def-context`

protocol (since expressions are legal wherever definitions are) and will likely also share many of the same attributes. This means that it is simpler to implement expressions and definitions as different varieties of the same kind of node. Since these nodes are used to implement Scheme they will have the kind `scheme`.

Since both `exp-context` and `def-context` will be used to create these nodes, both contexts will also have kind `scheme`. They will also both have the operator context `exp-context` because definitions are never allowed in the operator position of a form (whether or not the form itself can be a definition). They will share the namespace `scheme-namespace` because they both look up (and bind, in the case of definitions) identifiers in the same namespace. The two contexts will also share an unbound symbol handler, a list handler and a constant handler because unbound symbols, lists that do not begin with keywords, and constants are processed in the same way (and mean the same thing) in the two contexts.

Figure 2-14 contains source code that creates the contexts `exp-context` and `def-context` that are described above. The code assumes that the names `scheme-unbound-identifier`, `make-constant-exp`, and `make-combination-exp` are bound to appropriate handler procedures. The procedure `context/create` is used to assemble the various fields of the context into a context object. The constant `#f` is used as a placeholder for an invalid operator context when `exp-context` is created. On line 9 of the figure, `context/init-op-context!` is used to insert the recursive reference to `exp-context` as its own operator context. The procedure `definition-context?` will be used in the keyword procedure for `define` to check whether definitions are legal in some context.

Figures 2-15, 2-16, and 2-17 contain skeleton code that illustrates what constructing keyword nodes for some Scheme keywords (that implement the syntactic restrictions Scheme imposes on definitions) would look like. These keyword nodes use

```
(define $if
  (make-scheme-keyword
   (lambda (form env context)
     (if (not (= (length form) 4))
         (syntax-fail "If expression incorrect size" form))
     (let ((test-node (process (cadr form) env exp-context))
           (then-node (process (caddr form) env exp-context))
           (else-node (process (cadddr form) env exp-context)))
       (make-scheme-node
        ;; Omitted details
        )))))
```

Figure 2-15: Skeleton code for a Scheme `if` keyword that uses contexts

the new version of `process` as well as `exp-context` and `def-context`. The procedure `make-scheme-node` is like `make-exp-node` from the simplified framework, only it makes nodes of kind `scheme` (expressions and definitions) instead of just nodes of kind `exp` (expressions). The details of calls to `make-scheme-node` are omitted in all of the figures. The first thing to notice about these keyword procedures is that, like the new version of `process`, they take three arguments: a form, a syntactic environment and a context. The procedure `make-scheme-keyword` is a variant of `make-keyword-node`. It is used to create keyword nodes whose keyword procedures should return `scheme` nodes when called[18]. Figure 2-15 contains skeleton code for the `if` keyword. The interesting thing to notice about `if` is that all of its subexpressions are processed in `exp-context`, ensuring that definitions cannot be direct subforms of an `if`.

Figure 2-16 contains skeleton code for a primitive `define` keyword. This `define` keyword does not implement any of Scheme's syntactic sugar for defining procedures. On line 7 of the figure, the procedure `definition-context?` is used to check whether or not definitions are allowed in the context in which the `define` form is being processed. If definitions are not allowed, a syntax error is signaled. If they are, the node that represents the `define` expression is constructed. The variable that is bound by the `define` is added to the current syntactic environment using the procedure `scheme-bind!`, which is shorthand for calling the procedure `environment/bind!` with the namespace `scheme-namespace`. The call signals a syntax error if a variable bound in the innermost frame of the current syntactic environment is being redefined. The variable is bound before the expression that computes the initial value of the variable is processed, so that expression can contain recursive references to the variable. That expression is processed in `exp-context` so it cannot be another definition.

Figure 2-17 contains skeleton code for the `begin` keyword. Unlike the keywords we have seen so far, `begin` uses the context it receives to process its subexpressions (through the call to `process-scheme-body`). The use of the argument context is how `begin` expressions are made transparent to definitions. The procedure

[18]Creating keyword procedures with procedures like `make-scheme-keyword` will be useful if `process` is changed so that it internally checks keyword nodes (by checking the kind of node they promise to return).

```
(define $define
  (make-scheme-keyword
   (lambda (form env context)
     (if (or (not (= (length form) 3))
             (not (identifier? (cadr form))))
         (syntax-fail "Illegally formatted definition" form))
     (if (not (definition-context? context))
         (syntax-fail "Definition in illegal context" form))
     (let* ((var (cadr form))
            (dexp (caddr form))
            (var-node (make-variable var env)))
       (scheme-bind! env var var-node
                     (lambda (var)
                       (syntax-fail "Illegal redefinition: "
                                    form)))
       (let ((dexp-node (process dexp env exp-context)))
         (make-scheme-node
          ;; Omitted details ))))))
```

Figure 2-16: Skeleton code for a Scheme **define** keyword that uses contexts

```
(define $begin
  (make-scheme-keyword
   (lambda (form env context)
     (if (< (length form) 2)
         (syntax-fail
          "Begin expression has too few subexpressions"
          form)
         (make-sequence-node
          (process-scheme-body (cdr form) env context)
          form)))))
```

Figure 2-17: Skeleton code for a Scheme **begin** keyword that uses contexts

`process-scheme-body` is a variant of the `process-body` procedure discussed in the simplified framework. The change in name indicates that the new procedure can only be used to process lists of forms where each form represents a `scheme` node. The procedure `process-scheme-body` is also used to process the body of a `lambda` expression, where it would be called with `def-context` explicitly, to indicate that definitions are allowed in the body of a `lambda` (whether or not they are allowed in the context in which the `lambda` itself is processed).

2.5 Additional features

This section describes additional features of my framework that simplify the task of writing syntax extensions and syntax-extension interfaces. The first part of this section describes *node futures*, special objects that make it easier to recover nodes that correspond to parts of a form. The second part describes modifications to `process` that enable the automatic recovery of the source code that corresponds to a particular node. The following part describes advanced features of my framework's system for generating and handling syntax errors. These features allow users of my framework to replace low-level syntax errors with equivalent higher-level errors. The final part of this section describes the interface to the Scheme 48 module system that my framework provides. This interface is intended for the use of creators of syntax-extension interfaces. It allows them to precisely control the Scheme 48 environments in which syntax extensions are evaluated.

2.5.1 Node futures

In section 2.2.4, a simple strategy was presented to allow macros to recover nodes that represent parts of a form:

1. Process the parts of a form whose nodes are desired separately.
2. Build the node that represents the entire form by using node insertion to reuse the processed parts of the form.

In order for this strategy to work, a macro must be able to process the parts of a form correctly. This means that a macro must have access to the syntactic environment and context in which these forms should be processed. Accessing the context in which a form should be processed should not be difficult, because there should only be a limited number of contexts — roughly corresponding to the different kinds of language elements that can be created. Accessing the syntactic environment, on the other hand, can be more difficult. In particular, when a node that represents part of a local binding form is desired, recovering the node using the tools that have been described so far is cumbersome. This is demonstrated by the `catch` macro in Figure 2-18. `catch` is a convenient shorthand for a use of Scheme's `call-with-current-continuation` (abbreviated `call/cc`).

```
;; (catch <var> <body> ...) =>
;; (call/cc (lambda (<var>) <body> ...))

(define-syntax catch
  (lambda (form env context)
    (if (not (= (length form) 3))
        (syntax-fail "catch form incorrect size" form))
    (if (not (identifier? (cadr form)))
        (syntax-fail
         "catch: attempt to bind continuation to non-identifer"
         form))
    (let*
        ((%lambda (process 'lambda global-env exp-keyword))
         (%call/cc (process 'call/cc global-env exp-context))
         (var (cadr form))
         (body (caddr form))
         (body-node #f)
         (var-node #f)
         (output-node
          (process
           '(,%call/cc
             (,%lambda
              (,var)
              ,(capture-env
                (lambda (body-env)
                  (set! var-node
                        (process var body-env exp-context))
                  (set! body-node
                        (process body body-env exp-context))
                  body-node))))
                  env exp-context)))
      (set-form! output-node
                 `(catch ,(get-form var-node)
                         ,(get-form body-node)))
      output-node)))
```

Figure 2-18: Syntax definition for a source-code reconstructing `catch` macro

The `catch` macro wants nodes that represent the identifier bound and the body of the `catch` expression, so that it can use them to reconstruct its source code. The environment required to create these nodes is made available by the use of `capture-env` on lines 24–30. The nodes themselves are made available to the rest of the macro by mutating the placeholder variables `body-node` and `var-node`. As can be seen from figure, this is difficult code to write, but a bigger problem is that this code is even more difficult to automatically generate — as a syntax-extension interface might want to do. In order to automatically generate this code, it is necessary to know not just the subforms whose node is desired, but also the syntactic environment and context in which these forms should be processed.

Node futures provide a simpler way to access nodes that correspond to parts of a form. The idea behind node futures is to extend the strategy of node insertion to those cases when the appropriate node cannot be created in advance. Instead of inserting an unavailable node into a form, a node future can be inserted. After the form has been processed, the node future can be used to recover its corresponding node. From this point of view, the strategy from Figure 2-18 is a complex, brittle and error-prone way to construct a node future — using `capture-env` to interrupt processing (and recover an environment) and mutating a local variable to provide access to the new node.

Node futures are implemented using data structures that store a form and, once it has been discovered, its associated node. They are created using the procedure `node-future/create`. This procedure takes a single argument: the form around which to build a node future. When `process` discovers that the form it is processing is a node future, it extracts the form to process from it. The extracted source code is then processed and the resulting node is saved in the node future (before the node is returned). Since `process` has access to the current syntactic environment and context, they do not need to be deduced. The creator of a node future can access the saved node after `process` returns. The clause of the internal process procedure that handles node futures is presented in Figure 2-19.

```
((node-future? form)
 (let* ((source (node-future/source form))
        (processed (process-int source env context)))
   (node-future/set-node! form processed)
   processed))
```

Figure 2-19: Source code for the internal `process` clause that handles node futures

The clause uses the procedure `node-future/source` to extract the source form stored in a node future. It uses the procedure `node-future/set-node!` to save a node in the future. It is important to note that only one node can be saved in a node future. This should be expected because only one node can be the "future instance" of a particular node future. However, just as in certain cases S-expressions might be processed multiple times, node futures might be processed mode than once as well. As implemented, only the first call to `node-future/set-node!` (the first time the node future is processed) successfully saves a node. The nodes that later calls attempt

to save are lost. process could return the previously saved node when it encounters a node future that already has a node associated with it, but that would mean that node futures would not be transparent to the core keywords and syntax extensions that did not use them. Transparency would be lost because the node *returned* after processing a node future would depend on whether or not that node future had a saved node. This is a tradeoff made when using node futures instead of node insertion. In the case of node insertion, the inserted node is unambiguous and can be reused, as long as it can be created in the first place. In the case of node futures, a node does not have to be created, but when a node future is used more than once the desired node may be lost [19].

```
(define-syntax catch
  (lambda (form env context)
    (if (not (= (length form) 3))
        (syntax-fail "catch form incorrect size" form))
    (if (not (identifier? (cadr form)))
        (syntax-fail
         "catch: attempt to bind continuation to non-identifer"
         form))
    (let*
        ((%lambda (process 'lambda global-env exp-keyword))
         (%call/cc (process 'call/cc global-env exp-context))
         (var-fut (node-future/create (cadr form)))
         (body-fut (node-future/create (caddr form)))
         (output-node
          (process
           `(,%call/cc
             (,%lambda (,var-fut) ,body-fut))
           env
           exp-context))
         (var-node (node-future/node var-fut))
         (body-node (node-future/node body-fut)))
      (set-form! output-node
                 `(catch ,(get-form var-node)
                         ,(get-form body-node)))
      output-node)))
```

Figure 2-20: Syntax definition for a catch macro that uses node futures

A second version of the catch macro, that uses node futures, is presented in Figure 2-20. It uses the procedure node-future/create to create node futures that encapsulate the identifier bound and the body of the catch expression (lines 12–13). It uses the procedure node-future/node to extract the nodes that correspond to the

[19] Constructs (syntax extensions or core syntax) that reuse forms usually assign the same "meaning" (associate the same node) with that form, such as when a variable node is bound and that binding is used. In that case, the limitations of node futures are not a problem. The case where there is a problem is if a construct reuses a form with a different meaning for each use. One example of such a construct would be an assertion macro that uses an input form as an asserted predicate as well as part of its error message if the assertion fails.

subforms. One thing to mention is that the node future corresponding to the identifier bound does not get its node from `process`. The only way that node future is used is when the keyword procedure for `lambda` binds it to a variable node. However, binding an identifier is another way of associating a meaning with it. The future instance is no less valid because it comes from a binding, so the procedure `environment/bind!` must also be modified to handle identifier node futures. If a node future that contains an identifier is bound, the node to which it is bound to is also saved. Just as in the case of node futures that are processed, node futures that are bound only capture a single future instance.

It is possible for a syntax-extension interface to generate code similar to the `catch` macro in Figure 2-20. If the interface decomposes a macro use into its component subforms, they can be encapsulated in node futures. These node futures can be used to construct the expanded code that corresponds to a macro use. This expanded code can then be processed to create an output node that represents the macro use. After processing, the nodes that correspond to the subforms can be extracted and used to construct improved semantic information for that output node. An example of a syntax-extension interface that follows this strategy is the macro-by-attributes interface described in Chapter 4.

2.5.2 Automatic source code recovery

Static analyses which perform semantic error-checking, such as type-checking, manipulate nodes because they contain the semantic information that the analyses check. However, such analyses often want the source code that corresponds to a node. They can use a node's source code to generate more specific and more useful error messages for a user. As demonstrated by the sample static analysis, syntax extensions could reconstruct their source code for such a use. However, every syntax extension would have to include source-code reconstruction steps for this to work. Otherwise there would be nodes whose source code cannot be reconstructed. It is easier for the framework to automatically associate source code with nodes because the framework is responsible for processing source code into nodes in the first place.

The framework can be modified to automatically save source code in the `form` attribute of a node by making some small changes to the `process` procedure. First, the wrapper procedure is changed to set the `form` attribute of a node to the form processed, as long as that form is not a node, node future, or a form generated by the `capture-env` procedure (see section 2.3.2). Embedded nodes should already have their `form` attribute set because they are created by other calls to `process`. `capture-env` also uses `process` to transform the source code it generates into a node, so the nodes returned from its forms also have their `form` attribute set. Node futures, on the other hand, are directly processed by the internal part of `process` (because they may be a part of an expression whose kind should not be checked), so the code that handles node futures also has to set their `form` attribute. The changes to `process` that implement automatic source code recovery are listed in Figure 2-21.

```
(define (process form env context)
  (define (process-int form env context)
    ... ;; the syntax-object clause is the only one changed
    ((node-future? form)
     (let* ((source (node-future/source form))
            (processed (process-int source env context)))
       (node-future/set-node! form processed)
       (node/attribute-set! processed 'form source)
       processed))
    ... ;; Other omitted clauses
    )
  ;; The wrapper procedure is changed as well
  (let ((processed (process-int form env context)))
    ... ;; Omitted kind-checking
  (if (and (not (node? form))
           (not (node-future? form))
           ;; check for capture-env forms
           (not (special-form? form)))
      (node/attribute-set! processed 'form form))
  processed))
```

Figure 2-21: Changes to `process` that implement automatic source code recovery

2.5.3 Advanced syntax error handling

In discussing the framework so far, I have explained how the `syntax-fail` procedure can be used to generate syntax errors, but I have not explained how those syntax errors are handled. The reason this has not been explained is that handling syntax errors is one of the responsibilities of a user of the framework (a compiler driver, interpreter, etc.). However, the framework does have to supply a method for installing a syntax-error handler. This is provided through the fluid variable[20] `syntax-fail-handler`. The initial value of this fluid variable is a procedure that signals a system error because no syntax-error handler is installed. Users of the framework should set it to an escape procedure that handles syntax errors. Usually that means the procedure does not return to its caller and, instead, reports an error to the user. The escape procedure takes two arguments: a message that describes the syntax error and the form that is syntactically invalid.

The reason the syntax-error handler is stored in a fluid variable is so that the handler can be locally overridden. This is accomplished through the procedure `chain-syntax-fail`. This procedure takes two arguments: a local syntax-error handler and a thunk. The value `chain-syntax-fail` returns is the value obtained when its thunk argument is called — unless there is an error. Instead, all syntax errors generated while calling the thunk are handled by the local handler. Just as with the syntax-error handlers described in the previous paragraph, a local syntax-error handler takes two arguments: a message and a form. The difference is that the local

[20]Fluid variables are a feature of Scheme 48. Essentially, they are mutable, dynamically scoped objects. They are described in greater detail in the library-code section of the Scheme 48 user's manual [10].

handler is not an escape procedure. Instead, it also returns two values: a syntax-error message and a form that is syntactically invalid. This message and form are passed to the syntax-error handler that was in effect before the local handler was installed. This allows keyword procedures to raise the level of the syntax errors they generate, without allowing them to suppress syntax errors completely. More importantly, this allows a syntax-extension interface to check (and possibly improve) the syntax errors that syntax extensions created with that interface report. In section 4.2, `chain-syntax-fail` is used in this manner by the macro-by-renaming syntax-extension interface.

2.5.4 Using the Scheme 48 module system

Syntax-extension interfaces that are based on my framework must provide methods for creating keyword procedures that process syntax extensions[21]. As mentioned earlier, these keyword procedures are Scheme 48 procedures, so they need to be created by evaluating Scheme 48 expressions. Extension interfaces want to control the environments with which keyword procedures are created for two reasons:

1. They want to provide access to procedures (like `process`) and other data objects (like specific contexts) that keyword procedures need in order to function.

2. They want to deny access to procedures (like `make-exp-node`) and data that would allow keyword procedures to depend on internal parts of a language implementation.

My framework provides two procedures that allow extension interfaces to create controlled environments in which keyword procedures can be evaluated: `load-module` and `create-environment`. Both procedures depend on internal details of the version of Scheme 48 with which my framework was developed, so they would need to be rewritten to port the framework to another version of Scheme, or possibly even another version of Scheme 48.

The two procedures use the Scheme 48 module system [18] to enable the creation of controlled environments. The first procedure, `load-module`, takes a single argument: a file specification (a symbol or list that specifies a relative path to a file). It loads that file into the configuration package of the Scheme 48 system on which the framework is running. That makes the structures and interfaces defined in that file available to an extension interface. Through the use of `create-environment`, an extension interface can make those structures available to the syntax extensions defined using that interface. `create-environment` takes one argument: a list of symbols that name Scheme 48 structures. It returns an environment that contains the bindings exported by those structures, in addition to bindings for the standard Scheme procedures. One use of `load-module` and `create-environment` would be to create mechanisms to allow complex syntax extensions to use the Scheme 48 module system to structure their implementations.

[21] These keyword procedures are used to create keyword nodes that, when bound in an appropriate syntactic environment, implement a syntax extension.

Chapter 3

Language Implementation

In this chapter I use my syntax-extension framework to implement a language with a non-trivial static analysis. This serves two purposes. First, it demonstrates that my framework can handle complex static analyses. Second, it builds a foundation for the discussion of syntax-extension interfaces in Chapter 4.

The language I chose to implement is Scheme/R, a demonstration language discussed in MIT's graduate programming languages class. Scheme/R is a statically-typed variant of Scheme that uses Hindley-Milner type inference. My implementation of Scheme/R uses Hindley-Milner type inference for two separate, but closely related purposes: to assign types to program expressions and to explain what is wrong with program expressions that are not well-typed.

The first section of this chapter is an overview of the type-manipulation library used by my language implementation. The purpose of this discussion is to make the language representation and keyword procedures described in the rest of the chapter easier to understand. The second section describes how the representation of Scheme described in Chapter 2 is modified to represent Scheme/R. The third section describes the implementation of Scheme/R in more detail. This includes a discussion of several keywords and the syntactic environments in which those keywords are bound.

3.1 Type library

The type-manipulation library is a set of procedures for creating, printing and unifying Hindley-Milner types. The type library began as a group of procedures that manipulated S-expression representations of Hindley-Milner types.

Since that beginning, the evolution of the interface to the library has been driven by two desires. The first desire is, eventually, to provide parallel unification technology in the library. This technology can make it easier to distinguish between type errors that are contained within a part of an expression and type errors that result from the interactions between parts of an expression [15]. Distinguishing between these sorts of type errors makes it easier to distinguish between type errors caused by argument forms of a syntax extension and type errors caused by the use of the extension itself. The second desire is, in the future, to implement the type library using my syntax-

extension framework. Types could be represented using type nodes, constructed by processing type forms. This would make it easier to naturally represent languages where types are written down as parts of programs. In addition, implementing a type system using my framework would make it easier to make the type system extensible.

Each of the parts of this section describes a different aspect of the type library. The first part describes how types are represented. The second part describes type variables and the role they play. The final part of this section describes the routines provided for unifying types.

3.1.1 Describing types

The first goal of the library is to connect *type forms*, the S-expressions that describe types, and the types themselves. The procedure `create-type` can be used to turn a type form into a type. In the current implementation, `create-type` does nothing more than verify that a form describes a valid type before returning it. However, in a future version, `create-type` would be an interface to the `process` procedure. As it does with other forms, `process` would create a type node from a type form, using a syntactic environment and a type context (which could select a private namespace for types). However, if types are created by processing, the type described by a type form could depend on local environment bindings in the type namespace. This is a problem for syntax extensions that want to use types because they may not get the type they expect when they process type forms in the current syntactic environment. For example, if there is a local binding in the type namespace for the symbol `int`, a syntax extension might not be able to access the global integer type. This is another version of the hygiene problem described in section 2.3. To resolve these problems for syntax extensions, `create-type` would process type forms in a global syntactic environment. This means `create-type` would be an interface for hygienically creating types[1].

In addition to creating primitive types from symbols (such as `int` (integers), `bool` (booleans), `string` (strings), and `unit` (the unit type)), `create-type` can also be used to create types from the following S-expressions:

- (`listof` T) — lists of some type, T,
- (`pairof` T_1 T_2)— pairs with left elements of type T_1 and right elements of type T_2,
- (`contof` T) — continuations that expect the type T,
- (`->` (T_1...T_n) T_r) — procedures with argument types T_1...T_n, that return type T_r,

[1] `process` itself would also be available for syntax extensions that wanted access to local type bindings — to construct a type from an argument form, for example.

- and (**generic** ($I_1 ... I_n$) T_b) — polymorphic types with a body type T_b. T_b uses the identifiers $I_1 ... I_n$ as placeholders for the argument types that are substituted in that body type when the polymorphic type is instantiated.

Just as **process** supports node insertion, **create-type** supports type insertion: creating types from type forms that have types where subforms are expected. For example, a type that describes a list of integers could be created using the expression (**create-type** '(**listof** ,**int-type**), where the variable **int-type** is a variable that contains a reference to an integer type. The primary use of type insertion is to create types that contain type variables as placeholders for parts of their type. Type variables are a special variety of type discussed in more detail in the next subsection.

3.1.2 Type variables

Type variables are types that are used as placeholders for unknown types during type inference. In the library, type variables are created using the procedure **type-var**. This procedure takes a single argument, a form that is intended to describe the expression whose type for which the type variable is a placeholder. The form can be extracted using the procedure **type-var/form**. In the current implementation of my demonstration language this form is stored, but not used. It is intended to allow clients to generate clearer and more specific error messages when type errors are discovered by allowing them to connect the type variables associated with a type error to parts of a source program. Type variables are an important part of unification. They keep track of the constraints discovered on the unknown types for which they are placeholders.

3.1.3 Unification and type errors

The most important ability the type library provides is the ability to unify types. Unification is an essential part of Hindley-Milner type inference. It allows parts of a program to implement type inference by incrementally adding type constraints to the type variables that are placeholders for their types. The primary interface to the unification system is the procedure **unify-types**. It takes two arguments: the two types to be unified. There is also secondary interface to the unification system, **unify-type-lists**, that unifies the corresponding members of two lists of types. **unify-type-lists**, though currently implemented as a serial unification, allows users to express unifications that are logically parallel. This means they could take advantage of parallel unification technology in the future without changing their type-inference routines. With both procedures, if unification is successful, it mutates the type variables (as necessary) to make the types to which they refer equivalent and returns an unspecified value. If unification is unsuccessful, it generates a type error by (indirectly) calling the procedure **type-fail**.

The procedure **type-fail** is an interface to a type-error handler in the same way that **syntax-fail** is an interface to a syntax-error handler (section 2.5.3). A fluid variable, **type-fail-handler**, allows a user of the library to install a procedure that

handles type errors. The procedure `chain-type-fail` allows macros and extension interfaces to install methods for replacing low-level type errors with higher-level errors. Unlike `syntax-fail`, `type-fail` takes three arguments: a message describing the type error, a data structure describing the type conflict in the form (usually a pair of the two types that fail to unify), and the form that is blamed for the type error. To understand how the details of how type errors are generated, the representation of Scheme/R that is used with the library must first be considered.

3.2 Representing Scheme/R

This section describes how my demonstration language, Scheme/R, is represented using my framework. The first part of this section describes how the representation of Scheme from Chapter 2 is modified to represent Scheme/R. It focuses on the differences between Scheme and Scheme/R nodes and attributes. The second part of this section describes how the new Scheme/R nodes and attributes are used to build a type-inference (and type-error reporting) system. The final part of this section discusses Scheme/R variable nodes and the additional features they have in order to support inferring polymorphic types.

3.2.1 Scheme/R nodes and attributes

The representation of Scheme/R is based on the representation of Scheme that linked the examples in Chapter 2. As in section 2.4.4, two contexts (`exp-context` and `def-context`) are used to implement syntactic restrictions on definitions. Just as with Scheme, the core syntax of Scheme/R has two kinds of language elements represented by two different kinds of nodes: keywords and Scheme/R nodes. The Scheme/R nodes follow the Scheme nodes in representing expressions and definitions as different varieties of the same kind of underlying node.

However, there are also substantial differences between the Scheme and Scheme/R nodes. This is why the Scheme/R nodes described in this chapter have kind `scheme/r`, instead of the kind `scheme`, as the Scheme nodes have. This also means they are constructed using the procedure `make-scheme/r-node` instead of `make-scheme-node`. One difference between the Scheme and Scheme/R nodes is the machine each kind of node targets. The Scheme nodes from Chapter 2 targeted an unspecified Scheme debugger, so their conversion procedures were omitted and the conversion procedure protocol was not described. The Scheme/R nodes, on the other hand, target a machine that accepts Scheme programs, and their compilation model is described in more detail below. Following that, the attributes that Scheme/R nodes are discussed in more detail, to prepare for a discussion of type inference.

Compiling Scheme/R

The first difference between the Scheme nodes described in Chapter 2 and the Scheme/R nodes described here is found in their conversion procedures. The

Scheme/R nodes generate code for code for a virtual machine that accepts programs in R^{5}RS Scheme [9]. To be specific, the conversion procedures for Scheme/R nodes are procedures of no arguments that return target-machine Scheme code. The next question to consider is how the target-machine code will implement Scheme/R. The choices made include:

- The generated Scheme code will not use any of Scheme's syntax extension facilities (`define-syntax`, `let-syntax`, or `letrec-syntax`). This restriction ensures that the syntax-extension mechanisms provided by the framework are complete.
- Scheme/R procedures are represented using target Scheme procedures. For simplicity, the generated Scheme procedures have the same arguments, in the same order, as the original Scheme/R procedures[2]. Among other things, this makes it easy to use procedures from the target Scheme implementation as library procedures for Scheme/R (see section 3.3.5).
- Recursion is implemented using recursive forms from the target Scheme (`define` and `letrec`). The advantage of this is simplicity: Scheme/R conversion procedures for recursive constructs do not need to choose how to "unwind" the recursion. However, this means the run-time semantics of recursion are inherited from the target language as well. Recursive constructs that are illegal in the target language, such as `(define foobar (+ foobar 1))`, will generate target-language errors at run-time. To the extent that the goal of this exercise is to replace lower-level errors with higher-level ones, this is disappointing.

Scheme/R attributes

The second difference between the Scheme and Scheme/R nodes is that the Scheme/R nodes have two additional attributes: `w-check` and `unify-fail-msg`. Both of these attributes are used to control type inference. The `w-check` attribute contains a type-checking procedure that is used to implement a type-inference rule. The type-checking attribute is named `w-check` because the type-checking procedure it stores is used to implement type-inference Algorithm W for Scheme/R [15]. The `unify-fail-msg` attribute is used to control the generation of type-error messages — which are usually caused by unification failures. The attributes are discussed in greater detail in the next part of this section, which describes the type-inference system in greater detail.

3.2.2 Type inference and type errors

This section describes how the Scheme/R nodes and attributes described above are used to implement a type-inference and type-error system. First, the type-checking procedure stored in the `w-check` attribute is explained. Then that attribute is used to build an interface for type inference that allows authors of type-checking procedures to control how nodes and type errors are associated. Finally, the generation of specific type errors, using type-error codes and the `unify-fail-msg` attribute is explained.

[2]The order of operand evaluation is also inherited from the target Scheme machine.

The `w-check` attribute

As mentioned in section 3.2.1, the `w-check` attributes of a Scheme/R node stores a type-checking procedure that implements a type-inference rule. The implementation of an inference rule for some node has two parts:

1. Imposing the type constraints generated by the expression that the node represents. This is handled in a `w-check` procedure by having the procedure use unification (and, possibly, other parts of the type library) to impose constraints on the types of parts of an expression — recursively building the `w-check` attribute the way a static analysis should (see section 2.2.4). If the imposed constraints do not generate a type error, the expression is well-typed.

2. Deducing the type of the node's expression. This type is returned by the node's `w-check` procedure, assuming the node is well-typed.

Type-inference interface

The next step in building a type-inference system is to build an interface for using the `w-check` attribute. The purpose of this interface is to simplify the task of generating clear and specific error messages when type errors occur. The problem is that type errors are generated by the unification system, and it does not know anything about the connection between the types it is unifying and the program being type-checked. Nor should it — the unification system should be independent of any particular programming language, so it can be used for any Hindley-Milner language. The solution is to use a language-specific interface for type inference that keeps track of the connection between unification and the expression (or definition) being type-checked.

The type-inference interface consists of two procedures:

`type-check` The `type-check` procedure is a procedure of a single argument: a Scheme/R node. When called, `type-check` does two things. First, it saves the node being checked in the fluid variable `type-fail-node`. Second, `type-check` invokes the type-checking procedure of that expression node and, as long as type-checking succeeds, returns the resulting type. The dynamic binding of the fluid variable is what makes it possible to recover the problematic node when there is a type error. It makes the current value of `type-fail-node` mirror the node whose type-inference procedure is generating type constraints (i.e., calling the unification system). When a recursive call to `type-check` is made, the value of `type-fail-node` is *temporarily* overridden, because a new node is responsible for the constraints introduced. When `type-check` returns, it returns to a previous node's type-checking routine, and the corresponding node is restored in `type-fail-node`. Another way to think about the fluid variable `type-fail-node` is as a "hidden" parameter passed through the unification engine to the type-error system. It is hidden for two reasons:

1. As mentioned earlier, the unification system should not be responsible for the language-specific task of tracking the connections between type constraints and the program nodes that generate them.
2. The parameter has a reasonable default value: the node whose type-inference routine is currently executing. Hiding the parameter makes it easy to give `type-fail-node` its default value and only type-inference routines that need to override that default behavior need to worry about it.

One reason to override this default behavior is to "invisibly" type-check a node. Syntax extensions may want to do this because it allows them to hide their expanded code during type inference. With this interface, a node can be "invisibly" type-checked by directly using its `w-check` attribute. If the `w-check` attribute is used instead of the `type-check` procedure, then the value of `type-fail-node` is not changed. This means that type errors generated by newly imposed constraints will be connected with the parent of that node, not the node that is actually being checked, hiding the node whose type-inference routine is executing[3].

`unify-node` The second part of this interface is the `unify-node` procedure. It takes two arguments: a type and a Scheme/R node. The `unify-node` procedure uses `type-check` to infer the type of its argument node. As discussed above, this means that if the argument node is not well-typed, the type error will be associated with that argument node. `unify-node` then uses the unification system (`unify-types` in particular) to require that the type of its argument node be compatible with its argument type. If a type error is generated here, the error is associated with the node that was in `type-fail-node` when the call to `unify-node` was made — the node responsible for the new constraint. There is also a parallel version of `unify-node`, `unify-nodes`, that takes a list of types and a list of expression nodes and uses `unify-type-lists` to constrain the types of those nodes.

Generating type errors

Now that the interface for type inference has been explained, the generation of type errors can be discussed in greater detail. As mentioned in section 3.1.3, type errors are ultimately generated by calls to the procedure `type-fail`, the way syntax errors are generated by calls to `syntax-fail` (section 2.5.3). The unification system generates type errors by calling `type-fail` through the procedure `unify-fail` (Figure 3-1). `unify-fail` uses two fluid variables (`type-fail-node` and `type-fail-msg-code`)

[3]This use of the type-inference interface assumes that the subnodes of the node being "invisibly" type-checked can be exposed. This tends to be the case for syntax extensions because subnodes of expanded code tend to be generated from parts of a syntax extension's argument form — which can be exposed. If this is not the case, a syntax extension will need to use more complicated techniques to hide the type-checking of its expansion.

```
(define (unify-fail type1 type2)
  (let ((node (fluid type-fail-node))
        (types (cons type1 type2)))
    (let ((form (node/attribute-get node 'form))
          (msg (node/attribute-get node 'unify-fail-msg)))
      (if (string? msg)
          (type-fail msg types form)
          (type-fail (msg (fluid type-fail-msg-code))
                     types form))))))
```

Figure 3-1: The procedure `unify-fail` that reports unification errors

to generate type errors. It uses Scheme 48's `fluid` procedure to extract the values of these fluid variables (lines 2 and 8 of Figure 3-1). As discussed above, `type-fail-node` contains the node blamed for the type error. The `form` attribute of that node is reported as the source code responsible for the type error (line 4).

The `unify-fail-msg` attribute of the failure node is used to determine the error message to print. If that attribute is just a string (a common case), then that is the error message. Otherwise, the attribute must be a procedure of a single argument. That procedure is passed the failure code contained in `type-fail-msg-code`. Nodes that use failure codes are responsible for setting this fluid variable. For example of this, see the Scheme/R `if` keyword procedure (discussed in section 3.3.2). In this case, a default value of the "hidden" parameter `type-fail-msg-code` does not need to be provided. This is safe because the default action is to ignore it (when `unify-fail-msg` is a string).

3.2.3 Scheme/R variable nodes

Unfortunately, the representation of Scheme/R described so far is not powerful enough to implement Hindley-Milner type inference. Specifically, it cannot support inferred polymorphism. The problem with supporting inferred polymorphism is that there is no representation of a type environment[4]. A type environment is necessary for inferring polymorphism because, in Hindley-Milner type inference, the free type variables of the current type environment constrain how potentially polymorphic types can be generalized. Specifically, a type cannot be generalized along any type variables that are free in the current type environment. This restriction is necessary because those type variables could be constrained in ways that have not yet been discovered. If a type were generalized along those type variables future type constraints on those variables could be violated — allowing type-incorrect programs to type-check.

The solution to this problem is to find a way to represent a type environment for Scheme/R. Just like a syntactic environment, a type environment stores compile-time binding information about a program. Unlike a syntactic environment, a type environment only stores information about the variables of a program. It is not interested in any other parts of a program (keywords, embedded languages, etc.).

[4]an environment that binds program variables to types

```
(define (make-variable name type env)
  (let ((genvar (identifier->symbol name env)))
    (define self (make-scheme/r-node
                  'variable
                  (lambda ()
                    (instantiate
                     (node/attribute-get self 'type)))
                  "Type checking should not fail at a variable"
                  (lambda () genvar)))
    (node/attribute-set! self 'type type)
    (node/attribute-set! self 'name name)
    self))
```

Figure 3-2: The definition of the `make-variable` procedure for Scheme/R

This means a natural way to represent a type environment is to add a `type` attribute to Scheme/R variable nodes. The `type` attribute allows syntactic environments to double as type environments — so the appropriate set of free type variables can be determined by iterating over a syntactic environment's bindings. This is an example of variety-specific information that a static analysis (in this case, type inference) might use attributes to store.

To make the discussion of variable nodes more concrete, the definition of the `make-variable` procedure for Scheme/R is given in Figure 3-2. As mentioned in section 2.2.2, `make-variable` is the procedure used to create nodes that represent variables. The Scheme/R version of `make-variable` takes three arguments: an identifier to name the variable, the type that should be associated with that variable and the syntactic environment in which the variable node will be bound. The syntactic environment argument is necessary because, as discussed in sections 2.1.3 and 2.3, syntactic environments are used to ensure that unique target-machine code is generated for each identifier. This happens on line 2 of the `make-variable` definition, where the procedure `identifier->symbol` creates a unique symbol for a new variable node from that node's identifier and syntactic environment.

On lines 3–9 of `make-variable`, `make-scheme/r-node` is called to create the new Scheme/R variable node. This procedure takes four arguments: the variety of the Scheme/R node that is being created, a type-checking procedure for the `w-check` attribute of that node, a value for the `unify-fail-msg` attribute of that variable node, and a conversion procedure that generates Scheme code for that node. The type-checking procedure for a variable node, on lines 5–7 of Figure 3-2, is notable. It retrieves the type stored in the `type` attribute of its variable node and returns an instantiated version of that type[5]. This ensures that if the `type` attribute of a variable node is changed, the variable node will be type-checked according to its new type. Allowing the type of a variable to change as type inference proceeds, while ugly, is necessary to support type inference for polymorphic mutually recursive

[5]Instantiation is the conversion of a polymorphic type into a monomorphic type by substituting fresh type variables for the identifiers in the body of the polymorphic type.

```
(define (make-constant-exp datum)
  (if (null? datum)
      (syntax-fail "Empty expressions are illegal" datum))
  (make-scheme/r-node
   'constant
   (let ((datum-type (constant-type datum)))
     (lambda ()
       datum-type))
   "FATAL ERROR : Constants must type check!"
   (lambda () datum)))
```

Figure 3-3: `make-constant-exp` for Scheme/R

bindings (see the implementation of `letrec` in section 3.3.4). The instantiation itself is necessary to correctly constrain uses of variables that have polymorphic types. The rest of `make-variable` sets up the variable-specific attributes of the new node before returning it. The `name` attribute of a variable node stores the original identifier used to create that node. It is not a significant part of the current system.

3.3 Implementing Scheme/R

This section completes the discussion of my implementation of Scheme/R (and its associated static analysis) by describing the procedures that implement the core syntax of Scheme/R and how they are used. The first four parts of this section describe the procedures that implement constant expressions, conditionals (`if`), definitions, and the two different local binding forms (`let` and `letrec`). The implementations of procedure expressions (`lambda`), procedure calls (`make-combination-exp`) and sequence expressions (`begin`) are not discussed in greater detail because those implementations are straightforward modifications of the corresponding implementations for Scheme, described in sections 2.2.2 and 2.4.4. After the procedures have been described, the process of assembling them into a global environment that implements keywords and provides access to library procedures is discussed in greater detail. The chapter concludes with a discussion of the implementation of the `define-syntax` keyword, that is used to create syntax-extension interfaces in Chapter 4.

3.3.1 `make-constant-exp`

The most primitive elements of Scheme/R are its self-evaluating constant expressions. The Scheme/R version of the procedure `make-constant-exp` (first introduced in section 2.4.4) creates Scheme/R nodes that represent constant expressions. The source code for `make-constant-exp` is in Figure 3-3.

As expected, creating a constant node is a simple use of `make-scheme/r-node`. The procedure `constant-type`, exported by the type library, is used to recover the type of the constant expression (line 6). Since the implementation uses type inference Algorithm W, nodes that represent constant expressions can never generate type errors. This is why the `unify-fail-msg` attribute of a constant node (line 9) is a

string that indicates a fatal error in the type system. Instead, constant nodes constrain expressions that include them by returning their fixed type. The conversion procedure for constants (line 10) is also simple. Constants in Scheme (the target language) are also self-evaluating expressions, so the conversion procedure just generates the Scheme constant that corresponds to the Scheme/R constant. The only unusual part of `make-constant-exp` is the syntax check on lines 2–3. This is required because (as mentioned in section 2.1.2) the framework considers an empty list to be a constant expression. Since the empty list is not a valid Scheme/R expression, `make-constant-exp` must ensure that a syntax error is generated when it is encountered.

3.3.2 `if`

The implementation of the `if` keyword for Scheme/R fills in the skeleton implementation of `if` from Figure 2-15 (discussed in section 2.4.4). Minor differences between the two versions of `if` include the expression contexts and node creation procedures they use. The source code that creates the keyword node for the Scheme/R `if` is given in Figure 3-4.

Scheme/R `if` expressions have a more complex type-inference rule than the constant expressions described earlier. This inference rule is implemented by the procedure on lines 12–21 of Figure 3-4. The rule introduces two type constraints:

1. The predicate (first subexpression) of the `if` must have type `bool`.
2. The types of the consequent (second subexpression) and alternative (third subexpression) of the `if` must be compatible.

Constraint 1 is implemented on lines 13–16 of the figure. `unify-node` is used to constrain the type of the node that represents the predicate. Constraint 2 is implemented on lines 17–21. There, the type of the consequent expression is inferred and unified with the type of the alternative expression. If both constraints are satisfied, the type shared by the consequent and alternative expressions is returned as the type of the `if` (line 21).

In the type-checking procedure for `if`, the Scheme 48 procedure `let-fluid` is used to dynamically bind different values for the fluid variable `type-fail-msg-code`. The procedure `let-fluid` takes three arguments: the fluid variable to be bound, the value to bind it to, and the thunk to execute while that dynamic binding is active. The `unify-node` calls that impose the different constraints of the rule are inside thunks passed to different `let-fluid` bindings of `type-fail-msg-code`. This means they see different values of that fluid variable, allowing a conditional node to report different error messages depending on which constraint is violated. These error messages are generated by the `unify-fail-msg` attribute of the node, created on lines 22–27 of the keyword node definition.

The final part of the `if` keyword node to consider is its conversion procedure, on lines 28–31 of Figure 3-4. The Scheme code generated by this procedure to implement an `if` expression is not remarkable, but the use of the `codegen` procedure

```
(define $if
  (make-scheme/r-keyword
   (lambda (form env context)
     (if (not (= (length form) 4))
         (syntax-fail "Illegally formatted if expression"
                      form))
     (let ((test-node (process (cadr form) env exp-context))
           (then-node (process (caddr form) env exp-context))
           (else-node (process (cadddr form) env exp-context)))
       (make-scheme/r-node
        'conditional
        (lambda ()
          (let-fluid type-fail-msg-code 'test-fail
                     (lambda ()
                       (unify-node (create-type 'bool)
                                   test-node)))
          (let-fluid type-fail-msg-code 'branch-fail
                     (lambda ()
                       (let ((then-type (type-check then-node)))
                         (unify-node then-type else-node)
                         then-type))))
        (lambda (code)
          (case code
            ((test-fail)
             "If predicate not a boolean")
            ((branch-fail)
             "If branches have incompatible types")))
        (lambda ()
          `(if ,(codegen test-node)
               ,(codegen then-node)
               ,(codegen else-node)))))))))
```

Figure 3-4: The keyword node for the Scheme/R version of `if`

is. The `codegen` procedure is shorthand for invoking the conversion procedure of an expression node. The conditional node created by the `if` keyword applies `codegen` to its subnodes to construct the target-machine code it needs to create target-machine code for the `if` expression, in this case, a Scheme `if`.

3.3.3 `define`

Figure 3-5 contains Scheme 48 code that constructs a keyword node for the Scheme/R version of `define`. One difference between this version of `define` and the version in Figure 2-16 (discussed in section 2.4.4) is that, like `if`, the Scheme/R version of `define` uses `make-scheme/r-node` and `make-scheme/r-keyword` instead of `make-scheme-node` and `make-scheme-keyword`. Another difference between the two versions of `define` is that the Scheme/R version creates a type variable (on line 11 of Figure 3-5) as a placeholder for the type of the variable bound by the definition. This type variable must be created because the Scheme/R version of `make-variable` requires a type for the variable node it creates. For simplicity, this version of `define`

```
(define $define
  (make-scheme/r-keyword
   (lambda (form env context)
     (if (or (not (= (length form) 3))
             (not (identifier? (cadr form))))
         (syntax-fail "Illegally formatted definition" form))
     (if (not (definition-context? context))
         (syntax-fail "Definition in illegal context" form))
     (let* ((var (cadr form))
            (dexp-form (caddr form))
            (var-type (type-var var))
            (var-node (make-variable var var-type env)))
       (scheme/r-bind! env var var-node
                       (lambda (var)
                         (syntax-fail
                          "Illegal redefinition: "
                          form)))
       (let ((dexp-node (process dexp-form env exp-context)))
         (make-scheme/r-node
          'definition
          (lambda ()
                  (unify-node var-type dexp-node))
          (create-type 'unit))
          "define: inconsistent defined variable type"
          (lambda ()
            `(define
               ,(codegen var-node)
               ,(codegen def-node)))))))))
```

Figure 3-5: The keyword node for the Scheme/R version of `define`

does not generalize potentially polymorphic variable types. Instead, `let` and `letrec` are used for polymorphic bindings.

The type-inference routine for definitions, on lines 21–23 of the figure is straightforward. The only complication is that definitions can be recursive, so the type-inference routine must ensure that the definition is consistent with the assumed type of a defined variable. That is why it unifies the type of `dexp-node` (the node that represents the expression that provides a value for the definition's variable) with the type of the variable being bound. As long as this unification is successful, the definition itself is given the unit type[6].

[6]This suggests that the type system allows definitions to be used as expressions. This is usually not a problem because definitions are syntactically prevented from appearing as expressions. A more accurate, but also more complex, solution would be to create a distinguished "invalid-type" object and check that no program types are constructed with it. This could catch the (currently ignored) error when a definition is the last subform in the body of a procedure or local binding form. In this position a definition is *syntactically* legal, since definitions are permitted at the top level of bodies, but semantically illegal because a definition does not return a value.

3.3.4 `let` and `letrec`

Unlike Scheme, in Scheme/R, `let` and `letrec` are not derived expressions — they are part of the core language. The reason `let` and `letrec` are part of the core language is because, as built-in expressions, they can infer polymorphic types for the variables they locally bind. In a Hindley-Milner type system, procedures cannot infer polymorphic argument types. This means that if `let` and `letrec` were ultimately derived from `lambda`, they would not be able to provide inferred polymorphism[7].

The definition of the keyword node that implements `letrec` is given in Figure 3-6. Since the keyword node that implements `let` is very similar to the one that implements `letrec`, it is omitted. The keyword procedures of these keyword nodes follow the same pattern. First, they check the syntax of their input form. Then, as long as the input form is syntactically valid, they extract the different pieces of the input form. In particular, they extract a variable list that is used to create a set of variable nodes. These variable nodes are bound in a syntactic environment intended for the body of the form. They also both build nodes for their value expressions (the expressions that provide values for the locally bound variables) and build a node for the body of the form using the appropriate syntactic environment. Finally, these nodes are used to construct an output expression node for the `let` or `letrec`. The type of this output expression node is determined by the inferred type of the body node of the `let` or `letrec`.

The first difference between `let` and `letrec` is in the syntactic environment used to construct expression nodes for the value expressions. For `let`, this syntactic environment does not contain the bindings of the locally bound variables, and for `letrec` (since the binding form is recursive), it does. The second difference between `let` and `letrec` is found in their type-inference routines. In the case of `let`, the initial types given to the variable nodes are irrelevant. The types of the value expressions are inferred and then generalized (by using the `generalize` procedure) and the `type` attributes of the `let` variable nodes are mutated to give those nodes their true types. The `let` bindings do not need to be type-checked separately because they could only cause a type error when a value expression is not well-typed (and that case has been handled by inferring the types of the value expressions).

In the case of `letrec`, however, these initial types are significant. They are used to check the consistency of the `letrec` bindings. Since `letrec`, like `define`, is a recursive binding form it is necessary to check that inferred type of each bound variable is consistent with the assumed type that was used to generate that inferred type. This check is performed by the unification on line 37 of the `letrec` keyword-node definition. One important consequence of this consistency check is that `letrec`-bound variables can only be used monomorphically in the `letrec` bindings. After the check, the `letrec`-bound variables are generalized for use by the `letrec` body.

[7]This is not strictly true for the current version of the system. As discussed in Chapter 4, two of the syntax-extension interfaces I have built for Scheme/R do not protect the type system from macros that deliberately break it. This means `let` and `letrec` could be implemented as macros that break the type system, but that is obviously the wrong thing to do.

```
(define $letrec
  (make-scheme/r-keyword
   (lambda (form env context)
     (if (not (and (> (length form) 2)
                   (let-list? (cadr form))))
         (syntax-fail "Illegally formatted letrec expression"
                      form)
         (let* ((bindings (cadr form))
                (vars (map car bindings))
                (exps (map cadr bindings))
                (body (cddr form))
                (new-env (environment/extend env))
                (types (map type-var vars))
                (var-nodes (map (lambda (var type)
                                  (make-variable var type
                                                 new-env))
                                vars types)))
           (for-each (lambda (var var-node)
                       (scheme/r-bind! new-env var var-node
                                       (lambda (var)
                                         (syntax-fail
                                          "Duplicate variable - letrec: "
                                          var))))
                     vars var-nodes)
           (let ((exp-nodes (map (lambda (exp)
                                   (process exp new-env
                                            exp-context))
                                 exps))
                 (body-node (make-sequence-node
                             (process-scheme/r-body body new-env
                                                    def-context)
                             form)))
             (make-scheme/r-node
              'letrec
              (lambda ()
                (let ((var-types (map variable-type var-nodes)))
                  (unify-exps var-types exp-nodes)
                  (for-each (lambda (var-node type)
                              (node/attribute-set! var-node 'type
                                                   (generalize type env exp-context)))
                            var-nodes var-types)
                  (type-check body-node)))
              "Inconsistent variable types in letrec bindings"
              (lambda ()
                `(letrec ,(map (lambda (var-node exp-node)
                                 `(,(codegen var-node)
                                   ,(codegen exp-node)))
                               var-nodes exp-nodes)
                   ,(codegen body-node)))))))))))
```

Figure 3-6: The keyword node for the Scheme/R version of `letrec`

3.3.5 Populating the global environment

The global environment of Scheme/R contains bindings for keyword nodes that implement Scheme/R's core forms as well as variable nodes that make it possible for Scheme/R programs to use Scheme library procedures from the target machine. Populating the global environment with keyword bindings is straightforward — all that is necessary is to use `scheme/r-bind!` on the global environment to bind a keyword node to the desired identifier in the Scheme/R namespace. For Scheme/R, globally bound keywords include `if`, `lambda`, `let`, `begin`, `letrec`, `define`, and `define-syntax`.

Globally bound variable nodes (that do not have associated definitions) are used to access Scheme library procedures. The implementation trick that makes this possible is that globally bound identifiers that are symbols are not mangled in the target-machine Scheme code. This is only safe to do for a single frame (a single envid value) because otherwise, as suggested in section 2.3.4, environment manipulation by syntax extensions could create undesired identifier conflicts in the target-language code[8]. By globally binding variable nodes that name standard Scheme procedures (and giving those procedures appropriate Scheme/R types), these procedures can be "linked" to the target-machine code for a Scheme/R program.

3.3.6 `define-syntax`

The keyword node that implements the `define-syntax` form is defined in Figure 3-7. A `define-syntax` form has three parts: the `define-syntax` keyword, the identifier to be defined, and a macro specification. Just like `define`, `define-syntax` can only be used in a definition context. `define-syntax` uses the `make-macro` procedure (Figure 3-8) to turn a macro specification into a keyword procedure. It then takes that keyword procedure and uses it to create a keyword node that is bound to the supplied identifier, in the current environment and the expression namespace. `define-syntax` returns an empty node as its output expression node because there is no target-machine code that corresponds to a `define-syntax` form.

A macro specification is an S-expression whose first element is a symbol. That symbol is used by `make-macro` as an index into a table of macro-making procedures. If no corresponding procedure is found for some symbol, then `make-macro` signals a syntax error. If a macro-making procedure is found, it is called with the entire macro specification, as well as the currently active syntactic environment (the environment of macro definition). It is expected to return a keyword procedure that `define-syntax` will use. By adding macro-making procedures to the table of macro-makers, new syntax extension interfaces (for creating expression macros) can be implemented. This table of macro-makers is used to implement the three syntax-extension interfaces

[8]If a completely safe implementation is desired, the character used to mangle identifiers (in my implementation, the '.' character) should be disallowed in source language identifiers when this trick is used. Otherwise, an unintentional identifier conflict involving a globally bound identifier is possible, though it would probably be rare.

```
(define $define-syntax
  (make-scheme/r-keyword
   (lambda (form env context)
     (if (or (not (= (length form) 3))
             (not (identifier? (cadr form))))
         (syntax-fail "Illegally formatted syntax definition"
                      form))
     (if (not (definition-context? context))
         (syntax-fail "Syntax definition in illegal context"
                      form))
     (let ((transformer (make-macro (caddr form) env)))
       (scheme/r-bind! env
                       (cadr form)
                       (make-exp-keyword transformer)
                       (lambda (var)
                         (syntax-fail "Illegal redefinition" form)))
       $empty-node))))
```

Figure 3-7: The keyword node for the Scheme/R version of `define-syntax`

```
(define (make-macro form env)
  (if (or (null? form)
          (not (list? form))
          (not (symbol? (car form))))
      (syntax-fail "Illegally formatted macro specification"
                   form))

  ;; should be a macro-making procedure
  (let ((macro-maker (assq (car form) makers)))
    (if macro-maker
        ((cadr macro-maker) form env)
        (syntax-fail "Unknown macro specification" form))))
```

Figure 3-8: The `make-macro` procedure used by the implementation of `define-syntax`

described in Chapter 4[9].

[9]The ability to create new syntax-extension interfaces is not exported to a Scheme/R user in the current implementation (ignoring the ability to create forms that expand into uses of `define-syntax`, which cannot be easily used to build a general extension interface). This is unfortunate because, as the discussion in Chapter 4 will demonstrate, future experimentation with extension interfaces is desired so that this system can be made easier to use.

Chapter 4

Syntax-Extension Interfaces

This chapter describes the three syntax-extension interfaces that I have developed using the syntax-extension framework described in Chapter 2 and the implementation of Scheme/R described in Chapter 3. The first interface is the macro-by-procedure interface. It was developed to demonstrate that my framework can support type-aware macros for Scheme/R. However, while this interface was an important proof-of-concept, it is a needlessly complex interface. The second interface is the macro-by-renaming interface. It was developed as a simpler alternative to the macro-by-procedure interface. Unfortunately, for many macros, the macro-by-renaming interface is not powerful enough. The macro-by-attributes interface is a third attempt to strike a balance between power and usability.

4.1 Macro-by-procedure

The macro-by-procedure syntax-extension interface allows programmers to create syntax extensions by creating keyword procedures that implement those syntax extensions. Like the "raw" syntax-extension interface from Chapter 2, the macro-by-procedure interface is a syntax-extension interface (rather than a compiler-extension interface) because its keyword procedures are required to use `process` to create their output expression nodes. This restriction ultimately forces those nodes to be implemented in terms of existing syntax.

The first part of this section is an overview of the macro-by-procedure interface. After the overview, two examples of macros written using the interface are presented. These macros demonstrate that my framework can support type-aware macros for Scheme/R. Since the static analysis for Scheme/R (Hindley-Milner type-inference) is complex and my framework is not analysis-specific, this demonstration supports the claim that my framework can be used to implement arbitrary extensible static analyses. This section concludes by assessing the advantages and drawbacks of the macro-by-procedure interface.

4.1.1 Overview

The macro-by-procedure interface is a slightly more complicated version of the "raw" syntax-extension interface from Chapter 2. Like that interface, the goal of the macro-by-procedure interface is to allow macro writers to specify the keyword procedures that implement their syntax extensions as directly as possible. There are three major differences between macro-by-procedure macros and the "raw" macros discussed earlier:

1. Since this chapter discusses three different syntax-extensions interfaces, macro-by-procedure macros are specified using a list that begins with with the symbol `procedure`. This list also contains the Scheme 48 expression used to construct the keyword procedure.

2. Macro-by-procedure macros can contain an optional `open` clause that allows them to interface to the Scheme 48 module system. Neither of the sample macros uses the `open` clause, so this kind of clause will not be discussed in greater detail.

3. The keyword procedure is constructed from the macro-creating expression in two steps. First, the expression is evaluated to yield a Scheme 48 procedure of a single argument. Then the resulting procedure is called with the syntactic environment in which the macro is being defined. The value returned by that call should be the keyword procedure that implements a macro. The two-step creation allows syntax extensions to access the syntactic environments in which they are defined and to perform actions at the time of macro definition.

The bindings available in the Scheme 48 environment in which a macro-creating expression is evaluated are an important determinant of the power of the keyword procedures that can be created by that expression. By default, the Scheme/R implementation provides a rich set of bindings for macro-by-procedure macros to use. Some of the more important bindings provided (beyond the basic set of Scheme bindings) include:

- the procedures provided by the framework for node, context and environment manipulation,[1],
- the `process` procedure,

[1] In the current Scheme/R implementation, the availability of these procedures does allow macro-by-procedure macros to create expression nodes without using `process` (as long as they implement the correct conversion procedure protocol). However, this is a defect of the language implementation, not of the macro-by-procedure interface. The language implementation should protect itself from "forged" nodes (and framework support might make that task easier). The interface should not be forced to deny macro writers access to generic node-manipulation procedures because those generic procedures are part of the toolkit that allows macro writers to create extensible embedded languages.

- the procedures provided by the type library for type construction and manipulation (`create-type`, `type-fail`, etc.),
- and the procedures (such as `unify-exp` and `type-check`) and contexts (`exp-context`, `def-context`, and `exp-keyword`[2]) that are essential parts of the Scheme/R implementation.

4.1.2 Sample macros

Two sample type-aware macros were implemented using the macro-by-procedure interface: `n+` and `arith-if`. The `n+` macro implements an addition form that can total an arbitrary number of numeric subexpressions[3]. The `arith-if` macro implements conditional expressions that have three different branches. The "predicate" of an `arith-if` expression should evaluate to an integer and the branch of the `arith-if` to execute is chosen based on whether that integer is negative, positive or zero.

`n+`

The first sample macro is the `n+` macro. A macro-by-procedure definition of `n+` is given in Figure 4-1. The syntax check for `n+` (lines 6–9) is simple: an `n+` form must have subforms other than the keyword. After syntax-checking, the `n+` macro constructs an expansion (lines 10–19). It builds nodes corresponding to the subexpressions of the `n+` expression (using the current syntactic environment so these subforms have access to local bindings) and assembles those nodes into nested addition expressions. It creates its output node (the node that represents the `n+` expression itself) by processing this expansion. The `n+` macro is hygienic because the expansion is processed in the global environment (so the appropriate binding for the inserted identifier + is found). Processing the assembled form in the global environment is not a problem because the nodes that represent the `n+` subexpressions have already been constructed (so they are not affected by the environment in which they are processed). After the output node has been created, its `w-check` and `unify-fail-msg` attributes are mutated to improve the type-checking of the `n+` expression. This mutation is necessary because macros cannot (and, as syntax extensions, should not be able to) create nodes with improved attributes directly.

The type-checking rule for `n+` is straightforward. All of the subexpressions of an `n+` expression must have integer type, and so must the `n+` expression itself. However,

[2]The `exp-keyword` context is a part of the Scheme/R implementation that is only used by syntax extensions. It is needed because `process` checks the kind of the node it returns and some syntax extensions may need to use `process` to look up expression keywords they wish to insert hygienically. The context `exp-context` is not suitable because kind-checking will cause `process` to signal a syntax error if a keyword node is returned from processing in an expression context. As mentioned earlier, kind-checking does not normally cause problems with keywords because most keyword nodes are used internally by `process` and the internal subroutine of `process` does not implement kind-checking.

[3]An n-argument version of + must be implemented as a macro in my version of Scheme/R because my implementation of Scheme/R does not support procedures that take a variable number of arguments.

```
(define-syntax n+
  (procedure
   (lambda (def-env)
     (lambda (form env context)
       (let ((args (cdr form)))
         (if (null? args)
             (syntax-fail
              "n+ expression has no arguments"
              form))
         (let* ((nodes (map (lambda (arg)
                              (process arg env exp-context))
                            args))
                (expansion
                 (let loop ((nodes nodes))
                   (if (null? (cdr nodes))
                       (node/copy (car nodes))
                       `(+ ,(car nodes) ,(loop (cdr nodes))))))
                (output-node
                 (process expansion global-env exp-context)))
           (node/attribute-set!
            output-node 'w-check
            (lambda ()
              (unify-exps
               (map (lambda (node) (create-type 'int)) nodes)
               nodes)
              (create-type 'int)))
           (node/attribute-set!
            output-node 'unify-fail-msg
            "n+ subexpression does not have type int")
           output-node))))))
```

Figure 4-1: Defining an n+ macro using the macro-by-procedure interface

implementing this type-checking rule (lines 22–26) is not completely straightforward. In order to understand why, it is important to recall that type-inference for Scheme/R is implemented recursively: expressions use the type information of their subexpressions to type-check themselves. In the case of an `n+` expression, this means using the subexpression nodes (and their associated type-checkers) to construct an improved type-checker for the entire `n+` expression. This improved `n+` type-checker uses the subnode type-checkers to ensure that each subnode has integer type before returning the type for the `n+` expression itself. This implicitly assumes that the node that represents an `n+` expression is distinct from the nodes that represent its subexpressions. If this is not the case, the constructed type-checker for an `n+` expression would recursively call itself (as the type-checker of a subexpression), leading to an infinite loop.

In many typical uses of the `n+` macro, this problem does not arise because new nodes are constructed to add the subexpressions together. However, there is one boundary case that is a problem: `n+` of a single subexpression. In that case, no addition is necessary, so the node that represents the expanded `n+` expression is also the node that represents the single subexpression. The use of `node/copy` on line 16 of the `n+` macro prevents the potential conflict by copying the last node in a list of `n+` subexpressions. That way, even when `n+` is used with a single subexpression, the node that represents the `n+` (the copied node) and the node that represents its subexpression are distinct, so the type-checker does not go into an infinite loop. There are several other ways to solve this problem. Using the `n+` macro with a single subexpression could be made syntactically illegal, the expansion constructed could add all of the subexpressions to zero to ensure a new node is created, or the case of a single subexpression could be handled separately. This method was chosen to highlight the problems authors of syntax extensions face when mutating nodes in the macro-by-procedure interface.

Figure 4-2 contains some sample uses of the macro-by-procedure `n+` macro. Notice that the error messages for invalid uses are clear and specific and are in terms of the problematic source code expression. These error messages demonstrate that this `n+` macro is type-aware. The error-reporting does have one flaw: In the case of non-integer subexpressions, no specific source-code expression is highlighted as the cause of the problem. For simple uses of `n+`, as in the example, this is not an issue, but large `n+` expressions could be difficult to debug. Techniques similar to the ones used by the `do` macro in Appendix A could fix this, but they would make the macro definition more complex and harder to understand.

`arith-if`

Figure 4-3 contains a macro-by-procedure definition of the second sample macro: an `arith-if` macro. It is very similar to the Scheme code that implements the Scheme/R `if` keyword (Figure 3-4). Like `if`, `arith-if` processes its argument form in pieces and signals a syntax error if it has the wrong number of subforms (lines 5–13 of Figure 4-3). There is one key difference between the implementation of the `if` keyword and the implementation of this `arith-if` macro. The `arith-if` macro (on lines

```
Expression: (n+ 1 2 3 4)
Expression type: int

Expression: (n+)
Syntax error: n+ expression has no arguments
(n+)

Expression: (n+ 1 2 #t 4 5)
Type inference failed
n+ subexpression does not have type int
Type clash between:
int
bool
In form: (n+ 1 2 #t 4 5)

Expression: (n+ #t)
Type inference failed
n+ subexpression does not have type int
Type clash between:
int
bool
In form: (n+ #t)
```

Figure 4-2: Using the macro-by-procedure n+ macro

15–22) constructs its output node by processing its expansion. To ensure that the `arith-if` macro is hygienic, its expansion is processed in the global environment, ensuring that the inserted `let`, `if`, <, and = identifiers are resolved correctly. As with the `n+` macro, this is safe because the nodes that represent its subexpressions have already been processed. Even the inserted identifier `temp`, that is bound to the value of the "predicate" subexpression, is not a problem because the syntactic environment in which the subexpressions are processed does not contain a binding for the inserted `temp`.

The type-checking procedure for `arith-if` (lines 25–35) implements the expected type-checking rule for `arith-if` expressions. The first part of the rule (line 29) requires that the "predicate" expression of an `arith-if` have a numeric type[4]. The next part of the rule (lines 30–34) requires that the branches of an `arith-if` have compatible types. The final part of the rule (line 35) reports the type of an `arith-if` expression as the type shared by its branches. The rest of the type-checking procedure sets up failure codes used by the type-error-message procedure for `arith-if`.

Figure 4-4 contains some sample uses of the macro-by-procedure `arith-if` macro (both valid and invalid). The invalid uses of `arith-if` include both syntactically invalid and type-incorrect uses. Again, the error messages are clear and specific, making no reference to the macro-expanded code. This demonstrates that this `arith-if` macro is type-aware, as well.

[4]In my implementation of Scheme/R, the only available numeric type is integer.

```
(define-syntax arith-if
  (procedure
   (lambda (def-env)
     (lambda (form env context)
       (if (not (= (length form) 5))
           (syntax-fail "arith-if expression incorrect size"
                        form))
       (let* ((pred-node (process (cadr form) env exp-context))
              (minus-node (process (caddr form) env exp-context))
              (zero-node (process (cadddr form) env exp-context))
              (plus-node (process (car (cddddr form))
                                  env
                                  exp-context))
              (output-node
               (process
                `(let ((temp ,pred-node))
                   (if (< temp 0)
                       ,minus-node
                       (if (= temp 0)
                           ,zero-node
                           ,plus-node)))
                global-env exp-context)))
         (node/attribute-set!
          output-node 'w-check
          (lambda ()
           (let ((result-type (type-var form)))
             (let-fluid type-fail-msg-code 'pred-fail
              (lambda ()
                (unify-exp (create-type 'int) pred-node)))
             (let-fluid type-fail-msg-code 'branch-fail
              (lambda ()
                (unify-exps
                 (list result-type result-type result-type)
                 (list minus-node  zero-node   plus-node))))
             result-type)))
         (node/attribute-set!
          output-node 'unify-fail-msg
          (lambda (code)
            (case code
              ((pred-fail)
               "arith-if: predicate does not have integer type")
              ((branch-fail)
               "arith-if: branches have incompatible types"))))
         output-node)))))
```

Figure 4-3: Defining an `arith-if` macro using the macro-by-procedure interface

```
Expression: (arith-if 0 1 2 3)
Expression type: int

Expression: (arith-if #t 1 2 3)
Type inference failed
arith-if: predicate does not have integer type
Type clash between:
int
bool
In form: (arith-if #t 1 2 3)

Expression: (arith-if 0 #f 1 2)
Type inference failed
arith-if: branches have incompatible types
Type clash between:
bool
int
In form: (arith-if 0 #f 1 2)

Expression: (arith-if 0 1 2)
Syntax error: arith-if expression incorrect size
(arith-if 0 1 2)
```

Figure 4-4: Using the macro-by-procedure `arith-if` macro

4.1.3 Assessment

The most important thing that these macro-by-procedure macros demonstrate is that it is possible to create type-aware syntax extensions. In addition, these macros help demonstrate that the macro-by-procedure interface is a powerful one. In this interface, macros can precisely control conflicts between identifiers they insert and identifiers in their argument forms. Both sample macros were hygienic, but it should be easy to see how that could be changed. Macros can also precisely control the type errors they generate. For instance, the `arith-if` macro uses type-error codes to generate different error messages for different illegal uses. In fact, the `do` macro described in the introduction (the most powerful type-aware macro I have written to date) is implemented using this interface. In this interface, macros can also take advantage of the semantic information available in their expansion. The `arith-if` and `n+` macros use this information to deduce the types of subexpressions, but other macros could use it in more creative ways. In fact, macro-by-procedure syntax extensions are as powerful as syntax extensions can be in my framework because, however a syntax extension is specified, it eventually is implemented using a keyword procedure (or set of related keyword procedures).

However, specifying syntax extensions by directly writing keyword procedures has some disadvantages, as well. Each of the keyword procedures has to manipulate nodes and attributes "by hand" to generate its output nodes. This can be complex and error-prone, as the `n+` macro and its struggles with copying nodes demonstrate. Worse, this interface forces macro writers to think about their macros at the wrong level

of abstraction. They should be thinking in terms of expansions and type-checking rules without having to think about the details of how nodes and attributes must be assembled to implement those expansions and type-checking rules.

These keyword procedures can also be *too* powerful. For example, in the case of Scheme/R, they can break the type system by making "illegal" changes to Scheme/R nodes and attributes. A keyword procedure could mutate the `type` attribute of a variable node, changing the type associated with a variable in the middle of type inference. Or the type-checking procedure that a keyword procedure inserts (for its output node) could return a type that does not agree with the type of the expanded code. Removing this flexibility is difficult because these syntax extensions create their improvements by manipulating nodes and attributes. One solution might be to rigidly specify how syntax extensions can modify their output nodes (perhaps with attribute permissions) so syntax extensions cannot make "illegal" changes to a node. This is a general solution that could be adapted to protect the results of any static analysis. A disadvantage of this solution would be that it could make it more difficult for syntax extensions to improve their output nodes. Another solution might be to re-type-check the expanded version of a program to catch any type errors syntax extensions miss. An advantage of this solution would be that it would make it easier to debug the type-checking improvements of syntax extensions. Disadvantages would include the largely redundant work of type-checking and the difficulty of generalizing this strategy to other static analyses.

4.2 Macro-by-renaming

The second syntax extension interface I have implemented is the macro-by-renaming interface. This interface was created as a reaction to the disadvantages of the macro-by-procedure interface. This interface is easier to use than the macro-by-procedure interface, but it is also less flexible. It was inspired by the explicit-renaming macro system introduced by Clinger [4]. The first part of this section is an overview of the macro-by-renaming interface. The second part implements the same sample macros (`n+` and `arith-if`) using the new interface. The section concludes with an assessment of the power and usability of the macro-by-renaming interface.

4.2.1 Overview

A macro specification in the macro-by-renaming interface has three parts:

- the symbol `rename` that indicates the kind of macro specification,
- an explicit-renaming transformer that implements the syntax extension,
- and optional clauses that control the error messages generated by a syntax extension.

The explicit-renaming transformer is at the heart of the macro-by-renaming interface. It is a procedure that takes three arguments: the form to be transformed, a renaming

procedure and syntax-error procedure. It returns a transformed version of the argument form. This form is processed to create the node that corresponds to the use of a syntax extension. The syntax-error procedure argument is just the `syntax-fail` procedure (first described in section 2.1.2). It is necessary because the explicit-renaming transformer is evaluated in an environment that has only the basic Scheme bindings and macro-by-renaming macros must be able to generate syntax errors.

The renaming procedure is what allows hygienic macros to be defined in the explicit-renaming interface. The argument of the renaming procedure is an identifier to be renamed. It returns a new identifier that is meant to be used in the transformed version of the argument form. If this new identifier appears free in the output of the transformer, it has the same meaning as the original identifier has in the syntactic environment in which the macro was defined[5]. This identifier can also be bound, and bound uses of the identifier get their meaning from their binding. The renaming procedure allows the explicit-renaming transformer to control the conflicts between identifiers that appear in the transformed version of the argument form — making hygienic macros possible (as well as controlled unhygienic macros).

The optional error message clauses allow macro-by-renaming syntax extensions to improve the error messages they generate. Each clause consists of a symbol (that indicates the kind of clause) and a string (that is an improved error message). If an error is traced to code inserted by the explicit-renaming macro transformer, and there is a clause in the macro specification corresponding to that kind of error, then the clause overrides the error message generated. Specifically, the error message string is taken from the string of the clause and the form for which an error is reported is the original use of the syntax extension. This allows a macro-by-renaming macro to hide its expanded code when errors are generated by that code (by having clauses that correspond to the errors). In the current implementation there are two kinds of clauses, syntax-error clauses and type-error clauses, indicated by the symbols `syntax-error` and `type-error`, respectively. Type-error clauses are what make it possible to create type-aware macros with this interface. Each kind of clause can appear at most once in a macro-by-renaming specification, in any order.

4.2.2 Sample macros

This section will discuss the implementation of the two sample macros using the macro-by-renaming interface. As before, the first macro implemented will be an n+ macro, followed by an `arith-if` macro.

n+

Figure 4-5 contains a macro-by-renaming implementation of the n+ macro. The macro is broadly similar to the macro-by-procedure n+ macro in Figure 4-1. It performs the same syntax check and loops through the argument forms to build an expansion. The macro-by-renaming n+ macro also explicitly manages identifier conflicts, the way the

[5]This is true even if the original identifier was unbound in the environment of macro definition.

macro-by-procedure n+ macro does. The only difference in identifier management is that macro-by-procedure macro uses node insertion to control the identifier + and macro-by-renaming macro uses a renaming procedure.

```
(define-syntax n+
  (rename (lambda (form rename syntax-fail)
            (if (= (length form) 1)
                (syntax-fail
                 "n+ expression has no arguments"
                 form))
            (let ((%+ (rename '+)))
              (let loop ((args (cdr form)))
                (if (null? (cdr args))
                    (car args)
                    `(,%+ ,(car args) ,(loop (cdr args)))))))
          (type-error "n+ subexpression does not have type int")))
```

Figure 4-5: A macro-by-renaming version of the n+ macro

The major differences between the two versions of n+ are found in what the macro-by-renaming version excludes. It does not need to manage processing, explicitly copy nodes, or create a type-checking procedure the way the macro-by-procedure n+ macro does. Instead, the macro-by-renaming interface manages processing and error handling and provides simple hooks for user control of error messages. For n+, this simplicity does not adversely affect the resulting macro, as the sample uses of the macro-by-renaming n+ (in Figure 4-6) show[6]. The only odd case in those uses is the expression (n+ #t), which the macro-by-procedure macro rejects because n+ arguments should have integer type. In the case of the macro-by-renaming n+, since nothing is actually added to #t in the expansion, the macro-by-renaming interface accepts it as well-typed[7]. This highlights the difference between the macro-by-renaming interface and the macro-by-procedure interface. In the macro-by-procedure interface, a macro writer is in complete control of a macro's type-checking (and any other aspects of its expansion). In the macro-by-renaming interface, the interface type-checks the expansion and only after the fact decides whether or not to use a macro-writer's improvements.

arith-if

Figure 4-7 contains a macro-by-renaming implementation of the arith-if macro. Like the macro-by-renaming n+ macro, the macro-by-renaming arith-if macro is broadly similar to its macro-by-procedure counterpart, checking the syntax of its

[6]However, unlike the macro-by-procedure interface, a version of n+ that reported the specific subexpression responsible for a type error could not be implemented in the macro-by-renaming interface.

[7]One way to solve this problem would be to add the n+ subexpressions to zero, so the #t would generate a type error (because a boolean cannot be added to a number).

```
Expression: (n+ 1 2 3 4)
Expression type: int

Expression: (n+)
Syntax error: n+ expression has no arguments
(n+)

Expression: (n+ 1 2 #t 4 5)
Type inference failed
n+ subexpression does not have type int
In form: (n+ 1 2 #t 4 5)

Expression: (n+ #t)
Expression type: bool

Expression: (n+ 1 #t 2 #f 4 5)
Type inference failed
n+ subexpression does not have type int
In form: (n+ 1 #t 2 #f 4 5)
```

Figure 4-6: Using the macro-by-renaming n+ macro

```
(define-syntax arith-if
  (rename
   (lambda (form rename syntax-fail)
     (if (not (= (length form) 5))
         (syntax-fail "arith-if form incorrect size" form))
     (let ((%if (rename 'if)) (%temp (rename 'temp)))
       `(,(rename 'let)
         ((,%temp ,(cadr form)))
         (,%if
          (,(rename '<) ,%temp 0)
          ,(caddr form)
          (,%if
           (,(rename '=) ,%temp 0)
           ,(cadddr form)
           ,(car (cddddr form))))))))
   (type-error "Type error in an arith-if expression")
   (syntax-error "Syntax error in an arith-if expression")))
```

Figure 4-7: A macro-by-renaming arith-if macro

uses and explicitly managing identifier conflicts. Again, the first difference is in using a renaming procedure instead of node insertion to manage identifier conflicts. The other differences are found in the tasks omitted: processing subexpressions and manual attribute manipulation. Unlike the macro-by-renaming `n+` macro, this simplicity adversely affects the macro-by-renaming `arith-if` macro. This is illustrated in Figure 4-8, which contains some sample uses of the macro-by-renaming `arith-if`. Unlike the macro-by-procedure `arith-if`, the macro-by-renaming `arith-if` cannot distinguish between type errors caused by the "predicate" subexpression and type errors caused by conflicts between the types of the branch subexpressions. The reason for this is that in the macro-by-renaming interface a macro can only choose the error message to report when a type error is encountered — it cannot tailor the error message to the type error as a macro-by-procedure macro could.

```
Expression: (arith-if 1 2 3 4)
Expression type: int

Expression: (arith-if #t 1 2 3)
Type inference failed
Type error in an arith-if expression
In form: (arith-if #t 1 2 3)

Expression: (arith-if 0 #f 1 2)
Type inference failed
Type error in an arith-if expression
In form: (arith-if 0 #f 1 2)

Expression: (arith-if 1 2 3)
Syntax error: arith-if form incorrect size
(arith-if 1 2 3)
```

Figure 4-8: Using the macro-by-renaming `arith-if` macro

4.2.3 Assessment

The sample macro-by-renaming macros demonstrate three things:

1. Macro-by-renaming macros are much easier to create and understand than macro-by-procedure macros.

2. Type-aware macros can be created in the macro-by-renaming interface, since both sample macros reported type errors in terms of source code, not expanded code.

3. Macro-by-renaming macros give a macro writer much less control over type-checking than macro-by-procedure macros. This lack of control can be a disadvantage (because it makes error messages less detailed) and an advantage (a macro-by-renaming macro cannot break the type system the way a macro-by-procedure macro can).

In a sense, the macro-by-renaming interface is the opposite of the macro-by-procedure interface. It does not suffer from the major disadvantages of macro-by-procedure. Macros are written at a higher abstraction level (involving source-code transformations and error messages rather than nodes and attributes) and macros cannot damage the type system. However, the macro-by-renaming macros are also much less powerful than macro-by-procedure macros. This means that the macro-by-renaming interface is good for macros where users can easily deduce the sources of macro-related type errors. For macros with complex type rules, however, the macro-by-renaming interface limits what macros can do to help users understand type errors. One way to resolve this would be to extend the macro-by-renaming interface (though how to do so is not obvious). Another way to resolve this, demonstrated by the final interface, is to attempt to develop a new interface that balances the strengths of macro-by-renaming and macro-by-procedure, while minimizing their weaknesses.

4.3 Macro-by-attributes

The macro-by-attributes syntax-extension interface is the third syntax-extension interface I have implemented. It is a partially successful attempt to strike a balance between the power of the macro-by-procedure interface and the simplicity of the macro-by-renaming interface. It was inspired by the `syntax-case` macro system created by Hieb, Dybvig, and Bruggeman [7]. At this point, however, the relationship between the two interfaces is probably difficult to see. The first part of this section describes the macro-by-attributes interface. The second part of this section implements the `arith-if` and `n+` macros in the macro-by-attributes interface. The final part of this section assesses the macro-by-attributes interface.

4.3.1 Overview

A macro-by-attributes macro specification has up to four parts:

1. the symbol `attributes` that indicates the kind of macro specification,
2. a list of symbols, whose purpose will be explained later,
3. an optional `open` clause, that allows a macro-by-attributes macro to interface to the Scheme 48 module system (as in the discussion of macro-by-procedure, this clause will not be discussed in greater detail.),
4. and a list of transformation rules.

Each transformation rule begins with a *pattern*. The language that describes these patterns is closely related to the pattern language used by R^{5}RS Scheme's `syntax-rules` macros [9]. The differences between the two languages are that this pattern language includes wildcards and that it does not include vector or constant patterns. In this pattern language a pattern can be:

- an identifier,
- a wildcard (the symbol _),
- a list of one or more patterns,
- or a list of one or more patterns followed by an ellipsis (the symbol ...).

The pattern at the beginning of a transformation rule is matched against forms that attempt to use its associated syntax extension. The first transformation rule in a macro's list of rules whose pattern matches is the transformation rule that is used to create an output node. If no pattern matches, a syntax error is reported. The matching semantics for particular identifiers are controlled by the list of symbols that is the second part of the specification. If an identifier appears in that list of symbols, it only matches itself. If an identifier does not appear in that list, it matches any form and binds that identifier in the remainder of the transformation rule. The details of that binding are explained below. It is a syntax error for an identifier to be bound more than once by a pattern. Wildcards match any form and do not bind anything. A list of patterns matches a form if the form is a list of the same length and each part of the form matches its corresponding pattern.

A list of patterns is terminated by an ellipsis is the most complicated variety of pattern. They behave just like the ellipsis patterns in Scheme's `syntax-rules` macros. Ignoring the last pattern in the list, the first n-1 patterns of an ellipsis pattern must match the first n-1 elements of a form for a successful match. Any remaining elements of a form must each match the last pattern of an ellipsis pattern separately. Any variables bound by that last pattern are bound to lists of what they otherwise would have been bound to. Each element of any of these lists is taken from the match of the last pattern and the corresponding element of the tail of the form.

After the pattern, several different clauses can appear in a transformation rule. Collectively, these clauses specify how to construct the output node that corresponds to a particular transformation rule.

`bind` A `bind` clause binds variables that can be used by the rest of the transformation rule. `bind` clauses can appear anywhere in a transformation rule after that rule's pattern. A `bind` clause consists of the symbol `bind`, followed by a list of bindings. Each binding consists of a symbol and a Scheme expression. In the remainder of the transformation, each symbol is bound to the value obtained by evaluating the corresponding expression[8]. The bindings of a `bind` clause are evaluated sequentially so that earlier bindings are available to later expressions. Neither of the sample macros use a `bind` clause, but they are an important feature of the macro-by-attributes interface. They are essential when intermediate values need to be saved in a transformation rule.

[8]Variables bound by a `bind` clause should not conflict with variables bound by the pattern of a transformation rule, or there will be unpredictable results.

`form` Every transformation rule *must* have exactly one `form` clause. A `form` clause consists of the symbol `form`, followed by a Scheme expression. Before and inside the `form` clause of a transformation rule, the pattern variables of a pattern are bound to node futures (or trees whose leaves are node futures) that encapsulate the different forms (or trees of forms in the case of ellipsis patterns) into which a matching form was decomposed. The expression of a `form` clause is evaluated to produce the expansion corresponding to the use of the clause's transformation rule. Any identifiers in the expanded form will automatically be resolved relative to the defining environment of the syntax extension unless those identifiers are encapsulated in a node future. By default, only the decomposed parts of the input form are encapsulated in node futures, so macro-by-attributes macros are hygienic by default. This default can be overridden by encapsulating other parts of an expansion in node futures.

After any necessary identifier forwarding, this expanded form will be processed in the syntactic environment and context in which the original syntax extension (that contains the transformation rule) was used. After the `form` clause of a transformation rule, the pattern variables that were bound to node futures used in the expansion become bound to the corresponding future nodes (and similarly for variables bound to trees of node futures used in the expansion). Pattern variables that refer to node futures that were not used in the expansion should not be used in the remainder of the transformation rule, since they have no corresponding future nodes.

attribute clauses An attribute clause begins with a symbol that is not `bind` or `form`. This symbol names the attribute created by the clause. Attribute clauses can only appear after the `form` clause of a transformation rule. After the symbol, an attribute clause contains a Scheme expression. This expression is evaluated and the value is stored in the appropriate attribute of the output node of the syntax extension. Attributes should only be set by one clause of a transformation rule, though this restriction is not enforced. Attribute clauses should use the nodes that correspond to parts of the input form to construct their improved attributes.

4.3.2 Sample macros

As above, two sample macros are implemented using the macro-by-attributes interface: the `n+` macro and the `arith-if` macro.

`n+`

Figure 4-9 contains a macro-by-attributes version of the `n+` macro. This macro is implemented recursively using two transformation rules. The first rule (lines 4–10) is the base case for the `n+` macro. In this case, the `n+` macro has only one argument expression. This argument expression is the expansion of `n+`, and is required to be of numeric type. The second rule (lines 11–21) is the recursive case. This rule only

```
(define-syntax n+
  (attributes
   ()
   ((_ exp)
    (form exp)
    (w-check
     (lambda ()
       (unify-exp (create-type 'int) exp)
       (create-type 'int)))
    (unify-fail-msg "n+ subexpression must have numeric type"))
   ((_ exp exps ...)
    (form `(+ ,exp (n+ ,@exps)))
    (w-check
     (lambda ()
       (unify-exp (create-type 'int) exp)
       (for-each (lambda (exp)
                   (unify-exp (create-type 'int)
                              exp))
                 exps)
       (create-type 'int)))
    (unify-fail-msg "n+ subexpression must have numeric type"))))
```

Figure 4-9: Defining an n+ macro using the macro-by-attributes interface

matches n+ expressions that have at least two subexpressions because the case of one subexpression is handled by the first rule. The expansion adds the first subexpression to an n+ of the remaining subexpressions. The type-checking rule checks that *all* subexpressions (not just the first) have numeric type, so that type errors are not reported by the recursive uses of n+. As in the previous two interfaces, sample uses of the current n+ macro demonstrate that it is type-aware. Unlike the previous two interfaces, neither explicit syntax-checking nor manual identifier-management is necessary.

arith-if

The macro-by-attributes version of the arith-if macro (Figure 4-11) is a straightforward translation of the macro-by-procedure version of the arith-if macro (Figure 4-3). It consists of a single transformation rule, that decomposes an arith-if expression into its parts. The form clause of the rule reassembles these parts into an arith-if expansion (5–9). The only differences from the macro-by-procedure version are found in the lack of explicit syntax-checking and manual identifier-management. The type-checking procedure from the w-check attribute clause of the rule is almost identical to the type-checking procedure from the macro-by-procedure arith-if. Only the names of the variables that refer to the nodes that represent parts of an arith-if expression differ in the two procedures and corresponding variables refer to the same part of an arith-if. Similarly, the error-message procedure from the unify-fail-msg attribute clause of the macro-by-attributes arith-if is identical to the error-message procedure from the macro-by-procedure arith-if. As

```
Expression: (n+ 1 2 3)
Expression type: int

Expression: (n+)
Syntax error:
(n+)

Expression: (n+ 1 2 #t 4 5)
Type inference failed
n+ subexpression must have numeric type
Type clash between:
int
bool
In form: (n+ 1 2 #t 4 5)

Expression: (n+ #t)
Type inference failed
n+ subexpression must have numeric type
Type clash between:
int
bool
In form: (n+ #t)

Expression: (let ((+ 1)) (n+ 1 2 3 4))
Expression type: int
```

Figure 4-10: Using the macro-by-attributes n+ macro

```
(define-syntax arith-if
  (attributes
   ()
   ((arith-if pred minus zero plus)
    (form
     `(let ((temp ,pred))
        (if (< 0 temp)
            ,minus
            (if (= 0 temp) ,zero ,plus))))
    (w-check
     (lambda ()
       (let ((result-type (type-var form)))
         (let-fluid type-fail-msg-code 'pred-fail
                    (lambda ()
                      (unify-exp (create-type 'int) pred)))
         (let-fluid type-fail-msg-code 'branch-fail
                    (lambda ()
                      (unify-exps
                       (list result-type
                             result-type
                             result-type)
                       (list minus
                             zero
                             plus))))
         result-type)))
    (unify-fail-msg
     (lambda (code)
       (case code
         ((pred-fail)
          "arith-if: predicate does not have integer type")
         ((branch-fail)
          "arith-if: branches have incompatible types")))))))
```

Figure 4-11: Defining an `arith-if` macro using the macro-by-attributes interface

might be expected from similar implementations, the type-error messages from using the macro-by-attributes `arith-if` are the same as the type-error messages from the macro-by-procedure `arith-if`. The syntax-error messages only differ because the macro-by-attributes `arith-if` does not include any explicit syntax-checking.

```
Expression: (arith-if 0 1 2 3)
Expression type: int

Expression: (arith-if #t 1 2 3)
Type inference failed
arith-if: predicate does not have integer type
Type clash between:
int
bool
In form: (arith-if #t 1 2 3)

Expression: (arith-if 0 #f 1 2)
Type inference failed
arith-if: branches have incompatible types
Type clash between:
bool
int
In form: (arith-if 0 #f 1 2)

Expression: (arith-if 0 1 2)
Syntax error:
(arith-if 0 1 2)
```

Figure 4-12: Using the macro-by-attributes `arith-if` macro

4.3.3 Assessment

The `arith-if` macro illustrates the first advantage of the macro-by-attributes interface: it can be convenient shorthand for constructing expansions and nodes. The pattern-matching aspect of the interface provides a default level of syntax-checking that is better than that provided by the other two interfaces. In those interfaces, if an input form is syntactically invalid, syntax errors must manually be signaled while taking apart the form or constructing expanded code from it or syntax errors involving expanded code and/or system errors (like taking the `car` of an empty list) will be generated. In the macro-by-attributes interface, syntactically invalid forms generate their own syntax errors, and specific syntax-error handling is only needed if a macro writer wants to provide detailed syntax-error messages. Another convenience is the automatic identifier forwarding of the interface. That forwarding means that hygienic macros require no explicit identifier management. Attributes are specified quasi-declaratively, which at least obscures the details of mutating a node[9]. This

[9]In fact, the macro-by-attributes interface performs a small amount of node copying so that users of the interface do not have problems as the output node is mutated with new attributes.

shorthand can even be an advantage relative to them macro-by-renaming interface because that interface requires explicit syntax-checking and identifier management.

The n+ macro illustrates the second advantage of the macro-by-attributes interface: it makes it easier for a macro writer to structure the implementation of a syntax extension. If an implementation can be logically separated into distinct rules, like n+ was, the implementation will be easier to understand in macro-by-attributes than in macro-by-procedure or macro-by-renaming (where code that distinguishes different rules will be mixed in with code that implements different rules).

The advantages of the macro-by-attributes interface come from the new concept it introduces: the transformation rule. Transformation are rules simple, yet powerful units out of which macro implementations can be built. Transformation rules are also a framework that can incorporate many different conveniences including:

- pattern variables that connect pieces of a macro's source form to the nodes that represent their meaning, making it easier to access semantic information about those pieces,
- automatic renaming of inserted identifiers, so that, by default, macros are hygienic,
- and attribute clauses, a simpler syntax for modifying the attributes of the output node.

What is disappointing about the macro-by-attributes interface is that the transformation rule is the only new concept it introduces. There are no new concepts related to attribute-generation, in general, or type-checking, in particular, so writing macro type-checkers is no easier than in the macro-by-procedure interface. Attribute clauses may make attributes easier to specify, but they do not make attributes any easier to think about. This is in contrast to the macro-by-renaming interface, which introduces the concept of error-message clauses to allow macros to control their error messages. This concept makes it easy to write simple type-aware macros, at the cost of making complex type-aware macros harder to use. This assessment is, in the end, disappointing, not because the macro-by-attributes interface is itself disappointing, but rather because it says that the right set of concepts for creating type-aware macros has not yet been invented.

Chapter 5

Related Work

My syntax-extension framework and syntax-extension interfaces were invented to solve two problems that existing Scheme macro systems did not solve: combining syntax extension with static analyses and extending the syntax of languages with many kinds of language elements. There are many other systems that attempt to solve similar problems. Staged-computation systems are one kind of related system. Extensible-grammar systems are a second kind of related system. Two other systems related to my syntax-extension framework are not as easy to categorize.

5.1 Staged-computation systems

Staged-computation systems allow programmers to separate their programs into distinct stages. The benefit of this separation is that programs in earlier stages can construct and manipulate representations of programs in later stages, allowing them to control the code executed in those stages. This often leads to significant performance improvements, as earlier stages specialize later stages to the task at hand. Examples of staged-computation systems include Wickline, Lee and Pfenning's ML$^{\Box}$ [22] and Taha and Sheard's MetaML [21]. Syntax-extension systems, like my framework, deal with computations in two stages: compile-time, when syntax extensions are expanded, and run-time, when expanded code is executed. However, there are also significant differences that make syntax extension more than a limited form of staged computation:

- Staged-computation systems allow programmers to transform and manipulate programs. Syntax-extension systems, by contrast, allow programmers to transform and manipulate syntax. This means syntax-extension systems allow programmers to add new syntactic forms to a base language. Staged-computation systems, while they extend a base language with new program-manipulation operators, do not make the resulting language extensible. Some uses of staged computation may mimic new syntactic forms, but, among other limitations, they cannot mimic forms that bind variables.

- Staged-computation systems only permit limited cross-stage communication.

At best, when a representation of a later-stage program is constructed in an earlier stage, any earlier-stage variables it refers to are frozen. Those variables cannot be affected by the environment in which that program is eventually executed. From the viewpoint of hygienic macro systems, this means that staged-computation systems are automatically hygienic and that this hygiene cannot be overridden.

Since multi-stage programming is difficult, staged-computation systems usually have associated multi-stage type systems to assist multi-stage programmers. In particular, the staged-computation systems based on ML, mentioned above, have multi-stage type-inference systems. My syntax-extension interfaces, by contrast, do not provide a type system for macro writers, and rely on macro writers to correctly type-check the output of their syntax extensions. This is probably where future syntax-extension research can best learn from staged-computation research. Staged-computation systems are very close to automatically hygienic macro systems (like Scheme's `syntax-rules` system [9]) and their multi-stage type systems solve the problem of typing program transformations, which are similar to syntax transformations. This suggests that a system that infers typing rules for automatically hygienic macros could be developed by studying the multi-stage type systems used by staged-computation systems.

5.2 Extensible-grammar systems

Staged-computation systems are related to my framework because they try to solve problems associated with transformations and static analyses. Extensible-grammar systems, on the other hand, are related to my framework because they try to solve problems related to syntax extension and multiple language elements. The fundamental idea of extensible-grammar systems is to provide extended user languages through grammars that transform those user languages into simpler base languages. Cardelli, Matthes, and Abadi developed a particularly interesting extensible-grammar system [3]. Notable features of their system include:

- Their system automatically renames the identifiers in grammar rules, so that the transformations those rules create are hygienic. The system does not allow this automatic hygiene to be overridden, but that would be simple to add.
- In their system, grammar definitions are statically "type-checked." This "type-checking" is used to guarantee that the output of grammar productions is syntactically valid. By contrast, my untyped framework provides no guarantee that the expansion of a syntactically valid program will itself be syntactically valid.
- Their system keeps track of when identifiers are used as part of a binding and when identifiers are used to refer to a binding. This information makes correctly renaming the identifiers in grammar rules simpler.
- Their system allows grammar definitions to introduce syntax restrictions as well as syntax extensions. Syntax restrictions can be used to create languages that are easier to implement or understand.

One possible way to use extensible grammars would be to extend them with semantic actions and attribute manipulations. This would make it possible to adapt my framework to languages that are not based on S-expressions. Another interesting possibility suggested by extensible grammars is a typed interface for creating syntax extensions. A well-designed typed syntax-extension interface could guarantee that syntactically valid expanded code would be produced for every syntactically valid input program.

5.3 Other similar systems

Two other systems related to my syntax-extension framework have been based on hygienic macro technology. The first system is Maddox's semantic macro system [14]. His semantic macro system is the oldest system I am aware of that permits type-aware macros of any kind. The distinctive feature of Maddox's system is that his improved syntax-extension technology is embedded inside of a new programming language and its associated programming environment. My approach, on the other hand, is to provide a framework for constructing extensible programming languages and their associated extensible static analyses. These differences make Maddox's system more complete, since there is a base programming language and environment to work with, but also make his system less flexible because his improved extension technology cannot be easily separated from his programming language and programming environment.

The other related system is the **McMicMac** system of Krishanmurthi, Felleisen and Duba [13]. This system was also designed to allow embedded programming languages to be easily created. **McMicMac** and my syntax-extension framework are very similar — both can be used as general compiler-extension interfaces. The primary difference between the two systems is in how their technology is applied. In **McMicMac**, the power of a general compiler-extension interface is exposed through *micros* that allow authors of extensions to directly create and access elements of the compiler's intermediate representation of a program. My interfaces, on the other hand, limit access to the elements of my compiler's intermediate representation (nodes) so that general complier-extension features are suppressed and only syntax-extension features are exposed. The disadvantage of my approach is that compiler-extensions expressed through micros can be more powerful than syntax-extensions expressed through macros. The advantage of my approach is that the coupling of language extensions to the compiler implementation can be weaker.

Chapter 6

Future Work

The system described in this thesis is a proof-of-concept system. It was created to demonstrate that combining hygienic macro technology with extensible static analyses is possible and valuable. This proof opens up many possibilities for future research. Broadly speaking, this future work can be separated into three different categories: research on syntax-extension frameworks, research on user-language implementation and research on syntax-extension interfaces. These categories must overlap, however, because the goal of this research is to create better user languages through syntax extension, which necessarily involves a syntax-extension framework and a syntax-extension interface built on top of that framework. In the discussion below, ideas have been organized according to my impression of the focus of their work, but, because of the overlap, this organization can only be approximate.

6.1 Framework research

There are a many different projects that could be based on my syntax-extension framework. First, there are a number of different ways the existing framework can be made more powerful and usable. Second, there are interesting benefits that can result from combining this syntax-extension framework with other systems. Finally, an object-oriented re-implementation of my syntax-extension framework might make this new syntax-extension technology easier to develop and use.

6.1.1 Improving the existing framework

Probably the most significant way to improve the existing framework would be to re-design the context system. In the current implementation, contexts are very powerful, but their design is oriented towards the creation of full-fledged embedded languages. Simpler uses of contexts (like syntactically restricting definitions) are much harder than they need to be. With the exception of permitting definitions, the definition context created in section 2.4.4 is identical to the expression context it is based on. In the current system, however, making that small change required specifying every detail of the new context (see Figure 2-14). In a redesigned context system, it should

be easy to create variations of existing contexts. That sort of redesign would make it easier to take advantage of the potential power of contexts.

Another way to to improve the existing framework would be to simplify it. The existing framework has many partially redundant features. For example, both node insertion and node futures can be used to access nodes that represent parts of an output form. Both the procedure `capture-env` and generated identifiers can be used to hygienically create temporary identifiers for use in macro-expanded code. This redundancy suggests that a smaller, simpler framework could be as powerful as the existing one. The challenge here would be to create the smaller, simpler feature set for that framework.

One of the problems with the existing framework is that it is difficult for language implementations to protect their semantic information from syntax extensions that might damage it. In the current framework, all semantic information is stored as an attribute of some node. Once a syntax extension has access to a node, they can read or modify all of that node's attributes, so they can damage or destroy that node's semantic information. Syntax extensions do not even need to be malicious to cause problems, the complete lack of protection means that buggy syntax extensions can be just as bad as malicious ones. Denying syntax extensions access to nodes, as in the macro-by-renaming interface, solves this problem, but only at the cost of sharply limiting the power of syntax extension. A better solution would be to extend the framework with some way to finely control access to nodes and attributes. This would allow language implementations (and their associated static analyses) to create finer-grained interfaces to their semantic information. Through these interfaces syntax extensions could be prevented from damaging critical semantic information without significantly limiting their ability to improve static analyses.

The error-reporting system in the current framework is also one of its limitations. It mandates a fixed style of error reporting that limits languages and the programming environments built around them. Instead, the error system should mandate a minimum amount of error information that can be added to, if desired. This would allow clients of the framework to tailor error reporting to their needs without destroying the incentive to provide rich error information that is in the current version of the framework.

Another potential improvement to the existing framework would be to implement new node operations. New node operations might include extracting the parent node and child nodes of a particular node. Those operations would make it easier to write procedures (particularly static analyses) that traverse the nodes of a program. If the parent and child information could also be modified, traversals that rebuilt the nodes that constitute a program (such as an optimization pass of a compiler) would also be possible. Other new operations could generalize the conversion procedures of the current node implementation. In the current framework, since each node only has a single conversion procedure, each language implementation is limited to a single target machine language for its output code. If conversion procedures were generalized, then one language implementation could support many target languages, making languages implemented with the framework more flexible and useful. New node operations would make nodes a more powerful and more attractive intermediate

representation for a compiler.

6.1.2 Combining the framework with other systems

One aspect of the current framework implementation that will limit its use is the framework's dependence on S-expressions. Many popular programming languages are not based on S-expressions, and in its current form, the framework cannot be used to create syntax-extension interfaces for them. This is particularly unfortunate because the framework removes one important barrier to using syntax extension with popular programming languages: the conflict between syntax extension and static analyses. One solution to this problem would be to create S-expression variants of existing programming languages for the framework to operate on, but those variants might not be accepted. Another possibility would be to extend the framework so that it can support languages that are not based on S-expressions. This probably would involve combining the existing framework with extensible-grammar technology [3].

One of the things that is demonstrated by Maddox's semantic macro system [14] (and, to a lesser extent, by C compilers and Scheme systems) is that there is a great deal to gain from integrating syntax-extension technology into a programming environment. Among other things, integrated extension technology makes programming in extended languages seem natural. A well-designed programming environment would make language experimentation using this framework much easier. The challenge here is to retain the full flexibility of the original framework (including language and analysis independence) while taking advantage of what a programming environment can offer. Probably the simplest way to build a programming environment for this framework is to replace the syntax-extension system of Scheme 48 with a system based on this framework.

6.1.3 Object-oriented framework

An ambitious way to improve my syntax-extension framework would be to re-implement it in a statically-typed, object-oriented language. The main benefit of static typing would be in the compile-time checking of node and attribute manipulation. In the current framework, small errors in the kind of node produced or in the kind of attribute data used can be very difficult to understand or debug. On the other hand, one of the important sources of flexibility in the existing framework is found in the ability to create new kinds of nodes and to attach new varieties of information to existing nodes. In order to retain that flexibility in a statically-typed framework, an inheritance mechanism would be necessary. This is what suggests an object-oriented re-implementation. Even with inheritance, retaining the flexibility of the existing framework will be difficult because, as mentioned earlier, it is not obvious how to capture all of the important node relationships.

Many of the previously described improvements would fit naturally into an object-oriented re-implementation of the framework. Object-oriented languages allow objects to finely control access to their private data, so protecting semantic information would be easier. They usually have rich exception systems that could form the basis

of an improved error-reporting system and an object-oriented reconsideration of the node abstraction would be a natural time to add new node operations. Nevertheless, it is important to recognize that an object-oriented re-implementation, no matter how natural it may seem, is a radical reorganization of the framework which will present many new design challenges.

6.2 Language research

There are a number of ways that my demonstration language could be improved to make it a more interesting and usable language. One set of improvements centers around improving the type system that the language uses. Other improvements would allow the language to take greater advantage of possibilities offered by my syntax-extension framework.

6.2.1 Improving the type system

The most important change that could be made to my demonstration language's type system is to implement it within my syntax-extension framework, using type nodes and a type context or contexts. The immediate benefit of this change is that it would become possible to have language forms that include types. Among other things, these kinds of forms could be used for optional type annotation, either by programmers who don't want to use type inference or by type-aware macros that want to control how their expansions are type-checked. In particular, optional type annotation would be one way to improve the macro-by-renaming interface to make it more powerful.

Another change that should be made to the type system is the incorporation of parallel unification technology [15]. As described in Chapter 3, parallel unification is the only way to reliably distinguish between type errors caused by the argument forms of a macro and type errors caused by using a macro on a particular set of argument forms. The ability to make this distinction would make it possible to provide users of macros with more precise type-error messages.

The remaining problems of my demonstration language's type system come from the features it is missing because it is a prototype implementation. It supports only a limited set of primitive types, though that set could be easily extended. Its support for product types is weak (it only supports pairs) and it does not support any sum types. The type system also does not support recursive types, and the base set of types is not extensible. To a large extent, the lack of extensibility would be resolved by implementing the type system within the framework, but the other limitations need to be addressed separately.

6.2.2 Other improvements

Another way to improve the current language implementation would be to make it more comprehensively use the framework's support for multiple language elements.

There are a number non-expression language elements in the core language that are not accessible in the current implementation. They include: procedure parameter lists, the binding clauses used by `let` and `letrec`, and, most significantly, macro specifications. Exposing these things as language elements in their own right would make it much easier to experiment with and improve their syntax. This is particularly important in the case of macro specifications because experimenting with syntax-extension interfaces is probably the most important thing to do to make this technology more usable. What makes implementing macro specifications within the framework particularly difficult is that they include Scheme code interpreted by the underlying Scheme 48 system. One solution to this would be to change the framework so that it "bootstraps" its own extension language, the way the system the framework is based on [1] does. That way, the extension language would end up based on nodes, contexts and attributes and would be as extensible as the rest of the system. A closely related way to improve the current language implementation would be to implement Scheme's derived expressions as syntax extensions that introduce new language elements where appropriate.

Adding a module system to the language would be another way to improve the language implementation. It would be a large step towards making the demonstration language a reasonable language to program in. A module system would also make importing procedures from the underlying Scheme system simpler. Right now accessing these procedures depends on a trick using the global environment. Instead, the underlying system could be exposed as a Scheme module that other parts of the language could use. The biggest problem here is that the current implementation of syntactic environments may not interact well with modules, and so it may need to be changed.

6.3 Interface research

The final kind of research that my syntax-extension framework makes possible is research into syntax-extension interfaces for extensible static analyses and multiple language elements. This research is particularly important because all of the syntax-extension interfaces I have implemented so far have significant drawbacks that make them hard to use in some important cases.

One kind of interface that would be interesting to develop is an interface like Scheme's `syntax-rules` [9]. The results of staged-computation research suggest that in an interface where hygiene is enforced, like `syntax-rules`, it may be possible to infer the type-checking rule for a syntax extension. Even if that is not the case, a developing a high-level language for type-checking that goes with the high-level syntax-extension language of `syntax-rules` could be a very good way to create a powerful and usable extension interface for type-aware macros.

Another interesting interface experiment would be to create a statically-typed syntax-extension interface. A well-designed typed interface could help authors of syntax extensions control the transition from input syntax to objects that represent the meaning of that syntax. Then, instead of manipulating generic nodes to

improve the analysis of their expanded code, syntax extensions could manipulate specific language objects, making analysis improvements easier to understand and debug. This might work most naturally in an object-oriented re-implementation of the existing syntax-extension framework (see section 6.1.3), but a statically-typed syntax-extension interface need not depend on such a framework.

Another major limitation that all the current syntax-extension interfaces share is their weak support for developing and implementing (rather than using) embedded languages. The macro-by-renaming interface does not support developing or implementing embedded languages at all, and the other two interfaces rely on an author of a syntax extension manually controlling the contexts in which their syntax is processed. This is another improvement that could be guided by Maddox's semantic macro system [14]. Maddox's system has an elegant extension of quasiquotation that allows macros to manipulate multiple kinds of language elements at the same time.

Appendix A

Implementing do

The do macro is the most ambitious type-aware macro I have written for my demonstration language, Scheme/R. It implements an iteration form that is a variant of the R^{5}RS Scheme do macro [9]. Other than the underlying typed language, the major difference between this version of do and the Scheme do is that this version inserts a non-hygienic binding for the identifier `return`. That identifier is bound to a continuation that can be used to escape from the do loop. In Scheme/R, continuations cannot be typed correctly if they are represented as procedures, so they are given by types of the form (`contof` T) and are invoked using the `throw` procedure. Figure A-1 contains some sample legal and illegal uses of this do macro.

This do macro was implemented using the macro-by-procedure syntax-extension interface (section 4.1) because it was written before the other interfaces had been developed. The implementation of the do macro is complex, so it will be described in several stages. First, the syntax-checking and decomposition of a do form will be discussed. After that, the construction of the expanded code for do will be explained. Following that, the procedures that type-check a do expression will be described. Finally, the procedure that assembles these pieces into a functioning macro will be presented.

A.1 Syntax-checking and decomposition

The procedure `syntax-do` (Figure A-2) is used to syntactically check and decompose a do expression. It takes a single argument: the do form to be checked and decomposed. Lines 2–17 contain the syntax-checking code. Most of the syntax-checking is self-explanatory. A do expression must have at least three parts:

1. the do keyword itself,
2. a list of variable clauses,
3. and a termination clause.

The termination clause contains a test expression and a list of termination expressions. The test expression is evaluated at the beginning of every iteration of the loop. If

```
Expression: (do ()
                (#f 5))
Expression type: int
;; Infinite loop

Expression: (let ((x (cons 1 (cons 3 (cons 5 (null))))))
              (do ((x x (cdr x)) (sum 0 (+ sum (car x))))
                  ((null? x) sum)))
Expression type: int
Expression value: 9

Expression: (let ((x (cons 1 (cons 3 (cons 0 (cons 5 (null)))))))
              (do ((x x (cdr x)) (sum 0 (+ sum (car x))))
                  ((null? x) sum)
                (if (= (car x) 0)
                    (throw return sum)
                    (set! x x))))
Expression type: int
Expression value: 4

Expression: (let ((x (cons 1 (cons 3 (cons 0 (cons 5 (null)))))))
              (do ((x x (car x)) (sum 0 (+ sum (car x))))
                  ((null? x) sum)
                (if (= (car x) 0)
                    (throw return sum)
                    (set! x x))))
Type inference failed
do : types of variable and step expression incompatible
Type clash between:
(listof int)
int
In form: (x x (car x))

Expression: (let ((x (cons 1 (cons 3 (cons 0 (cons 5 (null)))))))
              (do ((x x (cdr x)) (sum 0 (+ sum (car x))))
                  (x sum)
                (if (= (car x) 0)
                    (throw return sum)
                    (set! x x))))
Type inference failed
do : test expression must have type bool
Type clash between:
bool
(listof int)
In form: x
```

Figure A-1: Sample uses of a type-aware do macro

the test expression evaluates to true, the termination expressions are evaluated in order and the value of the last expression is returned as the value of the entire do. This means there must be at least one termination expression after the test in a termination clause.

The procedure do-vars? used on line 6 of syntax-do checks the syntax of the do variable clauses. The first element of a variable clause must be the identifier bound by that clause. The second element of the variable clause is an init expression that supplies an initial value for that variable. Optionally, a variable clause can also have a third element, a step expression. The step expression of a variable clause is evaluated at the end of every loop iteration and supplies the value that the clause's variable is bound to in the next loop iteration. If a variable clause has no step expression, the variable itself is used as the step expression. In that case, except for side-effects, that value of that clause's variable does not change as the loop is executed. The procedure do-vars? also checks that no two variable clauses use the same identifier for their variable.

```
(define (syntax-do form)
  (if (not (> (length form) 2))
      (syntax-fail
       "do expression has incorrect size"
5      form))
  (if (not (do-vars? (cadr form)))
      (syntax-fail
       "do expression has illegally formatted variable bindings"
       (cadr form)))
10 (if (< (length (caddr form)) 1)
      (syntax-fail
       "do expression must have a test expression"
       (caddr form)))
  (if (< (length (caddr form)) 2)
15    (syntax-fail
       "do expression must have a termination expression"
       (caddr form)))
  (let ((do-vars (map do-var-fix (cadr form))))
    (let ((test (car (caddr form)))
20        (exps (cdr (caddr form)))
          (commands (cdddr form))
          (vars (map car do-vars))
          (inits (map cadr do-vars))
          (steps (map caddr do-vars)))
25    (values test exps commands vars inits steps))))
```

Figure A-2: The procedure syntax-do that syntactically checks and decomposes a do expression

After syntax-checking, the remainder of the syntax-do procedure separates the input form into the different components used to expand and type-check a do expression. The procedure do-var-fix (used on line 18) adds the default step expression (the variable itself) to variable clauses do not have a step expression. The remaining

bindings (lines 19–24) bind the various components of a `do` expression to different variables. The only variable that is not self-explanatory is the `commands` variable. This variable contains the list of commands for the `do` loop, which comes from the list of expressions after the termination clause of a `do` expression. When the test expression is false, these commands are evaluated for effect before the `step` expressions are evaluated to prepare for the next loop iteration. At the end of `syntax-do` (line 25) a multiple-value return is used to give the different components of the input `do` expression to the caller of `syntax-do`.

A.2 Expansion

The procedure `expand-do` (Figure A-3) is used to assemble the pieces of a `do` expression into the expanded code that implements it. It takes as arguments the pieces of a `do` expression that are returned by `syntax-do` with one addition: an identifier to bind the return continuation to[1]. The first part of the `expand-do` definition (lines 3–7) is evaluated before the `expand-do` procedure is created. This preamble binds variables that capture the global meanings of some identifiers used in `do` expansions[2]. This preamble is used because these global meanings should not change, so the `do` macro should not look them up every time it is expanded. The preamble is awkward and inconvenient and there are two other ways the same effect could have been achieved:

1. Save these global meanings through an action performed at the time the `do` macro is defined (taking advantage of the two-step keyword procedure creation of the macro-by-procedure interface). This was not done because the only convenient way to access these saved meanings would be to include `expand-do` in the keyword-creating procedure of the `do` macro and that procedure is already too large and too complex.

2. Change the language implementation and the macro-by-procedure interface so that important global meanings like these are included in the environment made available to macro-by-procedure macros. This improvement to the current macro-by-procedure interface implementation has not been made because I was more interested in exploring new syntax-extension interfaces than I was in making small improvements to the macro-by-procedure interface.

[1] As will be seen in section A.4, the arguments of `expand-do` are not the same values that are returned from `syntax-do`. Instead, node futures encapsulate the forms contained in those values so that the nodes corresponding to those forms can be recovered for type-checking. This detail does not affect `expand-do`, however, because node futures are processed exactly the way the forms they encapsulate would be processed.

[2] An careful reader will notice that `call/cc`, an identifier that refers to the global `call-with-current-continuation` procedure, is also accessed as an expression keyword, using the context `exp-keyword`, instead of the expected context `exp-context`. The reason `call/cc` is accessed through the context `exp-keyword` is that, due to a quirk in the language implementation, `call/cc` is implemented as an expression keyword.

After the preamble, the actual **expand-do** procedure is created. When called, the first thing this procedure does is generate a fresh identifier to name the procedure that implements iterations of the **do** loop (line 9). A quasiquoted template is used to construct the **do** expansion on lines 10–22 of **expand-do**. The first part of the expansion uses **call/cc** to capture and bind a return continuation for the **do** loop. Then the procedure that implements an iteration of the loop is recursively bound. The body of that procedure evaluates the test expression and then evaluates the termination expression or the rest of the loop, as appropriate. The loop is started by calling its implementation procedure with the appropriate initial values (line 22).

```
(define expand-do
  (let
      ((%begin (process 'begin global-env exp-keyword))
       (%letrec (process 'letrec global-env exp-keyword))
       (%lambda (process 'lambda global-env exp-keyword))
       (%if (process 'if global-env exp-keyword))
       (%call/cc (process 'call/cc global-env exp-keyword)))
    (lambda (test exp commands vars return inits steps)
      (let ((%do-loop (genid 'do-loop)))
        `(,%call/cc
          (,%lambda
           (,return)
           (,%letrec ((,%do-loop
                       (,%lambda
                        ,vars
                        (,%if ,test
                              ,exp
                              (,%begin
                               ,@(append
                                  commands
                                  `((,%do-loop ,@steps)))))))))
                     (,%do-loop ,@inits)))))))))
```

Figure A-3: The procedure **expand-do** that assembles the expansion of a **do** expression

A.3 Type-checking

There are four procedures used to implement the type-checking and type-error messages of the **do** macro. They are named **return-check**, **make-var-check**, **make-test-check**, and **do-fail-msg**. The definitions of these procedures are given in Figures A-4–A-7. The code that explains how these procedures are assembled into a complete type-inference routine will be explained in the following section, but these procedures are responsible for most of the type-checking and type-error generation in this **do** implementation.

The first procedure is the **return-check** procedure (Figure A-4). This procedure ensures that the variable node that represents the variable bound to the return continuation of a **do** expression has a continuation type. Before **return-check**, the type

of that variable node is unconstrained. In `return-check`, that variable node is unified against a continuation of some specific, but currently unconstrained type. This unification will never fail because the variable node starts out unconstrained. The important thing is that this constraint ensures that any non-continuation use of the variable that is bound to the return continuation will generate a type error.

```
(define (return-check return-node)
  (unify-exp
   (create-type '(contof ,(type-var 'return)))
   return-node))
```

Figure A-4: The `return-check` procedure used in the type-checking of a `do` expression

The next part of type-checking is the `make-var-check` procedure (Figure A-5). This procedure creates a procedure that can type-check a variable clause and can generate a specific error message if that variable clause is not well-typed. The `make-var-check` procedure takes a single argument: the `do` expression being type-checked. This form is used to distinguish type errors blamed on subexpressions of the `do` expression from type errors blamed on the `do` itself. This information will be used to improve type errors caused by incorrect variable clauses. The type-checking procedure returned by `make-var-check` takes four arguments: nodes that represent the variable, `init` expression, and `step` expression of a variable clause, as well as the original variable clause those nodes were derived from.

The body of the type-checking procedure is a call to `chain-type-fail` (section 3.1.3). The handler procedure passed to `chain-type-fail` (lines 4–7) is the part of the variable-clause type-checker that improves the type-error messages associated with variable clauses. When a type error is signaled, the form blamed for that type error is compared to the `do` form being type-checked. If they are different, the type error was not caused by the `do` (and, hence, was not caused by a variable clause of the `do`), so the type-error information is not changed. When the forms match, the current variable clause being type-checked must have caused the type error since that is the only part of the `do` that introduced a new type constraint. To make the error message more specific, the form blamed for the type error is replaced with the variable clause being type-checked.

The thunk passed to `chain-type-fail` (lines 8–15) contains the code that introduces the type constraints associated with a variable clause. The first constraint is that the type of the variable of a clause must be compatible with the type of the `init` expression of that clause. The second constraint is that the type of the variable of a clause must be compatible with the type of the `step` expression for that clause. Fluid bindings of `type-fail-msg-code` are used to generate different type-error messages when those two constraints are violated.

Since the test expression of a `do` follows the variable clauses of a `do`, `make-test-check` is used after `make-var-check`. This sort of ordering is one way to make the results of a type-checking procedure easier for programmers to understand. If a type-checking procedure introduces type constraints in the order a programmer reading the code would introduce type constraints, then the type-error messages gen-

```
(define (make-var-check form)
  (lambda (var-node init-node step-node do-var)
    (chain-type-fail
     (lambda (current-msg current-types current-form)
       (if (eq? current-form form)
           (values current-msg current-types do-var)
           (values current-msg current-types current-form)))
     (lambda ()
       (let ((var-type (variable-type var-node)))
         (let-fluid type-fail-msg-code 'init-fail
                    (lambda ()
                      (unify-exp var-type init-node)))
         (let-fluid type-fail-msg-code 'step-fail
                    (lambda ()
                      (unify-exp var-type step-node)))))))))
```

Figure A-5: The make-var-check procedure used to type-check do variable clauses

```
(define (make-test-check test-node form)
  (lambda ()
    (chain-type-fail
     (lambda (current-msg current-types current-form)
       (if (eq? current-form form)
           (values current-msg current-types (car (caddr form)))
           (values current-msg current-types current-form)))
     (lambda ()
       (unify-exp (create-type 'bool) test-node)))))
```

Figure A-6: The make-test-check procedure used to type-check the test of a do expression

erated from those constraints will seem more natural than the type-error messages generated from any other order of type constraints.

The make-test-check procedure (Figure A-6) takes two arguments: the node that represents the test expression to check and the do form being type-checked. It returns a procedure of no arguments that type-checks the test expression of that do. Like make-var-check, the body of the type-checking procedure created by make-test-check is a call to chain-type-fail. Also as in make-var-check, that call is used to improve the form associated with type-error messages caused by the do. In the case of make-test-check, the new form is the form from which the test node was created. The only constraint introduced by this type-checking procedure is that the test expression must have boolean type (line 9).

The final procedure used by the type-inference routine for do is the do-fail-msg procedure (Figure A-7). This is the procedure that generates different type-error messages for the different reasons a do expression can fail to type-check. The procedure has four different cases since there are four reasons a do expression might not be well-typed. The do-fail-msg procedure will be stored in the unify-fail-msg attribute of the node that represents the do, so that the type-error messages it generates will be used (see section 3.2.2).

```
(define (do-fail-msg code)
  (case code
    ((test-fail)
     "do : test expression must have type bool")
    ((init-fail)
     "do : types of variable and init expression incompatible")
    ((step-fail)
     "do : types of variable and step expression incompatible")
    ((call/cc-fail)
     "do : incompatible type thrown to return continuation")))
```

Figure A-7: The `do-fail-msg` procedure used to generate type-error messages for `do` expressions

A.4 Assembling the keyword procedure

The final step in implementing the `do` macro is creating a keyword procedure from the pieces that have been described so far. The procedure, `$do`, that creates the `do` keyword procedure is given in Figure A-8. The first thing that `$do` does is save the global meaning of the `begin` keyword in the variable `%begin` (line 3). The rest of the `$do` procedure consists of a `lambda` expression for the keyword procedure that actually implements `do` (lines 4–45).

The keyword procedure begins by calling `syntax-do` to syntactically check and decompose its input form. The components of the input `do` form returned by `syntax-do` are bound using a `receive` macro. The next thing the keyword procedure does is to create node futures that will be used to recover the future nodes corresponding to the forms contained in the different components of the input form (lines 8–13). Parts of the input form that consist of a single subform (like the test expression) are encapsulated inside of a single node future. Parts of the input form that are lists of subforms (like the `init` and `step` expressions of a `do`) turn into lists of node futures. The list of termination expressions contained in the input form is an exception. The type-inference routine for `do` does not need to access the termination expressions individually, so they are encapsulated inside of a `begin` expression contained in a single node future (line 9). A node future for the identifier that will be bound to the loop's return continuation is also created (line 11).

After the node futures have been created, they are passed to the procedure `expand-do` so that they will be used in the expanded code for the `do` expression (lines 14–15). After the expansion has been created, it is processed, and then the future nodes are extracted from the node futures that were used in the processed expansion (lines 16–26). The unimproved type-inference routine for the `do` expression is also extracted from the node that represents the expanded `do` (lines 19–20).

The type-inference routine for `do` expressions (lines 30–40) uses the procedures described in section A.3 to implement the appropriate type-checking rule. The first step is to use the procedure `return-check` to constrain the type of the node that represents the loop's return continuation (line 31). The second step is to create a variable-clause type-checker and use it to type-check the variable clauses of the `do`

```
(define $do
  (lambda (def-env)
    (let ((%begin (process 'begin global-env exp-keyword)))
      (lambda (form env context)
        (receive
         (test exps commands vars inits steps)
         (syntax-do form)
         (let* ((test (node-future/create test))
                (exp (node-future/create `(,%begin ,@exps)))
                (vars (map node-future/create vars))
                (return (node-future/create 'return))
                (inits (map node-future/create inits))
                (steps (map node-future/create steps))
                (output-code (expand-do test exp commands vars
                                        return inits steps))
                (output-node (process output-code
                                      env
                                      context))
                (output-check (node/attribute-get output-node
                                                  'w-check))
                (return-node (node-future/node return))
                (test-node (node-future/node test))
                (var-nodes (map node-future/node vars))
                (init-nodes (map node-future/node inits))
                (step-nodes (map node-future/node steps))
                (exp-node (node-future/node exp)))
           (node/attribute-set!
            output-node
            'w-check
            (lambda ()
              (return-check return-node)
              (for-each
               (make-var-check form)
               var-nodes init-nodes step-nodes (cadr form))
              (let-fluid
               type-fail-msg-code 'test-fail
               (make-test-check test-node form))
              (lets-fluid
               type-fail-msg-code 'call/cc-fail
               output-check)))
           (node/attribute-set!
            output-node
            'unify-fail-msg
            do-fail-msg)
           output-node))))))
```

Figure A-8: The $do procedure that creates a keyword procedure that implements do expression

expression (lines 32–34). The variable-clause type-checker is called with each variable clause from the input `do` form (to ensure any clause blamed for a type error does not contain an originally omitted `step` expression) as well as the corresponding elements of the `var-nodes`, `init-nodes`, and `step-nodes` lists. After the variable clauses have been checked, the type-checker moves on to check the test expression of the `do` expression. It uses `make-test-check` to make a thunk that type-checks the test expression and calls that thunk inside a fluid binding for `type-fail-msg-code` so that an appropriate type-error message is generated if the test expression does not type-check (lines 35–37).

At this point there is only one more type-constraint left to implement in the type-inference rule for `do` expressions: the types of any values thrown to the loop's return continuation must be compatible with the type of any values returned from the `do` (through the last termination expression). Instead of directly checking this constraint, the improved type-inference routine for `do` uses the unimproved type-inference routine to ensure that this remaining constraint is satisfied (lines 38–40). Since all the other type constraints have been checked, if the unimproved type-checker signals a type error (that is blamed on the `do` expression) that type error must be caused by a conflict between the type of the return continuation and the type of any values returned. The final part of the keyword procedure returns the improved node that represents the expanded `do` expression (line 45).

Bibliography

[1] Alan Bawden. Macros and modules made (fairly) simple. Unpublished macro system code, available at `http://www.bawden.org/mtt/mmms.tgz`.

[2] Alan Bawden and Jonathan Rees. Syntactic closures. In *ACM Conference on LISP and functional programming*, pages 86–95, 1988.

[3] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible syntax with lexical scoping. Technical Report 121, Digital Equipment Corporation Systems Research Center, February 1994.

[4] William Clinger. Hygienic macros through explicit renaming. *Lisp Pointers*, IV(4):17–23, December 1991.

[5] William Clinger and Jonathan Rees. Macros that work. In *ACM Conference on Principles of Programming Languages*, pages 155–162, 1991.

[6] Chris Hanson. A syntactic closures macro facility. *Lisp Pointers*, IV(4):9–16, October-December 1991.

[7] Robert Hieb, Kent Dybvig, and Carl Bruggeman. Syntactic abstraction in scheme. Technical Report 355, Indiana University, Computer Science Department, July 92.

[8] Peter Housel. An introduction to macro expansion algorithms. Incomplete article available at `http://www.cs.indiana.edu/scheme-repository/doc.misc.html`.

[9] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998.

[10] Richard Kelsey and Jonathan Rees. A user's guide to scheme 48. Converted to HTML by Margaret Fleck and available at `http://www.cs.hmc.edu/~fleck/envision/scheme48/user-guide.html`.

[11] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[12] Eugene M. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *ACM Converence on LISP and Functional Programming*, pages 151–161. acm, August 1986.

[13] Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From macros to reusable generative programming. Technical Report TR 00-364, Department of Computer Science, Rice University, 2000.

[14] William Maddox. Semantically-sensitive macroprocessing. Technical Report CSD-89-545, University of California, Berkeley, December 1989.

[15] Bruce J. McAdam. On the unification of substitutions in type inference. Technical Report ECS-LFCS-98-384, Laboratory for Foundations of Computer Science, The University of Edinburgh, March 1998.

[16] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, Cambridge, UK, second edition, 1992.

[17] Eric Raymond and Guy L. Steele. *The New Hacker's Dictionary.* MIT Press, Cambridge, MA, USA, second edition, 1993.

[18] Jonathan Rees. Another module system for scheme. A memo describing the Scheme 48 module system, available as part of Scheme 48 version 0.53, which can be obtained at `http://s48.org/0.53/scheme48-0.53.tgz` or at `http://www.neci.nj.nec.com/homepages/kelsey/scheme48-0.53.tgz`.

[19] Olin Shivers and Brian Carlstrom. Scsh reference manual. The reference manual for the Scheme shell release 0.5.2, available at `ftp://www-swiss.ai.mit.edu/pub/su/scsh/scsh-manual.ps`.

[20] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.

[21] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. Technical Report CSE-99-007, Oregon Graduate Institute of Science and Technology, January 6, 1999.

[22] Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and Modal-ML. *ACM SIGPLAN Notices*, 33(5):224–235, May 1998.