Refining Syntactic Sugar:
Tools for Supporting Macro Development

A dissertation presented
by

Ryan Culpepper

to the Faculty of the Graduate School
of the College of Computer and Information Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Northeastern University
Boston, Massachusetts

April, 2010

Abstract

Over the past two decades, Scheme macros have evolved into a powerful API for the compiler front-end. Like Lisp macros, their predecessors, Scheme macros expand source programs into a small core language; unlike Lisp systems, Scheme macro expanders preserve lexical scoping, and advanced Scheme macro systems handle other important properties such as source location. Using such macros, Scheme programmers now routinely develop the ultimate abstraction: embedded domain-specific programming languages.

Unfortunately, a typical Scheme programming environment provides little support for macro development. The tools for understanding macro expansion are poor, which makes it difficult for experienced programmers to debug their macros and for novices to study the behavior of macros. At the same time, the language for specifying macros is limited in expressive power, and it fails to validate syntactic correctness of macro uses.

This dissertation presents tools for macro development that specifically address these two needs. The first is a stepping debugger specialized to the pragmatics of hygienic macros. The second is a system for writing macros and specifying syntax that automatically validates macro uses and reports syntax errors.

Acknowledgments

I offer many thanks to my advisor, Matthias Felleisen, for his guidance, perspective, inspiration, and correction, all in large quantities.

I thank my committee for their help in shaping my research. Mitch Wand planted seeds of subtle and beautiful ideas in his programming languages classes. Olin Shivers has repeatedly demonstrated the utility of little embedded languages. Matthew Flatt constructed the system that is the context of the work described in this dissertation.

I thank my contemporaries at Northeastern University. We learn more from each other than from anyone else. In particular, I thank Richard Cobbe, Dave Herman, Sam Tobin-Hochstadt, Carl Eastlund, Stevie Strickland, Felix Klock, and Christos Dimoulas for research collaborations, for interesting discussions, and for friendship.

I thank the Fortress team at Sun: Guy Steele, Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, and Sukyoung Ryu.

I thank my many excellent teachers, especially Ann Padgett, Can Akkoç, Ian Barland, Keith Cooper, and Frank Jones.

I thank my family for their love and support.

Contents

List of Figures

CHAPTER 1

Introduction

Modern programming languages support a variety of abstraction mechanisms: higher-order functions, class systems, expressive type systems, module systems, and many more. With these, programmers can develop code for reuse, establish single points of control for a piece of functionality, and decouple distinct components for separate development. These mechanisms, however, require the programmer to adapt the program to the language. The phrasing of the program must fit the structure fixed by the designers of the language. As Paul Hudak [42] has argued, the ultimate abstraction is to adapt the language to the program. The ideal programming language should therefore allow programmers to develop and reuse entire sub-languages.

The Lisp and Scheme family of languages empower programmers to do just that. Through *macros*, they offer the programmer the ability to define new forms of expressions and definitions with custom behavior. These *syntactic* abstractions may introduce new binding positions, perform analyses on their subterms, and change the context of their subexpressions. In some systems, macros can share information, perform sophisticated computation, and detect and report errors, all at compile time. As some Scheme implementers have put it, macros are a true API to the front-end of the compiler.

When attached to an expressive language [24] macros suffice to implement many general-purpose abstraction mechanisms as libraries. For example, programmers have used macros to extend Scheme with constructs for pattern matching [74], relations in the spirit of Prolog [23, 38, 63], extensible looping constructs [22, 62], class systems [5, 56], component systems [17, 33, 70], and software contract checking [28], among others. As a result, a macro-enabled language can have a core of a dozen or so syntactic constructs but appear to implement a language as rich in features as Common Lisp [65].

Programmers have also used macros to handle metaprogramming tasks traditionally implemented outside the language using preprocessors or special compilers: such as parser generators [54], tools for engineering semantics [25], and specifications of compiler transformations [60]. Macros are used to transform programs to support novel forms of execution, such as servlets with serializable continuations [55, 51] and programs linked to a

theorem-proving system [21]. Finally, macros enable the creation of full-fledged *sister languages* of the host language, such as extensions of the base language with types [68], laziness [6], and functional reactivity [15]. Since the dialects are defined by translation to a common host, programmers can link together modules written in different dialects.

In order to support these increasingly ambitious applications, macro systems have had to evolve, too, because true syntactic abstractions must be indistinguishable from features directly supported by the language implementation. Historically, macros had humble beginnings in Lisp systems, where they were compile-time functions over program fragments, usually plain S-expressions. Unfortunately, these simplistic macros do not really define abstractions. For example, because Lisp macros manipulate variables by name alone, references they introduce can be captured by bindings in the context of the macro's use. This tangling of scopes reveals implementation details instead of encapsulating them. In response, researchers developed the notions of macro hygiene [46], referential transparency [14, 20], and phase separation [30] and implemented macro systems guaranteeing those properties. Now macros manipulate program representations that carry information about lexical scope; affecting scope takes deliberate effort on the part of the macro writer and becomes a part of the macro's specification.

Unfortunately, typical Scheme programming environments do not provide sufficient support for macro programming. Generic programming tools for the host language are not sufficient; macro programming is a specialized task and it needs specialized programming language support.

Macro systems lack usable inspection tools. Programmers find debugging macros difficult, and novice macro writers find the expansion process bewildering and opaque. Other languages have debuggers that expose the execution model to the programmer and enable interactive exploration. A debugger that steps through the implementation of macro expansion would be misguided; the focus should be on the properties of macro expansion, not the properties of its implementation.

Macro systems also provide poor support for writing robust macros. Language constructs implemented via macros should be indistinguishable from those supported directly by the compiler. In particular, they should clearly specify their valid syntax, reject invalid programs, and produce diagnostic messages in terms of the appropriate constructs and concepts. Existing macro systems make validation and error reporting either impossible or laborious. When error checking must be coded by hand, in many cases it is not done at all. Functional languages have developed techniques for describing and enforcing program structure, including types and dynamic software contracts. Such tools cannot be simply used without adaptation, however; a system for structuring macros should describe syntax at the level that macros deal with, not at the level of its representation as data.

My work addresses these two problems facing macro writers, the lack of inspection capabilities and the poor support for developing robust macros. I

present two tools tailored to macro programming.

To address the lack of inspection tools for the macro expansion process, I have developed a stepping debugger specialized to the concerns of macro expansion. This debugger presents the macro expansion process as a linear sequence of rewriting steps of annotated terms; it graphically illustrates the binding structure of the program as expansion reveals it; and it adapts to the programmer's level of abstraction, hiding details of syntactic forms that the programmer considers built-in.

To assist programmers with the design of robust syntactic abstractions, I have created a new system for specifying and implementing macros. It incorporates an expressive pattern language for specifying syntax, an abstraction facility that reflects existing informal practice in structuring syntax specification, and a matching algorithm that automatically generates appropriate syntax error messages when a macro is misused.

The rest of my dissertation consists of two parts. Chapters 2–5 discuss the macro stepper, including its design, implementation, and basis in models of macro expansion. Chapters 6–8 discuss my system for specifying syntax, named `syntax-parse`. Chapter 9 concludes with a discussion of related work.

CHAPTER 2

Debugging Macros

A debugger must be tailored to its target language. Debugging a logic program is fundamentally different from debugging assembly code. Most importantly, a debugger should reflect the programmer's mental model of the language. This model determines what information the debugger displays as well as how the programmer accesses it. It determines whether the programmer inspects variable bindings resulting from unification or memory locations and whether the programmer explores a tree-shaped logical derivation or steps through line-sized statements.

Following this analysis, the design of our debugger for macros is based on three principles of Scheme macros:

1. Macro expansion is a rewriting process.
2. Macros respect lexical scoping.
3. Macros create layered language abstractions.

This chapter elaborates these three principles and their influence on the design of our debugger. The final section examines existing tools in light of these principles and discusses their deficiencies.

2.1 Rewriting

Intuitively, macro definitions specify rewriting rules, and macro expansion is the process of applying these rules to the program until all of the macros have been eliminated and the program uses only primitive forms.

The macro expander does not rewrite terms freely. Rather, it looks at terms only in *expansion contexts* such as at the top-level and in certain positions within primitive syntactic forms. The body of a `lambda`-form, for example, is an expansion position; its formal parameter list is not. A term that is not in an expansion context may eventually be placed into such a context as enclosing macros are rewritten. We reserve the words *expression* and *definition* for terms that ultimately occur in an expansion context. The expression structure of a program, then, is gradually discovered during expansion.

```
;; (with-lock expr) acquires a lock, evaluates the expression,
;; and releases the lock.
(define-syntax with-lock
  (syntax-rules ()
    [(with-lock body)
     (dynamic-wind
       (lambda () (acquire-the-lock))
       (lambda () body)
       (lambda () (release-the-lock)))]))
```

Figure 2.1: A simple macro

Terms that are not expressions are simply part of the syntactic structure of the macro or primitive form, such as the formal parameter list of a `lambda` expression or a `cond` clause. Macros commonly perform case analysis on their syntax, extracting subexpressions and other meaningful syntax such as parameter lists and embedding these fragments in a new expression. Modern macro systems provide convenient notations for this case analysis and transformation of syntax.

R^{6}RS Scheme [64] provides two ways of writing macro definitions. One is a limited pattern-based macro facility [47] called `syntax-rules`. The other, called `syntax-case` [20], allows programmers to compute syntax transformations using the full power of Scheme plus pattern-matching.

A macro definition using `syntax-rules` has the following form:

```
(define-syntax macro
  (syntax-rules (literal ...)
    [pattern_1 template_1]
    ...
    [pattern_1 template_n]))
```

This definition directs the macro expander to replace any occurrences of terms matching one of the listed patterns with the corresponding template. In the process, the expander substitutes the occurrence's subterms for the pattern variables. The `literals` determine which identifiers in the pattern are matched for equality rather than treated as pattern variables.

For example, Figure 2.1 shows the definition of `with-lock`, a macro that helps a programmer protect shared resources. With this macro, the programmer ensures that the enclosed expression is executed only in the proper context: with the lock acquired. In addition, the programmer can be sure that the function calls to acquire and release the lock are balanced.

Given the definition of `with-lock`, macro expansion consists roughly of scanning through a program to determine which occurrences of `with-lock` to replace. An example expansion is shown in Figure 2.2. The initial pro-

```
(define (print-items header items)
  (with-lock
   (begin (print header) (for-each print items))))

⟶

(define (print-items header items)
  (dynamic-wind
    (lambda () (acquire-the-lock))
    (lambda () (begin (print header) (for-each print items)))
    (lambda () (release-the-lock))))
```

Figure 2.2: Expansion of `with-lock`

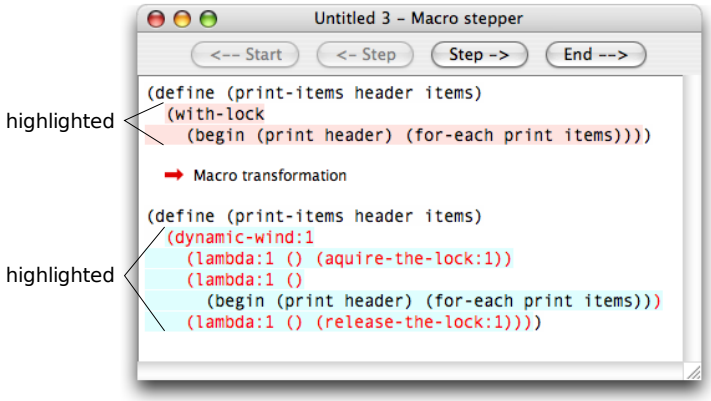

Figure 2.3: Macro stepper expansion of `with-lock`

gram contains one occurrence of the macro. It is rewritten to the second program, which contains no occurrences of the macro. At that point, macro expansion is complete.

The macro stepper shows the macro expansion process as a sequence of rewriting steps as in Figure 2.2. Each step consists of two program texts that differ by the application of a single macro rewriting rule. The stepper highlights the site of the rule's application in the first program and the site of the macro's result in the second program.

The tool's graphical user interface permits programmers to go back and forth in an expansion and also to skip to the beginning or end of the rewriting sequence. When a program contains multiple top-level terms, additional buttons appear that allow navigation between terms. The ideas for this interface have been borrowed from the run-time stepper for PLT Scheme and similar steppers for Lisp-like languages [12, 50, 69].

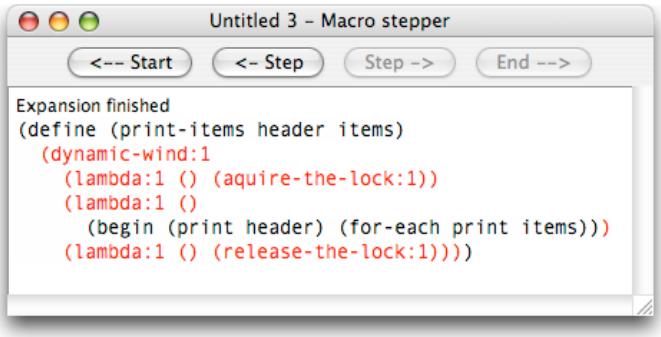

Figure 2.4: Macro stepper expansion of `with-lock`, final state

Figure 2.3 shows the program from above in the macro stepper. The stepper shows the original code and the result of applying the `with-lock` rewriting rule. The figure indicates the highlighted regions: the macro use and its expansion.

If the programmer navigates forward, the stepper indicates that expansion is complete and shows the final program, as shown in Figure 2.4. The final program displayed by the stepper is *almost* what one would expect from naively applying the rewrite rules, but some of the identifiers have a "`:1`" suffix. These come from the macro system's lexical scoping mechanism. In this particular case, the marks do not affect the meaning of the program, but often they do and the next section explains when and how.

2.2 Lexical scoping

Scheme requires macro expansion to be hygienic [14, 46]—to respect lexical scoping according to the following two principles:[1]

1. References introduced by a macro refer to the binding occurrence apparent at the site of the macro's definition.

2. Binding occurrences of identifiers introduced by the macro (not taken from the macro's arguments) bind only other identifiers introduced by the same macro transformation step.

An identifier is introduced by a macro if it comes from the macro definition, not from the macro's arguments. For `syntax-rules` macros, this amounts to the identifier appearing in the template.

The gist of these requirements is that macros act "like closures" at compile time. For example, the writer of `with-lock` can be certain of the meaning of `acquire-the-lock`, `release-the-lock`, and even `lambda`, regardless

[1] Unless the programmer explicitly programs such a violation using the hygiene-breaking features of a system such as the `syntax-case` system.

of whether some of those names have different bindings where the macro is used. Similarly, a macro can create a temporary variable binding without capturing references in expressions that the macro receives as arguments.

Thinking of macro expansion in terms of substitution provides additional insight into the problem and its solution. A single macro expansion step involves two sorts of substitution and thus there are two sorts of capture to avoid. The first substitution brings the macro body to the use site; this substitution must not allow bindings in the context of the use site to capture names present in the macro body (condition 1). The second places the macro's arguments into the body; names in the macro's arguments must avoid capture by bindings in the macro body (condition 2), even though the latter bindings might not be immediately apparent.

To see how bindings might not be apparent, consider this definition:

```
(define-syntax (munge stx)
  (syntax-rules ()
    [(munge e) (mangle (x) e)]))
```

It is not clear what role the template's occurrence of `x` plays. If `mangle` acts like `lambda`, then `x` would be a binding occurrence, and by the hygiene principle it must not capture any free occurrences of `x` in the expression `e`.[2] If, however, `mangle` acts like `begin`, then the occurrence of `x` is a reference to a name defined in the context of `munge`'s definition. The first hygiene principle guarantees that if `x` is a reference, it refers to the `x` in scope where the macro was *defined*. Without performing further expansion steps, the macro expander cannot tell the role `x` plays and consequently which hygiene principle governs its behavior. Thus it must delay the treatment of `x` until the use of `mangle` has been replaced with core syntax. The program representation must contain enough information to allow for both possibilities.

In short, the binding structure of a program is gradually discovered during macro expansion. The complete structure of the program is not known until it is completely expanded.

Hygienic macro systems usually implement the proper lexical scoping by annotating their program representations with timestamps, or *marks*. An identifier's mark, if one is present, indicates which macro rewriting step introduced it. Even though we customarily represent marks as numbers, we care only about their identity, not the order in which they occur.

For illustration, Figure 2.5 shows the macro `myor`, which introduces a binding for a temporary variable. Here is a program that uses the macro:

```
(define (nonzero? r)
  (myor (negative? r) (positive? r)))
```

[2]There are devious macros, related to the "ill-behaved macros" discussed in Section 2.3, that challenge conventional intuitions about hygiene by searching through expressions looking for identifiers to capture [45]. Regardless, the hygiene principles we describe give a useful, if imperfect, guide to the scoping properties of macros.

```
;; (myor e1 ... eN) evaluates its subexpressions in order
;; until one of them returns a non-false value. The result
;; is that non-false value, or false if all produced false.
(define-syntax myor
  (syntax-rules ()
    [(myor e) e]
    [(myor e1 e ...)
     (let ([r e1])
       (if r r (myor e ...)))]))
```

Figure 2.5: Definition and use of `myor`

If macro expansion followed the naive rewriting process, the temporary variable `r` introduced by the macro would capture the reference to the formal parameter named `r`:

```
;; RESULT OF NAIVE EXPANSION, CAUSES CAPTURE
(define (nonzero? r)
  (let ([r (negative? r)])
    (if r r (positive? r))))
```

Instead, the macro expander adds a mark to every macro-introduced identifier. Using our notation for marks, the Scheme expander gives us this term:

```
(define (nonzero? r)
  (let:1 ([r:1 (negative? r)])
    (if:1 r:1 r:1 (myor:1 (positive? r)))))
```

The expander marks introduced identifiers indiscriminately, since it cannot predict which identifiers eventually become variable bindings. When a marked identifier such as `let:1` does not occur in the scope of a binding of the same marked identifier, the mark is ignored; hence the `let:1` above means the same thing as `let`. In contrast, the two references to `r:1` occur inside of a binding for `r:1`, so they refer to that binding. Finally, the occurrence of the unmarked `r` is *not* captured by the binding of `r:1`; it refers to the formal parameter of `nonzero?`.

Marks are a crucial part of Scheme's macro expansion process. Our macro debugger visually displays this scope information at every step. The display indicates from which macro expansion step every subterm originated by rendering terms from different steps in different colors. If there are too many steps to give each one a distinguishable color, then the stepper adds numeric suffixes to marked identifiers, such as `let:1`.[3]

[3]Because of the colors, it is never unclear whether a suffix is part of the actual identifier or a macro stepper annotation.

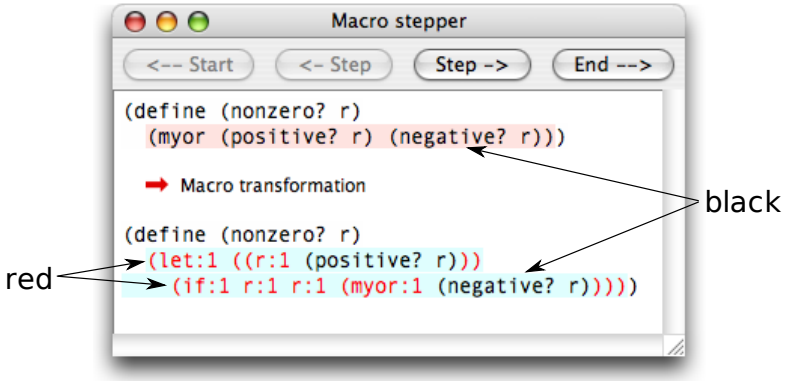

Figure 2.6: Macro stepper expansion of `myor`

Figure 2.6 shows what the macro stepper displays for the expansion of `nonzero?`. The introduced binding has a mark, and the mark matches the introduced references. The marked identifiers are rendered in red and the unmarked identifiers are rendered in black.

Recall that macro expansion gradually discovers the binding structure of the program. The discovery process occurs in two parts. When the macro expander sees a primitive binding form, it discovers the *binding site* of a variable. It does not yet know, however, what matching occurrences of those identifiers constitute *references* to those bindings. Other intervening macros may introduce bindings of the same name. Only when the macro expander encounters a variable reference does the binding become definite.

The macro stepper displays its knowledge of the program's binding structure through binding arrows, inspired by the arrows of DrScheme's Check Syntax tool [27]. When the macro expander uncovers a binding, the macro stepper draws a *tentative* binding arrow from the binding occurrence to every identifier that it may possibly bind.[4] The stepper annotates tentative binding arrows with a "?" symbol at the bound end. When the macro expander finally resolves a variable to a specific binding, the arrow becomes *definite* and the question mark disappears. Alternatively, macro expansion may uncover a closer binding occurrence of the same name, in which case the target of the tentative arrow changes.

The next step in the expansion, including some of the binding arrows, is shown in Figure 2.7. Note that the references to the marked `r:1` variables definitely belong to the binding of `r:1`, and the first reference to the unmarked `r` is definitely bound by the procedure's formal parameter. The final occurrence of `r` is still tentative because macro expansion is not fin-

[4]The macro stepper cannot predict references created using hygiene-breaking facilities like `datum->syntax`, of course, but once they are created the macro stepper does identify them as potential references.

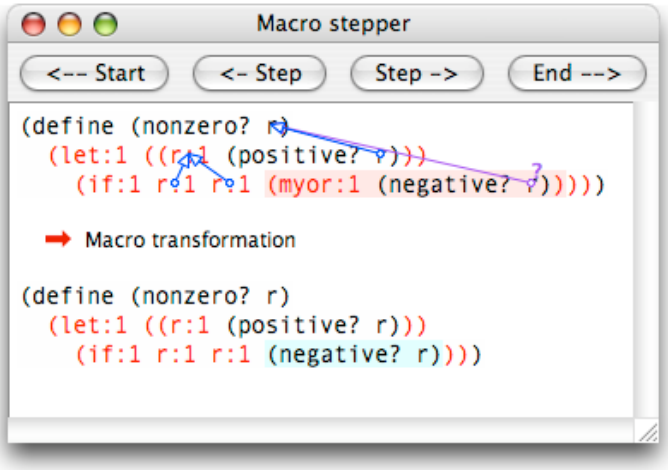

Figure 2.7: Macro stepper expansion of `myor`, step 2

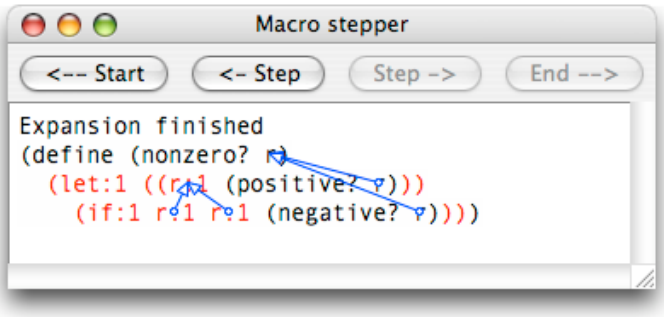

Figure 2.8: Macro stepper expansion of `myor`, final state

ished with its enclosing term. Stepping forward once more reveals the fully expanded program. Since all of the references have been uncovered by the macro expander, all of the binding arrows are now definite. The final state is shown in Figure 2.8.

Binding information is important in debugging macros. Because macros often manipulate the lexical scope in a subtle manner, the shape of an intermediate term alone may not reveal an error. The program seems correct, but variables do not get bound or get bound in the wrong place. In these cases, the stepper's visual presentation of lexical binding is critical.

To illustrate this point, consider the creation of an `if-it` macro, a variant of `if` that binds the variable named `it` to the result of the test expression for the two branches. Figure 2.9 shows a naive attempt, called `if-it1`, that tries simply inserting the identifier `it` in the macro's result.

```
(define-syntax if-it1
  (syntax-rules ()
    [(if-it1 test then else)
     (let ([it test]) (if it then else))]))
```

Figure 2.9: First attempt to write `if-it`

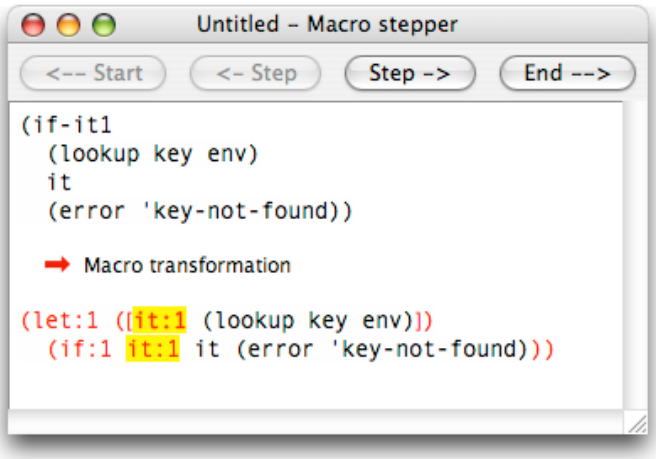

Figure 2.10: Macro stepper: expansion of `if-it1`

```
(define-syntax (if-it2 stx)
  (syntax-case stx ()
    [(if-it2 test then else)
     (with-syntax ([it (datum->syntax #'if-it2 'it)])
       #'(let ([it test]) (if it then else)))]))
```

Figure 2.11: Second attempt to write `if-it`

This implementation of `if-it1` fails to achieve the desired behavior. Using `if-it1` in a program reveals that `it` is not bound as expected, leading to an error. The macro stepper, as shown in Figure 2.10 illustrates the problem. The `it` inserted by the macro is marked with a timestamp, indicated by the color and the "`:1`" suffix. A marked identifier cannot capture an unmarked reference, however.

A correct implementation of `if-it` must insert an `it` identifier without marks, or rather with the *same* set of marks as the original expression using the macro. This cannot be done using `syntax-rules`, but can be done using the `syntax-case` macro system [20, 64]. A `syntax-case` macro's right-hand side contains Scheme code that performs the macro transformation, and it

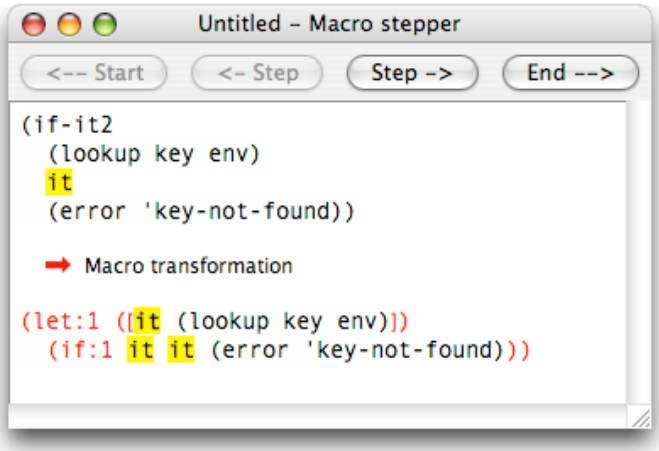

Figure 2.12: Macro stepper: expansion of `if-it2`

can use syntax-manipulating procedures such as `datum->syntax` and binding forms such as `with-syntax`. These facilities offer the programmer explicit control over lexical scoping, which is necessary for "breaking hygiene" as the programmer intends with this macro. Figure 2.11 shows the correct implementation of the macro, and Figure 2.12 shows the macro stepper, which confirms that the inserted binding occurrence of `it` is unmarked.

In addition to acting as a diagnostic tool for incorrect bindings, the macro stepper also provides a visual explanation of the behavior of `datum->syntax`: it transfers the marks from its first argument to its second. That is, the occurrence of `it` in `if-it2`'s expansion is unmarked because the `if-it2` keyword was unmarked. Thus the macro stepper helps reinforce and enhance the programmer's mental model of hygienic macro expansion.

Note: PLT Scheme attaches additional properties to lexical tokens and syntax trees during macro expansion. The programmer can view those properties, on demand, in a separate panel of the macro stepper.

2.3 Layered language abstractions

Preprocessors and compilers translate from a source language to a target language. So do macros, but in a language with macros, there are many source languages and many target languages.

For example, PLT Scheme's trait library [34] is implemented as a macro that translates trait declarations to mixin declarations [35], i.e., as functions from classes to classes. Mixins have been part of PLT Scheme's standard library for a long time—and they are implemented as macros in terms of "basic" Scheme. This "basic" language, however, is itself implemented using macros atop a smaller language kernel, following Steele's precedent [66].

```
(define (aeval expr)
  (match expr
    [(cons op args)
     (apply (aeval op) (map aeval args))]
    [(? number? n)
     n]
    [(? symbol? x)
     (if-it2 (lookup x) (fetch it) (error 'unbound))]))
```

Figure 2.13: A use of multiple macros

Such layers of linguistic abstraction are common in languages that support macros, and they demand special support from debuggers. After all, a debugger for a high-level language should not bother the programmer with low-level details. In a program with layered abstractions, however, the line between high-level and low-level is fluid. It varies from one debugging session to another. The debugger must be able to adjust accordingly.

A macro debugger that operates at too low a level of abstraction burdens the programmer with extra rewriting steps that have no bearing on the programmer's problem. In addition, by the time the stepper reaches the term of interest, the context has been expanded to core syntax. Familiar syntactic landmarks may have been transformed beyond recognition. Naturally, this prevents the programmer from understanding the macro as a linguistic abstraction in the original program. For the `class` example, when the expander is about to elaborate the body of a method, the `class` keyword is no longer visible; field and access control declarations have been compiled away; and the definition of the method no longer has its original shape. In such a situation, the programmer cannot see the forest for all the trees.

The macro debugger overcomes this problem with *macro hiding*. Specifically, the debugger implements a policy that determines which macros the debugger considers *opaque*. Designating a macro as opaque effectively removes its rewriting rules and adds expansion contexts, as if the macro were a language primitive. The programmer can modify the policy as needed. The macro debugger does not show the expansion of opaque macros, but if an occurrence of an opaque macro has subexpressions, it does display the expansions of those subexpressions in context.

Consider the `if-it2` macro from the previous subsection. After testing the macro itself, the programmer wishes to employ it in the context of a larger program, an evaluator for arithmetic expressions. The code, which uses the pattern-matching form called `match` from a macro library, is shown in Figure 2.13.

Since `match` is a macro, the stepper would normally start by showing the expansion of the `match` expression. It would only show the expansion

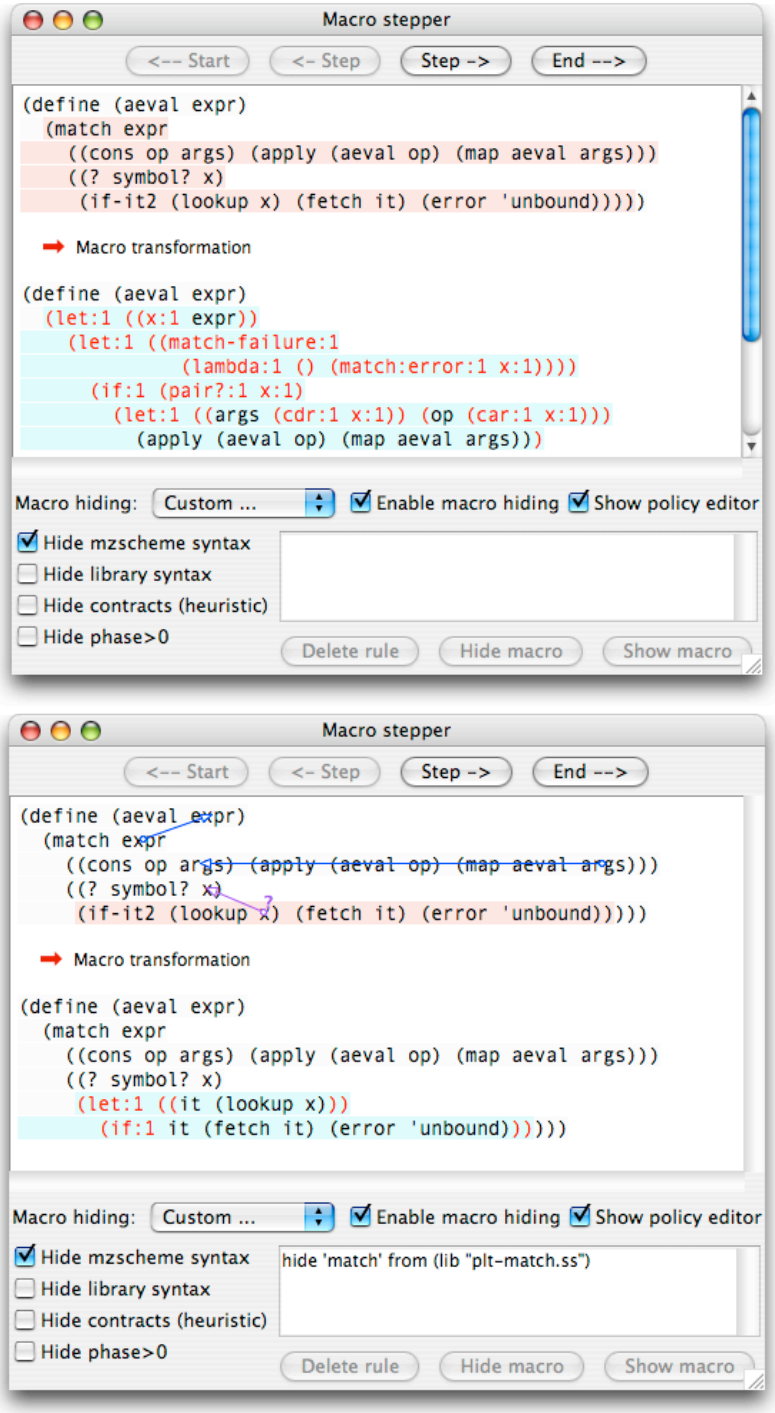

Figure 2.14: Example of macro hiding

of `if-it2` later, within the code produced by `match`. That code is of course a tangled web of nested conditionals, intermediate variable bindings, and failure continuations, all of which is irrelevant and distracting for someone interested in inspecting the behavior of `if-it2`. The left-hand side of Figure 2.14 shows this view of the program's expansion. Note the hiding policy editor, which includes no directive about `match`.

To eliminate the noise and to focus on just the behavior of interest, the programmer instructs the macro stepper to consider `match` opaque. The macro stepper allows the programmer to update the policy by clicking on an occurrence of a macro and changing its designation in the policy editor. When the policy changes, the stepper immediately updates the display.

When the programmer hides `match`, the stepper no longer treats `match` as a rewriting rule but as a primitive form that contains expansion contexts: the expression following the `match` keyword and the right-hand side of each clause. With `match` considered primitive, there is only one rewriting rule that applies to this program. See the right-hand side of Figure 2.14.

Now the programmer can concentrate on `if-it2` in its original context, without the distraction of irrelevant expansion steps. As the screenshot shows, macro hiding does not interfere with the macro stepper's ability to determine the program's binding structure.

In principle, a macro hiding policy is simply a predicate on macro occurrences that determines whether to show or hide its details. In practice, programmers control the macro hiding policy by designating particular macros or groups of macros as opaque, and that designation applies to all occurrences of a macro. The policy editor allows programmers to specify the opacity of macros at two levels. The coarse level includes two broad classes of macros: those belonging to the base language [31] and those defined in standard libraries; these classes of macros correspond to the "Hide mzscheme syntax" and "Hide library syntax" checkboxes. Hiding these allows programmers to focus on the behavior of their own macros, which are more likely to contain bugs than library macros. For finer control, programmers can override the designation of individual macros.

We could allow policies to contain complex provisions, and we are still exploring mechanisms for specifying these policies. We expect that additional user feedback will be necessary to find the best ways of building policies. So far we have discovered only two additional useful policies. The first hides a certain class of macros that implement run-time contract checking [28]. The second hides the expansion of macros used in the right-hand sides of macro definitions, e.g., uses of `syntax-rules` and `syntax-case`, which are actually macros. We have added ad hoc support for these cases.

2.3.1 Limitations

Because of macro hiding, our debugger presents steps that actually never happen and it presents terms that the expander actually never produces. In

most cases these spurious terms are plausible and instructive, but for some macros, macro hiding produces terms that are not equivalent to the original program. These faulty terms generally arise when a macro inspects one of its subexpressions or otherwise treats it as more than just an expression. In a sense, such macros violate the expression abstraction by inspecting the expression's underlying representation.

Macro hiding relies on the expression abstraction to justify its reordering of expansion steps. One way of breaking the expression abstraction is to quote a subexpression, as in this example:

```
(define-syntax time
  (syntax-rules ()
    [(time e) (time-apply (quote e) (lambda () e))]))
```

If `time` is opaque, then macro hiding shows expansion in the context of the `time` macro. But the resulting terms do not preserve the meaning of the original term, because they report different quoted constants. Thus we regard this macro as ill-behaved *with respect to the expression abstraction*.

A more subtle way of breaking the expression abstraction is analyzing a subexpression or a subterm that is only sometimes used as an expression:

```
(define-syntax no-constants
  (syntax-rules (quote)
    [(no-constants (quote c)) (error "no constants please")]
    [(no-constants e) (list e)]))
```

Here's the expansion of one use of `no-constants`:

```
    (no-constants (quasiquote x))
⟶  (list (quasiquote x))
⟶  (list (quote x))
```

If `no-constants` is hidden, the expansions of its subexpressions are shown as if they occurred in the context of the `no-constants` special form, thus:

```
    (no-constants (quasiquote x))
⟶  (no-constants (quote x))
```

But if the final term were evaluated, it would result in an error rather than the correct result, a list containing the symbol `'x`.

The macro debugger also cannot hide a macro that clones a subexpression. Since the subexpression occurs multiple times in the output, the macro debugger finds multiple expansion subsequences associated with the same context, the context of the subexpression in the original macro use. When it discovers the contexts are the same (or overlap), the macro stepper issues a warning and continues, showing that instance of the macro. This limitation is indirectly beneficial, since expression cloning is almost always an error in the macro definition.

In general, the expansions of a cloned expression may be different, so there is no good way to fit them into the same context. As a special case, however, duplicated variable references are allowed, since their expansions are trivial and no conflicts arise.

Note that not every duplication of a subterm is a duplication of an expression. For example, consider the following macro:

```
(define-syntax push!
  (syntax-rules ()
    [(push! s e)
     (set! s (cons e s))]))
```

The macro duplicates `s`. But since `s` appears as the target of an assignment, `s` must be a variable name, and that occurrence is not an expression. The other occurrence of `s` is as a variable reference expression, which has no expansion steps. So this macro is regarded as well-behaved by macro hiding. If, however, `set!` allowed targets that included subexpressions (as Lisp's `setf` does, for example), then this `push!` macro would be capable of duplicating subexpressions.

2.4 Existing tools

Most Scheme implementations provide limited support for inspecting macro expansion. Typically, this support consists of two procedures, called `expand` and `expand-once` or `macroexpand` and `macroexpand-1` [65]. Some implementations also provide a procedure called `expand-only`.

When applied to a term, `expand` runs the macro expander on the term to completion. It does not give the programmer access to any intermediate steps of the expansion process. If expansion aborts with an error, `expand` usually shows the faulty term, but it does not show the term's context. The second common tool, `expand-once`, takes a term and, if it is a macro use, performs a single macro transformation step. Since `expand-once` returns only a simple term, it discards contextual information such as the syntactic environment and even the place where the macro expander left off. Finally, `expand-only` acts like `expand` but takes as an extra argument the list of macros to expand.

Neither of these tools suffices for debugging complex macros. In general, macro programmers need to debug both syntactic and logical errors in their macros. Some macro bugs, such as using a macro in a way that does not match its rewriting rules, cause the macro expander to halt with an error message. Other bugs pass through the expander undetected and result in a well-formed program—but the *wrong* well-formed program.

Programmers using `expand` may catch syntactic bugs, but they usually have a difficult time figuring out what sequence of transformations produced the term that finally exposed the flaw. The error raised by `expand` only re-

ports the faulty term itself, not the context the term occurs in. Furthermore, `expand` races past logical errors; it is up to the programmer to decipher the fully-expanded code and deduce at what point the expansion diverged from the expected path.

Using `expand-once`, on the other hand, leads the programmer through expansion step by step—but only for the root term. Thus `expand-once` is useless for finding errors that occur within nontrivial contexts. Such errors can occur in macros that depend on bindings introduced by other macros.

In fact, `expand` suffers from one more problem: it reveals too much. For example, Scheme has three conditional expressions: `if`, `cond`, and `case`. Most Scheme implementations implement only `if` as a primitive form and define `cond` and `case` as macros. Whether a special form is a primitive form or a macro is irrelevant to the programmer, but macro expansion reveals the difference. It is thus impossible to study the effects of a single macro or a group of related macros in an expansion, because `expand` processes *all* macros, even those the programmer is not concerned with. Using `expand-only` avoids this problem. but it otherwise exhibits the same flaws as `expand`. Our idea of macro hiding can be seen as a generalization of `expand-only`.

Implementing better debugging tools than `expand` and `expand-once` is surprisingly difficult. It is impossible to apply many techniques that work well in run-time debugging. For example, any attempt to preprocess the program to attach debugging information or insert debugging statements fails for two reasons: first, until parsing and macro expansion happens, the syntactic structure of the tree is unknown; second, because macros can inspect their arguments, annotations or modifications are likely to change the result of the expansion process [71].

These limitations imply that a proper debugger for macros cannot be built atop the simplistic tools exposed to Scheme programmers; debugging macros requires cooperation from the macro expander itself.

CHAPTER 3

Models of Macro Expansion

The reliability of the macro stepper depends on the faithfulness of its rendition of the expansion process. This chapter provides models of hygienic macro expansion for a small subset of Scheme. In Chapter 4 we explain the implementation of the macro stepper and show how the expansion models support the implementation design.

The language of our model is a greatly simplified subset of Scheme. It has only a few special forms, including restricted variants of the binding forms `lambda` and `let-syntax`; we omit constants and we give application an explicit keyword (`app`). We omit a formal account of individual macro transformers; the transformation process is represented by an undefined auxiliary relation.

Our models of macro expansion are adaptations of the `syntax-case` model of Dybvig et al. [20]. We first present the syntax algebra, a datatype for representing program terms with lexical scoping information. Next we present two reformulations of the macro expansion process, a big-step semantics and a small-step reduction semantics, and we prove the two models equivalent. Both are used in the validation of the macro stepper's design.

3.1 Syntax algebra

The unique challenge of representing terms in the presence of macro expansion is that the binding structure of the program is obscured by macros. Binding structure is gradually revealed by the expansion process, but the expansion process must respect the programs binding structure. For this reason neither simple textual representations nor binding-dependent representations such as de Bruijn-indexed terms or higher-order abstract syntax are adequate. The syntax datatype must accommodate incomplete binding information that is incrementally updated as expansion progresses.

Figure 3.1 shows our syntax datatype, Term, an adaptation of the algebra developed by Dybvig et al. [20]. It extends the traditional S-expression representation by generalizing from symbols to "identifiers" and adding two new variants of identifier:

Term	e	::=	x
		|	$(e \ \cdots \ e)$
Identifier	x, y, z	::=	s
		|	$\mathrm{mark}(x, mark)$
		|	$\mathrm{subst}(x, y, a)$
Symbol	s		
Address	a		
Mark	$mark$		

Figure 3.1: Syntax datatype

- $\mathrm{mark}(x, mark)$ consists of an identifier coupled with a timestamp, or *mark* in `syntax-case` terminology. It has the potential to act either like x or like a fresh name, depending on how it is ultimately used.

- $\mathrm{subst}(x, y, a)$ consists of an underlying identifier, x, with a delayed alpha-renaming $y \mapsto a$.

Our datatype differs from that of Dybvig et al. in two ways. First, we distinguish symbols—undecorated identifiers—from the fresh tokens created during expansion as the targets of renamings; we call the latter *addresses*. This simplifies the management of fresh names and makes it easier to describe fully-expanded programs as a grammar. Second, our models of expansion produce syntax, rather than a different sort of abstract syntax tree.

Figure 3.2 defines the relevant operations on identifiers:

- $\mathrm{resolve}(x)$ applies all delayed alpha-renaming operations and produces the symbol for the binding that x refers to.

- $\mathrm{marksof}(x)$ computes the set of marks that the macro expander has applied to x; identifiers in the original program have no marks.

- $x =_{\mathrm{free}} y$ if x and y are equivalent as references; that is, if they both refer to the same binding.

- $x =_{\mathrm{bound}} y$ if x and y are equivalent for the purposes of creating new bindings; that is, if a binding of one would capture the other.

We also extend mark and subst to compound terms; they simply propagate the marks and renamings to the identifier leaves of the term

The syntax datatype and its operations are designed to keep track of the *identity* of bindings. What those bindings *mean* is managed by the macro expander through some separate mechanism. For the sake of simplicity, however, we enforce the invariant that identifiers whose *meanings* are the primitive syntactic forms are recognizable by their binding identities.

$$
\begin{array}{rcl}
\mathrm{resolve}(s) & = & s \\
\mathrm{resolve}(\mathrm{mark}(x, \mathit{mark})) & = & \mathrm{resolve}(x) \\
\mathrm{resolve}(\mathrm{subst}(x, y, a)) & = & \begin{cases} a & \text{if } x =_{\mathrm{bound}} y \\ \mathrm{resolve}(x) & \text{otherwise} \end{cases} \\
\\
\mathrm{marksof}(s) & = & \emptyset \\
\mathrm{marksof}(\mathrm{mark}(x, \mathit{mark})) & = & \mathrm{marksof}(x) \uplus \{\mathit{mark}\} \\
\mathrm{marksof}(\mathrm{subst}(x, y, s)) & = & \mathrm{marksof}(x) \\
A \uplus B & = & (A \cup B) - (A \cap B) \\
\\
\mathrm{subst}(\texttt{(}e_1 \cdots e_n\texttt{)}, y, a) & = & \texttt{(}\mathrm{subst}(e_1, y, a) \cdots \mathrm{subst}(e_n, y, a)\texttt{)} \\
\mathrm{mark}(\texttt{(}e_1 \cdots e_n\texttt{)}, \mathit{mark}) & = & \texttt{(}\mathrm{mark}(e_1, \mathit{mark}) \cdots \mathrm{mark}(e_n, \mathit{mark})\texttt{)}
\end{array}
$$

$$
\begin{array}{rcl}
x =_{\mathrm{free}} y & \text{iff} & \mathrm{resolve}(x) = \mathrm{resolve}(y) \\
y =_{\mathrm{bound}} y & \text{iff} & x =_{\mathrm{free}} y \text{ and } \mathrm{marksof}(x) = \mathrm{marksof}(y)
\end{array}
$$

Figure 3.2: Operations and relations on syntax

Fully-expanded term	g	::=	x
		|	$\texttt{(}\mathit{kw}_{\mathit{lambda}}\ \texttt{(}x \cdots\texttt{)}\ g\texttt{)}$
		|	$\texttt{(}\mathit{kw}_{\mathit{app}}\ g \cdots\texttt{)}$
Keywords	$\mathit{kw}_{\mathit{lambda}}$	::=	x where $\mathrm{resolve}(x) = \texttt{lambda}$
	$\mathit{kw}_{\mathit{app}}$	::=	x where $\mathrm{resolve}(x) = \texttt{app}$
	$\mathit{kw}_{\mathit{letstx}}$	::=	x where $\mathrm{resolve}(x) = \texttt{let-syntax}$
PrimitiveName	$\mathit{primitive}$	::=	`lambda` | `app` | `let-syntax`

Figure 3.3: Grammar of core terms

$$\alpha\text{-conversion Environment } \xi = \mathrm{Address} \to \mathrm{Symbol}$$

$$\begin{aligned}
[\![x]\!]_\xi &= \begin{cases} \xi(\mathrm{resolve}(x)) & \text{when } \mathrm{resolve}(x) \in \mathrm{Address} \\ \mathrm{resolve}(x) & \text{when } \mathrm{resolve}(x) \in (\mathrm{Symbol} - \mathrm{PrimitiveName}) \end{cases} \\
[\![(kw_{lambda}\ (x)\ g_b)]\!]_\xi &= \texttt{(lambda (}s\texttt{) } [\![g_b]\!]_{\xi[\mathrm{resolve}(x) \mapsto s]}\texttt{)} \\
&\quad \text{where } s \text{ does not occur in } g_b \text{ or } \mathrm{rng}(\xi) \\
[\![(kw_{app}\ g_1\ g_2)]\!]_\xi &= \texttt{(app } [\![g_1]\!]_\xi\ [\![g_2]\!]_\xi\texttt{)} \\
\xi_0 &= \emptyset
\end{aligned}$$

Figure 3.4: Interpretation of fully-expanded terms

Invariant 1. *If the identifier x occurs in a position such that the macro expander treats it as a reference, then the following propositions are equivalent:*

- $\mathrm{resolve}(x) \in \mathrm{PrimitiveName}$
- *x refers to the primitive named by* $\mathrm{resolve}(x)$

That is, the macro expander's context-sensitive lookup mechanism maps primitive names to the corresponding primitive forms.

Both of our formulations of macro expansion (big-step and reduction semantics) enforce this invariant. Note that this invariant holds only for identifiers in suitable positions. For example, in the program

```
(lambda (app) app)
```

the property does not hold for the second occurrence of `app`, because the macro expander is not ready to treat that identifier as a reference. The macro expander treats only the *renamed* identifier as a reference, and the renamed identifier does not resolve to `app`, but to a fresh address.

Invariant 1 allows us to describe fully-expanded terms as a grammar, shown in Figure 3.3. Note that not every fully-expanded term is a legal program; for example, the grammar permits "variable references" that refer to primitive forms or nonexistent addresses. Also note that a fully-expanded term does not contain any `let-syntax` expressions; macro definitions are stripped out by expansion.

We define the interpretation of fully-expanded terms $[\![e]\!]_\xi$ via a translation to simple S-expressions. That is, the decorated syntax variants are removed. The translation is shown in Figure 3.4. It is parameterized by an environment ξ mapping addresses to symbols; the symbols are chosen to avoid conflicts with existing references in the program. The function is

Environment	ρ	=	$(\text{Symbol} \cup \text{Address}) \to \text{Denotation}$
Denotation	*denotation*	=	$\mathsf{Prim}(\text{Symbol})$
		|	$\mathsf{Macro}(\text{Transformer})$
		|	Var
Store	S	=	set of Marks and Addresses
Transformer	t		

Expansion relation	$\rho \vdash e, S \Downarrow g, S'$
Transformation relation	$t, e \Downarrow_{tr} e'$

Figure 3.5: Domains and relations

undefined on terms containing invalid variable references. If the reference resolves to an address a, $\xi(a)$ is undefined if the address is not bound; if the reference resolves to a symbol, it is checked against the primitive names.

3.2 Big-step semantics

We first reformulate the recursive equations of Dybvig et al. [20] as a big-step semantics, similar to an earlier formulation of a different expansion algorithm by Clinger and Rees [14].

The expansion process uses an environment to record the meanings of bound names and a store for the generation of fresh addresses and marks. The basic judgment thus has the shape

$$\rho \vdash e, S \Downarrow g, S'$$

and says that the term e fully macro expands into g in the environment ρ, transforming the store S into S'. Figure 3.5 summarizes the domains and metavariables of the big-step semantics.

This model of macro expansion uses an environment to carry the meaning of identifiers. The environment maps a symbol to its denotation: a primitive syntactic form, a macro transformer, or the variable designator. Determining the meaning of an identifier involves first resolving the substitutions to a symbol, and then consulting the environment for the meaning of the symbol.

$$\rho_0(s) = \begin{cases} \mathsf{Prim}(s) & \text{when } s \in \text{PrimitiveName} \\ \mathsf{Var} & \text{otherwise} \end{cases}$$

The environment's usage satisfies Invariant 1. The environment represents the macro expander's context-sensitive meaning function; it is consulted to determine the meaning of a reference. The environment is never extended with a new Prim mapping and existing mappings are not changed

or removed, so the environment maps the names of primitives, and no other names, to the corresponding primitive forms, as the invariant requires.

Figure 3.6 lists the expansion rules for macros and primitive forms. Ignore the shaded parts of the semantics until Section 4.4. Determining which rule applies to a term involves resolving its leading keyword and consulting the environment for its meaning.

A macro use is distinguished by a leading keyword that is mapped to transformation rules in the syntactic environment. The expander uses those rules to transform the macro use into a new term. We use the $t, e \Downarrow_{tr} e'$ judgment to indicate this process. We omit the definition of the transformation function, since it does not affect the structure of our debugger. The expander marks the expression before transformation and then marks the transformation's result. Identifiers in the original term get the mark twice, and these marks cancel out (marks are self-canceling). Identifiers introduced by the rule's template get the mark only once, and the mark thus acts as a timestamp for the identifier's introduction. When a marked identifier is used in a binding construct, the alpha-renaming affects only identifiers with the same name *and* the same relevant renamings *and* the same marks.

The LAMBDA rule generates a fresh address for the `lambda` expression's formal parameter and creates a new formal parameter and a new body term with the old formal mapped to the address. Then it extends the environment, mapping the address to the Var designator, and expands the new body term in the extended environment. Finally, it reassembles the `lambda` term with the new formal parameter and expanded body.

When the macro expander encounters an occurrence of a `lambda`-bound variable in the body expression, it resolves the identifier to the address from the renaming step and discovers that the environment maps it to the variable designator (otherwise it is a misused macro or primitive name—an error). It then returns the identifier unchanged. This case is handled by the VARIABLE rule. The same rule also covers global variables, since the initial environment considers all identifiers resolving to symbols—other than the names of primitive syntax—to be global variables.

The APP (application) rule simply expands the application's subexpressions without extending the environment.

The LET-SYNTAX rule works like the LAMBDA rule but maps the fresh address to the macro transformer instead of the variable designator. Once the body has been fully expanded, there is no more need for the macro bindings, so it does not recreate the `let-syntax` expression.

The store is used to record the marks and addresses that have been used thus far during macro expansion. The initial store is the empty set, and the initial program must not contain any marks or addresses.

$$\frac{\begin{array}{c} e = (x_{macro}\ \ e_1 \cdots e_n) \qquad \rho(\text{resolve}(x_{macro})) = \mathsf{Macro}(t) \\ \langle mark, S_1\rangle = \text{fresh-mark}(S) \qquad e_{mark} = \text{mark}(e, mark) \\ t, e_{mark} \Downarrow_{tr} e'_{mark} \qquad e' = \text{mark}(e'_{mark}, mark) \qquad \rho \vdash e', S_1 \Downarrow g, S_2, T_k \\ T = \mathsf{Visit}(e); \mathsf{EnterMacro}; \mathsf{Mark}(mark); \mathsf{ExitMacro}(e'); T_k \end{array}}{\rho \vdash e, S \Downarrow g, S_2, T} \quad \textsc{Macro}$$

$$\frac{\begin{array}{c} e = (kw_{lambda}\ \ (x)\ \ e_b) \qquad \rho(\text{resolve}(kw_{lambda})) = \mathsf{Prim}(\texttt{lambda}) \\ \langle a, S_1\rangle = \text{fresh-address}(S) \qquad x' = \text{subst}(x, x, a) \\ e'_b = \text{subst}(e_b, x, a) \qquad \rho' = \rho\,[a \mapsto \mathsf{Var}] \qquad \rho' \vdash e'_b, S_1 \Downarrow g_b, S_2, T_b \\ T = \mathsf{Visit}(e); \mathsf{PrimLambda}; \mathsf{Rename}(a); T_b; \mathsf{Return}((kw_{lambda}\ \ (x')\ \ g_b)) \end{array}}{\rho \vdash e, S \Downarrow (kw_{lambda}\ \ (x')\ \ g_b), S_2, T} \quad \textsc{Lambda}$$

$$\frac{\rho(\text{resolve}(x)) = \mathsf{Var} \qquad T = \mathsf{Visit}(x); \mathsf{Variable}; \mathsf{Return}(x)}{\rho \vdash x, S \Downarrow x, S, T} \quad \textsc{Variable}$$

$$\frac{\begin{array}{c} e = (kw_{app}\ \ e_1\ \ e_2) \qquad \rho(\text{resolve}(kw_{app})) = \mathsf{Prim}(\mathsf{app}) \\ \rho \vdash e_1, S_0 \Downarrow g_1, S_1, T_1 \qquad \rho \vdash e_2, S_1 \Downarrow g_2, S_2, T_2 \\ T = \mathsf{Visit}(e); \mathsf{PrimApp}; T_1; T_2; \mathsf{Return}((kw_{app}\ \ g_1\ \ g_2)) \end{array}}{\rho \vdash e, S_0 \Downarrow (kw_{app}\ \ g_1\ \ g_n), S_2, T} \quad \textsc{App}$$

$$\frac{\begin{array}{c} e = (kw_{letstx}\ \ ((x\ \ t))\ \ e_b) \qquad \rho(\text{resolve}(kw_{letstx})) = \mathsf{Prim}(\texttt{let-syntax}) \\ \langle a, S_1\rangle = \text{fresh-address}(S) \qquad \text{ValidRule(t)} \\ \rho' = \rho\,[a \mapsto \mathsf{Macro}(t)] \qquad e'_b = \text{subst}(e_b, x, a) \qquad \rho' \vdash e'_b, S_1 \Downarrow g_b, S_2, T_b \\ T = \mathsf{Visit}(e); \mathsf{PrimLetSyntax}; \mathsf{Rename}(a); T_b; \mathsf{Return}(g_b) \end{array}}{\rho \vdash e, S \Downarrow g_b, S_2, T} \quad \textsc{Let-Syntax}$$

Figure 3.6: Semantics of macro expansion with expansion events

$$
\begin{array}{lrcl}
\text{Context} & E & ::= & [\,] \\
 & & | & (kw_{lambda}\ (x \cdots)\ E) \\
 & & | & (kw_{app}\ E\ {}^{\triangleright}e) \\
 & & | & (kw_{app}\ g\ E) \\
 & & | & (kw_{letstx}\ ((x\ t))\ E) \\
\text{MixTerm} & \hat{e} & ::= & {}^{\triangleright}e \\
 & & | & (kw_{lambda}\ (x \cdots)\ \hat{e}) \\
 & & | & (kw_{app}\ \hat{e}\ {}^{\triangleright}e) \\
 & & | & (kw_{app}\ g\ \hat{e}) \\
 & & | & (kw_{letstx}\ ((x\ t))\ \hat{e}) \\
 & & | & g \\
\text{Machine state} & MS & ::= & \langle \hat{e}, S \rangle
\end{array}
$$

Figure 3.7: Reduction semantics language

3.3 Reduction semantics

Our second formulation of macro expansion is a small-step rewriting semantics on pairs of terms and stores. The relation is on pairs of "mixed terms", written $\hat{e}$, and stores; it has the form

$$\langle \hat{e}, S \rangle \longrightarrow \langle \hat{e}', S' \rangle$$

The final states of the machine are those of the form

$$\langle g, S \rangle$$

There are also stuck states corresponding to failed expansions, such as a misuse of a primitive form or a macro transformer that fails to transcribe. We retain the same definition of stores as in the big-step semantics. Figure 3.7 introduces the grammar of expansion contexts, mixed terms ($\hat{e}$), and machine states.

Note the addition of the auxiliary ${}^{\triangleright}$ marker. These mark the frontier of expansion; a term immediately following the ${}^{\triangleright}$ marker is ready for expansion. Proper subterms of a ${}^{\triangleright}$-marked term are not ready for expansion; they may be in the context of a syntactic form that will transform them or apply renamings to them. Terms outside of the ${}^{\triangleright}$ markers have already been expanded, and they must not be expanded again; otherwise the bookkeeping maintained by the syntax algebra would be disrupted.

Figure 3.8 gives the definition of the lookup auxiliary function. Whereas in the big-step semantics the meaning of identifiers is carried by an environment, in the small-step semantics the meaning is extracted from the expansion context. Note that contexts are traversed inside-out; that is, context "frames" nearest the reference are searched first. For locally-bound names there is no difference, since every name bound in the context is unique, but searching in this order makes the base case easier to specify.

$$
\begin{array}{rcl}
\text{lookup}(s, [\,]) & = & \begin{cases} \mathsf{Prim}(s) & \text{when } s \in \text{PrimitiveName} \\ \mathsf{Var} & \text{otherwise} \end{cases} \\
\text{lookup}(s, E[(kw_{lambda}\ \ (x)\ \ [\,])]) & = & \begin{cases} \mathsf{Var} & \text{when } s = \text{resolve}(x) \\ \text{lookup}(s, E) & \text{otherwise} \end{cases} \\
\text{lookup}(s, E[(kw_{app}\ \ [\,]\ \ {}^{\triangleright}e)]) & = & \text{lookup}(s, E) \\
\text{lookup}(s, E[(kw_{app}\ \ g\ \ [\,])]) & = & \text{lookup}(s, E) \\
\text{lookup}(s, E[(kw_{letstx}\ \ ((x\ \ t))\ \ [\,])]) & = & \begin{cases} \mathsf{Macro}(t) & \text{when } s = \text{resolve}(x) \\ \text{lookup}(s, E) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 3.8: Reduction semantics auxiliaries

$$
\begin{array}{rcl}
\langle E[{}^{\triangleright}(x_{macro}\ \ e_1 \cdots e_n)], S\rangle & \longrightarrow & \langle E[{}^{\triangleright}e'], S'\rangle \\
& & \quad \textsf{where } \text{lookup}(\text{resolve}(x_{macro}), E) \\
& & \qquad\qquad = \mathsf{Macro}(t) \\
& & \qquad e = (x_{macro}\ \ e_1 \cdots e_n) \\
& & \qquad \langle mark, S'\rangle = \text{fresh-mark}(S) \\
& & \qquad t, \text{mark}(e, mark) \Downarrow_{tr} e'_{mark} \\
& & \qquad e' = \text{mark}(e'_{mark}, mark) \\
\langle E[{}^{\triangleright}(kw_{lambda}\ \ (x)\ \ e_b)], S\rangle & \longrightarrow & \langle E[(kw_{lambda}\ \ (x')\ \ {}^{\triangleright}e'_b)], S'\rangle \\
& & \quad \textsf{where } \langle a, S'\rangle = \text{fresh-address}(S) \\
& & \qquad x' = \text{subst}(x, x, a) \\
& & \qquad e'_b = \text{subst}(e_b, x, a) \\
\langle E[{}^{\triangleright}x], S\rangle & \longrightarrow & \langle E[x], S\rangle \\
& & \quad \textsf{where } \text{lookup}(\text{resolve}(x), E) = \mathsf{Var} \\
\langle E[{}^{\triangleright}(kw_{app}\ \ e_1\ \ e_2)], S\rangle & \longrightarrow & \langle E[(kw_{app}\ \ {}^{\triangleright}e_1\ \ {}^{\triangleright}e_2)], S\rangle \\
\langle E[{}^{\triangleright}(kw_{letstx}\ \ ((x\ \ t))\ \ e_b)], S\rangle & \longrightarrow & \langle E[(kw_{letstx}\ \ ((x'\ \ t))\ \ {}^{\triangleright}e'_b)], S'\rangle \\
& & \quad \textsf{where } \langle a, S'\rangle = \text{fresh-address}(S) \\
& & \qquad x' = \text{subst}(x, x, a) \\
& & \qquad e'_b = \text{subst}(e_b, x, a) \\
\langle E[(kw_{letstx}\ \ ((x\ \ t))\ \ g)], S\rangle & \longrightarrow & \langle E[g], S\rangle
\end{array}
$$

Figure 3.9: Reduction semantics

The small-step semantics performs essentially the same operations as the big-step semantics. Instead of applying the environment to pick the appropriate rule, the small-step semantics uses the lookup function and the context. Fresh marks and fresh addresses are obtained from the store in the same way, and the store is updated in the machine configuration. Marking, macro transcription, and unmarking all happen in a single step. As before, we rely on the unspecified $t, e \Downarrow_{tr} e'$ relation for macro transcription.

The flow of $^{\triangleright}$ markers in the small-step semantics generally reflects the structure of derivations in the big-step semantics. Each small-step rule applies to a $^{\triangleright}$-marked term in an expansion context. It removes that marker and produces some number of new markers. Each new marker corresponds to a subderivation position in the corresponding big-step rule, and it is placed before the term corresponding to that subderivation. The sole exception is the final rule, which merely cleans up unnecessary `let-syntax` forms once their bodies have been fully expanded.

An initial machine state consists of a mixed term $^{\triangleright}e$, where e contains no marks or addresses, and the empty store. Reduction stops on a machine state containing a fully-expanded term.

Theorem 2. *For any e and g, and S', the following propositions are equivalent:*

- $\rho_0 \vdash e, S_0 \Downarrow g, S'$
- $\langle {}^{\triangleright}e, S_0\rangle \longrightarrow^* \langle g, S'\rangle$

The big-step and small-step semantics are equivalent on whole programs.

Proof. The theorem follows from Lemma 3 because the initial environment ρ_0 and lookup on the empty context agree on all mappings. □

Lemma 3. *For any e, g, S, S', ρ, and E such that for all s, $\rho(s) = \mathrm{lookup}(s, E)$, the following propositions are equivalent:*

- $\rho \vdash e, S \Downarrow g, S'$
- $\langle E[{}^{\triangleright}e], S\rangle \longrightarrow^* \langle E[g], S'\rangle$

The big-step and small-step semantics are equivalent in equivalent contexts.

Proof. ($\Rightarrow$) By induction on the structure of the derivation and case analysis on the final inference rule:

MACRO The premises of the big-step rule satisfy the side conditions for the step

$$\langle E[{}^{\triangleright}e], S\rangle \longrightarrow \langle E[e'], S_1\rangle$$

Then by the IH on the subderivation, we get

$$\langle E[{}^{\triangleright}e'], S_1\rangle \longrightarrow^* \langle E[g], S'\rangle$$

(For future cases, we elide the left-hand side when it is the same as the previous right-hand side.)

LAMBDA The premises of the rule satisfy the side conditions for the step

$$\langle E[\,^{\triangleright}e], S\rangle \longrightarrow \langle E[(kw_{lambda}\ \ (x')\ \ ^{\triangleright}e'_b)], S_1\rangle$$

Then by the IH on the body expression's subderivation (checking that the extended context matches the extended environment) we get

$$\longrightarrow^* \langle E[(kw_{lambda}\ \ (x')\ \ g_b)], S'\rangle$$

and so $g = (kw_{lambda}\ \ (x')\ \ g_b)$.

APP The premises give us the step

$$\langle E[\,^{\triangleright}e], S\rangle \longrightarrow \langle E[(kw_{app}\ \ ^{\triangleright}e_1\ \ ^{\triangleright}e_2)], S\rangle$$

Then applying the IH once at context $E[(kw_{app}\ \ [\,]\ \ ^{\triangleright}e_2)]$ gives

$$\longrightarrow^* \langle E[(kw_{app}\ \ g_1\ \ ^{\triangleright}e_2)], S_1\rangle$$

Applying the IH once more at context $E[(kw_{app}\ \ g_1\ \ [\,])]$ gives

$$\longrightarrow^* \langle E[(kw_{app}\ \ g_1\ \ g_2)], S_2\rangle$$

and so $g = (kw_{app}\ \ g_1\ \ g_2)$.

VARIABLE (Base case.) Trivial.

LET-SYNTAX Similar to LAMBDA, except that after producing

$$\longrightarrow^* \langle E[(kw_{letstx}\ \ ((x'\ \ t))\ \ g_b)], S'\rangle$$

we tack on

$$\longrightarrow \langle E[g_b], S'\rangle$$

by the final reduction rule, to eliminate the `let-syntax` form.

($\Leftarrow$) By induction on the length of the reduction sequence and case analysis on the first step:

MACRO

$$\langle E[\,^{\triangleright}e], S\rangle \longrightarrow \langle E[\,^{\triangleright}e'], S_1\rangle \longrightarrow^* \langle E[g], S'\rangle$$

By the IH on the tail of the sequence, we get $\rho \vdash e', S_1 \Downarrow g, S'$ and by the side conditions and the derivation above we get $\rho \vdash e, S \Downarrow g, S'$.

LAMBDA

$$\begin{aligned}\langle E[\,^{\triangleright}(kw_{lambda}\ \ (x)\ \ e_b)], S\rangle &\longrightarrow \langle E[(kw_{lambda}\ \ (x')\ \ ^{\triangleright}e_b)], S_1\rangle \\ &\longrightarrow^* \langle E[g], S'\rangle\end{aligned}$$

By Lemma 4, the tail of the sequence contains the machine configuration $\langle E[(kw_{lambda}\ \ (x')\ \ g_b)], S_2\rangle$, and in fact we can tell it must end with it; $g = (kw_{lambda}\ \ (x')\ \ g_b)$ and $S_2 = S'$. The extended context corresponds to the extended environment ρ' in the big-step rule. By the IH we get $\rho' \vdash e'_b, S_1 \Downarrow g_b, S'$, and by that and the side conditions we can derive $\rho \vdash e, S \Downarrow g, S'$.

APP

$$\langle E[{}^{\triangleright}(kw_{app}\ e_1\ e_2)], S\rangle \longrightarrow \langle E[(kw_{app}\ {}^{\triangleright}e_1\ {}^{\triangleright}e_2)], S\rangle \longrightarrow^* \langle E[g], S'\rangle$$

By Lemma 4, we know the tail of the sequence contains the configuration $\langle E[(kw_{app}\ g_1\ {}^{\triangleright}e_2)], S_1\rangle$. By applying Lemma 4 again we know that after that occurs $\langle E[(kw_{app}\ g_1\ g_2)], S_2\rangle$, and in fact that must be the end of the whole sequence. From the first subsequence we get $\rho \vdash e_1, S \Downarrow g_1, S_1$, and from the second subsequence we get $\rho \vdash e_2, S_1 \Downarrow g_2, S'$. From those we can form the whole derivation.

VARIABLE (Base case.) Trivial.

LET-SYNTAX Similar to LAMBDA, except that there is an extra step at the end that discards the `let-syntax` form; this makes up for the difference between the term found by Lemma 4 and the expected term $g = g_b$.

□

Lemma 4. *For any E_1, E_2, e, g, S, and S', if*

$$\langle E_1[E_2[{}^{\triangleright}e]], S\rangle \longrightarrow^+ \langle E_1[g], S'\rangle$$

then there are g' and S'' such that

$$\langle E_1[E_2[{}^{\triangleright}e]], S\rangle \longrightarrow^+ \langle E_1[E_2[g']], S''\rangle \longrightarrow^* \langle E_1[g], S'\rangle$$

Expansion within a primitive context finishes within that context before moving on to a different context.

Proof. By induction on the length of the reduction sequence and case analysis of the first step:

MACRO

$$\langle E_1[E_2[{}^{\triangleright}e]], S\rangle \longrightarrow \langle E_1[E_2[{}^{\triangleright}e']], S_1\rangle \longrightarrow^* \langle E_1[g], S'\rangle$$

By IH on the tail of the sequence. Since no g is ${}^{\triangleright}$-tagged, the sequence must have at least one step.

LAMBDA

$$\begin{aligned}\langle E_1[E_2[{}^{\triangleright}(kw_{lambda}\ (x)\ e_b)]], S\rangle &\longrightarrow \langle E_1[E_2[(kw_{lambda}\ (x')\ {}^{\triangleright}e_b)]], S_1\rangle \\ &\longrightarrow^* \langle E_1[g], S'\rangle\end{aligned}$$

By IH on the tail of the sequence, we get that it passes through the state $\langle E_1[E_2[(kw_{lambda}\ (x')\ g_b)]], S_2\rangle$ along the way. The inner term is fully expanded. So $g = (kw_{lambda}\ (x')\ g_b)$.

VARIABLE (Base case.) Trivial.

APP

$$\begin{aligned}\langle E_1[E_2[{}^{\triangleright}(kw_{app}\ \ e_1\ \ e_2)]], S\rangle &\longrightarrow \langle E_1[E_2[(kw_{app}\ \ {}^{\triangleright}e_1\ \ {}^{\triangleright}e_2)]], S\rangle \\ &\longrightarrow^* \langle E_1[g], S'\rangle\end{aligned}$$

By IH on the tail of the sequence, we get that it passes through the state $\langle E_1[E_2[(kw_{app}\ \ g_1\ \ {}^{\triangleright}e_2)]], S_1\rangle$ along the way. By IH on the tail starting at that point, we get that it passes though $\langle E_1[E_2[(kw_{app}\ \ g_1\ \ g_2)]], S_2\rangle$. So $g = (kw_{app}\ \ g_1\ \ g_2)$.

□

CHAPTER 4

Implementation of the Macro Stepper

The structure of the macro stepper is dual[1] to the structure of a compiler: both have a front end that sits between the user and the intermediate representation (IR), a "middle end" or optimizer that performs advantageous transformations on the intermediate representation, and a back end that connects the intermediate representation to the program execution.

Whereas in a compiler information flows from the front end to the back end, in a debugger it starts at the back end and flows to the front end. The macro stepper's back end monitors the execution of the program; the front end displays the steps to the user. If enabled, the macro stepper's middle end is responsible for "optimizing" the IR for user comprehension.

In the macro stepper, the back end is connected to the macro expander, which is instrumented to emit events that describe three aspects of the process: its progress, the intermediate terms, and the choices of fresh names and marks. The back end parses this stream of events into a structure that represents the call-tree of the expander functions for the expansion of the program. This structure serves as the macro stepper's intermediate representation. The middle end traverses the tree, hiding extraneous details. Finally, the front end turns the tree representation into a sequence of rewriting steps.

The structure of the debugger motivates our two formulations of macro expansion from Chapter 3. The intermediate representations constructed by the back end correspond to derivations in our big-step semantics, whereas the rewriting sequences presented by the front end correspond to reduction sequences of the small-step semantics. We validate the design of the macro stepper by proving the faithfulness of these correspondences.

Alas, we have no technique for proving the correctness of the middle end. Indeed, as discussed in Section 2.3, for some macros the middle end *does not preserve the meaning* of the original program. Our proof of the front end's correctness assumes that the middle end is skipped.

In this chapter we discuss the implementation of the macro stepper for a subset of Scheme. We show the instrumentation of the macro expander and we describe the parser that reconstructs the expander call tree. We explain

[1] *dual*: the arrows go the other way 'round.

how the macro hiding algorithm—the middle end—transforms the IR tree by detecting the contexts of hidden macros, and we discuss how the front end turns the IR into the rewriting sequences shown in Chapter 2. Finally we state and prove the correspondences between our expansion models and the structures produced by the macro stepper.

4.1 The Back End

The macro stepper's back end is the bridge between the macro expansion process and the rest of the macro stepper. It consists of two parts: instrumentation embedded in the macro expander and a parser connected to the stepper itself. The two parts communicate via a side-channel. When the macro expander runs, the instrumentation sends events over the channel to the parser, which reconstructs the essence of the expansion process for interpretation by the macro stepper. The additions to the macro expander are unobtrusive and the impact on expansion small.

An alternative design would be to change the implementation of the macro expander to compute both the expanded program and a data structure representing the expansion. This would have involved changing the signatures of every internal procedure involved in macro expansion. This approach would have required pervasive changes to the expander compared to the current approach, and it would have incurred additional costs on all executions of the macro expander. Furthermore, the PLT Scheme expansion API allows macro transformers to re-enter the expander. Threading expansion records through interleaved expander and macro transformer code would require either changing the interface between them or creating a channel within the expander to track subexpansion records.

In this section we first present a simplified version of the macro expander and the instrumentation added to support the macro stepper. Then we show the grammar of events produced by the instrumented macro expander and show how to reconstruct a record of the expansion process.

4.1.1 Instrumenting the Macro Expander

The macro expander is implemented as a set of mutually recursive procedures. PLT Scheme uses the `syntax-case` algorithm [20] for macro expansion, and we describe our implementation in the context of that algorithm, but our implementation strategy applies to other macro expanders, too.

Figures 4.1–4.3 lists pseudocode for `expand-term`, the main expander procedure, and its auxiliary procedures for expanding macros and primitive syntactic forms. The primitive expander procedures recursively call `expand-term` on subterms as needed.

The `expand-term` procedure distinguishes macro applications, primitive syntax, and variable references with a combination of pattern matching on

```
expand-term(term, env) =
  emit-event(Visit, term)
  case term of
    (kw . _), when lookup(kw, env) = Macro(rules)
      => let term2 = expand-macro(term, env, rules)
         return expand-term(term2, env)
    (kw . _), when lookup(kw, env) = Primitive(expander)
      => let term2 = expander(term, env)
         emit-event(Return(term2))
         return term2
    id, when lookup(id, env) = Variable
      => emit-event(Variable)
         emit-event(Return(term))
         return term
    else
      => emit-event(Error)
         raise syntax error

expand-macro(term, env, rules) =
      => emit-event(EnterMacro)
         let M = fresh-mark()
         let termM = mark(term, M)
         emit-event(Mark(M))
         let term2M = transform(rules, termM)
         let term2 = mark(term2M, M)
         emit-event(ExitMacro(term2))
         return term2
```

Figure 4.1: Expansion procedures

```
expand-prim-lambda(term, env) =
  emit-event(PrimLambda)
  case term of
    (kw (formal) body), when formal is an identifier
      => let newvar = freshname()
         let env2 = extend-env(env, newvar, Variable)
         let formal2 = rename(formal, formal, newvar)
         let body2 = rename(body, formal, newvar)
         emit-event(Rename(newvar))
         let body3 = expand-term(body2, env2)
         return (kw (formal2) body3)
    else => emit-event(Error)
            raise syntax error

expand-primitive-application(term, env)
  emit-event(PrimApp)
  case term of
    (app op arg)
      => let op2 = expand-term(op, env)
         let arg2 = expand-term(arg, env)
         return (app op arg)
    else => emit-event(Error)
            raise syntax error
```

Figure 4.2: Expansion procedures (cont.)

the structure of the term and environment lookup of the leading keyword.

Macro uses are handled according to the transformation rules associated with the macro. First, the expander creates a fresh mark and stamps it on the given term. Second, it transforms the term using the macro's transformation rules. Finally, it applies the mark again, canceling out marks on subterms of the original term. Naturally, the expander recurs on the result to eliminate macros in the resulting term. The self-canceling marking is an efficient implementation of timestamps due to Dybvig et al. [20].

Primitive forms are handled by calling the primitive expander procedure associated with the keyword. The initial environment maps the name of each primitive (such as `lambda`) to its primitive expander (`expand-lambda`). When the primitive expander returns, expansion of that term is complete.

The shaded fragments in Figures 4.1–4.3 represent our additions to the

```
expand-primitive-let-syntax(term, env)
  emit-event(PrimLetSyntax)
  case term of
    (kw ([var rhs]) body)
      when name is an identifier
       and rhs is a valid transformer
      => let newvar = freshname()
         let env2 = extend-env(env, newvar, Macro(rhs))
         let body2 = rename(body, var, newvar)
         emit-event(Rename(newvar))
         return expand-term(body2, env2)
    else => emit-event(Error)
            raise syntax error
```

Figure 4.3: Expansion procedures (cont.)

expander to emit debugging events. These debugging events are received and interpreted by the macro stepper's back end.

The calls to `emit-event` send events through a private channel of communication to the macro stepper. The events carry data about the state of the macro expander. Figure 4.4 shows the event variants and the types of data they contain.

A Visit event indicates the beginning of an expansion step, and it contains the term being expanded. Likewise, the expansion of every term ends with a Return event that carries the final expanded term.

The EnterMacro and ExitMacro events surround macro transformations. The Mark event carries the fresh mark for that transformation step, and the ExitMacro event carries the term produced by the transformation. The macro-handling case of `expand-term` does not include a Return event because the expander has not completed expansion of that term.

For every primitive, such as `app`, there is an event (PrimApp) that indicates that the macro expander is in the process of expanding that kind of primitive form. Primitives that create and apply renamings to terms, such as `lambda`, send Rename events containing the fresh names to which the bound names are renamed.

4.1.2 Reconstruction

The back end of the macro stepper consumes the sequence of low-level events from the instrumented macro expander, parsing it into the intermediate representation's tree structures. The kinds of events in the stream determine the variants of the tree's nodes, and the nodes' slots contain the

$$
\begin{array}{llcl}
\text{Event} & \mathit{event} & = & \mathsf{Visit}(\text{Term}) \mid \mathsf{Return}(\text{Term}) \\
 & & \mid & \mathsf{EnterMacro} \mid \mathsf{Mark}(\text{Mark}) \mid \mathsf{ExitMacro}(\text{Term}) \\
 & & \mid & \mathsf{PrimLambda} \mid \mathsf{PrimApp} \mid \mathsf{PrimLetSyntax} \\
 & & \mid & \mathsf{Variable} \mid \mathsf{Rename}(\text{Address}) \\
\text{Traces} & T & = & \text{lists of Event}
\end{array}
$$

Figure 4.4: Expansion events

$$
\begin{array}{rcl}
\text{IR } \mathit{ir} & = & \mathsf{MacroNode}(\text{Term}, \text{Mark}, \text{Term}, \text{IR}, \text{Term}) \\
 & \mid & \mathsf{VariableNode}(\text{Term}, \text{Term}) \\
 & \mid & \mathsf{LambdaNode}(\text{Term}, \text{Address}, \text{IR}, \text{Term}) \\
 & \mid & \mathsf{AppNode}(\text{Term}, \text{IR}, \text{IR}, \text{Term}) \\
 & \mid & \mathsf{LetSyntaxNode}(\text{Term}, \text{Address}, \text{IR}, \text{Term})
\end{array}
$$

Figure 4.5: Intermediate representation structures

$$
\begin{array}{rcl}
\mathit{Expand} & \rightarrow & \mathsf{Visit}\ \mathsf{EnterMacro}\ \mathsf{Mark}\ \mathsf{ExitMacro}\ \mathit{Expand} \\
 & \mid & \mathsf{Visit}\ \mathsf{PrimLambda}\ \mathsf{Rename}\ \mathit{Expand}\ \mathsf{Return} \\
 & \mid & \mathsf{Visit}\ \mathsf{PrimApp}\ \mathit{Expand}\ \mathit{Expand}\ \mathsf{Return} \\
 & \mid & \mathsf{Visit}\ \mathsf{PrimLetSyntax}\ \mathsf{Rename}\ \mathit{Expand}\ \mathsf{Return} \\
 & \mid & \mathsf{Visit}\ \mathsf{Variable}\ \mathsf{Return}
\end{array}
$$

Figure 4.6: Grammar of event streams

$$
\begin{array}{rcl}
\mathit{Expand} & \rightarrow & e_1 : \mathsf{Visit}\ \mathsf{EnterMacro}\ m : \mathsf{Mark}\ e_2 : \mathsf{ExitMacro}\ \mathit{ir} : \mathit{Expand} \\
 & & \triangleright\ \mathsf{MacroNode}(e_1, m, e_2, \mathit{ir}, \text{finalterm}(\mathit{ir})) \\
 & \mid & e_1 : \mathsf{Visit}\ \mathsf{PrimLambda}\ f : \mathsf{Rename}\ \mathit{ir} : \mathit{Expand}\ e_2 : \mathsf{Return} \\
 & & \triangleright\ \mathsf{LambdaNode}(e_1, f, \mathit{ir}, e_2) \\
 & \mid & e_1 : \mathsf{Visit}\ \mathsf{Variable}\ e_2 : \mathsf{Return} \\
 & & \triangleright\ \mathsf{VariableNode}(e_1, e_2) \\
 & \mid & e_1 : \mathsf{Visit}\ \mathsf{PrimApp}\ \mathit{ir}_1 : \mathit{Expand}\ \mathit{ir}_2 : \mathit{Expand}\ e_2 : \mathsf{Return} \\
 & & \triangleright\ \mathsf{AppNode}(e_1, \mathit{ir}_1, \mathit{ir}_2, e_2) \\
 & \mid & e_1 : \mathsf{Visit}\ \mathsf{PrimLetSyntax}\ f : \mathsf{Rename}\ \mathit{ir} : \mathit{Expand}\ e_2 : \mathsf{Return} \\
 & & \triangleright\ \mathsf{LetSyntaxNode}(e_1, f, \mathit{ir}, e_2)
\end{array}
$$

Figure 4.7: Grammar with intermediate representations

data associated with the events.

Figure 4.5 lists the variants of the IR tree datatype. There is one variant per primitive syntactic form plus one variant for a macro rewriting step. Every variant in the tree datatype starts with a field containing the initial term and ends with a field containing the final term for that expansion. The functions initialterm and finalterm serve as accessors for those fields. The remaining fields of each variant are determined by the expander's behavior on that term variant. The MacroNode variant contains a field for the mark associated with that step, the result of the macro transformation, and the subtree corresponding to the expansion of the macro's result. The LambdaNode variant contains the fresh name for the `lambda`-bound variable and an IR field that represents the expansion of the renamed body expression. The VariableNode variant contains no additional information.

Given the instrumented macro expander, we can easily read off the grammar for event streams. The terminals of the grammar consist of exactly the expansion events described in Figure 4.4. There is only one nonterminal: *Expand*, which corresponds to the events produced by an invocation of `expand-term`. Each call to `emit-event` corresponds to a terminal on the right hand side, and each recursive call to `expand-term` corresponds to an occurrence of *Expand* on the right hand side. Figure 4.6 shows the grammar; the single nonterminal name is italicized.

The back end parses the event stream into the intermediate representation. Figure 4.7 shows the grammar augmented with action routines that produce the intermediate representation. Each terminal or nonterminal in a production alternative is optionally annotated with a variable that gets bound to the associated value, and the result of parsing that alternative is given by the action routine following the ▷ marker.

The environment is not explicit in the IR datatype, but it can be reconstructed from a node's context via a traversal of the IR tree from the root (the node corresponding to the expansion of the entire program) to the given node, adding to the environment on every LambdaNode or LetSyntaxNode node.

4.2 The Middle End: Macro Hiding

Once the back end has created an IR structure, the macro stepper processes it with the user-specified macro hiding policy to produce a new IR tree. Because the user can change the policy during a debugging session, the macro stepper retains the original IR tree, so that updating the display involves only reprocessing the tree and re-running the front end.

The tree produced by applying the macro policy is called a *synthetic tree*, as it contains nodes that have been synthesized by the macro stepper rather than representing actual steps of expansion. The datatype of synthetic trees contains an additional IR node variant, SyntheticNode, used to represent

$$\begin{array}{llcl} \text{IR} & \mathit{ir} & = & \mathsf{SyntheticNode}(\text{Term}, \text{Subs}, \text{Term}) \\ & & | & \ldots \\ \text{Sub} & \mathit{sub} & = & \mathsf{SubExpandNode}(\text{Path}, \text{IR}) \\ & & | & \mathsf{SubRenameNode}(\text{Paths}, \text{Identifier}, \text{Symbol}) \end{array}$$

Figure 4.8: Extension of IR datatype for macro hiding

$$\text{Path } \mathit{path} ::= [] \mid (\text{-}^* \mathit{path} \text{ -}^*)$$

$$\begin{array}{rcl} \text{substitute-path} & : & (\text{Term}, \text{Path}, \text{Term}) \to \text{Term} \\ \text{map-at-paths} & : & (\text{Term}, \text{Paths}, (\text{Term} \to \text{Term})) \to \text{Term} \end{array}$$

Figure 4.9: Path grammar and functions

opaque macros. The extension of the IR datatype is shown in Figure 4.8.

A SyntheticNode contains a list of subterm expansions and renamings. A subterm expansion consists of an IR tree and a path representing the context of the subterm in the node's original syntax. Renamings are accompanied by a list of paths, because they typically affect multiple non-contiguous subterms. For example, a renaming for a `lambda` expression includes the formal parameter and the body expression.

In principle, the primitive forms could be represented as synthetic nodes. A `lambda` expansion, for example, would be represented as a renaming at paths (- [] -) and (- - []), followed by a subexpansion at (- - []). An `app` expansion would simply contain two subexpansions. In practice, there is no advantage to this additional translation step.[2]

Figure 4.9 shows the grammar for paths and the signatures of a few functions on paths. A path represents the location of a subterm within a term. Unlike contexts, paths do not represent the contents of the term surrounding the location. This makes paths useful for identifying subterm positions when other parts of the term are (yet) unknown.

We write $\mathit{path}(e)$ to indicate the subterm of e at the location denoted by path. The function substitute-path replaces the indicated subterm of a given term with a new subterm. The function map-at-paths generalizes this operation, replacing the subterms at the given paths, which must not overlap, with the results of the given function applied to the existing subterms.

The macro hiding algorithm is a traversal of the IR tree in two alternating modes, hide and seek. In hide mode, the algorithm looks for macro-rewriting nodes to hide; when it finds a macro expression to hide, it switches

[2] In fact, the full macro stepper handles primitive forms and expansion features that would make this difficult; see Section 5.2.

$$
\begin{array}{rcl}
\text{hide} & : & (\text{IR}, (\text{Term} \rightarrow \text{boolean})) \rightarrow \text{IR} \\
\text{substitute-subs} & : & (\text{Term}, \text{Subs}) \rightarrow \text{Term} \\
\text{substitute-sub} & : & (\text{Term}, \text{Sub}) \rightarrow \text{Term} \\
\text{seek} & : & (\text{IR}, \text{Term}, (\text{Term} \rightarrow \text{boolean})) \rightarrow \text{Subs} \\
\text{seek-within} & : & (\text{IR}, \text{Term}, (\text{Term} \rightarrow \text{boolean})) \rightarrow \text{Subs}
\end{array}
$$

Figure 4.10: Signatures of macro hiding functions

to seek mode to build a synthetic node. In seek mode, the algorithm traverses the tree corresponding to a hidden macro's expansion, looking for subtrees to show. When it finds a subtree corresponding to a term that exists in the most recent hidden macro expression, it includes that subtree in the synthetic node and switches back to hide mode. Figure 4.10 shows the signatures of significant functions in the hiding algorithm.

Figure 4.11 shows the hide function, which consumes an IR tree and the hiding policy and produces an IR tree. The hiding policy is represented by the *show* predicate, a parameter to both hide and seek. If the input is a macro node and the policy says to hide the macro, the stepper switches to seek to continue the traversal of the macro node. The seek function returns a list of subtrees with paths, and the hide function replaces the original macro node with a synthetic node containing the subtrees. In all other cases, i.e., for primitive nodes and macro nodes that should not be hidden, hide just recurs on the node's subtrees.

In addition to removing macro expansion nodes from the tree, the hide function must also recompute the final term for every node that it produces to account for hidden macros. This calculation involves the node's processed subtrees and their contexts, which are fixed for primitive nodes and calculated by seek for synthetic nodes.

The auxiliary function substitute-subs (omitted), which computes the final term of a synthetic node, checks that the contexts of all of the sub-expansions it receives are disjoint. (Contexts of renamings may overlap with each other and with sub-expansions; in fact, this occurs often.) If sub-expansion contexts overlap (due to cloned subexpressions), the macro stepper overrides the policy and shows the current node instead. The auxiliary substitute-sub (also omitted) performs a single substitution on a Sub node, and substitute-path performs a single substitution given a path.

The seek algorithm (see Figure 4.12) looks for subtrees of terms that occurred in the original macro expression, represented as the e_{vis} parameter. The original macro expression is also called the "visible expression." The existence of such a subtree implies that the subterm occupies an *expansion context* of the hidden macro. When the stepper finds these subtrees, it

$$\begin{aligned}
&\text{hide}(\mathsf{MacroNode}(e_a, \mathit{mark}, e_m, \mathit{ir}_k, e_z),\ \mathit{show}), \quad \text{when } \mathit{show}(e_a)\\
&\quad = \mathsf{MacroNode}(e_a, \mathit{mark}, e_m, \mathit{ir}'_k, \text{finalterm}(\mathit{ir}'_k))\\
&\qquad \text{where } \mathit{ir}'_k = \text{hide}(\mathit{ir}_k, \mathit{show})\\
&\text{hide}(\mathsf{MacroNode}(e_a, \mathit{mark}, e_m, \mathit{ir}_k, e_z),\ \mathit{show}), \quad \text{when } \neg\mathit{show}(e_a)\\
&\quad = \mathsf{SyntheticNode}(e_a, e'_z, \mathit{subs})\\
&\qquad \text{where } \mathit{subs} = \text{seek}(\mathsf{MacroNode}(e_a, \mathit{mark}, e_m, \mathit{ir}_k, e_z), e_a, \mathit{show})\\
&\qquad\qquad e'_z = \text{substitute-subs}(e_z, \mathit{subs})\\
&\text{hide}(\mathsf{LambdaNode}(e_a, s, \mathit{ir}_b, e_z),\ \mathit{show})\\
&\quad = \mathsf{LambdaNode}(e_a, s, \mathit{ir}'_b, e'_z)\\
&\qquad \text{where } \mathit{ir}'_b = \text{hide}(\mathit{ir}_b, \mathit{show})\\
&\qquad\qquad e'_z = \text{substitute-path}(e_z, (\text{- - []}), \text{finalterm}(\mathit{ir}'_b))\\
&\text{hide}(\mathsf{AppNode}(e_a, \mathit{ir}_1, \mathit{ir}_2, e_z),\ \mathit{show})\\
&\quad = \mathsf{AppNode}(e_a, \mathit{ir}'_1, \mathit{ir}'_2, e'_z)\\
&\qquad \text{where } \mathit{ir}'_1 = \text{hide}(\mathit{ir}_1, \mathit{show})\\
&\qquad\qquad e_{z,1} = \text{substitute-path}(e_z, (\text{- [] -}), \text{finalterm}(\mathit{ir}'_1))\\
&\qquad\qquad \mathit{ir}'_2 = \text{hide}(\mathit{ir}_2, \mathit{show})\\
&\qquad\qquad e_z = \text{substitute-path}(e_{z,1}, (\text{- - []}), \text{finalterm}(\mathit{ir}'_2))\\
&\text{hide}(\mathsf{VariableNode}(e_a, e_z), \mathit{show})\\
&\quad = \mathsf{VariableNode}(e_a, e_z)\\
&\text{hide}(\mathsf{LetSyntaxNode}(e_a, s, \mathit{ir}_b, e_z),\ \mathit{show})\\
&\quad = \mathsf{LetSyntaxNode}(e_a, s, \mathit{ir}'_b, e'_z)\\
&\qquad \text{where } \mathit{ir}'_b = \text{hide}(\mathit{ir}_b, \mathit{show})\\
&\qquad\qquad e'_z = \text{substitute-path}(e_z, (\text{- - []}), \text{finalterm}(\mathit{ir}'_b))
\end{aligned}$$

Figure 4.11: Macro hiding: the hide function

switches back to hide for the subtree, then pairs the resulting synthetic tree with the path of the hidden macro expression in which it occurred.

When the seek function encounters a binding form, it must record the visible effects of the binding form's renaming. For `lambda`, for example, it searches for maximal subterms of the renamed formal parameter and body occurring in the original macro expression, and it records the paths to those subterms in a SubRenameNode. It continues the search by applying the renaming to the appropriate parts of the visible expression and recurring on the body IR.

To illustrate, suppose that we are processing the following expression, a use of the macro `or`, whose expansion we would like to hide:

```
(or X Y) ⟶* ((lambda (t) (if t t Y)) X)
```

When the algorithm initially enters seek mode, it is searching for subexpres-

$\text{seek}(ir, e_{vis}, show)$, when $\text{initialterm}(ir)$ occurs in e_{vis} at $path$
  $=$ $\mathsf{List}(\mathsf{SubExpandNode}(path, \text{hide}(ir, show)))$
$\text{seek}(ir, e_{vis}, show)$, when $\text{initialterm}(ir)$ does not occur in e_{vis}
  $=$ $\text{seek-within}(ir, e_{vis}, show)$

$\text{seek-within}(\mathsf{MacroNode}(e_a, mark, e_m, ir_k, e_z), e_{vis}, show)$
  $=$ $\text{seek}(ir_k, e_{vis}, show)$
$\text{seek-within}(\mathsf{LambdaNode}(e_a, s, ir_b, e_z), e_{vis}, show)$
  $=$ $\mathsf{Cons}(sub, \text{seek}(ir_b, e'_{vis}, show)$
    where $renamed = \{\text{lambda-formal}(e_a), \text{lambda-body}(e_a)\}$
      $paths = \{path \mid e$ occurs in e_{vis} at $path,$
        e maximal subterm of $renamed\}$
      $sub = \mathsf{SubRenameNode}(paths, \text{lambda-formal}(e_a), s)$
      $e'_{vis} = \text{substitute-sub}(e_{vis}, sub)$
$\text{seek-within}(\mathsf{AppNode}(e_a, ir_1, ir_2, e_z), e_{vis}, show)$
  $=$ $\text{append}(subs_1, subs_2)$
    where $subs_1 = \text{seek}(ir_1, e_{vis}, show)$
      $subs_2 = \text{seek}(ir_2, e_{vis}, show)$
$\text{seek-within}(\mathsf{VariableNode}(e_a, e_z), e_{vis}, show)$
  $=$ Empty
$\text{seek-within}(\mathsf{LetSyntaxNode}(e_a, s, ir_b, e_z), e_{vis}, show)$
  $=$ $\mathsf{Cons}(sub, \text{seek}(ir_b, e'_{vis}, show)$
    where $renamed = \{\text{let-syntax-name}(e_a), \text{let-syntax-body}(e_a)\}$
      $paths = \{path \mid e$ occurs in e_{vis} at $path,$
        e maximal subterm of $renamed\}$
      $sub = \mathsf{SubRenameNode}(paths, \text{let-syntax-name}(e_a), s)$
      $e'_{vis} = \text{substitute-sub}(e_{vis}, sub)$

Figure 4.12: Macro hiding: the seek function

$$
\begin{array}{llcl}
\text{Step} & \mathit{step} & = & \mathsf{Step}(\text{Side}, \text{Side}) \\
 & & | & \mathsf{AdminStep}(\text{Side}, \text{Side}) \\
\text{Side} & s & = & (\text{Term}, \text{State}) \\
\text{State} & \mathit{state} & = & (\text{Identifiers}, \text{Identifiers}) \\
\text{Ctx} & \mathit{ctx} & = & (\text{Term} \rightarrow \text{Term})
\end{array}
$$

Figure 4.13: Datatypes for reduction sequences

sions of the original `or` expression. But `lambda` is a binding form and applies a renaming to its body, including the subexpression `Y`. So in the body of the `lambda` form, the algorithm should not be looking for the original `Y`, but for `Y` wrapped with the `lambda`'s renaming.

The algorithm tracks the identity of the syntax objects rather than comparing them for textual equality. Specifically, in Figure 4.12 the phrase "e occcurs in e_{vis}" means that the actual syntax object representing e is contained within the syntax object representing e_{vis}. This relation could be formalized by introducing unique labels on every term and subterm, but for simplicity we treat it informally.

Consequently, macro hiding never confuses subterms of a hidden macro with terms arising later in the macro's expansion. It also enforces a strict interpretation of "subexpression"; for example, if a macro takes apart a term and reassembles it, the macro hiding algorithm does not consider them the same term. We consider this the proper behavior; by analyzing and rebuilding an expression a macro takes responsibility for its contents. Of course, untouched subexpressions of an analyzed term are considered the same.

4.3 The Front End: Reduction Sequences

Once macro hiding has produced an IR tree adjusted to the programmer's level of abstraction, the macro stepper's front end translates the tree into a sequence of rewriting and administrative steps for display by the macro stepper's graphical user interface. Rewriting steps correspond to macro transcriptions, and administrative steps correspond to renaming operations and uncovering primitive forms.[3]

Each step contains the program before expansion and the program after expansion. Each step also records the bindings and references known immediately before and after the step. The macro stepper uses this information to draw binding arrows and distinguish between tentative and definite references. For simplicity, we do not model the recording of the context and redex as separate data, although the macro stepper does this in practice so that it can highlight the redex and contractum. Figure 4.13 shows the representation of steps.

[3]Typically, however, the macro stepper displays only the rewriting steps.

$$
\begin{aligned}
&\text{reductions} : (\text{IR}, \text{Ctx}, \text{State}) \rightarrow (\text{Term}, \text{Steps}, \text{State}) \\
&\text{reductions}(\mathsf{MacroNode}(e_a, \mathit{mark}, e_m, \mathit{ir}_k, e_z), \mathit{ctx}, \mathit{state}) \\
&\quad = \quad (e'_z, \mathsf{Cons}(\mathit{step}_1, \mathit{steps}_2), \mathit{state}_2) \\
&\qquad \text{where } \mathit{state}_1 = \text{learn-reference}(\mathit{state}, \text{keyword-of}(e_a)) \\
&\qquad\qquad \mathit{step}_1 = \mathsf{Step}((\mathit{ctx}(e_a), \mathit{state}), (\mathit{ctx}(e_m), \mathit{state}_1)) \\
&\qquad\qquad (e'_z, \mathit{steps}_2, \mathit{state}_2) = \text{reductions}(\mathit{ir}_k, \mathit{ctx}, \mathit{state}_1) \\
&\text{reductions}(\mathsf{LambdaNode}(e_a, s, \mathit{ir}_b, e_z), \mathit{ctx}, \mathit{state}) \\
&\quad = \quad (e'_z, \mathsf{Cons}(\mathit{step}_r, \mathit{steps}), \mathit{state}') \\
&\qquad \text{where } \mathit{state}_u = \text{learn-reference}(\mathit{state}, \text{keyword-of}(e_a)) \\
&\qquad\qquad r(e) = \text{subst}(e, \text{lambda-formal}(e_a), s) \\
&\qquad\qquad e_r = \text{substitute-path}(e_a, (\text{- [] -}), r(\text{lambda-formal}(e_a))) \\
&\qquad\qquad e'_r = \text{substitute-path}(e_r, (\text{- - []}), r(\text{lambda-body}(e_r))) \\
&\qquad\qquad \mathit{state}_r = \text{learn-binder}(\mathit{state}_u, \text{lambda-formal}(e'_r)) \\
&\qquad\qquad \mathit{step}_r = \mathsf{AdminStep}((\mathit{ctx}(e_a), \mathit{state}_u), (\mathit{ctx}(e'_r), \mathit{state}_r)) \\
&\qquad\qquad \mathit{ctx}' = \text{extend-context}(\mathit{ctx}, (\text{- - []}), e_r) \\
&\qquad\qquad (e'_z, \mathit{steps}, \mathit{state}') = \text{reductions}(\mathit{ir}_b, \mathit{ctx}', \mathit{state}_r) \\
&\text{reductions}(\mathsf{AppNode}(e_a, \mathit{ir}_1, \mathit{ir}_2, e_z), \mathit{ctx}, \mathit{state}) \\
&\quad = \quad (e'_z, \mathsf{Cons}(\mathit{step}_a, \text{append}(\mathit{steps}_1, \mathit{steps}_2)), \mathit{state}_2) \\
&\qquad \text{where } \mathit{state}' = \text{learn-reference}(\mathit{state}, \text{keyword-of}(e_a)) \\
&\qquad\qquad \mathit{step}_a = \mathsf{AdminStep}((\mathit{ctx}(e_a), \mathit{state}), (\mathit{ctx}(e_a), \mathit{state}_1)) \\
&\qquad\qquad \mathit{ctx}_1 = \text{extend-context}(\mathit{ctx}, (\text{- [] -}), e_a) \\
&\qquad\qquad (e_{1,z}, \mathit{steps}_1, \mathit{state}_1) = \text{reductions}(\mathit{ir}_1, \mathit{ctx}_1, \mathit{state}') \\
&\qquad\qquad e'_a = \text{substitute-path}(e_a, (\text{- [] -}), e_{1,z}) \\
&\qquad\qquad \mathit{ctx}_2 = \text{extend-context}(\mathit{ctx}, (\text{- - []}), e'_a) \\
&\qquad\qquad (e_{2,z}, \mathit{steps}_2, \mathit{state}_2) = \text{reductions}(\mathit{ir}_2, \mathit{ctx}_2), \mathit{state}_1) \\
&\qquad\qquad e'_z = \text{substitute-path}(e'_a, (\text{- - []}), e_{2,z}) \\
&\text{reductions}(\mathsf{VariableNode}(e_a, e_z), \mathit{ctx}, \mathit{state}) \\
&\quad = \quad (e_z, \text{list}(\mathit{step}_a), \mathit{state}') \\
&\qquad \text{where } \mathit{state}' = \text{learn-reference}(e_z, \mathit{state}) \\
&\qquad\qquad \mathit{step}_a = \mathsf{AdminStep}((\mathit{ctx}(e_a), \mathit{state}), (\mathit{ctx}(e_z), \mathit{state}'))
\end{aligned}
$$

Figure 4.14: Reduction

$$
\begin{aligned}
&\text{reductions}(\mathsf{SyntheticNode}(e_a, subs, e_z), ctx, state)\\
&\quad = \text{reductions-subs}(e_a, subs, ctx, \text{learn-reference}(state, \text{keyword-of}(e_a)))\\
&\text{reductions-subs}(e_a, \mathsf{Empty}, ctx, state)\\
&\quad = (e_a, \mathsf{Empty}, state)\\
&\text{reductions-subs}(e_a, \mathsf{Cons}(sub_1, subs_2), ctx, state)\\
&\quad = (e_2, \text{append}(steps_1, steps_2), state_2)\\
&\qquad \text{where } (e_1, steps_1, state_1) = \text{reductions-sub}(e_a, sub_1, ctx, state)\\
&\qquad\qquad (e_2, steps_2, state_2) = \text{reductions-subs}(e_1, subs_2, ctx, state_1)\\
&\text{reductions-sub}(e_a, \mathsf{SubExpandNode}(path, ir), ctx, state)\\
&\quad = \text{reductions}(ir, \text{extend-context}(ctx, e_a, path), state)\\
&\text{reductions-sub}(e_a, \mathsf{SubRenameNode}(paths, x, s), ctx, state)\\
&\quad = (e_r, \mathsf{List}(\mathsf{AdminStep}((e_a, state), (e_r, state_r))), state_r)\\
&\qquad \text{where } r(e) = \text{subst}(e, x, s)\\
&\qquad\qquad e_r = e_a \text{ with } r \text{ applied at each } path \in paths\\
&\qquad\qquad state_r = \text{learn-binder}(state, x)
\end{aligned}
$$

Figure 4.15: Auxiliaries for reductions of synthetic nodes

Figure 4.14 shows the reductions function, which represents the front end's traversal of the IR tree to produce reduction steps.[4] The macro stepper calls this function with the processed IR tree, the empty context, and the empty state. The function produces the final, expanded term, a list of reduction steps, and the final state containing all known binding occurrences and references. Figure 4.15 shows auxiliary functions for processing synthetic nodes created by macro hiding.

When reductions encounters a macro node, it creates a rewriting step and adds it to the sequence generated by the macro node's continuation subtree. The step contains the program term before and after applying the macro. The front end constructs the second program term by inserting the macro's result term into the inherited program context.

When the front end encounters a primitive node in the IR tree, it generally recurs on the subtrees of the primitive node, extending the expansion context and passing along the binding information, and the subsequences are concatenated. The context passed to each subtree incorporates previous updates to the term; for example, the context used for processing a `lambda` expression's body includes the renamed formal parameter.

The rewriting sequence also stores information about the known bindings and references. Whenever the front end encounters a binding node—LambdaNode, LetSyntaxNode, or SyntheticNode containing renaming steps—it

[4]The LetSyntaxNode case is omitted; it is similar to the LambdaNode case.

adds the bound occurrences to the accumulator for known bindings. Likewise, for every kind of node it adds the variable or special form's keyword to the list of known references.

A step is displayed by showing the term of the first side and the term of the second side, separated by an arrow. The corresponding states are used to illustrate the macro expander's knowledge of the program's binding structure at the time of the step. If an identifier appears in the known binding list, the macro stepper adds an arrow to that identifier from any identifier that might refer to it. (This relationship can be calculated from the term alone.) For those potential references appearing in the known references list, the arrow is "definite"; for the rest, the arrow is "tentative."

4.4 Correctness

Following Clements et al. [12], a correct stepper must display the elements of a program's reduction according to a small-step semantics. In order to establish this kind of correctness for our stepper, we prove two claims. First, the back end constructs an intermediate representation faithful to the big-step semantics. Second, the front end constructs a rewriting sequence faithful to the small-step semantics. This section sketches those proofs.

4.4.1 The back end and the big-step semantics

The back end comprises the instrumented expander and the event stream parser. Rather than work directly with the instrumented expander functions, we take the big-step semantics of Chapter 3 (page 27) as a model of the recursive expansion functions. We modify this model of macro expansion to generate the same event streams as our instrumentation. The shaded parts of Figure 3.6 represent the event sequence emitted by the macro expander. The back-end correctness theorem establishes a correspondence between the IR trees produced by the back end's parser and the extended big-step derivations.[5]

To show the faithfulness of the back end we need to introduce a representation function from big-step derivations to IR trees. Then we show that the resulting IR trees are a suitable representation of the derivations and prove that the back end implements the representation function.

Given a derivation Δ, we write $[\![\Delta]\!]$ for its representation as an IR tree. The mapping is the obvious interpretation of the IR trees, discussed informally in Section 4.1.2. A precise definition is given in Figure 4.16.

Every inference rule has a corresponding datatype variant, and the fields of the variants suffice to determine the entire contents of the derivation if

[5]It is straightforward to verify that the structure of the recursive expansion functions matches the structure of the big-step semantics, including the event streams. Thus we consider the big-step semantics an adequate model of instrumented expansion.

Let $\Delta : \rho \vdash e, S \Downarrow g, S'$. The IR approximation $[\![\Delta]\!]$ is defined by case analysis on the final inference rule of Δ. Metavariables refer to the corresponding parts of the derivation, as given by the rules in Figure 3.6.

MACRO $[\![\Delta]\!] = \mathsf{MacroNode}(e, mark, [\![\Delta_k]\!], g)$, where Δ_k is the single subderivation of Δ.

LAMBDA $[\![\Delta]\!] = \mathsf{LambdaNode}(e, a, [\![\Delta_b]\!], g)$, where Δ_b is the single subderivation of Δ.

VARIABLE $[\![\Delta]\!] = \mathsf{VariableNode}(e, g)$

APP $[\![\Delta]\!] = \mathsf{AppNode}(e, [\![\Delta_1]\!], [\![\Delta_2]\!], g)$, where Δ_1 is the first subderivation and Δ_2 is the second subderivation of Δ.

LET-SYNTAX $[\![\Delta]\!] = \mathsf{LetSyntaxNode}(e, a, [\![\Delta_b]\!], g)$, where Δ_b is the single subderivation of Δ.

Figure 4.16: IR representation function

the initial store and environment are known.

Informally, we want to prove that when the macro expander runs, starting with the initial environment and store, and the stepper constructs an IR tree, the IR tree corresponds to the derivation in the big-step semantics.

The mapping from derivations to IR trees is not injective; IR trees do not represent the environment and the store. If, however,

$$\Delta : \rho \vdash e, S \Downarrow g, S'$$

then $[\![\Delta]\!]$, ρ, and S together completely determine Δ. In particular, it is possible to recover the derivation for whole-program expansions, because the initial environment and store are known. Consequently, IR trees are a faithful approximation of big-step derivations.

Let $\mathcal{G}$ be the grammar of Figure 4.7, and define $\mathrm{parse}(T) = ir$ when grammar $\mathcal{G}$ produces the value ir for the input stream T (using *Expand* as the start symbol).

Theorem 5 (Back end correctness)**.** *If* $\Delta : \rho_0 \vdash e, S \Downarrow g', S', T$ *then* $[\![\Delta]\!] = \mathrm{parse}(T)$.

Proof. By Lemma 6 we know that $\mathcal{G}$ is unambiguous and thus the parse function is well-defined on all traces in the language of $\mathcal{G}$. By Lemma 7 we know that T is in the language of $\mathcal{G}$. Consequently, $\mathrm{parse}(T)$ is well-defined.

We proceed by induction on the structure of Δ and case analysis of the last inference rule used:

MACRO The definition of T in the rule matches exactly with the MacroNode production in $\mathcal{G}$, and since the $\mathcal{G}$ is unambiguous, that must be the

$$
\begin{aligned}
[\![\mathsf{Step}((e, state), (e', state'))]\!] &= (e, e') \\
[\![\mathsf{AdminStep}((e, state), (e', state'))]\!] &= (e, e') \\
[\![\langle e, S\rangle \longrightarrow \langle e', S'\rangle]\!] &= (\text{no-markers}(e), \text{no-markers}(e'))
\end{aligned}
$$

$$
\frac{\forall i \leq n : [\![step_i]\!] = [\![MS_i \longrightarrow MS_{i+1}]\!]}{\mathsf{List}(step_1, \cdots, step_n) \sim MS_1 \longrightarrow \cdots \longrightarrow MS_{n+1}}
$$

Figure 4.17: Small-step approximation

production used to parse T. The values of the fields as dictated by $\mathcal{G}$ correspond exactly to the values as dictated by the definition of $[\![\cdot]\!]$. We use the IH once, on the subderivation for the macro's result.

the other rules The proofs for the other cases proceed in an analogous fashion.

□

Lemma 6. parse *is well-defined on all event sequences in the language of* $\mathcal{G}$.

Proof. The grammar $\mathcal{G}$ is LR(1)—this property is machine-checked by the parser generator—and thus unambiguous. □

Lemma 7. *If* $\rho \vdash e, S \Downarrow g, S', T$*, then* T *is in the language of* $\mathcal{G}$.

Proof. By induction on the derivation of the judgment and case analysis of the last inference rule used. Each inference rule defines its trace in a manner that corresponds exactly to an alternative of *Expand* in $\mathcal{G}$. □

4.4.2 The front end and the small-step semantics

The front end is faithful to the small-step semantics in the sense that for any complete reduction sequence produced by the small-step semantics, the macro stepper's front end produces the same sequence when given the appropriate IR tree for that expansion. Figure 4.17 formalizes the comparison of the sequences.

Theorem 8 (Front end correctness)**.** *If* $\langle {}^{\triangleright}e, S_0\rangle \longrightarrow^* \langle g, S'\rangle$*, then there exists* $\Delta : \rho_0 \vdash e, S_0 \Downarrow g, S'$ *such that*

$$
\begin{aligned}
&steps \;\sim\; \langle {}^{\triangleright}e, S_0\rangle \longrightarrow^* \langle g, S'\rangle \\
&\quad \textit{where } (g, steps, state') = \text{reductions}([\![\Delta]\!], ctx_0, state_0)
\end{aligned}
$$

Proof. The existence of $\Delta : \rho_0 \vdash e, S_0 \Downarrow g, S'$ is guaranteed by Theorem 2, and since both semantics are deterministic, there is exactly one such Δ. Lemma 9 does the rest of the work. □

Lemma 9. *If*

- $\Delta : \rho \vdash e, S \Downarrow g, S'$
- *for all symbols* s, $\rho(s) = \text{lookup}(s, E)$
- *for all expressions* e, $ctx(e) = E[e]$

then

$$steps \;\sim\; \langle E[{}^{\triangleright}e], S\rangle \longrightarrow^* \langle E[g], S'\rangle$$
$$\textit{where } (g, steps, state') = \text{reductions}([\![\Delta]\!], ctx, state_0)$$

Proof. This inductive proof follows the structure of the proof that given Δ there exists a reduction sequence of the same expansion (Theorem 2). The rest is case analysis of the last rule used.

MACRO $[\![\Delta]\!] = \mathsf{MacroNode}(e, mark, e_m, ir_k, g)$. From the front end we have the first step from $ctx(e_a)$ to $ctx(e_m)$, which agrees with the first step taken by the small-step sequence. The remainder of the steps come from ir_k, which corresponds to the subderivation of Δ, so by the IH the rest of the steps agree also.

LAMBDA $[\![\Delta]\!] = \mathsf{LambdaNode}(e, a, ir_b, g)$. The first step from the front end is an administrative (renaming) step where the formal parameter and body are renamed; that agrees with the first step taken by the small-step semantics. The rest of the steps are generated by ir_b; the two contexts are extended in the same way for the formation of the rest of the sequences, so by the IH they agree.

VARIABLE $[\![\Delta]\!] = \mathsf{VariableNode}(e, g)$. The front end and small-step semantics both take a single step, and they agree.

APP $[\![\Delta]\!] = \mathsf{AppNode}(e, ir_1, ir_2, g)$. The front end's first step is an administrative step that does not change the term; it agrees with the first step of the small-step semantics, which only moves ${}^{\triangleright}$ markers around. We apply the IH to ir_1 and then to ir_2, with the appropriate extended contexts.

□

CHAPTER 5
The Macro Stepper in Practice

We have implemented a macro stepper for PLT Scheme following the design presented in Chapters 2–4. The actual macro stepper handles the full PLT Scheme language, including modules, procedural macro transformers and their accompanying phase separation [30], source location tracking [20], and user-defined syntax properties. The macro stepper also handles additional powerful operations available to macro transformers such as performing local expansion and lifting expressions to top-level definitions.

This chapter discusses practical issues beyond the idealized design. We present a technique for extending the macro stepper to handle syntax errors and a discussion of how macro hiding must be adapted to scale up to the full language. Finally, we report on the experiences we and others have had using the macro stepper.

5.1 Handling syntax errors

Expansion can fail if either a primitive form or a macro transformer signals a syntax error. The macro stepper must also handle failed expansions. This requires additional instrumentation of the macro expander and extensions to both the intermediate representation and parser.

Syntax errors can only occur during the execution of a macro transformer or when a primitive expander validates the use of a primitive form. In both cases, the macro expander emits an Error event containing the exception describing the syntax error.

The most straightforward representation would include an additional IR node variant for each position a failure could occur in. For example, the `lambda` primitive would generate three variants, corresponding to the following three cases:

- the `lambda` expression is ill-formed, such as if the formal parameters position does not contain exactly one identifier, or if the body expression is missing
- the `lambda` expression is well-formed, but expansion fails in the body expression

FailExpand ::= Visit EnterMacro Mark Error
| Visit EnterMacro Mark ExitMacro *FailExpand*
| Visit PrimLambda Error
| Visit PrimLambda Rename *FailExpand* Return
| ...

Figure 5.1: Error-handling grammar

- the expansion succeeds

The `app` primitive would have four variants: one for success, one for early failure, and two for failure within each of the two subexpressions. This representation, however, obscures the similarity of the related variants. The grammar would include separate rules for building each variant, and the processing of IR trees would have to either duplicate code or create some mechanism for sharing the processing of related prefixes of structures.

Instead, we extend the IR with a notion of an *incomplete* node. We keep the same variants (one per primitive form plus one for macros) and add fields at every position where an error can occur. For example, the `lambda` keyword checks to make sure it has a list of identifiers for the formal parameters before starting to expand the body expression, so LambdaNode gets one additional field:

$$\mathsf{LambdaNode}(\mathrm{Term}, \mathrm{Option}(\mathrm{Exn}), \mathrm{Address}, \mathrm{IR}, \mathrm{Term})$$

If an optional syntax error field holds an exception value, then all the fields that follow it hold a distinguished Absent value and the node is considered incomplete. Furthermore, if an IR-valued field holds an incomplete node, then the fields that follow it hold the absent value.

For example, the three possible cases of `lambda` expansion enumerated above have representations of the following forms, respectively:

$$\mathsf{LambdaNode}(e, exn, \mathsf{Absent}, \mathsf{Absent}, \mathsf{Absent})$$
$$\mathsf{LambdaNode}(e, \mathsf{None}, a, ir, \mathsf{Absent})$$
$$\mathsf{LambdaNode}(e, \mathsf{None}, a, ir, e')$$

We extend the grammar with a new nonterminal, *FailExpand*, that represents the event streams of failed expansions. A new start symbol is added to the grammar that goes to either *Expand* or *FailExpand*. As with *Expand*, the alternatives of *FailExpand* can be read off the instrumented expander code, picking only the cases where an error is raised or a recursive call fails. Figure 5.1 shows some of the productions of *FailExpand*.

The error-handling grammar is roughly twice the size of the original grammar. Furthermore, the new alternatives share a great deal of structure with the original alternatives, and we have designed our intermediate

Grammar	$grammar$	$=$	$production^*$
Production	$production$	$=$	$NT \rightarrow I^* \triangleright A$
Nonterminal	NT	$=$	$N \mid \mathsf{Fail}(N)$
Item	I	$=$	$x : NT \mid x : ?(name) \mid x : T$
Core item	C	$=$	$x : NT \mid x : T$
Nonterminal name	N		
Terminal name	T		
Variable	x, y, z		
Action	A		

Figure 5.2: Syntax of annotated grammars

representation to exploit that shared structure. Rather than write down every alternative, we generate the full grammar from annotations on the "optimistic" grammar.

5.1.1 Generation of Error Alternatives

Before we can explain how the annotated grammar is elaborated into the ordinary sort of grammar accepted by our parser generator, we must define annotated grammars. The syntax of annotated grammars is shown in Figure 5.2. Note that instead of allowing alternation within productions, we write each variant of a nonterminal as a separate production. Contrast Figures 4.7 and 5.3. A nonterminal name is either a simple name, such as *Expand*, or a simple name tagged with either *?* or *Fail*. A nonterminal name tagged with *?* can only occur on the right-hand side of a production, not in definition position, and it can only occur in the right-hand-side of a production for a simply-named nonterminal.

A right-hand side item is either a nonterminal or terminal, prefixed with a variable bound in the action routine. In our examples, we abuse the notation slightly by omitting the variable if the value carried by the nonterminal or terminal is not used in the action routine.

An annotated grammar that contains no *?*-tagged nonterminal names can be transformed into an ordinary grammar by simply mapping every tagged name to a fresh simple name.

The elaboration of the annotated grammar involves splitting every production alternative containing a *?* ("maybe") annotation into its successful and failed parts. The successful traces are then ones in which potential errors do not occur. The failed traces are the traces in which exactly one error occurs, and it ends the trace.

Here is the splitting relation $\longrightarrow_{\text{split}}$:

$$\begin{array}{rcl} N_0 & \rightarrow & x_1 : C_1 \ \cdots \ x_m : C_m \ \ y : ?(N) \ \ z_1 : I_1 \ \cdots \ z_n : I_n \triangleright A \\ & & \longrightarrow_{\text{split}} \\ N_0 & \rightarrow & x_1 : C_1 \ \cdots \ x_m : C_m \ \ y : \mathsf{N} \ \ z_1 : I_1 \ \cdots \ z_n : I_n \triangleright A \\ \mathit{Fail}(N_0) & \rightarrow & x_1 : C_1 \ \cdots \ x_m : C_m \ y : \mathit{Fail}(\mathsf{N}) \ z_1 : \mathit{Skip}_1 \ \cdots \ z_n : \mathit{Skip}_n \triangleright A \end{array}$$

A production is split at its first *?*-tagged item. The *?*-tagged item is either successful or failed. The successful alternative remains in the production for the successful (untagged) nonterminal, and the failed alternative becomes a production of the *Fail*-tagged version of the nonterminal being defined.

The splitting relation relies on a special nonterminal, *Skip*, that represents items skipped after a failure. Here is the definition of *Skip*:

$$\mathit{Skip} \ \rightarrow \ \epsilon \triangleright \mathit{Absent}$$

The grammar of event traces uses an additional special nonterminal, *?*(*Check*), to represent places where syntax errors might occur. *Check* means no error occurred and returns None; *Fail*(*Check*) means an error occurred and returns the error. Both *Check* and *Fail*(*Check*) are defined explicitly:

$$\begin{array}{rcl} \mathit{Check} & \rightarrow & \epsilon \triangleright \mathit{None} \\ \mathit{Fail}(\mathit{Check}) & \rightarrow & \mathit{exn} : \mathsf{Error} \triangleright \mathit{exn} \end{array}$$

Finally, the start symbol is defined to be either a successful event trace or a failed event trace:

$$\begin{array}{rcl} \mathit{Start} & \rightarrow & \mathit{ir} : \mathit{Expand} \triangleright \mathit{ir} \\ \mathit{Start} & \rightarrow & \mathit{ir} : \mathit{Fail}(\mathit{Expand}) \triangleright \mathit{ir} \end{array}$$

Note that the start symbol could not be defined by the single right-hand side *?*(*Expand*), because the splitting rule would move the failed trace alternative to a different nonterminal, *Fail*(*Expand*).

Figure 5.3 shows the annotated grammar. It differs from the grammar in Figure 4.7 only in the addition of checks for syntax errors, with the corresponding additions to the action routines, and tagging of recursive references to *Expand*.

The actual parser covers 26 primitive syntactic forms—many of them with far more complicated syntax than the primitives used for demonstration here—and multiple modes of expansion (full recursive expansion versus partial head expansion to uncover core syntactic forms). It also supports the API available to macro writers, which allows macros to expand subterms and manipulate syntactic environments. The full grammar, including nonterminal definitions with action routines, is 500 lines long. The implementation of error-case splitting and a few other extensions takes another 300 lines. The rewritten grammar is fed to PLT Scheme's macro-based LALR(1) parser generator library [54].

Expand

→ e_1 : Visit EnterMacro c : *?*(*Check*) $mark$: Mark e_2 : ExitMacro ir : *?*(*Expand*)
▷ MacroNode$(e_1, c, mark, e_2, ir, \text{finalterm}(ir))$

| e_1 : Visit PrimLambda c : *?*(*Check*) a : Rename ir : *?*(*Expand*) e_2 : Return
▷ LambdaNode(e_1, c, a, ir, e_2)

| e_1 : Visit Variable c : *?*(*Check*) e_2 : Return
▷ VariableNode(e_1, c, e_2)

| e_1 : Visit PrimApp c : *?*(*Check*) ir_1 : *?*(*Expand*) ir_2 : *?*(*Expand*) e_2 : Return
▷ AppNode$(e_1, c, ir_1, ir_2, e_2)$

| e_1 : Visit PrimLetSyntax c : *?*(*Check*) a : Rename ir : *?*(*Expand*) e_2 : Return
▷ LetSyntaxNode(e_1, c, a, ir, e_2)

Figure 5.3: Annotated grammar

The middle end and front end are updated to handle incomplete nodes. Macro hiding checks for errors and ceases processing when one is encountered. The front end records a special kind of "error step" for syntax errors and then immediately stops processing the rest of the IR tree.

5.2 Handling partial expansion

The macro hiding algorithm presented in Chapter 4 works beautifully for the tiny language used in the presentation. It works well for most of Scheme; internal definitions are a challenge, but they can be preprocessed into a form suitable for the macro hiding algorithm. The problems with internal definitions are simple compared to the challenges posed by the the full language. PLT Scheme partially expands internal definitions so that they can be transformed into a `letrec` expression. Modules similarly perform two passes of expansion over their contents: one to uncover definitions, including macro definitions, and one expand expressions.

Macro writers also have access to partial expansion via PLT Scheme's rich macro API. Partial expansion is used by special forms to enable macro-extensible subforms. For example, the `class` macro allows macros to expand into method definitions, initialization argument and field declarations, etc. This is accomplished using partial expansion.

The clean separation between macro hiding and rewriting sequence generation breaks down in the presence of partial expansion. The implementation of macro hiding presented in Chapter 4 exploits the fact that different branches of an IR tree do not revisit the same terms or the same contexts. This is visible in the lack of threading updates to the visible term in seek and seek-within. The only dependence is caused by the renamings of binding

forms, and that is purely local. Contrast that with the heavy threading of updates in the reductions function.

It is possible to change the implementation of macro hiding to account for the additional dependencies introduced by partial expansion. But that results in a complicated traversal that is remarkably similar to the one already performed by the reductions function. Maintaining two traversals proved to be an enormous burden, and it resulted in a brittle, buggy implementation of macro hiding.

The current implementation of the macro stepper fuses macro hiding and reduction sequence generation. One traversal of the IR is performed that tracks the hiding mode, propagates updates to the "visible" term, and produces rewriting steps directly without generating synthetic IR trees.

A single construct, "two-pass expansion," suffices to model most forms of partial expansion. A two-pass expansion frame is dropped around any form that partially expands a subterm, inspects it, and optionally transforms it. The partial expansion goes in the first pass, and the inspection and subsequent transformation goes in the second pass. For example, in a `lambda` body, the partial expansion to uncover internal definitions constitutes the first pass, and the transformation into a `letrec` form and subsequent expansion of that form constitute the second pass.

Consider the following example of internal definition expansion:

```
(lambda (f x)
  (defthunk g (or x 12))
  (f g))
⟶
(lambda (f x)
  (define g (lambda () (or x 12)))
  (f g))
⟶  ;; Start of pass 2
(lambda (f x)
  (letrec ([g (lambda () (or x 12))])
    (f g)))
⟶
(lambda (f x)
  (letrec ([g (lambda () (let ([t x]) (if t t 12)))])
    (f g)))
```

Notice that the `defthunk` form is partially expanded into a `define` form, but the `or` expression in the right-hand side is not expanded until after the definition is turned into a `letrec` expression.

To handle macro hiding, the implementation proceeds as follows. If any macros are hidden in the first pass, the second pass immediately starts in seek mode with a visible term computed from the result of the first pass. Continuing with the example above, if the policy says to hide the expansion of `defthunk`, it would be inappropriate to produce a `letrec` form revealing

parts of `defthunk`'s implementation. Instead, in the second pass only expressions visible at the end of the first pass should be considered. The resulting expansion looks like this:

```
(lambda (f x)
  (defthunk g (or x 12))
  (f g))
⟶  ;; Start of pass 2; hiding has occurred
(lambda (f x)
  (defthunk g (let ([t x]) (if t t 12)))
  (f g))
```

The `letrec` transformation is not shown, but since the `or` expression was visible at the end of the first pass, its expansion is shown in the second.

If the traversal encounters a two-pass expansion while already in seek mode, the two passes are just processed sequentially. Two-pass expansions can be nested. For example, both modules and internal definitions use two-pass expansion, so every `lambda` body within a module—or another `lambda` body, for that matter—involves nested two-pass expansion.

5.3 Experience

The macro stepper is available in the standard distribution of PLT Scheme. Macro programmers have been using it since its first release, and their practical experience confirms that it is an extremely useful tool. It has found use in illustrating macro expansion principles on small example programs, occasionally supplementing explanations given on the mailing list in response to macro questions. In Kathi Fisler's accelerated introductory course at Worcester Polytechnic Institute, a course that covers macros, the students discovered the macro stepper on their own and found it "cool."

Our users report that the macro stepper has significantly increased their productivity. It has helped several fellow researchers debug large multi-module systems of co-operating macros that implement their language extensions. One example is Typed Scheme [68], a typed dialect of PLT Scheme. The implementation of Typed Scheme is a large project that makes intensive use of the features of the PLT Scheme macro system, and its development benefited greatly from the macro stepper.

User reports support the importance of macro hiding, which allows programmers to work at the abstraction level of their choice. The feature is critical for dealing with nontrivial programs. Otherwise, programmers are overwhelmed by extraneous details, and their programs change beyond recognition as the macro stepper elaborates away the constructs that give programs their structure. At the same time, the need for more structured navigation is clear, especially when debugging large modules. This is a priority of the continuing development of the macro stepper.

The macro stepper aided in its own development. We used macros to implement the grammar transformation for handling errors (see Section 5.1), and our first attempt produced incorrect references to generated nonterminal names—a kind of lexical scoping bug. Fortunately, we were able to use a primitive version of the macro stepper to debug the grammar macros.

CHAPTER 6

Fortifying Macros

Existing systems for expressing macros force macro programmers to make a choice between clarity and robustness. If they choose clarity, they must forgo validating significant parts of the macro's specification, leading to second-class language extensions of generally low quality. If they choose robustness, they must write in a style that obscures the relationship between the implementation and the specification. The key problem is that macro systems lack a sufficiently expressive way of defining, parsing, and validating syntax.

We introduce a new domain specific language for writing macros based on declarative syntax specifications. Our system uses these specifications to detect misuses—including violations of context-sensitive constraints—and synthesize appropriate feedback, reducing the need for ad hoc validation code. In this chapter we describe the limitations of existing systems and present the motivations behind our system's design. Sections 6.3 and 6.4 describe our system and illustrate its features with a series of examples. Chapter 7 presents a semantics, and Chapter 8 discusses case studies.

6.1 What is a macro?

It is well known that `let` can be expressed as the immediate application of a `lambda` abstraction [48]. The `let` expression's variables become the formal parameters of the `lambda` expression, the initialization expressions become the application's arguments, and the body becomes the body of the `lambda` expression. Here is a quasi-formal expression of the idea:

```
(let ([var rhs] ...) body) = ((lambda (var ...) body) rhs ...)
```

It is understood that each `var` is an identifier and each `rhs` and `body` is an expression. This might be stated as an aside to the above equation or it might be a consequence of metavariable conventions. The variables also must be distinct—no duplicates are allowed.

Languages such as Lisp and Scheme have long supported these kinds of *macros*, a mechanism that allows *programs* to define new notations. Essentially, macros are an API for extending the front end of the compiler. A

macro definition associates a name with a compile-time callback, a *syntax transformer* function. When the compiler encounters a use of the macro name, it calls the associated macro transformer to rewrite the expression. Because macros are defined by translation, rather than by a plugin that directly manipulates the compiler's data structures, they are often called *derived syntactic forms* [52]. In the example above, the derived form `let` is expanded into the primitive forms `lambda` and application.

Existing systems make it surprisingly difficult to write macros that properly validate their syntax. Without validation, macros allow faulty terms to trigger errors in other syntactic forms that might not even appear in the original program, leading to incoherent error messages. Such macros do not create seamless linguistic abstractions, but become sources of confusion and distraction for programmers.

6.2 Expressing macros

To illustrate the problems with existing macro systems, let us examine them in the context of the ubiquitous `let` example:

```
(let ([var rhs] ...) body) = ((lambda (var ...) body) rhs ...)
                                        the vars are distinct identifiers
                                      body and the rhss are expressions
```

A macro's syntax transformer is essentially a function from syntax to syntax. Many Lisp and Scheme dialects take that as the entirety of the interface: macros are just distinguished functions, introduced with `define-macro` instead of `define`, that consume and produce S-expressions representing terms. Macros in such systems typically use standard list and S-expression functions to "parse" syntax, and they use `quasiquote` to build up the desugared expression:[1]

```
(define-macro (let form)
  `((lambda ,(map car (cadr form)) ,@(caddr form))
    ,@(map cadr (cadr form))))
```

A well-organized implementation would extract and name the subterms before assembling the result, separating *parsing* from *code generation*:

```
(define-macro (let form)
  (define bindings (cadr form))
  (define body (caddr form))
  (define vars (map car bindings))
```

[1] Some hygienic macro systems, such as those based on explicit renaming [13] or syntactic closures [7], use mechanisms other than `quasiquote` to construct result terms, but they use the same *parsing* mechanisms as other systems we discuss here.

```
(define rhss (map cadr bindings))
`((lambda ,vars ,body) ,@rhss))
```

These definitions do not resemble the specification, however, and they do not even properly implement it. The parsing code does not validate the basic syntax of `let`. For example, extra terms appearing after a binding pair are simply ignored:

```
(let ([x 1] [y 3 'what-about-me?]) (+ x y))
```

Macro writers, eager to move on after checking that the macro "works," will continue to write sloppy macros like these unless their tools make it easy to write robust ones.

To that end, Kohlbecker and Wand [47] created a domain-specific language called Macro-By-Example (MBE). In MBE, macros are specified in a notation close to the initial informal equation, and the parsing and code generation code is produced automatically. The generated parsing code enforces the declared syntax, rejecting malformed uses such as the one above.

MBE replaces the procedural code with a sequence of clauses, each consisting of a pattern and a template. The patterns describe the macro's syntax. A pattern generally contains *pattern variables*, and when a pattern matches, the pattern variables are bound to the corresponding subterms of the actual macro use. These subterms are substituted into the template where the pattern variables occur to produce the macro's expansion result.

Here is `let` expressed with `syntax-rules` [44], one of many implementations of MBE:

```
(define-syntax let
  (syntax-rules ()
    [(let ([var rhs] ...) body)
     ((lambda (var ...) body) rhs ...)]))
```

This macro has one clause. The pattern variables are `var`, `rhs`, and `body`.

The crucial innovation of MBE is the use of ellipses (`...`) to describe sequences of subterms with homogeneous structure. Such sequences occur frequently in S-expression syntax. Sometimes they are simple sequences, such as the list of identifiers of a `lambda` expression's formal parameters, but often the sequences have more complex structure, such as binding pairs associating `let`-bound variables with their values.

Every syntax variable has an associated *ellipsis depth*. A depth of 0 means the variable contains a single term, a depth of 1 indicates a list of terms, and so on. Syntax templates are statically checked to make sure the ellipsis depths are in order. We do not address template checking and transcription in this work.

Ellipses do not add expressive power to the macro system, but they do add expressiveness to *patterns*. Without ellipses, the `let` macro could still be expressed via auxiliary clauses and explicit recursion. The structure of valid

`let` expressions would be obscured; instead of residing in a single pattern, it would be distributed across multiple clauses of a recursive macro. That is, ellipses help close the gap between specification and implementation.

Yet even MBE enables the macro writer to specify only part of the information present in the informal description at the start of this section. The example macros presented up to this point omit two critical aspects of the `let` syntax from validation: the first term of each binding pair must be an identifier, and those identifiers must be distinct.

Consider these misuses of `let`:

```
(let ([x 1] [x 2]) (h x))
(let ([(x y) (f 7)]) (g x y))
```

In neither case does the `let` macro report that it has been used with invalid syntax. Both times it inspects the syntax, says "close enough," and produces an invalid `lambda` expression. Then `lambda`, implemented by a careful compiler writer, signals an error, such as "lambda: duplicate identifier in: x" in PLT Scheme [36] for the first term and "invalid parameter list in `(lambda ((x y)) (g x y))`" in Chez Scheme [10] for the second.

Source location tracking [20], in macro systems that offer it, helps the situation somewhat. For example, the DrScheme [27] programming environment highlights the duplicate identifier. But this is not a solution. Macros should report errors on their own terms.

There is also a danger that a macro might pass through syntax that has an unintended meaning. In PLT Scheme, the second example above produces the error "unbound variable in: `y`." The pair `(x y)`, rather than being rejected by `lambda`, is accepted as an *optional parameter* with a default expression, and the error occurs during the expansion of `y`. If `y` were a legal expression, the second example would be silently accepted. In this case the behavior is innocuous, but in another macro the result of injecting unvalidated syntax could be more sinister.

The traditional solution to this problem is to include a guard expression, sometimes called a "fender," that is run after the pattern matches but before the transformation expression is evaluated. The guard expression produces true or false to indicate whether those constraints inexpressible as patterns are satisfied. If the guard expression fails, the pattern is skipped and the next pattern is tried. If all of the patterns fail, the macro raises a generic syntax error.

The `syntax-case` [20, 64] implementation of MBE provides guard expressions:

```
(define-syntax (let stx)
  (syntax-case stx ()
    [(let ([var rhs] ...) body)
     ;; Guard expression
     (and (andmap identifier? (syntax->list #'(var ...)))
```

```
(define-syntax (let stx)
  (syntax-case stx ()
    [(let ([var rhs] ...) body)
     (begin
      ;; Error-checking code
      (for-each (lambda (var)
                  (unless (identifier? var)
                    (syntax-error "expected identifier" stx var)))
                (syntax->list #'(var ...)))
      (let ([dup (check-duplicate #'(var ...))])
        (when dup
          (syntax-error "duplicate variable name" stx dup)))
      ;; Result term
      #'((lambda (var ...) body) rhs ...))]
    [(let bad-bindings body)
     (syntax-error "bad binding sequence" stx #'bad-bindings)]
    [(let bindings)
     (syntax-error "missing body expression" stx stx)]
    ;; ad nauseam
    ))
```

Figure 6.1: Explicit error coding

```
          (not (check-duplicate #'(var ...))))
      ;; Transformation expression
      #'((lambda (var ...) body) rhs ...)]))
```

A `syntax-case` clause consists of a pattern, an optional guard expression, and a transformation expression. Syntax templates within expressions are marked with a `#'` prefix.

Guard expressions suffice to prevent macros from accepting invalid syntax, but they suffer from two flaws. First, since guard expressions are separated from transformation expressions, work needed both for validation and transformation must be done twice. Second and more important, guards provide no way of explaining why the syntax was invalid. That is, they only control parsing; they do not track specific causes of failure.

To provide high-quality error feedback, explicit error coding is necessary, as shown in Figure 6.1. Notice, however, that of the fourteen non-comment lines of the macro's clauses, three of them are patterns, one is a template, and *ten* are dedicated to parsing and validation. Two of the three clauses exist only for error-reporting, and only the most conscientious macro writers are likely to take the time to enumerate all the ways the syntax could be invalid, or even a reasonable subset.

Certainly, the code for `let` could be simplified. Macro writers could build libraries of common error-checking routines. Such an approach, however,

```
(define-struct struct-id (field-id ...) option ...)
  where option = #:mutable
               | #:super super-struct-expr
               | #:inspector inspector-expr
               | #:property property-expr value-expr
               | #:transparent
               | ...
```

Figure 6.2: The syntax of `define-struct`

would still mix the error-checking code with the transformation code, rather than bringing the macro's implementation closer to its specification. Furthermore, abstractions that focused solely on raising syntax errors would not address the other purpose of guards, the selection between multiple valid alternatives.

Even ignoring the nuances of error reporting, another problem remains. Some syntax is simply hard to parse using MBE patterns. Macro writers cope in two ways: either they compromise on the user's convenience and simplify the syntax to something easier to parse, or they hand-code the parser.

The ability to parse complicated syntax is needed, for example, to support keyword arguments. Keyword arguments help prevent the proliferation of families of similarly-named procedures with minor differences. They serve the same purpose for macros.

An example of a keyword-enhanced macro is the `define-struct` form, described in Figure 6.2. It has several keyword options, which can occur in any order. The `#:transparent` and `#:inspector` keywords control when structure values can be inspected using reflective APIs. The `#:mutable` option makes fields mutable; the `#:property` allows structure types to override behavior such as how they are printed; and so on. The different keywords have different numbers of arguments, too: `#:mutable` and `#:transparent` have none, but others take one and `#:property` takes two.

Parsing a `define-struct` form gracefully is simply beyond the capabilities of MBE's pattern language, with its narrow focus on homogeneous sequences. A single optional keyword argument can be supported by simply writing two clauses—one with the argument and one without. At two arguments, calculating out the patterns becomes onerous, and the macro writer is likely to make odd, expedient compromises—arguments must appear in some order, or if one argument is given, both must be. Beyond two arguments, the approach is simply unworkable. The alternative is, again, to move part of the parsing processing into the transformer code. The macro writer sketches the rough structure of the syntax in broad strokes with the pattern, then settles in to fill in the details with procedural parsing code:

```
(define-syntax (define-struct stx)
```

```
(syntax-case stx ()
  [(define-struct name (field ...) . keyword-options)
   ——— #'keyword-options ———]))
```

In the actual implementation of `define-struct`, the parsing of keyword options alone takes over one hundred lines of code. Our system can shorten that code by an order of magnitude.

In summary, MBE offers weak syntax patterns, causing the work of validation and error-reporting to overflow into the guard and transformer expressions. Furthermore, guard expressions accept or reject entire clauses, and rejection comes without information as to why it failed. Finally, MBE lacks the vocabulary to describe a broad range of syntaxes that have recently appeared in languages with macros. Our domain-specific language for macros is designed to address all these issues.

6.3 The Design of `syntax-parse`

Our system, dubbed `syntax-parse`, uses a domain-specific language to support precise parsing and validation. It features three significant advances:

- An expressive language of syntax patterns, including the ability to annotate pattern variables with the classes of syntax they can match.
- A facility for defining new syntax classes as abstractions over syntax patterns. This allows the disciplined interleaving of declarative specifications and procedural constraint-checking code.
- A backtracking matching algorithm that tracks progress to rank and report failures and a notion of failure that carries error information.

Our syntax classes serve a role similar to nonterminals in traditional grammars. Guard expressions are retained and enhanced to provide proper rejection messages. They can additionally be located in syntax class definitions.

In this section we illustrate the design of `syntax-parse` with a series of examples. We start with the `let` example to illustrate the basic workings of patterns and syntax classes. Then we extend the example to explain the advanced features of our system.

6.3.1 Revisiting `let`

The syntax of `syntax-parse`—see Figure 6.3—is similar to `syntax-case`. As a starting point, here is the `let` macro transliterated from the `syntax-rules` version:

```
(define-syntax (let stx)
  (syntax-parse stx
```

```
(syntax-parse stx-expr clause ...)
  where clause = [pattern side-clause ... expr]
        side-clause = #:fail-when condition-expr message-expr
                    | #:with pattern stx-expr
```

Figure 6.3: Syntax of `syntax-parse`

```
     [(let ([var rhs] ...) body)
      #'((lambda (var ...) body) rhs ...)]))
```

It enforces only the same basic structural correctness as the MBE implementation. That is, it omits the two side conditions listed at the beginning of Section 6.2.

To this skeleton we add the constraint that every term labeled `var` must be an identifier. Likewise, `rhs` and `body` should be annotated to indicate that they are expressions. These annotations are written thus:

```
(define-syntax (let stx)
  (syntax-parse stx
    [(let ([var:identifier rhs:expr] ...) body:expr)
     #'((lambda (var ...) body) rhs ...)]))
```

Note that the syntax class annotations—`identifier` and `expr`—are not part of the pattern variable name, and they do not appear in the template.

The final constraint, that the identifiers are unique, can be expressed as a *side condition* using a `#:fail-when` clause, which is like a `syntax-case` guard expression but conveys additional information:

```
(define-syntax (let stx)
  (syntax-parse stx
    [(let ([var:identifier rhs:expr] ...) body:expr)
     #:fail-when (check-duplicate #'(var ...))
                 "duplicate variable name"
     #'((lambda (var ...) body) rhs ...)]))
```

The call to `check-duplicate` acts as a condition; if it returns `#f`, failure is averted and control continues to the template expression. But if it returns any other value, parsing fails with a "duplicate variable name" message; furthermore, when the condition value is a syntax object, that syntax is included as the specific site of the failure. In short, our side-conditions are like guard expressions, but the failures they generate carry information describing the specific reasons for the failure.

The other kind of side clause is the *sub-parse* clause, introduced by the `#:with` keyword; it is discussed in the next section.

6.3.2 Defining syntax classes

Macros tend to share common syntactic structure. For example, the binding pair syntax, consisting of an identifier for the variable name and an expression for its value, occurs in other variants of `let`: `let*` and `letrec`. It can be abstracted out into a separate reusable syntax class thus:

```
(define-syntax-class binding
  #:description "binding pair"
  #:attributes (var rhs)
  (pattern [var:identifier rhs:expr]))
```

The syntax class is named `binding`, but for the purposes of error reporting it is known as "binding pair." Since the pattern variables `var` and `rhs` have moved out of the main pattern into the syntax class, they must be exported as *attributes* of the syntax class so that their bindings are available to the main pattern. The name of the `binding`-annotated pattern variable, `b`, is combined with the names of the attributes to form the *nested attributes* `b.var` and `b.rhs`:

```
(define-syntax (let stx)
  (syntax-parse stx
    [(let (b:binding ...) body:expr)
     #:fail-when (check-duplicate #'(b.var ...))
                 "duplicate variable name"
     #'((lambda (b.var ...) body) b.rhs ...)]))
```

In addition to patterns, syntax classes may contain side conditions. For example, both the `let` and `letrec` forms require that their variable bindings be distinct. Here is an appropriate syntax class:

```
(define-syntax-class distinct-bindings
  #:description "sequence of binding pairs"
  #:attributes ([var 1] [rhs 1])
  (pattern (b:binding ...)
           #:fail-when (check-duplicate #'(var ...))
                       "duplicate variable name"
           #:with (var ...) #'(b.var ...)
           #:with (rhs ...) #'(b.rhs ...)))
```

The attributes of `distinct-bindings` are `var` and `rhs`. They are bound by the `#:with` clauses, each of which consists of a pattern followed by an expression, which may refer to previously bound attributes such as `b.var`. The expression's result is matched against the pattern, and the pattern's attributes are available for export or for use by subsequent side clauses. Unlike the `var` and `rhs` attributes of `binding`, these have an *ellipsis depth* of 1, so they can be used within ellipses in the macro's template, even though `distinct-bindings` does not occur within ellipses in the macro's pattern:

```
(define-syntax (let stx)
  (syntax-parse stx
    [(let bs:distinct-bindings body:expr)
     #'((lambda (bs.var ...) body) bs.rhs ...)]))
```

Many built-in syntax classes do little more than use predicates in their side conditions. For example, here is the definition of `identifier`:

```
(define-syntax-class identifier
  #:opaque
  #:attributes ()
  (pattern x
           #:fail-when (not (identifier? #'x)) #f))
```

This definition uses a new keyword option, `#:opaque`. An *opaque* syntax class hides internal error details. Consequently, the side condition that implements the `identifier?` check need not supply an informative error message. If the side condition fails, its message—`#f`, here—is discarded, and the error is reported by the syntax class instead, using the string "expected identifier." Since the `#:description` option is omitted, the description defaults to the name of the syntax class, `identifier`.

Another way of describing a sequence of distinct bindings is as a syntax class *parameterized* over a list of identifiers that have already been seen. Using a parameterized syntax class requires the explicit `~var` pattern constructor.[2] A `~var` pattern consists of a pattern variable and a syntax class applied to arguments, which are expressions in the scope of all attributes bound so far in the pattern. The colon notation is just shorthand for a `~var` pattern with no arguments; for example, the pattern `x:identifier` is the same as `(~var x (identifier))`.

Figure 6.4 shows the alternative definition of `distinct-bindings`. Note that in the second pattern of `bindings-excluding` the pattern variable `var0` is used in the argument to the `bindings-excluding` class. Parsing happens left to right; consequently, pattern variable binding also runs left to right, following the principle of scope being determined by control domination [62]. The auxiliary `id-excluding` syntax class is a trivial combination of `identifier` and a side condition on the `seen` parameter.

The second definition of `distinct-bindings` is not equivalent to the first. The first definition verifies the structure of the binding pairs first, then checks for a duplicate name. The second checks the structure and checks duplicates in the same pass. They report different errors for the following term:

```
(let ([a 1] [a 2] [x y z]) a)
```

The first version reports that the third binding pair is ill-formed. The second reports that the second occurrence of `a` is a duplicate. The macro writer must decide the most suitable order of validation.

[2] All pattern constructor keywords start with a tilde (`~`).

```
(define-syntax-class distinct-bindings
  #:attributes ([var 1] [rhs 1])
  (pattern (~var bs (bindings-excluding '()))
           #:with (var ...) (bs.var ...)
           #:with (rhs ...) (bs.rhs ...)))

(define-syntax-class (bindings-excluding seen)
  #:description "sequence of binding pairs"
  #:attributes ([var 1] [rhs 1])
  (pattern ()
           #:with (var ...) '()
           #:with (rhs ...) '())
  (pattern ([(~var var0 (id-excluding seen)) rhs0]
            . (~var rest (bindings-excluding (cons #'var0 seen))))
           #:with (var ...) #'(var0 rest.var ...)
           #:with (rhs ...) #'(rhs0 rest.rhs ...)))

(define-syntax-class (id-excluding seen)
  #:attributes ()
  (pattern x:identifier
           #:fail-when (identifier-member #'x seen)
                       "duplicate variable name"))
```

Figure 6.4: Parameterized syntax classes

6.3.3 Backtracking with progress

Syntax class annotations and side conditions serve a dual role in our system. When they fail, they trigger errors that report the specific site and cause of the failure. A failure triggers an immediate error, however, only if there are no *choices* in the parsing process. If there are choices, failure causes backtracking, and errors are deferred until all viable choices are exhausted. Thus annotations and side conditions act not only as a checklist for reporting errors, but as a way of guiding parsing through a sequence of choices.

In Scheme, `let` has another variant, in which the implicit procedure has a name. This notation provides a handy loop-like syntax. Here are the two variants and their translations expressed using `syntax-parse`:

```
(define-syntax (let stx)
  (syntax-parse stx
    [(let loop:identifier bs:distinct-bindings body:expr)
     #'(letrec ([loop (lambda (bs.var ...) body)])
         (loop bs.rhs ...))]
    [(let bs:distinct-bindings body:expr)
     #'((lambda (bs.var ...) body) bs.rhs ...)]))
```

$$\begin{array}{lrcl} \text{Progress} & \pi & ::= & ps^* \\ \text{Progress Step} & ps & ::= & \textsc{First} \mid \textsc{Rest} \mid \textsc{Side} \end{array}$$

$$\textsc{First} < \textsc{Rest} < \textsc{Side}$$

$$\epsilon < ps \cdot \pi \qquad \frac{\pi_1 < \pi_2}{ps \cdot \pi_1 < ps \cdot \pi_2} \qquad \frac{ps_1 < ps_2}{ps_1 \cdot \pi_1 < ps_2 \cdot \pi_2}$$

Figure 6.5: Progress datatype

The macro uses the syntax class annotations to pick the applicable pattern, "named `let`" if the first argument is an identifier and "normal `let`" if it is a binding list. The use of syntax class annotations to select the clause to match must be reconciled with the role of annotations in error reporting. Pattern-matching failure clearly cannot always immediately signal an error. But the annotations must retain their error-reporting capacity; if the whole parsing process fails, the annotations must be used to generate a specific error.

The dual role of failure is supported using the following approach. When there are multiple alternatives, such as multiple clauses in a `syntax-parse` expression or multiple variants of a syntax class definition, they are tried in order. As alternatives are tried, a list of failures is accumulated, and each failure contains a measure of the *progress* made into the term. If parsing fails, the attempts that made the most progress are chosen to explain the syntax error. Usually, but not always, there is a unique maximum, resulting in a single, useful error explanation.

Figure 6.5 defines our notion of progress as variable-length sequences of individual progress steps. The progress steps FIRST and REST indicate the first part and rest of a compound term, respectively. Since parsing is performed left to right, if the parser is looking at the rest of a term, the first part must have been parsed successfully. The SIDE progress step indicates a side condition or sub-parsing process.

Progress is ordered lexicographically. Steps are recorded left to right, so for example the second term in a sequence is written REST · FIRST; that is, take the rest of the full term and then the first part of that.

Consider the following erroneous `let` term:

```
(let ([a 1] [2 b]) (* a b))
```

The first clause of the new `let` macro fails at the second subterm with the progress string REST·FIRST. The second clause, however, fails deeper within the second argument, at REST · FIRST · REST · FIRST · FIRST, which is strictly more progress than REST · FIRST. So the second failure is reported as the

error, and the first failure is deemed irrelevant. The macro reports that it expected an identifier in place of 2.

If `identifier` were not an *opaque* syntax class, its internal progress, SIDE, would be included and the first clause would have instead failed at REST · FIRST · SIDE, and it would have been chosen instead of the second clause. That error does not reflect our expectations; we regard the `identifier` check as a simple, *early* check. The need to distinguish different kinds of side-condition check motivated the `#:opaque` option.

Sometimes multiple alternatives fail at the same place. For example, consider the following term:

```
(let 5)
```

Both clauses make the same amount of progress with this term: REST·FIRST. As a result, the error message says `let` expected, in place of `5`, either an "identifier" or a "sequence of binding pairs." The phrasing is synthesized from the descriptions of the syntax classes.

Backtracking can also occur within a pattern. For instance, in an ellipsis pattern, there is a choice between parsing another repetition of the ellipsis head or parsing the ellipsis tail. So while the pattern `(a ... b c)` matches any proper list of at least two terms, it does so only after backtracking a few times.

6.3.4 Error messages

Failures contain, in addition to progress, messages that indicate the location and nature of the error. A typical error message is

```
let: expected identifier or sequence of binding pairs in: 5
```

for the last example in the previous section. This message consists of the macro's expectations (an identifier or a sequence of binding pairs) and the specific term where parsing failed (`5`).

Our system produces messages from a strictly limited set of ingredients. It automatically synthesizes messages for literal and datum patterns. As a special case, it knows how to report certain errors involving the datum pattern `()` as "unexpected terms." The other ingredients are the descriptions attached to syntax classes and `~describe` forms, the messages included in explicit side conditions such as `#:fail-when` clauses, and the labels given in repetition constraints. That's it.

In particular, `syntax-parse` does not synthesize messages to describe compound patterns. For example, consider the following pattern:

```
(var:identifier rhs:expr)
```

If a term such as `5` is matched against this pattern, it fails to match the compound structure of the pattern. The matching process does not reach the

identifier or expression check. One possible error message is "expected list of an identifier and an expression"; another is "expected (identifier expr)." Macro writers occasionally write error messages of both forms in practice. We have chosen not to generate such messages for two reasons: first, it is technically difficult; and second, we believe such messages are misguided.

Generating messages from patterns is feasible when the pattern is simple, such as the example above. For patterns with deeper nesting and patterns using advanced features, however, generating an accurate message is tantamount to simply displaying the pattern itself. While showing patterns in failures is a useful debugging aid for macro developers, it is a bad way to construct robust linguistic abstractions. Errors should be based on documented concepts, not implementation details.

A syntax error should identify the faulty term and concisely explain what was expected. It should not recapitulate the documentation or suggest a remedy; rather, it should make it easy to locate the appropriate documentation and devise a remedy. Consequently, our system reports errors only using descriptions and messages supplied by the macro writer, who should document the macro using those same descriptions and messages.

When a compound pattern such as the one above fails, the pattern's context is searched and the nearest enclosing description, whether from a syntax class or `~describe` form, is used to report the error. If there is no enclosing description, the generic message "bad syntax" is produced along with the name of the macro, indicating that the programmer should look at the macro's syntax documentation.

6.4 Syntax patterns

The power of `syntax-parse` is due to its expressive pattern language, an extension of the syntax patterns of MBE with additional pattern forms. This section describes the additional forms and explains how they are used.

6.4.1 Single-pattern variants

The syntax patterns of MBE include pattern variables, literal data, literal identifiers, pairs, and ellipses (repetitions). Our main category of syntax patterns, called "single-term patterns" (S), extends MBE patterns with a number of new variants, summarized in Figure 6.6. As demonstrated, pattern variables may be annotated with syntax classes. Single-term patterns (S) may contain action patterns (A), head patterns (H), and ellipsis head patterns (EH); we discuss those patterns in the following sections. Here are the new variants of single-term patterns:

`(~or` S^+`)` matches if any of the subpatterns S match. This pattern introduces a backtracking choice point where each S is an alternative. The

Single-term pattern S ::= x
| x:$class_S$
| (~var x ($class_S$ $expr^*$))
| (~literal x)
| $datum$
| (H . S)
| (A . S)
| (~and S $\{S|A\}^*$)
| (~or S^+)
| (~not S)
| (EH ... . S)
| (~describe $expr$ S)
| (~describe #:opaque $expr$ S)

Figure 6.6: Single-term patterns

~or-pattern binds all of its subpatterns' attributes, but some of those attributes may have "absent" values.

(~and S $\{S|A\}^*$) matches if every subpattern matches. The attributes of each subpattern S are bound within every subsequent subpattern, and the attributes of all subpatterns are bound by the ~and-pattern.

(~not S) matches if S fails to match. The ~not-pattern binds no attributes.

(~literal x) matches any identifier that refers to the same binding as x. This is a standard notion of identifier equivalence in hygienic macro systems [46, 14, 20]; see $=_{\text{free}}$ in Chapter 3.

(~describe $expr$ S) matches whatever S matches and binds the same attributes as S. If matching fails within S without a "good" explanation, the description produced by $expr$ is attached to the failure.

(~describe #:opaque $expr$ S) matches like S, but if matching fails, then both internal progress and messages generated by S are discarded and matching fails with the message $expr$.

The "logical" pattern constructors enable concise specifications. Consider the following definition of a syntax class that recognizes a simple language of types:

```
(define-syntax-class type
  (pattern (dom:type ... (~literal ->) rng:type ...))
  (pattern (~and ground:id (~not (~literal ->)))))
```

In the second variant, the ~and pattern constructor is used to say that a ground type is an identifier that is not the reserved -> symbol.

Using ~or involves a bit of extra machinery. In many pattern matching systems, if local disjunction is supported at all, it is supported only under the condition that the disjuncts bind the same pattern variables. Our disjunctions are more flexible: the ~or-pattern binds the *union* of the attributes of the disjuncts, and those attributes that do not occur in the selected disjunct get the "absent" value of #f.

It is illegal to use a non-syntax-valued attribute in a syntax template. The attribute form accesses the value of the attribute, allowing the programmer to check whether it is safe to use as syntax. Here is an auxiliary function for parsing field declarations for a class macro, where a field declaration contains either a single name or distinct internal and external names:

```
(define (parse-field-declaration stx)
  (syntax-parse stx
    [(field (~or field:id [internal:id field:id]))
     (make-field (if (attribute internal) #'internal #'field)
                 #'field)]))
```

Many potential uses of ~or-patterns are better expressed as syntax classes, not least because a syntax class can use a #:with clause to bind missing attributes:

```
(define-syntax-class field-declaration
  #:attributes (internal field)
  (pattern field:id
           #:with internal #'field)
  (pattern [internal:id field:id]))
```

There is a pattern form that has the same effect as a #:with clause, however; we discuss it in Section 6.4.2.

The ~describe form is used to give a description to a complex pattern. Our system does not synthesize error messages for compound patterns such as (var:identifier rhs:expr) or (x:identifier ...). When these patterns fail, there is no good message to report; we call such failures "ineffable." Within a ~describe form, however, when an ineffable failure occurs it is given the ~describe form's description. If the #:opaque option is supplied, the failure description is always replaced, even if the failure is not ineffable; in addition, local progress is discarded.

If parsing a ~describe pattern succeeds, the choices made in the subpattern are committed [53, 41] and subsequent failures do not backtrack to within the subpattern. Commitment is incorporated into ~describe to simplify error selection and the syntax class protocol.

Syntax class bodies act as though they are wrapped with ~describe forms. Recall the syntax class definitions from the let example, binding and distinct-bindings. The binding syntax class could be inlined into distinct-bindings as follows:

Action pattern A ::= `(~fail #:when` *condition message*`)`
| `(~fail #:unless` *condition message*`)`
| `(~parse` S *expr*`)`

Figure 6.7: Action patterns

```
(define-syntax-class distinct-bindings
  #:description "sequence of distinct binding pairs"
  (pattern ((~describe "binding pair" [var:id rhs:expr]) ...)
           #:fail-when ———))
```

In fact, `distinct-bindings` can be inlined into the definition of `let` as well, but that also requires a way to inline the syntax class's side condition into the pattern at the right place. (If it were simply made a side condition of the macro clause, the ordering of checking would be slightly disrupted.) This task is accomplished by *action patterns*.

6.4.2 Action patterns

Action patterns, shown in Figure 6.7, do not directly describe syntax; rather, they affect the parsing process in other ways. Here are the action patterns:

`(~fail #:when` *condition message*`)` evaluates *condition*; if true, the matching process backtracks with a failure containing *message*. The variant with `#:unless` inverts the meaning of the *condition*.

`(~parse` S *expr*`)` evaluates *expr* and matches it against S.

The `~fail` and `~parse` patterns do not introduce new forms of behavior; instead they allow programmers to express side clauses as patterns. With action patterns we can inline both `binding` and `distinct-bindings` into the definition of `let`:

```
(define-syntax (let stx)
  (syntax-parse stx
    [(let (~describe "sequence of distinct bindings"
            (~and ((~describe "binding pair"
                     [var:identifier rhs:expr])
                   ...)
                  (~fail #:when (check-duplicate #'(var ...))
                         "duplicate variable")))
          body:expr)
     #'((lambda (var ...) body) rhs ...)]))
```

A common use of the `~parse` pattern form is to bind default values within an `~or` pattern, avoiding the need for explicit `attribute` checks later. Re-

Head pattern	H	$::=$ $x : class_H$
		$\mid$ (~var x ($class_H$ $expr^*$))
		$\mid$ (~seq . L)
		$\mid$ (~and H $\{H \mid A\}^*$)
		$\mid$ (~or H^+)
		$\mid$ (~describe $expr$ H)
		$\mid$ (~describe #:opaque $expr$ H)
		$\mid$ S
List pattern	L	$::=$ ()
		$\mid$ (H . L)
		$\mid$ (A . L)
		$\mid$ (EH ... . L)

Figure 6.8: Head patterns

call the example of `parse-field-declaration` from Section 6.4.1. Here `internal` is bound in both alternatives, simplifying the result template:

```
(define (parse-field-declaration stx)
  (syntax-parse stx
    [(field (~or (~and field:id (~parse internal #'field))
                 [internal:id field:id]))
     (make-field #'internal #'field)]))
```

6.4.3 Head patterns

The patterns of Sections 6.4.1 and 6.4.2 do not provide the power needed to parse macros like `define-struct` (see Figure 6.2). There are elements of `define-struct`'s syntax that comprise multiple consecutive terms, but syntax patterns so far describe only single terms. An occurrence of the inspector option, for example, consists of two adjacent terms: the keyword `#:inspector` followed by an expression, such as in the following definition:

```
(define-struct point (x y) #:inspector an-inspector #:mutable)
```

No single-term pattern describes the inspector option. In particular, the pattern `(#:inspector insp:expr)` does not, because the parenthesized term `(#:inspector an-inspector)` does not appear in the example above.

Our solution is to introduce *head patterns* (see Figure 6.8), which describe sequences of terms. The primary head pattern constructor is `~seq`, which is followed by a proper list pattern (L). For example, `(~seq x y z)` matches a sequence of any three terms, and `(~seq x:id ... 5)` matches a sequence consisting of any number of identifiers followed by the term `5`.

A head pattern must be combined with a tail pattern, which is a normal single-term pattern. The combined pattern matches by attempting to split

a single term into a prefix sequence of terms that matches the head pattern and a suffix term that matches the tail. For example, the term `(1 2 3 a b)` matches the pattern `((~seq x y z) w:id ...)` because the term can be split into the prefix of three terms `1 2 3` matching `(~seq x y z)` and the suffix `(a b)` matching `(w:id ...)`. Of course, `((~seq x y z) w:id ...)` is equivalent to `(x y z w:id ...)`.

The `~seq` pattern is useful primarily when combined with other pattern constructors, such as `~and` and `~or`. Here is a macro with an optional keyword argument:

```
(define-syntax (test-case stx)
  (syntax-parse stx
    [(test-case (~or (~seq #:around proc) (~seq)) e:expr)
     ——— (attribute proc) ———]))
```

Head patterns are not intrinsically tied to keywords, of course. We could describe the syntax of full Scheme `let`, accommodating both normal and "named `let`" syntax, with the following pattern:

```
(let (~or (~seq loop:id) (~seq)) bs:distinct-bindings
  body:expr)
```

An abstraction of a head pattern is called a *splicing syntax class*. Each of its variants is a head pattern (H), most often a `~seq` pattern, although other kinds of head pattern are possible. The optional keyword argument above could be extracted thus:

```
(define-splicing-syntax-class optional-around
  #:attributes (proc)
  (pattern (~seq #:around proc))
  (pattern (~seq)
           #:with proc #'(lambda (p) (p))))
```

A pattern variable annotated with a splicing syntax class can represent multiple terms. In the example below, `ka` matches two terms:

```
(define-syntax (test-case stx)
  (syntax-parse stx
    [(test-case ka:optional-around e) ——— #'ka.proc ———]))
(test-case #:around call-with-connection ———)
```

Head patterns can also occur in front of ellipses. In those cases, a few additional variants are available that enable macro writers to support multiple optional arguments occurring in any order.

Ellipsis-head pattern EH ::= `(~or` EH^+`)`
| `(~once` H `#:name` $expr$`)`
| `(~optional` H `#:name` $expr$`)`
| H

Figure 6.9: Ellipsis-head patterns

```
(define-splicing-syntax-class define-struct-options
  (pattern (~seq (~or (~optional (~seq #:mutable))
                      (~optional (~seq #:super super-expr))
                      (~optional
                       (~or (~seq #:inspector inspector-expr)
                            (~seq #:transparent)))
                      (~seq #:property pkey:expr pval:expr))
                 ...)))
```

Figure 6.10: Keyword options of `define-struct`

6.4.4 Ellipsis-head patterns

Ellipsis-head patterns are the final ingredient necessary to specify syntax like the keyword options of `define-struct`. An ellipsis-head patterns may have multiple alternatives combined with `~or`; each alternative is a head pattern. It specifies sequences consisting of some number of instances of the alternatives joined together. An alternative may be annotated with one of two additional pattern forms, `~optional` and `~once`, that restrict the number of times that alternative may appear in the sequence.

The meaning of an `~or`-pattern changes slightly when it occurs immediately before ellipses. Instead of "absent" values accruing for every alternative that is not chosen, only the chosen alternative accrues attribute values. Consequently, when the term `(1 a 2 b c)` is matched against the pattern `((~or x:identifier y:number) ...)`, `x` matches `(a b c)` and `y` matches `(1 2)`.

These extensions to ellipses and head patterns finally provide enough power to write an apt specification of `define-struct`'s options. For brevity, we extract the options as a splicing syntax class; see Figure 6.10.

Most of the options can occur at most once, so they are wrapped with `~optional` patterns. The exception is the `#:property` option, which can occur any number of times. The `#:inspector` and `#:transparent` options are mutually exclusive, so they are grouped together under one `~optional` disjunct.

Given the syntax class `define-struct-options`, the following pattern describes the syntax of `define-struct`:

```
(define-struct name:identifier (field:identifier ...)
  opts:define-struct-options)
```

Of course, the definition of `define-struct-options` needs a bit more work to provide attributes for all the information that the macro definition needs. All this takes, though, is a few uses of `~and` and `~parse`.

CHAPTER 7

Pattern Matching with Failure

This chapter discusses the mechanics of pattern matching with failure information. We present a semantics of a core version of the `syntax-parse` pattern language and a sketch of the implementation in PLT Scheme.

7.1 Backtracking with failure information

We model backtracking with failure information with a "single-elimination" monad, a variant of well-known backtracking monads [72, 43]. A single-elimination (SE) sequence consists of a (finite) list of successes terminated by at most one failure. We write the single-elimination sequences consisting of successes a_1 through a_n punctuated by failure ϕ thus:

$$\langle a_1, \cdots, a_n; \phi \rangle$$

The monad is parameterized by a type from which the success elements are drawn. We discuss particular success types in Section 7.2. The sequences of successes may be empty. For simplicity we always include the failure and adjoin $\bot$, representing "no failure."

The important aspect of this monad is its handling of failures, which models our macro system's error selection algorithm. A failure (other than $\bot$) consists of a progress (π) together with a *reason*. For modeling our macro system, we use sets of messages as failure reasons. When sequences are combined, their failures are joined according to the following rules. If one failure has greater progress (according to the order of Figure 6.5), it is selected; if they have the same progress, their message sets are combined. The identity element is $\bot$; it is considered to have less progress than any other failure. $\mathrm{Failure}$ is a bounded join-semilattice with least element $\bot$.

Figure 7.1 defines the monad's operations, including unit, bind (written $\blacktriangleright$), and disjoin (written $\hat{}$). The unit operation creates a sequence of one success and no failure. Disjoin ($\hat{}$) concatenates successes and joins ($\vee$) the failures, and bind ($\blacktriangleright$) applies a function to all successes in a sequence and disjoins the resulting sequences with the original failure. This monad is similar to the standard list monad except for the way it handles failures.

$$
\begin{array}{llll}
\text{SE(A)} & se & ::= & \langle a_1, \cdots, a_n; \phi \rangle \qquad \text{where } a_i \in A \\
\text{Failure} & \phi & ::= & \bot \mid \textsc{Fail}(\pi, \mathit{reason}\}) \\
\text{Progress} & \pi & ::= & \epsilon \mid \pi \cdot \textsc{First} \mid \pi \cdot \textsc{Rest} \mid \pi \cdot \textsc{Side} \\
\text{Reason} & \mathit{reason} & ::= & \{\mathit{msg}_1, \cdots, \mathit{msg}_n\} \\
\text{Message} & \mathit{msg} & &
\end{array}
$$

$$
\begin{array}{rcl}
\text{unit}(a) & = & \langle a; \bot \rangle \\
\langle a_1, \cdots, a_n; \phi \rangle \blacktriangleright f & = & f(a_1) \mathbin{\hat{}} \cdots \mathbin{\hat{}} f(a_n) \mathbin{\hat{}} \langle\, ; \phi \rangle \\
A \otimes B & = & A \blacktriangleright \lambda x.\, B \\
\langle a_1, \cdots, a_k; \phi_1 \rangle \mathbin{\hat{}} \langle a_{k+1}, \cdots, a_n; \phi_2 \rangle & = & \langle a_1, \cdots, a_k, a_{k+1}, \cdots, a_n; \phi_1 \vee \phi_2 \rangle \\
\text{once}(\langle a_1, a_2, \cdots, a_n; \phi \rangle) & = & \langle a_1; \bot \rangle \\
\text{once}(\langle\, ; \phi \rangle) & = & \langle\, ; \phi \rangle \\
\text{replace-fail}(\langle a_1, \cdots, a_n; \phi \rangle, \phi') & = & \langle a_1, \cdots, a_n; \phi' \rangle
\end{array}
$$

Figure 7.1: Single-elimination sequences and operations

The monad supports a few additional operations. The function once truncates the sequence after the first success, if possible, discarding the rest of the successes and failure; otherwise it simply returns the failure. The replace-fail function replaces a SE-sequence's failure.

One might expect that a simpler model would be a list of successes *or* a failure. After all, if a pattern succeeds, backtracking typically only occurs when triggered by a failure of greater progress, which would make any failure in the prior pattern irrelevant. This is not always the case, however; the first conjunct of an `~and` pattern can succeed once, then fail at a later point than the second conjunct that triggered the backtracking. Here is a contrived pattern that has such behavior given the term `(1 1)`, for example:

```
(~and (~or (x 1) (x 2))
      (0 y))
```

On the other hand, it is not clear that the error selected for this example, `"expected the literal 2"`, is the most appropriate. A possible remedy is to order failures by a combination of term progress and pattern progress, enforcing the invariant that if a pattern succeeds, any subsequent failure must be greater than any failure within the pattern.

Nonetheless, we stick with the monad presented above for three reasons. First, ranking failures purely by progress is compelling and easy for programmers to understand. Second, our monad is more general than the refinement suggested above, giving us greater flexibility in exploring other failure-ranking options. Finally, this monad corresponds neatly to a two-continuation implementation [72].

Single pattern	S	::=	$\mathsf{Var}(x) \mid \mathsf{Class}_{\mathrm{S}}(x, c_{\mathrm{S}}(e)) \mid \mathsf{Datum}(d)^{\ell}$
			$\mid \mathsf{Literal}(x)^{\ell} \mid \mathsf{Pair}(S_1, S_2)^{\ell} \mid \mathsf{Head}(H_1, S_2)$
			$\mid \mathsf{And}(S_1, S_2) \mid \mathsf{Or}(S_1, S_2) \mid \mathsf{Rep}(EH_1, S_2)$
			$\mid \mathsf{Opaque}(S)^{\ell} \mid \mathsf{Action}(A)$
Action patterns	A	::=	$\mathsf{Fail}(e)^{\ell} \mid \mathsf{Parse}(S, e)$
Head pattern	H	::=	$\mathsf{Class}_{\mathrm{H}}(x, c_{\mathrm{H}}(e)) \mid \mathsf{Seq}(L)^{\ell}$
			$\mid \mathsf{And}(H_1, S_2) \mid \mathsf{Or}(H_1, H_2)$
List pattern	L	::=	$\mathsf{Datum}(())^{\ell} \mid \mathsf{Pair}(S_1, L_2)^{\ell} \mid \mathsf{Head}(H_1, L_2)$
			$\mid \mathsf{Rep}(EH_1, L_2)$
Ellipsis head pattern	EH	::=	$\mathsf{Between}(N_{min}, N_{max}, H)_R^{\ell_1,\ell_2} \mid \mathsf{Or}(EH_1, EH_2)$
Expression	ℓ, e		
RepLabel	R		

Figure 7.2: Core pattern syntax

7.2 Pattern matching

We explain the semantics of `syntax-parse` via reduction to the core pattern language of Figure 7.2. The core language eliminates syntactic sugar. For example, the shorthand for annotated pattern variables is eliminated, and And and Or patterns are binary. All syntax classes have a single parameter. The disjuncts of core Or patterns bind the same variables, using And and Parse if necessary to bind "absent" attributes.

All patterns that can fail are decorated with message expressions (ℓ). While we do not impose any particular structure on messages, we intend them to represent combinations of terms with error text. We assume the presence of an eval function for evaluating expressions. A `~describe` pattern is represented by distributing its message expression to its subpatterns.

We omit the ellipsis nesting depth of attributes; it is a static property, easy to compute separately. We generalize the repetition constraint forms (`~once` and `~optional`) to a Between form. An unconstrained ellipsis head pattern is modeled as a Between pattern with $N_{min} = 0$ and $N_{max} = \infty$. Each repetition disjunct has a distinct label (R) used to track repetitions and two message expressions, one to report too few repetitions and one for too many.

Figure 7.3 defines the additional domains and operations used by the semantics. Terms consist of atoms and "dotted pairs" of terms.[1] Parsing success is represented by an attribute record σ mapping names to terms. Records are combined by the $\sqcup$ operator, which produces a record with the union of the two records' attribute bindings. We overload the record combination operator notation; when the right-hand side is a SE-sequence, it indicates that the left-hand record is combined with every record in the se-

[1] We use the standard abbreviation, writing `(a b c)` for `(a . (b . (c . ())))`.

$$
\begin{array}{lll}
\text{Term} & z & ::= x \mid \mathit{datum} \mid \texttt{()} \mid \texttt{(}z_1\ \texttt{.}\ z_2\texttt{)} \\
\text{Record} & \sigma, \rho & ::= \{x_1 \mapsto z_n, \cdots, x_n \mapsto z_n\} \\
\text{RepEnv} & \xi & ::= \{R_1 \mapsto N_1, \cdots, R_n \mapsto N_n\}
\end{array}
$$

$$
\begin{array}{rcl}
\sigma \sqcup \langle \sigma_1, \cdots, \sigma_n; \phi \rangle & = & \langle \sigma \sqcup \sigma_1, \cdots, \sigma \sqcup \sigma_n; \phi \rangle \\
\mathrm{zipcons}(\sigma, \langle \sigma_1, \cdots, \sigma_n; \phi \rangle) & = & \langle \mathrm{zipcons}(\sigma, \sigma_1), \cdots, \mathrm{zipcons}(\sigma, \sigma_n); \phi \rangle \\
\mathrm{zipcons}(\sigma_1, \sigma_2) & = & \{x \mapsto \mathsf{Cons}(\sigma_1(x), \sigma_2(x)) \mid x \in \mathrm{dom}(\sigma_1)\} \\
 & & \sqcup\, (\sigma_2 - \mathrm{dom}(\sigma_1))
\end{array}
$$

Figure 7.3: Domains and operations for pattern semantics

quence. We also define the $\mathrm{zipcons}$ function, which combines two records by Consing the values of the first record onto the values of the second.

All of the pattern denotation functions are parameterized over a set of syntax definitions Δ and a record ρ containing attributes from patterns already matched. Pattern matching of single-term patterns produces SE-sequences of records. The other pattern variants produce sequences with additional results:

$$
\begin{array}{rcl}
\mathcal{S}[\![S]\!]^{\rho}_{\Delta} z \pi & : & \mathrm{SE}(\mathrm{Record}) \\
\mathcal{A}[\![A]\!]^{\rho}_{\Delta} \pi & : & \mathrm{SE}(\mathrm{Record}) \\
\mathcal{H}[\![H]\!]^{\rho}_{\Delta} z \pi & : & \mathrm{SE}(\mathrm{Record}, \mathrm{Term}, \mathrm{Progress}) \\
\mathcal{REP}[\![EH]\!]^{\rho}_{\Delta} z \pi \xi & : & \mathrm{SE}(\mathrm{Record}, \mathrm{Term}, \mathrm{Progress}) \\
\mathcal{EH}[\![EH]\!]^{\rho}_{\Delta} z \pi & : & \mathrm{SE}(\mathrm{Record}, \mathrm{Term}, \mathrm{Progress}, \mathrm{RepEnv}) \\
\mathcal{STOP}[\![EH]\!]^{\rho}_{\Delta} z \pi \xi & : & \mathrm{SE}()
\end{array}
$$

Action patterns also produce sequences of records. Head patterns include a term and progress at which to resume matching. The various functions for handling ellipses consume and produce repetition labels (R) and repetition label environments (ξ); these are used to check repetition constraints.

7.2.1 Single-term patterns

Figure 7.4 shows the denotations of single-term patterns.

A variable pattern always matches, and it produces a record mapping the pattern variable to the input term. A class pattern matches according to the pattern recorded in the syntax class environment Δ. Only one result is taken from the syntax class's pattern; if it is a record, its domain attributes are prefixed with the pattern variable using the pfx function.

Both Datum and Pair patterns can fail; if they do, their label expressions are evaluated using the inherited record ρ, modeling the fact that messages are computed and can depend on attributes of earlier patterns. A Pair pattern matches a pair by matching the first component, then for every success

$$\mathcal{S}[\![\mathsf{Var}(x)]\!]^{\rho}_{\Delta} z\pi$$
$$= \ \mathrm{unit}(\{x \mapsto z\})$$
$$\mathcal{S}[\![\mathsf{Class}_{\mathrm{S}}(x, c_{\mathrm{S}}(e))]\!]^{\rho}_{\Delta} z\pi$$
$$= \ \mathrm{once}(\mathcal{S}[\![S]\!]^{\{y \mapsto \mathrm{eval}(e,\rho)\}}_{\Delta} z\pi) \blacktriangleright \lambda\sigma.\ \mathrm{pfx}(x, \sigma) \sqcup \mathrm{unit}(\{x \mapsto z\})$$
$$\text{where } \{c_{\mathrm{S}}(y) = S\} \in \Delta$$
$$\mathcal{S}[\![d^{\ell}]\!]^{\rho}_{\Delta} z\pi$$
$$= \begin{cases} \mathrm{unit}(\emptyset) & \text{when } z = d \\ \langle\ ; \textsc{Fail}(\pi, \{\mathrm{eval}(\ell, \rho)\})\rangle & \text{otherwise} \end{cases}$$
$$\mathcal{S}[\![\mathsf{Pair}(S_1, S_2)^{\ell}]\!]^{\rho}_{\Delta} z\pi$$
$$= \begin{cases} \mathcal{S}[\![S_1]\!]^{\rho}_{\Delta} z_1(\pi \cdot \textsc{First}) \blacktriangleright \lambda\sigma.\ \sigma \sqcup \mathcal{S}[\![S_2]\!]^{\rho \sqcup \sigma}_{\Delta} z_2(\pi \cdot \textsc{Rest}) \\ \qquad \text{when } z = (z_1\ .\ z_2) \\ \langle\ ; \textsc{Fail}(\pi, \{\mathrm{eval}(\ell, \rho)\})\rangle \quad \text{otherwise} \end{cases}$$
$$\mathcal{S}[\![\mathsf{Head}(H_1, S_2)]\!]^{\rho}_{\Delta} z\pi$$
$$= \ \mathcal{H}[\![H_1]\!]^{\rho}_{\Delta} z\pi \blacktriangleright \lambda(\sigma, z', \pi').\ \sigma \sqcup \mathcal{S}[\![\sigma(S_2)]\!]^{\rho \sqcup \sigma}_{\Delta} z'\pi'$$
$$\mathcal{S}[\![\mathsf{And}(S_1, S_2)]\!]^{\rho}_{\Delta} z\pi$$
$$= \ \mathcal{S}[\![S_1]\!]^{\rho}_{\Delta} z\pi \blacktriangleright \lambda\sigma.\ \sigma \sqcup \mathcal{S}[\![\sigma(S_2)]\!]^{\rho \sqcup \sigma}_{\Delta} z\pi$$
$$\mathcal{S}[\![\mathsf{Or}(S_1, S_2)]\!]^{\rho}_{\Delta} z\pi$$
$$= \ \mathcal{S}[\![S_1]\!]^{\rho}_{\Delta} z\pi \ \hat{}\ \mathcal{S}[\![S_2]\!]^{\rho}_{\Delta} z\pi$$
$$\mathcal{S}[\![\mathsf{Opaque}(S)^{\ell}]\!]^{\rho}_{\Delta} z\pi$$
$$= \ \mathrm{once}(\text{replace-fail}(\mathcal{S}[\![S]\!]^{\rho}_{\Delta} z\epsilon, \textsc{Fail}(\pi, \{\mathrm{eval}(\ell, \rho)\})))$$
$$\mathcal{S}[\![\mathsf{Action}(A)]\!]^{\rho}_{\Delta} z\pi$$
$$= \ \mathcal{A}[\![A]\!]^{\rho}_{\Delta} \pi$$
$$\mathcal{S}[\![\mathsf{Rep}(EH_1, S_2)]\!]^{\rho}_{\Delta} z\pi$$
$$= \ \mathcal{RE}\mathcal{P}[\![EH_1]\!]^{\rho}_{\Delta} z\pi\xi_0 \blacktriangleright \lambda(\sigma, z', \pi').\ \mathcal{S}[\![S_2]\!]^{\rho \sqcup \sigma}_{\Delta} z'\pi'$$

Figure 7.4: Semantics of S-patterns

attempting to match the second component. The attributes from the first component are visible to the second, and the records from the two parts are combined.

Head patterns return not only records, but also terms and progress. These are used to resume matching with the tail pattern. In And patterns, like in Pair patterns, the attributes of the first conjunct flow to the second. An Or pattern matches by disjoining its disjuncts' sequences.

An Opaque pattern takes a single result from its subpattern, either success or failure. If the subpattern failed, the failure is replaced with the Opaque pattern's progress and label.

A Rep pattern matches some number of repetitions of the ellipsis head pattern followed by the tail pattern. As with head patterns, each success includes a term and progress in addition to a record.

$$\mathcal{A}[\![\mathsf{Fail}(e)^{\ell}]\!]^{\rho}_{\Delta}\pi$$
$$= \begin{cases} \mathrm{unit}(\emptyset) & \text{if } \mathrm{eval}(e,\rho) \text{ is a true value} \\ \langle\ ;\textsc{Fail}(\pi\cdot\textsc{Side},\{\mathrm{eval}(\ell,\rho)\})\rangle & \text{otherwise} \end{cases}$$
$$\mathcal{A}[\![\mathsf{Parse}(S,e)]\!]^{\rho}_{\Delta}\pi$$
$$= \mathcal{S}[\![S]\!]^{\rho}_{\Delta}(\mathrm{eval}(e,\rho))(\pi\cdot\textsc{Side})$$

Figure 7.5: Semantics of A-patterns

7.2.2 Action patterns

Action patterns, unlike the other kinds of patterns, do not depend on the term currently being matched. Like single-term patterns, however, they produce records. The denotations of action patterns are shown in Figure 7.5.

A Fail pattern evaluates its condition expression in the context of the previous attributes. Depending on the result, it either succeeds with an empty record or fails with the associated label.

A Parse pattern evaluates its subexpression to a term and then matches that term against the subpattern.

7.2.3 Head patterns

A Seq pattern matches a sequence of terms if the embedded list pattern (a subset of single-term patterns) would match the list of those terms. Rather than duplicating and modifying the denotation function for single-term patterns to work with list patterns, we reuse $\mathcal{S}$ and add a new variant of single-term pattern, EndOfHead, that sneaks the additional information into the record.[2] We assume that `term` and `pr` are attribute names not occurring in the pattern.

For head And patterns, we perform the opposite transformation; after the first conjunct matches a sequence of terms, we convert that sequence into a term via the take function. We convert the second conjunct from a head pattern to a single-term pattern and use it to match the new term.

7.2.4 Ellipsis-head patterns

The $\mathcal{REP}$ function joins two possibilities. The term can match the ellipsis head pattern, yielding a new term, progress, and repetition count environment, which are all fed back into the $\mathcal{REP}$ function. Or if the repetition constraints of the ellipsis head pattern are satisfied (checked with the $\mathcal{STOP}$ function) it can stop and match the tail pattern.

[2] This lack of compositionality can be removed, but we present it thus both for conciseness and because it suggests a simple implementation.

$$
\begin{array}{l}
\mathcal{H}[\![\mathsf{Seq}(L)]\!]^{\rho}_{\Delta} z\pi \\
\quad = \quad \mathcal{S}[\![S]\!]^{\rho}_{\Delta} z\pi \blacktriangleright \lambda\sigma.\ (\sigma - \{\mathtt{pr}, \mathtt{term}\}, \sigma(\mathtt{pr}), \sigma(\mathtt{term})) \\
\qquad \text{where } S = \text{rewrite-L-pattern}(L) \\
\mathcal{S}[\![\mathsf{EndOfHead}]\!]^{\rho}_{\Delta} z\pi \\
\quad = \quad \text{unit}(\{\mathtt{pr} = \pi, \mathtt{term} = z\}) \\
\mathcal{H}[\![\mathsf{And}(H_1, H_2)]\!]^{\rho}_{\Delta} z\pi \\
\quad = \quad \mathcal{H}[\![H_1]\!]^{\rho}_{\Delta} z\pi \blacktriangleright \lambda(\sigma, z', \pi').\ \sigma \sqcup \mathcal{S}[\![S_2]\!]^{\rho}_{\Delta}(\text{take}(z, \pi, \pi'))\pi \\
\qquad \text{where } S_2 = \mathsf{Head}(H_2, \mathtt{()}) \\
\mathcal{H}[\![\mathsf{Or}(H_1, H_2)]\!]^{\rho}_{\Delta} z\pi \\
\quad = \quad \mathcal{H}[\![H_1]\!]^{\rho}_{\Delta} z\pi \mathbin{\hat{}} \mathcal{H}[\![H_2]\!]^{\rho}_{\Delta} z\pi \\
\mathcal{H}[\![\mathsf{Class}_{\mathrm{H}}(x, c_{\mathrm{H}}(e))]\!]^{\rho}_{\Delta} z\pi \\
\quad = \quad \mathcal{H}[\![H]\!]^{\{y \mapsto \text{eval}(e,\rho)\}}_{\Delta} z\pi \blacktriangleright \mathrm{f} \\
\qquad \text{where } \{c_{\mathrm{H}}(y) = H\} \in \Delta \\
\qquad\qquad \mathrm{f}(\sigma, z', \pi') = \text{unit}(\mathrm{g}(\sigma, \pi'), z', \pi') \\
\qquad\qquad \mathrm{g}(\sigma, \pi') = \{x \mapsto \text{take}(z, \pi, \pi')\} \sqcup \text{pfx}(x, \sigma)
\end{array}
$$

$$
\begin{array}{rcl}
\text{rewrite-L-pattern}(\mathtt{()}) & = & \mathsf{EndOfHead} \\
\text{rewrite-L-pattern}(\mathsf{Pair}(S_1, L_2)) & = & \mathsf{Pair}(S_1, \text{rewrite-L-pattern}(L_2)) \\
\text{rewrite-L-pattern}(\mathsf{Head}(H_1, L_2)) & = & \mathsf{Head}(H_1, \text{rewrite-L-pattern}(L_2)) \\
\text{rewrite-L-pattern}(\mathsf{Rep}(EH_1, L_2)) & = & \mathsf{Rep}(EH_1, \text{rewrite-L-pattern}(L_2))
\end{array}
$$

Figure 7.6: Semantics of H-patterns

The function "terminates" if the term z always gets smaller each iteration. This can be assured either by a conservative static analysis (similar to "nullability" in context-free grammars) or by a dynamic check. We omit the check for the sake of brevity. More precisely, the function reaches a point where no head matches, and the equation simplifies to the following:

$$\mathcal{REP}[\![EH]\!]^{\rho}_{\Delta} z\pi\xi = \langle\ ;\phi\rangle \mathbin{\hat{}} \mathcal{REP}[\![EH]\!]^{\rho}_{\Delta} z\pi\xi \mathbin{\hat{}} (\mathcal{STOP}[\![EH]\!]^{\rho}_{\Delta}\pi\xi \otimes \langle(\emptyset, z, \pi); \bot\rangle)$$

We can compact the disjuncts, resulting in the following:

$$\mathcal{REP}[\![EH]\!]^{\rho}_{\Delta} z\pi\xi = (\mathcal{STOP}[\![EH]\!]^{\rho}_{\Delta}\pi\xi \otimes \langle(\emptyset, z, \pi), \phi\rangle) \mathbin{\hat{}} \mathcal{REP}[\![EH]\!]^{\rho}_{\Delta} z\pi\xi$$

which has the solution

$$\mathcal{REP}[\![EH]\!]^{\rho}_{\Delta} z\pi\xi = \mathcal{STOP}[\![EH]\!]^{\rho}_{\Delta}\pi\xi \otimes \langle(\emptyset, z, \pi), \phi\rangle$$

We take this as the intended result of the $\mathcal{REP}$ function.

The $\mathcal{EH}$ function recurs through Or patterns. For Between patterns, it uses $\mathcal{H}$ to match the pattern and attaches the repetition label (R) to the result. The $\mathcal{STOP}$ function simply checks whether the accumulated repetitions satisfy the repetition constraints. It encodes constraint satisfaction as a single success and constraint violation as an failure referring to the repetition constraint label.

$$
\begin{array}{l}
\mathcal{REP}[\![EH]\!]^{\rho}_{\Delta} z\pi\xi \\
\quad = \quad (\mathcal{EH}[\![EH]\!]^{\rho}_{\Delta} z\pi \blacktriangleright \lambda(\sigma, z', \pi', R).\ \mathrm{zipcons}(\sigma, \mathcal{REP}[\![EH]\!]^{\rho}_{\Delta} z'\pi'\xi')) \\
\qquad \hat{}\ (\mathcal{STOP}[\![EH]\!]^{\rho}_{\Delta} \pi\xi \otimes \mathrm{unit}(\emptyset, z, \pi)) \\
\qquad\quad \text{where } \xi' = \xi[R \mapsto \xi(R) + 1] \\
\\
\mathcal{EH}[\![\mathsf{Or}(EH_1, EH_2)]\!]^{\rho}_{\Delta} z\pi \\
\quad = \quad \mathcal{EH}[\![EH_1]\!]^{\rho}_{\Delta} z\pi \ \hat{}\ \mathcal{EH}[\![EH_2]\!]^{\rho}_{\Delta} z\pi \\
\mathcal{EH}[\![\mathsf{Between}(N_{min}, N_{max}, H)^{\ell_1,\ell_2}_{R}]\!]^{\rho}_{\Delta} z\pi \\
\quad = \quad \mathcal{H}[\![H]\!]^{\rho}_{\Delta} z\pi \blacktriangleright \lambda(\sigma, z', \pi').\ \mathrm{unit}(\sigma, z', \pi', R) \\
\\
\mathcal{STOP}[\![\mathsf{Or}(EH_1, EH_2)]\!]^{\rho}_{\Delta} \pi\xi \\
\quad = \quad \mathcal{STOP}[\![EH_1]\!]^{\rho}_{\Delta} \otimes \mathcal{STOP}[\![EH_2]\!]^{\rho}_{\Delta} \\
\mathcal{STOP}[\![\mathsf{Between}(N_{min}, N_{max}, H)^{\ell_1,\ell_2}_{R}]\!]^{\rho}_{\Delta} \pi\xi \\
\qquad\quad \begin{cases} \mathrm{unit}(\emptyset) & \text{when } N_{min} \le \xi(R) \le N_{max} \\ \langle\ ; \textsc{Fail}(\pi, \{\mathrm{eval}(\ell_1, \rho)\})\rangle & \text{when } \xi(R) < N_{min} \\ \langle\ ; \textsc{Fail}(\pi, \{\mathrm{eval}(\ell_2, \rho)\})\rangle & \text{when } \xi(R) > N_{max} \end{cases} \\
\xi_0(R) \quad = \quad 0
\end{array}
$$

Figure 7.7: Semantics of EH-patterns

7.3 Implementation

We have implemented `syntax-parse` and syntax classes in PLT Scheme using the pre-existing `syntax-case` macro system. Our implementation interoperates with the existing macro infrastructure. The attributes bound by `syntax-parse` are implemented as `syntax-case`-compatible syntax variables, so we can simply reuse use the existing syntax template system. Consequently, macro writers can easily migrate from `syntax-case`.

7.3.1 Pattern compilation

Syntax patterns are compiled to implement a backtracking automaton parsing algorithm. Our initial work on this system was done using the classical pattern matrix compilation algorithm [2, 26]. We soon found, however, that we were paying the costs of a complicated implementation strategy without reaping its benefits. The matrix approach impeded our exploration of expressive pattern forms and made it difficult to understand progress tracking. At the same time, few of our new pattern variants can be combined to reduce backtracking, the primary benefit of the matrix-based compilation scheme.

Our current implementation uses a simpler compilation strategy similar to the two-continuation implementation of backtracking [72]. We tackle a single pattern and continuation expression at a time; the failure continuation is an implicit parameter. There is a separate compilation scheme for

single-term patterns, head patterns, action patterns, and ellipsis-head patterns. These compilation schemes are implemented as macros.

`(parse-S` S x π k`)` compiles single-term patterns. The parameters have the following meanings: S is the pattern to match, x is a variable holding the syntax object to be matched, π holds the current progress representation, and k is the (success) continuation expression. The `parse-S` form is responsible for binding all of S's attributes in k.

`(parse-A` A π k`)` compiles action patterns. The parameters have similar meanings to those of `parse-S`, except that the syntax object argument is not present, and the pattern argument is an action pattern.

`(parse-H` H x π x_{rest} π_{rest} k`)` compiles head patterns. The new parameters x_{rest} and π_{rest} are additional binding obligations; `parse-H` is responsible for binding them to the remainder of the syntax object and the progress string after the head is matched, respectively.

`(parse-EH` EH x π R x_{rest} π_{rest} k`)` compiles ellipsis-head *disjuncts*; `~or`-patterns are flattened and handled by an auxiliary to the `parse-S` macro. The new parameter rep is a variable that holds the number of previous repetitions of the pattern that have been matched.

In general, it is the obligation of each step of a compilation scheme to ensure that all of the current pattern's attributes are bound in the continuation expression. Some patterns bind attributes directly via the `let-attrs` form, which uses low-level macro machinery to create syntax variables. For example, here is the compilation of a simple pattern variable:

```
(parse-S a x π k)
⇒
(let-attrs ([(a 0) x]) k)
```

The pattern has a single attribute, `a` with depth 0. In the result of the translation, k occurs within an attribute binding for `a`, so the obligation is satisfied. In other cases, the obligation is filled by deferring to other translation steps. For example,

```
(parse-S (~and S1 S2) x π k)
⇒
(parse-S S1 x π (parse-S S2 x π k))
```

The attributes bound by `(~and` S_1 S_2`)` are the union of the attributes of S_1 and those of S_2, which must be disjoint. Since k occurs as the continuation expression of parsing S_2, which itself occurs as the continuation expression of parsing S_1, the obligation is satisfied by induction.

To prevent code duplication, each compilation step must use its continuation expression exactly once in its output. When multiple alternatives share

a continuation, a separate procedure is created to wrap the continuation expression, and the alternatives are passed invocations of the new procedure:

```
(parse-S (~or S1 S2) x π k)
⇒
(let ([continue (lambda-attrs (attr ···) k)])
  (try (parse-S x π S1 (continue (attribute attr) ···))
       (parse-S x π S2 (continue (attribute attr) ···))))
```

where $\{attr \cdots\} = \text{attrs}[\![(\texttt{~or}\ S_1\ S_2)]\!]$.

All compilation schemes are implicitly parameterized over two bindings, `fail` and `fail*`, which refer to the enclosing failure handler and the failure handler of the nearest enclosing `~describe` form, respectively. These bindings are implemented via PLT Scheme's *syntax parameters*, special macros whose meanings depend on the expansion context. We write `with` as shorthand for binding a temporary variable to an expression and updating the syntax parameter to expand into a reference to the variable.

Choice points are implemented by the `try` form, which locally updates `fail` in the first subexpression to try the second, and if both fail, join the failures together and invoke the original failure handler:

```
(try expr1 expr2)
⇒
(with ([fail (lambda (f1)
               (with ([fail (lambda (f2)
                              (fail (join-failures f1 f2)))])
                 expr2))])
  expr1)
```

7.3.2 Syntax class compilation

A syntax class name is bound at compile time to a record containing the names of its attributes and the name of its parser function.

First, the side clauses in each variant are eliminated by transforming them into action patterns and absorbing them into the variant's main pattern via `~and`. Second, a result expression is generated that returns a list of the class's exported attributes. The same result expression is used for each variant.

The variants are tried in order, in the context of `fail` and `fail*` bindings that return a failure, wrapped with the syntax class's description, instead of raising it as an error. The parser returns a list on success; any other value is a failure representation. Figure 7.8 shows an example of syntax class compilation.

Splicing syntax classes are compiled similarly, except that their result list includes the progress and suffix after matching.

```
(define-syntax-class (widget p q r)
  #:attributes (a b)
  #:description the-description
  (pattern P1)
  (pattern P2))
⇒
(define (parse-widget stx p q r)
  (define pr (empty-progress))
  (define (fail-k failure)
    (expected the-description failure))
  (with ([fail fail-k]
         [fail* fail-k])
    (try (parse-P stx pr P1 (list (attribute a) (attribute b)))
         (parse-P stx pr P2 (list (attribute a) (attribute b))))))
(define-syntax widget
  (make-syntax-class —— #'parse-widget ——))
```

Figure 7.8: Syntax class compilation

CHAPTER 8

syntax-parse in Practice

We have implemented our new macro system in PLT Scheme, and it has been available since August 2009. Experience confirms that `syntax-parse` make it easier for macro writers to implement macros with complex syntax. For example, it is now feasible for casual macro writers to write macros with multiple keyword options. For macros with complicated syntax, our system can cut parsing code by several factors up to an order of magnitude. The primary benefit, however, is increased robustness.

This chapter presents a series of case studies illustrating a practical application of `syntax-parse`. Each case study starts with a general discussion of the macros involved, their purpose, and their history. Next is a discussion of the changes made in the port to `syntax-parse` and a comparison of the macros before the port to the macros after. We compare what syntax errors are detected and how they are specified. We also point out drawbacks and opportunities for improvement uncovered by the port, as well as useful features and new techniques using `syntax-parse`.

8.1 Case study: Interface-oriented class macros

The first example is the `class-iop` library, which consists of a set of macros to enforce interface-oriented programming with PLT Scheme's class system. In PLT Scheme, classes and interfaces are first class values. An interface is essentially a catalog of method names. Classes are checked to ensure that they define all methods in the interfaces they declare. Method calls, however, do not involve any interface checks. A method invocation is written using the `send` form:

```
(send object-expr method arg-expr ...)
```

Interface definitions tend to fall out of sync, since new methods can simply be added to classes and used by clients.

The `class-iop` library provides a compile-time notion of interfaces and a variant of the method call syntax that checks the association of methods with interfaces. The checks occur at compile time, and so by using only checked

Normal `class` code

```
(define stack<%>
  (interface ()
    (empty? push pop)))

(define s%
  (class* object% (stack<%>)
    (define items null)
    (define/public (empty?)
      (null? items))
    (define/public (push item)
      (set! items (cons item items)))
    (define/public (pop)
      (begin0 (car items)
              (set! items (cdr items))))
    (define/public (add-all s)
      (unless (send s empty?)
        (push (send s pop))
        (add-all s)))
    (super-new)))

(define s1 (new s%))
(define s2 (new s%))

(send s1 push 'a)
(send s1 push 'b)
(send s2 add-all s1)
```

`class-iop` code

```
(define-interface stack<%> ()
  (empty? push pop))

(define s%
  (class* object% (stack<%>)
    (define items null)
    (define/public (empty?)
      (null? items))
    (define/public (push item)
      (set! items (cons item items)))
    (define/public (pop)
      (begin0 (car items)
              (set! items (cdr items))))
    (define/public (add-all s)
      (unless (send/i s stack<%> empty?)
        (push (send/i s stack<%> pop))
        (add-all s)))
    (super-new)))

(define s1 (new s%))
(define s2 (new s%))

(send/i s1 stack<%> push 'a)
(send/i s1 stack<%> push 'b)
(send/i s2 stack<%> add-all s1)
;; error: method not in static
;; interface in: add-all
```

Lines that differ are set in *slanted* type.

Figure 8.1: Example of `class-iop`

method calls, the programmer can prevent the program from diverging from the organization specified by the interfaces. The library shows one way in which rich macro systems can be used to establish static properties of programs even in the absence of a built-in type system.

The `define-interface` defines a name as a *static interface*, in contrast to ordinary interfaces, which are first-class values. An interface definition is written thus:

```
(define-interface interface (super-interface ...)
  (method ...))
```

The checked method call form, `send/i`, requires a static interface name after the object expression. It has the following form:

```
(send/i object-expr interface method arg-expr ...)
```

If the given method is not a member of the static interface, `send/i` raises a compile-time error. The form must also check that the object is an instance

of the interface; that check, however, is typically done at run time.[1]

Figure 8.1 shows two versions of the same program, one using the normal class system and the other using checked method calls. The program defines an interface `stack<%>` for mutable stacks and a class `stack%` implementing that interface.[2] The program then creates two instances of stacks, `s1` and `s2`, and calls their methods. The left-hand program uses the normal method call form, `send`, and the right-hand program uses the checked form, `send/i`. The final method call in the right-hand program raises an error, since `add-all` is not declared in the interface `stack<%>`.

The `class-iop` library, implemented with `syntax-case`, was released first on PLaneT, then moved to the core distribution to support the macro stepper's GUI—which until then had often suffered from out-of-sync interface definitions.

8.1.1 Comparison

The library defines two main sets of macros:

- The interface definition forms, `define-interface` and an auxiliary macro called `define-interface/dynamic`, which creates a static interface given method names and an interface value.

- The checked method forms, `send/i`, `send*/i`, and `send/appy/i`, as well as the private auxiliary macro `check-method`.

Interface definition forms The original version of the `define-interface` macro performs no validation checks. It produces a use of the auxiliary macro `define-interface/dynamic`, which uses guards, so a faulty use of `define-interface` can produce a generic error message referring to an auxiliary macro.

The `syntax-parse` version uses pattern variable annotations to check that the name to be defined is an identifier, the super-interfaces are identifiers bound as static interfaces, and the method names are identifiers:

```
(define-interface name:id (super:static-interface ...)
  (method:method-entry ...))
```

The `id` syntax class is familiar; the others are discussed later. Generic errors are still given in some cases, such as if the methods are not parenthesized. They can be made specific by adding descriptions to these patterns.

[1] The library also offers checked binding forms such as `define/i` that do the object-interface check at binding time so it can be skipped on method calls.

[2] PLT Scheme convention dictates that interface names end with a `<%>` suffix and class names end with a `%` suffix.

Ideas for `syntax-parse` One kind of misuse of `define-interface` that leads to a suboptimal error is omitting the super-interfaces entirely. In that case, the method names are interpreted as interface names. For example, given this use:

```
(define-interface stack<%> (push pop empty?))
```

the error says that `push` is not a static interface, whereas an omniscient validation system would indicate that the super-interface list was missing. This example illustrates a limitation in our strict left-to-right parsing and error-reporting algorithm.

Programmers can modify the parsing order by specifying it explicitly via patterns. One approach is to write an `~and` pattern that checks the number of forms first and then parses each one. Another approach is to to wrap the super-interface list pattern in an `~optional` pattern and then fail if it is absent:

```
(~and (define-interface name:id
        (~optional (super:static-interface ...))
        (method:method-entry ...))
      (~fail #:unless (attribute super)
             "missing super-interfaces list"))
```

Both of these approaches, however, involve cluttering the macro's specification with operational parsing hints. In the future such devices should be incorporated into the language as declarative specifications.

Checked method call forms The original versions of the checked method call forms use guards to check that the interface and method are identifiers. The helper macro `check-method` further checks that the interface argument is a static interface that contains the method. Consequently, there are two possible errors for passing a bad interface argument, a generic error if it is not an identifier and a specific error if it is an identifier but not bound as a static interface. The `syntax-parse` variant defines a syntax class for identifiers bound as static interfaces. Here is its definition:

```
(define-syntax-class static-interface
  (pattern (~var x (static static-interface?
                           "static interface"))
           #:attr value (attribute x.value)))
```

It uses the built-in `static` syntax class, which is parameterized over a predicate and a description. The attribute `value` is bound to the static interface record itself, which contains the method names and a reference to a variable bound to the dynamic interface value.

Figure 8.2 compares the original implementation of `send/i` with the new version. Note the original `check-method`; it uses `syntax-local-value` to

original version:

```
(define-syntax (send/i stx)
  (syntax-case stx ()
    [(send/i obj iface method . args)
     (and (identifier? #'iface) (identifier? #'method))
     #`(begin (check-method method iface)
              (send (check-object:interface send: obj iface)
                    method . args))]))

(define-syntax (check-method stx)
  (syntax-case stx ()
    [(sci method iface)
     (let ([si (syntax-local-value #'iface (lambda () #f))])
       (unless (static-interface? si)
         (raise-syntax-error 'checked-send
           "expected static interface" #'iface))
       (unless (member (syntax-e #'method)
                       (static-interface-members si))
         (raise-syntax-error 'checked-send
           "method not in static interface" #'method))
       #''okay)]))
```

syntax-parse version:

```
(define-syntax (send/i stx)
  (syntax-parse stx
    [(send/i obj:expr iface:static-interface method:id . args)
     (begin
      (check-method #'send/i #'method (attribute iface.value))
      #'(send (check-object<:interface send: obj iface)
              method . args))]))

(define-for-syntax (check-method for-whom method si)
  (unless (member (syntax-e method) (static-interface-members si))
    (raise-syntax-error (syntax-e for-whom)
      "method not in static interface" method))))
```

Figure 8.2: Comparison of checked method calls

look up the binding of the interface name, then it checks that it is actually a static interface record. The `static` syntax class provides a declarative interface to the same task. The new version of `check-method`, an auxiliary procedure rather than a macro, checks that the method is in the interface. The check could be expressed as a `#:fail-unless` side condition in the macro, but since there are three checked method call forms, it is more convenient to make it a common procedure.

Extensible method entries The new version of `define-interface` supports extensible method entry syntax, which is useful for libraries that define sets of methods following some convention.

For example, there is a library for creating mutable cells that allow listener registration. The listeners of a cell are called when the cell's contents change. A common pattern is for one "configuration" class to contain many such cells as fields. To simplify access to the cells, the library provides a macro that defines methods of the form `get-`X, `set-`X, and `listen-`X. Rather than write all three method names for each field—and there are classes that contain many of these fields—we can use a custom *interface macro* to produce the method names. Figure 8.3 shows the implementation of interface macros and the example `methods:notify` macro.

The `interface-macro` syntax class recognizes identifiers statically bound to `iface-macro` values, which are structures containing just a transformer procedure. The `method-entry` syntax class defines an attribute `methods` containing the fully-elaborated list of method names for that entry. The base case is an identifier, a simple method name. The alternative is a use of an interface macro; in that case, the transformer procedure is extracted from the structure, conveniently bound to the `macro.value` attribute. It is applied to the method entry to get a new list of method entries, and those are recursively elaborated. This custom macro expander is simple—for example, it is not fully hygienic—but a full-fledged expander could be built the same way.

8.1.2 Summary

With `syntax-parse`, macro writers can easily add error checking to macros originally written with little or incomplete checking, although truly comprehensive error catching requires verbose annotations.

The built-in `static` syntax class provides a concise declarative layer over the `syntax-local-value` procedure, making macros that share compile-time information easier to write. It also supports syntax class extensibility via lightweight custom macro expanders.

8.2 Case study: units

PLT Scheme's units [33]—first-class, dynamically linkable components—are implemented via a prime example of conscientiously-written macros. The

```
(define-struct iface-macro (proc))

(define-syntax-class interface-macro
  (pattern x
           #:declare x (static iface-macro? "interface macro")
           #:attr value (attribute x.value)))

(define (iface-expand macro stx)
  (let ([transformer (iface-macro-proc macro)])
    (transformer stx)))

(define-syntax-class method-entry
  (pattern m:id
           #:with (method ...) #'(m))
  (pattern (macro:interface-macro . args)
           #:with (method ...)
                  (apply syntax-list-append
                   (map (lambda (m)
                          (syntax-parse m
                            [m:method-entry #'(m.method ...)]))
                        (syntax->list
                         (iface-expand (attribute macro.value)
                                       #'(macro . args)))))))

;; Interface macro example

(define-syntax methods:notify
  (make-iface-macro
    (lambda (stx)
      (syntax-parse stx
        [(methods:notify name)
         (list (symbol-append 'get- #'name)
               (symbol-append 'set- #'name)
               (symbol-append 'listen- #'name))]))))
```

Figure 8.3: Extensible method entries

unit system was first implemented as a core part of the interpreter, then reimplemented as a library using macros. Since then, it has undergone several rewrites to support new features, most recently macros in signatures [17] and contracts [67]. The unit system is second only to the class system in size and complexity, and the needs of both systems have driven the evolution of the macro system.

Most macros of the unit system use `syntax-case`. Extending and modifying these macros—such as the recent addition of macro definitions to unit interfaces [17]—demands great care and often requires a polishing pass by the original implementor [29].

The implementation of a contract system for units uses `syntax-parse` for the separation of contract specifications from ordinary unit clauses. The latter are still sent to the old parsing procedures for additional validation.

8.2.1 Comparison

Figure 8.4 shows the syntax of the `unit` form; the new `unit/c` form, which creates a unit contract; and the `define-unit/contract` form, which optionally imposes contracts on `import` and `export` clauses. The text of the figure is taken directly from the documentation.

The unit library contains two versions of the code for parsing and validating the `unit` forms: `sig-id`, `tagged-sig-id`, `sig-spec`, `tagged-sig-spec`, and `init-depends-decl`. The first is used by the `unit` macro; it is written in `syntax-case` with explicit error checking. The second is used by the contract forms, since their syntax builds upon the basic `unit` syntax; it is written using syntax classes.

The `sig-id` syntax The original code only checks that it is an identifier, raising the error "not an identifier" otherwise. The `unit` macro checks that the identifier is bound as a signature later, in a different section of code. The new code defines a syntax class for `sig-id` that uses `static` to do the signature check immediately. Unlike the `class-iop` example, however, the signature is not provided as an attribute; it is looked up again later by the existing unit infrastructure.

The `tagged-sig-id` syntax In the original code, `check-tagged-id` is implemented by combining the higher-order `check-tagged` procedure with `check-id`. The `check-tagged` procedure contains two error messages:

- `tag must be a symbol`
- `expected (tag <identifier> <syntax>)`

The terminology, symbol versus identifier, is inconsistent, and the reference to "syntax" in the second message is vague. Creating syntax-checking abstractions is complicated; the `check-tagged` procedure should take another

```
(unit
  (import tagged-sig-spec ...)
  (export tagged-sig-spec ...)
  init-depends-decl
  unit-body-expr-or-defn
  ...)

tagged-sig-spec = sig-spec
                | (tag id sig-spec)

sig-spec = sig-id
         | (prefix id sig-spec)
         | (rename sig-spec (id id) ...)
         | (only sig-spec id ...)
         | (except sig-spec id ...)

init-depends-decl =
                  | (init-depend tagged-sig-id ...)

tagged-sig-id = sig-id
              | (tag id sig-id)


(unit/c (import sig-block ...) (export sig-block ...))

sig-block = (tagged-sig-id [id contract] ...)
          | tagged-sig-id

(define-unit/contract unit-id
  (import sig-spec-block ...)
  (export sig-spec-block ...)
  init-depends-decl
  unit-body-expr-or-defn
  ...)

sig-spec-block = (tagged-sig-spec [id contract] ...)
               | tagged-sig-spec
```

Figure 8.4: Unit syntax

argument describing the untagged syntax. In our system, syntax classes encompass both parsing and description. On the other hand, syntax classes cannot currently be parameterized over other syntax classes.[3] The abstraction could of course be done by a macro.

The `sig-spec` syntax Two forms, `only` and `except`, are only allowed in import clauses, not export clauses. The `check-spec-syntax` procedure takes an argument that tells whether it should accept import-only keywords. The syntax class does not distinguish between import and export clauses. Neither does the grammar presented in the documentation; the restrictions are mentioned later in the descriptions of the forms. In the current arrangement, the syntax class can afford to be liberal in always accepting import clauses, since they are rejected by the second round of validation.

The `check-spec-syntax` procedure has nine clauses, five of which are dedicated exclusively to errors. The error clauses deal with mistakes such as missing arguments and extra arguments and give detailed error messages. For example, a `prefix` form with no arguments results in the error "missing prefix identifier and import spec." In contrast, `syntax-parse` does not identify the missing forms; it merely says the `prefix` form is a bad `sig-spec`. The `check-spec-syntax` procedure also dedicates clauses to detecting too many arguments, but `syntax-parse` handles these well already.

Syntax list checking Many of the parsing procedures in the original code use a helper, `checked-syntax->list`, that detects improper list structure and raises the error "bad syntax (illegal use of '.')." This error is unrelated to the particular syntax being matched, whereas `syntax-parse` parses one pattern at a time without looking ahead.

This check can be simulated via an `~and` pattern that checks the list structure, but this is verbose. The `syntax-parse` pattern language would benefit from better support for catching improper list syntax early.

8.2.2 Summary

A syntax class definition does not naturally replicate every error that a conscientious macro writer can write. On the other hand, as Section 6.3.4 explains, the design of `syntax-parse` is based on the decision that instead of spending effort anticipating and explaining every possible deviation from correct syntax, a macro should simply determine the problematic subform and report its expectation. Syntax classes and descriptions usually suffice to generate that level of error reporting.

[3] We plan to support this feature soon.

8.3 Case study: parser

The final case study is the `parser` macro from the `parser-tools` library [54]. This macro takes a description of a grammar and produces a parser (or a set of parsers, if multiple start symbols are given). The `parser` macro cooperates with other macros that define tokens and lexers using regular expressions, but this case study focuses on just the `parser` macro.

The `parser` macro, due to Scott Owens, uses `syntax-case`. Other developers have made minor changes and contributions.

The original version of the `parser` macro parses and validates its syntax in two stages. The first stage performs shallow syntactic checks, and the second stage checks more complicated constraints, such as all production items must be terminals or nonterminals. The first level of checking is performed in the macro itself. The second level of checking is scattered throughout the auxiliary procedures used by the macro to construct the parser. The final level, which checks that the grammar is in fact LALR(1), is beyond the scope of our system.

This case study consists of two porting steps. The first covers only the shallow syntactic checks. The second incorporates the deeper, interdependent checks.

8.3.1 First stage

Figure 8.5 shows the documented syntax of `parser`. It contains the following clauses:

- `grammar` (mandatory) defines the nonterminals via productions.
- `tokens` (mandatory) imports terminal names via *token groups*, names statically bound to sets of token names.
- `start` (mandatory) lists the start symbols.
- `end` (mandatory) lists the tokens that end parsing.
- `error` (mandatory) determines the procedure called on a parse error.
- `precs` (optional) specifies the precedences of terminals.
- There are several other optional clauses.

The original version of the `parser` macro includes a large loop that processed clauses one by one. The loop records information from a clause by mutating local variables. The mutable variables are also used to detect duplicate and missing mandatory clauses. The two checks occur in different parts of the loop. Order of checking is inconsistent; for some clauses the

```
(parser clause ...)

clause = (grammar (non-terminal-id
                   ((grammar-id ...) maybe-prec expr)
                   ...)
                  ...)
       | (tokens group-id ...)
       | (start non-terminal-id ...)
       | (end token-id ...)
       | (error expr)
       | (precs (assoc token-id ...) ...)
       | (src-pos)
       | (suppress)
       | (debug filename)
       | (yacc-output filename)

maybe-prec =
           | (prec token-id)

assoc = left
      | right
      | nonassoc
```

Figure 8.5: Parser syntax

clause is validated before checked for duplication, but in others the duplicate checking occurs first. Some of the optional clauses, like `suppress`, permit duplicates. The macro defers validation of some clauses, including the `grammar` clause. Finally, the `parser` macro has a catch-all clause for nonsense clauses that raises an error enumerating the possible clause forms, but a few are left out.

The first stage of the port defines syntax classes describing each clause. The syntax classes correspond almost exactly to the syntax of Figure 8.5, except that the `grammar` clause has several auxiliary syntax classes, including `ntdef` and `production`.

The new macro's main pattern consists of an ellipsis pattern with one disjunct per clause. Each disjunct has a repetition constraint, either `~once`, for the mandatory clauses, or `~optional`. The repetition constraints handle both duplicate and missing clauses, using the `#:name` to generate the error messages. For example, a missing grammar clause produces the error message "missing grammar clause."

The original macro took 127 lines of code to parse and validate the clauses. After the first stage of the port, parsing is handled by 54 lines of code including 13 syntax class definitions. Some of the syntax classes per-

```
(parser (~or (~once g:grammar-clause #:name "grammar clause")
             (~once t:tokens-clause #:name "tokens clause")
             (~once s:start-clause #:name "start clause")
             ---
             (~optional p:precs-clause
                        #:name "precedence clause")
             --- )
        ...)
```

Figure 8.6: Main parser pattern

form more rigorous checking than the original macro's parsing loop. For example, the `grammar-clause` syntax class checks the well-formedness of productions, whereas the original macro performs that check later. The ported macro still calls the same auxiliary functions for interpreting the clauses and generating the parser table, so some syntax validation is now performed twice.

8.3.2 Second stage

The purpose of the second stage of porting was to rewrite the `parse` macro's auxiliary functions to move checks to syntax classes and to create the macro's intermediate data structures in the syntax classes. The clauses, however, cannot be parsed in a single pass. The `end` and `precs` clauses depend on the `tokens` clause; the `grammar` clause depends on both `tokens` and `end`; and the `start` clause depends on the `grammar` clause. But these clauses may occur *in any order*.

The solution is to parse the clauses in two passes. The first pass recognizes just the `grammar`, `tokens`, and `end` clauses with custom syntax classes that extract only the information needed by the second pass, when all of the clauses are fully validated. Figure 8.7 shows the main macro pattern and the pass-one version of the `grammar-clause` syntax class. The syntax class provides an attribute, `nts`, that contains a table of the nonterminal names. The pass-one versions of the `token-clause` and `end-clause` classes provide similar attributes.

In the second pass, represented by the second conjunct of the main `~and` pattern, the syntax classes are given these tables to satisfy their dependencies. Figure 8.8 shows the definition of the pass-two syntax class for the `grammar` clause. Each production (`ntprod`) checks that the defined nonterminal is not already defined as a token, and each item in the right-hand side must be either a nonterminal or a non-end terminal.

Note the relative verbosity of the repeated parameters and the use of the `#:declare` directive to avoid writing the parameters in the pattern via `~var`

```
(~and
 (_ (~or (~once pg:p1-grammar-clause #:name "grammar clause")
         (~once pt:p1-token-clause #:name "token clause")
         (~once pe:p1-end-clause #:name "end tokens clause")
         _)
    ...)
 (_ (~or (~once (~var g (grammar-clause (attribute pg.nts)
                                        (attribute pt.ts)
                                        (attribute pe.ends)))
                #:name "grammar clause")
         (~once _:token-clause #:name "token clause")
         (~once (~var s (start-clause (attribute pg.nts)))
                #:name "start symbols clause")
         (~once (~var e (end-clause (attribute pt.ts)))
                #:name "end tokens clause")
         ___)
    ...))

(define-syntax-class p1-grammar-clause
  (pattern ((~datum grammar) [nt:id . _] ...)
           #:fail-when (check-duplicate (syntax->list #'(nt ...))
                                        #:key syntax->datum)
                       "duplicate nonterminal definition"
           #:attr nts (let ([table (make-hasheq)])
                        (for ([nt (syntax->datum #'(nt ...))])
                          (hash-set! table nt #t))
                        table)))
```

Figure 8.7: Two-pass parsing `parser` clauses

patterns. Other grammar systems [75] have introduced auto-copy inherited attributes to make this pattern of parameterization more concise.

Attribute grammars inspire a tantalizing possibility, which is to eliminate the explicit separation of passes entirely and automate the flow of attributes between the necessary clauses. It is unclear whether we can extend our error-selection algorithm to handle new attribute-solving orders.

The original version of `parser` explicitly detects 39 different syntax errors. A full third of these (13) are handled by repetition constraints in the new version. Slightly fewer (11) are handled by annotating pattern variables; this set of errors includes the simple, such as

```
Debugging filename must be a string
```

and the complex and context-dependent, such as

```
Start symbol not defined as a non-terminal
```

```
(define-syntax-class (grammar-clause nts ts ends)
  (pattern ((~datum grammar) prod ...)
           #:declare prod (ntprod nts ts ends)))

(define-syntax-class (ntprod nts ts ends)
  #:attributes (nt [i 2])
  (pattern (nt:id ((i ...) prec rhs:expr) ...)
           #:declare i (item nts ts ends)
           #:declare prec (maybe-prec ts)
           #:fail-when (and (hash-ref ts (syntax-e #'nt) #f) #'nt)
                       "already declared as a terminal"))

(define-syntax-class (item nts ts ends)
  #:description "terminal or nonterminal name"
  (pattern i:id
           #:when (or (hash-ref nts (syntax-e #'i) #f)
                      (hash-ref ts (syntax-e #'i) #f))
           #:fail-when (hash-ref ends (syntax-e #'i) #f)
                       "end token cannot be used in a production"))
```

Figure 8.8: Second-pass syntax class for `grammar` clause

The latter kind of error is handled by a syntax class parameterized over the declared nonterminals. Eight errors are handled by explicit `#:fail-when` or `#:fail-unless` checks, such as the errors in 8.8. In most cases the error messages are rephrased according to `syntax-parse` conventions.

Seven of the thirty-nine errors reported by the original macro have no counterpart in the new version. Every one of these dropped errors is a local catch-all that explains what valid syntax looks like for the given clause or subform. For example:

```
parser must have form (parser args ...)
```

In a few messages the explanation is incorrect; programmers who revised the `parser` macro failed to update the error message. Most of these catch-all errors could be simulated via syntax class descriptions. As we argued in Section 6.3.4, however, such errors are misguided. A better strategy is to identify the faulty syntax in enough detail that the programmer can find the proper syntax in the documentation. I hope to integrate future versions of `syntax-parse` with the PLT documentation system [32]. Syntax class descriptions should link to documentation, not recapitulate it.

The original version devoted 570 lines to parsing and processing, counting the macro and its auxiliary functions. The line count leaves out separate modules such as the one that implements the LALR algorithm. In the original code, parsing and processing are tightly intertwined, and it is impossible

to directly count the lines of code dedicated to each. After the second stage of porting, the total lines of code for parsing and processing was 376, of which 23 were the macro's main pattern and 99 were syntax class definitions, leaving 254 lines for processing. By reasoning that the lines dedicated to processing should be roughly equivalent in both versions, we can estimate 300 lines for processing in the original version, leaving 270 for parsing. This is roughly twice the number of lines for parsing in the new version (122), and the new parsing code consists of modular, declarative specifications.

8.3.3 Summary

Our system can express a wide range of errors that must be checked explicitly in `syntax-case` macros. Using `~and` patterns a macro can perform parsing in multiple passes, although our system needs a more concise mechanism for parameterized syntax classes.

CHAPTER 9

Related Work

Scheme macros are defined by two intertwined lines of research on hygienic macro expansion and on how to express macro transformations.

Hygienic macro expansion was invented by Kohlbecker et al. [46] as a solution to the problem of inadvertent variable capture in Lisp macros, which essentially operated on trees of symbols. Likening macro rewriting steps to substitution in the λ-calculus, they devised an expansion algorithm that effectively performed α-conversion concurrently with expansion. Shortly thereafter, Kohlbecker and Wand [47] devised a system for writing macro transformations in a high-level language rather than direct manipulation of S-expression trees. In their system, called Macro-By-Example, macros are specified as a series of clauses, where each clause consists of a pattern and a template. MBE was independent of hygiene; it could be used in a hygienic or nonhygienic system. In fact, it was developed before hygienic macro expansion but published after.

Clinger and Rees [14] brought the two ideas together in a system (MTW) that used the knowledge of the template's contents to avoid the repeated, expensive traversal of terms performed by the original hygiene algorithm. MTW also incorporated techniques developed for syntactic closures [7], another system for implementing macros that respect lexical scoping. Dybvig et al.[20] devised a different hygiene algorithm based on an algebraic formulation of lazy α-conversion. Their algorithm was accompanied by a pattern-matching form called `syntax-case` that permitted MBE-style templates to occur within expressions; `syntax-case` combined the convenience of MBE's pattern matching and templating with the power of the full Scheme language.

Allowing macro transformers to be implemented using arbitrary Scheme expressions enables powerful macros, but it makes compiling such macros difficult. The compiler must execute parts of the program it is compiling, but it must only execute the correct parts. Queinnec [57] described such macros as a tower of languages; the implementation of each language must depend only on higher levels in the tower. Furthermore, all but a finite number of the tower's levels must be trivial. The tower principle was codified by Flatt [30] as a module system that allowed modules to import and export

macros. In Flatt's system, modules use a distinct import form to indicate that a dependency must be executed for the module to be compiled. Together, the scoping and module instantiation rules ensure that a program's meaning is independent of the vagaries of compilation order.

Several other language communities have recognized the value of syntactic abstraction and extensible languages and have started to explore their use. Researchers have crafted various macro-like systems for languages with conventional C-style syntax [3, 4, 73]. None of the systems achieves the expressive power of Scheme macros. The MacroML system by Ganz et al. is a type-safe macro system based on MetaML. In fact, their type discipline guarantees that ML-style macros generate proper, well-typed ML code. MacroML achieves soundness, however, by severely restricting the expressive power of the macro system, offering only a few prefabricated syntaxes, so its macros do not deal with the challenges of parsing. Nor do compile-time metaprogramming systems such as Template Haskell [61], which do not define syntax extensions at all.

Other languages offer extensibility via external pre-processing tools. For example, Camlp4 [18] is a pre-processing tool for OCaml [49] that lets programmers extend OCaml with new languages via grammars and a quasiquotation syntax. MetaBorg [9] is a technique using Stratego/XT to define the syntax and "assimilation" rules for language extensions. Assimilation rules play a role similar to macros; they rewrite derived syntax into the core syntax of the host system. Macros can be added to a system by assimilating the language extension tools themselves [58, 59]. The result is a syntax definition and rewriting system available within the language itself.

9.1 Semantics of macro expansion

Our semantics of macro expansion is an adaptation from the presentation of the `syntax-case` macro system by Dybvig et al. [20]. We take their syntax algebra with only minor modifications for clarity. Their presentation models expansion using recursive functions whose result is an AST, not a term. In contrast, we model expansion as relating terms to terms. The models based solely on terms provide a better basis for constructing and verifying the macro stepper.

Our big-step semantics is similar to the "Macros That Work" (MTW) big-step semantics of Clinger and Rees [14]. We use a similar environment structure, although in our model the environment's domain is an auxiliary set of symbols and "addresses" rather than identifiers. The MTW environment contains mappings for all fresh identifiers generated by macro transcription steps, whereas our environment contains mappings only for binding occurrences variables; fresh identifiers are resolved to binding occurrences using the Dybvig et el. syntax algebra instead of the environment.

Another line of work in semantics for Scheme-like macros consists of the

work of Bove and Arbilla [8] and the work of Gasbichler [39]. Both focus on exploring hygiene and template rewriting via small-step semantics based on explicit substitution. Both model a different hygiene mechanism from the one our system uses, and so neither one would make a good basis for the macro stepper.

9.2 Specifying syntax

Another line of research in macro specifications began with static checking of syntactic structure [16] and evolved to encompass binding information and hygienic expansion [40]. These systems, however, are incapable of verifying a broad range of widely used macro programming idioms, and they do not address the issues of error feedback or of modular syntax specification addressed by our system.

The Fortress language [1] provides a syntax extension mechanism based on extensible grammars and macros. The grammar system offers an abstraction mechanism—nonterminal definition—and an expressive syntax description language, but it lacks a means of programmatically enforcing constraints and controlling error messages. Fortress follows the tradition of Cardelli's extensible syntax [11], which had similar strengths and weaknesses.

The device of tracking failure information through backtracking and ordering failures appears in Ford's work on packrat parsing [37], where the notion of a failure's progress is simply characters consumed. Our ordering of parse failures is also similar to Despeyroux's work on partial proofs in logic programming [19]. In that work, a set of inference rules is extended with "recovery" rules that prove any proposition. The partial proofs are ordered so that use of a recovery rule has less progress than any real rule, and uses of different rules are incomparable; only the maximal proofs are returned. In contrast to the partial order of that system, our system uses a total order based on the pragmatics of parsing tree-structured input syntax.

CHAPTER 10

Conclusion

Macro developers benefit from macro-specific development tools.

The macro stepper enables macro writers to see how their macros expand and to quickly diagnose errors. The macro stepper displays binding information and other syntax information that influences expansion, and macro hiding lets programmers debug their macros without being distracted by the macros implementing their base language.

Syntax classes and `syntax-parse` enable macro writers to write high-level specifications of syntax. These specifications are used to validate uses of macros and automatically report specific error messages. Syntax classes and the `syntax-parse` notation makes writing robust macros quick, concise, and painless.

Bibliography

[1] Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. Growing a syntax. In *Foundations of Object-Oriented Languages*, January 2009.

[2] Lennart Augustsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, pages 368–381, 1985.

[3] Jonthan Bachrach and Keith Playford. The java syntactic extender (JSE). In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 31–42, 2001.

[4] Jason Baker and Wilson C. Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–281, June 2002.

[5] Eli Barzilay. Swindle. `http://www.barzilay.org/Swindle`.

[6] Eli Barzilay and John Clements. Laziness without all the hard work: combining lazy and strict languages for teaching. In *Functional and declarative programming in education*, pages 9–13, 2005.

[7] Alan Bawden and Jonathan Rees. Syntactic closures. In *Symposium on Lisp and Functional Programming*, pages 86–95, 1988.

[8] Ana Bove and Laura Arbilla. A confluent calculus of macro expansion and evaluation. In *Symposium on Lisp and Functional Programming*, pages 278–287, 1992.

[9] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 365–383, 2004.

[10] Cadence Research Systems. *Chez* Scheme Reference Manual, 1994.

[11] Luca Cardelli, Florian Matthes, Martin Abadi, and Robert W. Taylor. Extensible syntax with lexical scoping, 1994.

[12] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, pages 320–334, 2001.

[13] William Clinger. Hygienic macros through explicit renaming. *Lisp Pointers*, IV(4):25–28, 1991.

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

[15] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, 2006.

[16] Ryan Culpepper and Matthias Felleisen. Taming macros. In *International Conference on Generative Programming and Component Engineering*, pages 225–243, 2004.

[17] Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic abstraction in component interfaces. In *International Conference on Generative Programming and Component Engineering*, pages 373–388, 2005.

[18] Daniel de Rauglaudre. *Camlp4 - Reference Manual*, 2003.

[19] Thierry Despeyroux. Logical programming and error recovery. In *Industrial Applications of Prolog*, October 1995.

[20] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[21] Carl Eastlund and Matthias Felleisen. Automatic verification for interactive graphical programs. In *ACL2 Workshop*, pages 33–41, 2009.

[22] Sebastian Egner. Eager comprehensions in Scheme: The design of SRFI-42. In *Workshop on Scheme and Functional Programming*, pages 13–26, September 2005.

[23] Matthias Felleisen. Transliterating Prolog into Scheme. Technical Report 182, Indiana University, 1985.

[24] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[25] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, August 2009.

[26] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *International Conference on Functional Programming*, pages 26–37, 2001.

[27] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[28] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, pages 48–59, 2002.

[29] Matthew Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, June 1999.

[30] Matthew Flatt. Composable and compilable macros: you want it when? In *International Conference on Functional Programming*, pages 72–83, 2002.

[31] Matthew Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR2006-1-v352, PLT Scheme Inc., 2006. `http://www.plt-scheme.org/techreports/`.

[32] Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: closing the book on ad hoc documentation tools. In *International Conference on Functional Programming*, pages 109–120, 2009.

[33] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[34] Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems (APLAS) 2006*, pages 270–289, November 2006.

[35] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. *Formal Syntax and Semantics of Java*, 1523:241–269, 1999. Preliminary version appeared in proceedings of *Principles of Programming Languages*, 1998. Revised version is Rice University TR 97-293, June 1999.

[36] Matthew Flatt and PLT. Reference: PLT scheme. Reference Manual PLT-TR2010-reference-v4.2.4, PLT Scheme Inc., January 2010. `http://plt-scheme.org/techreports/`.

[37] Bryan Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, September 2002.

[38] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, July 2005.

[39] Martin Gasbichler. *Fully-parameterized, First-class Modules with Hygienic Macros*. PhD thesis, Eberhard-Karls-Universität Tübingen, February 2006.

[40] David Herman and Mitchell Wand. A theory of hygienic macros. In *European Symposium on Programming*, pages 48–62, March 2008.

[41] Patricia Hill and John Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[42] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[43] John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 53–96, London, UK, 1995. Springer-Verlag.

[44] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report of the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[45] Oleg Kiselyov. How to write seemingly unhygienic and referentially opaque macros with syntax-rules. In *Workshop on Scheme and Functional Programming*, October 2002.

[46] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *Symposium on Lisp and Functional Programming*, pages 151–161, 1986.

[47] Eugene E. Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Symposium on Principles of Programming Languages*, pages 77–84, 1987.

[48] Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101, 1965.

[49] Xavier Leroy. *The Objective Caml system, Documentation and User's guide*, 1997.

[50] Henry Lieberman. Steps toward better debugging tools for LISP. In *Proc. 1984 ACM Symposium on LISP and Functional Programming*, pages 247–255, 1984.

[51] Jay McCarthy. Automatically RESTful web applications: marking modular serializable continuations. In *International Conference on Functional Programming*, pages 299–310, 2009.

[52] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[53] Lee Naish. Pruning in logic programming. Technical report, University of Melbourne, 1995.

[54] Scott Owens, Matthew Flatt, Olin Shivers, and Benjamin McMullan. Lexer and parser generators in Scheme. In *Workshop on Scheme and Functional Programming*, pages 41–52, September 2004.

[55] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *International Conference on Functional Programming*, pages 216–227, 2005.

[56] PLT. PLT MzLib: Libraries manual. Technical Report PLT-TR2006-4-v352, PLT Scheme Inc., 2006. `http://www.plt-scheme.org/techreports/`.

[57] Christian Queinnec. Macroexpansion reflective tower. In *Proceedings of the Reflection'96 Conference*, pages 93–104, 1996.

[58] Jonathan Riehl. Assimilating MetaBorg: Embedding language tools in languages. In *International Conference on Generative Programming and Component Engineering*, pages 21–28, October 2006.

[59] Jonathan Riehl. *Reflective techniques in extensible languages*. PhD thesis, University of Chicago, August 2008.

[60] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass framework for compiler education. *Journal of Functional Programming*, 15(5):653–667, September 2005. Educational Pearl.

[61] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In *Haskell Workshop*, pages 1–16, October 2002.

[62] Olin Shivers. The anatomy of a loop: a story of scope and control. In *International Conference on Functional Programming*, pages 2–14, 2005.

[63] Dorai Sitaram. Programming in schelog. `http://www.ccs.neu.edu/home/dorai/schelog/schelog.html`.

[64] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. Revised^6 report of the algorithmic language Scheme. *Journal of Functional Programming*, 19(S1):1–301, August 2009.

[65] Guy L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, 1990.

[66] Guy Lewis Steele, Jr. Rabbit: a compiler for Scheme. Technical Report 474, MIT Artificial Intelligence Laboratory, May 1978.

[67] T. Stephen Strickland and Matthias Felleisen. Contracts for first-class modules. In *Symposium on Dynamic Languages*, pages 27–38, 2009.

[68] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Symposium on Principles of Programming Languages*, pages 395–406, 2008.

[69] Andrew P. Tolmach and Andrew W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[70] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Symposium on Principles of Programming Languages*, pages 203–215, 1999.

[71] Mitchell Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10(3):189–199, 1998.

[72] Mitchell Wand and Dale Vaillancourt. Relating models of backtracking. In *International Conference on Functional Programming*, pages 54–65, 2004.

[73] Daniel Weise and Roger Crew. Programmable syntax macros. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, June 1993.

[74] Andrew Wright and Bruce Duba. Pattern matching for Scheme, 1995. Unpublished manuscript.

[75] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1-2):39 – 54, 2010. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).