Fortifying Macros [*]

Ryan Culpepper [†] Matthias Felleisen
Northeastern University
{ryanc,matthias}@ccs.neu.edu

Abstract

Existing macro systems force programmers to make a choice between clarity of specification and robustness. If they choose clarity, they must forgo validating significant parts of the specification and thus produce low-quality language extensions. If they choose robustness, they must write in a style that mingles the implementation with the specification and therefore obscures the latter.

This paper introduces a new language for writing macros. With the new macro system, programmers naturally write robust language extensions using easy-to-understand specifications. The system translates these specifications into validators that detect misuses—including violations of context-sensitive constraints—and automatically synthesize appropriate feedback, eliminating the need for ad hoc validation code.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

1. What is a macro?

Every functional programmer knows that a **let** expression can be expressed as the immediate application of a λ abstraction [Landin 1965]. The **let** expression's variables become the formal parameters of the λ expression, the initialization expressions become the application's arguments, and the body becomes the body of the λ expression. Here is a quasi-formal expression of the idea:

$$(\textbf{let}\ ([\mathit{var}\ \mathit{rhs}]\ \ldots)\ \mathit{body}) = ((\lambda\ (\mathit{var}\ \ldots)\ \mathit{body})\ \mathit{rhs}\ \ldots)$$

It is understood that each *var* is an identifier and each *rhs* and *body* is an expression; the variables also must be distinct. These constraints might be stated as an aside to the above equation, and some might even be a consequence of metavariable conventions.

New language elements such as **let** can be implemented via *macros*, which automate the translation of new language forms into simpler ones. Essentially, macros are an API for extending the front end of the compiler. Unlike many language extension tools, however, a macro is part of the program whose syntax it extends; no separate pre-processor is used.

[*] The research was partially supported by NSF infrastructure grants.

[†] New address: School of Computing, 50 Central Campus Drive (Rm 3190), Salt Lake City, UT 84112-9205

A macro definition associates a name with a compile-time function, i.e., a *syntax transformer*. 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, they are often called *derived syntactic forms*. In the example above, the derived form **let** is expanded into the primitive forms λ and function application. Due to the restricted syntax of macro uses—the macro name must occur in operator position—extensions to the language easily compose. Since extensions are anchored to names, extensions can be managed by controlling the scope of their names. This allows the construction of a tower of languages in layers.

Introducing new language elements, dubbed *macros*, has long been a standard element of every Lisper's and Schemer's repertoire. Racket [Flatt and PLT 2010], formerly PLT Scheme, is a descendent of Lisp and Scheme that uses macros pervasively in its standard libraries. Due in part to its pedagogical uses, Racket has high standards for error behavior. Languages built with macros are held to the same standards as Racket itself. In particular, syntactic mistakes should be reported in terms of the programmer's error, not an error discovered after several rounds of rewriting; and furthermore, the mistake should be reported in terms documented by the language extension.

Sadly, existing systems make it surprisingly difficult to produce easy-to-understand macros that properly validate their syntax. These systems force the programmer to mingle the declarative specification of syntax and semantics with highly detailed validation code. Without validation, however, macros aren't true abstractions. Instead, erroneous terms flow through the parsing process until they eventually trip over constraint checks at a low level in the language tower. Low-level checking, in turn, yields incoherent error messages and leaves programmers searching for explanations. In short, such macros do not create seamless linguistic abstractions but sources of confusion and distraction.

In this paper, we present a novel macro system for Racket that enables the creation of true syntactic abstractions. Programmers define modular, reusable specifications of syntax and use them to validate uses of macros. The specifications consist of grammars extended with context-sensitive constraints. When a macro is used improperly, the macro system uses the specifications to synthesize an error message at the proper level of abstraction.

2. Expressing macros

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

$$(\textbf{let}\ ([\mathit{var}\ \mathit{rhs}]\ \ldots)\ \mathit{body}) = ((\lambda\ (\mathit{var}\ \ldots)\ \mathit{body})\ \mathit{rhs}\ \ldots)$$
the *var*s are distinct identifiers
body and the *rhs*s are expressions

A macro's syntax transformer is essentially a function from syntax to syntax. Many Lisp 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 S-expression functions to "parse" syntax, and they use quasiquotation to build up the desugared expression:

```
(define-macro (let bindings body)
  `((λ ,(map first bindings) ,body)
    ,@(map second bindings)))
```

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

```
(define-macro (let bindings body)
  (define vars (map first bindings))
  (define rhss (map second bindings))
  `((λ ,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, the macro simply ignores extra terms in a binding pair:

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

Macro writers, eager to move on as soon as "it works," will continue to write sloppy macros like these unless their tools make it easy to write robust ones.

One such tool is the so-called Macro-By-Example (MBE) notation by Kohlbecker and Wand [1987]. In MBE, macros are specified in a notation close to the initial informal equation, and the parsing and transformation 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 contains *syntax pattern variables*, and when a pattern matches, the pattern variables are bound to the corresponding sub-terms of the macro occurrence. These sub-terms are substituted into the template where the pattern variables occur to produce the macro's expansion result.

Here is **let** expressed with **syntax-rules** [Sperber et al. 2009], one of many implementations of MBE:

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

The pattern variables are *var*, *rhs*, and *body*.

The crucial innovation of MBE is the use of ellipses (...) to describe sequences of sub-terms with homogeneous structure. Such sequences occur frequently in S-expression syntax. Some sequences have simple elements, such as the parameters of a λ expression, but often the sequences have non-trivial structure, such as binding pairs associating **let**-bound variables with their values.

Every pattern 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 consistent. We do not address template checking and transcription in this work; see Kohlbecker and Wand [1987] for details.

Ellipses do not add expressive power to the macro system but do add expressiveness to *patterns*. Without ellipses, the **let** macro can still be expressed via explicit recursion, but in a way that obscures the nature of valid **let** expressions; instead of residing in a single pattern, it would be distributed across multiple clauses of a recursive macro. In short, ellipses help close the gap between specification and implementation.

Yet MBE lacks the power to express all of the information in the informal description of **let** above. The example macros presented so far neglect to validate two critical aspects of the **let** syntax: the first term of each binding pair must be an identifier, and those identifiers must be distinct.

Consider these two misuses of **let**:

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

In neither case does the **let** macro report that it has been used incorrectly. Both times it inspects the syntax, approves it, and produces an invalid λ expression. Then λ, implemented by a careful compiler writer, signals an error, such as "λ: duplicate identifier in: *x*" in Racket for the first term and "invalid parameter list in (λ ((*x y*)) (*g x y*))" in Chez Scheme [Cadence Research Systems 1994] for the second. Source location tracking [Dybvig et al. 1993] improves the situation somewhat in macro systems that offer it. For example, the DrRacket [Findler et al. 2002] programming environment highlights the duplicate identifier. But this is not a good solution. Macros should report errors on their own terms.

Worse, a macro might pass through syntax that has an unintended meaning. In Racket, the second example above produces the surprising error "unbound variable in: *y*." The pair (*x y*) is accepted as an *optional parameter with a default expression*, a feature of Racket's λ syntax, and the error refers to the free variable *y* in the latter portion. If *y* were bound in this context, the second example would be silently accepted. A slight variation demonstrates another pitfall:

```
(let ([(x) (f 7)]) (g x x))
```

This time, Racket reports the following error: "λ: not an identifier, identifier with default, or keyword at: (*x*)." The error message not only leaks the implementation of **let**, it implicitly obscures the legal syntax of **let**.

```
(define-syntax (let stx)
  (syntax-case stx ()
    [(let ([var rhs] ...) body)
     ;; Guard expression
     (and (andmap identifier? (syntax→list #'(var ...)))
          (not (check-duplicate #'(var ...))))
     ;; Transformation expression
     #'((λ (var ...) body) rhs ...)]))
```

Figure 1. let with guards

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 its constraints are satisfied. If the guard expression fails, the pattern is rejected and the next pattern is tried. If all of the patterns fail, the macro raises a *generic syntax error*, such as "bad syntax."

Figure 1 shows the implementation of **let** in **syntax-case** [Dybvig et al. 1993; Sperber et al. 2009], an implementation of MBE that provides guard expressions. A **syntax-case** clause consists of a pattern, an optional guard, 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 performed *twice* and code is often duplicated. Second and more important, guards do *not* explain why the syntax was invalid. That is, they only control matching; they do not track causes of failure.

```
(define-syntax (let stx)
  (syntax-case stx ()
    [(let ([var rhs] ...) body)
     (begin
      ;; Error-checking code
      (for-each (λ (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
      #'((λ (var ...) body) rhs ...))]))
```

Figure 2. let with hand-coded error checking

To provide precise error explanations, explicit error checking is necessary, as shown in figure 2. Of the ten non-comment lines of the macro's clause, one is the pattern, one is the template, and *eight* are dedicated to validation. Furthermore, this macro only reports errors that match the shape of the pattern. If it is given a malformed binding pair with extra terms after the right-hand side expression, the clause fails to match, and **syntax-case** produces a generic error. Detecting and reporting those sorts of errors would require even more code. Only the most conscientious macro writers are likely to take the time to enumerate all the ways the syntax could be invalid and to issue appropriate error reports.

Certainly, the code for **let** could be simplified. Macro writers could build libraries of common error-checking routines. Such an approach, however, would still obscure the natural two-line specification of **let** by mixing the error-checking code with the transformation code. Furthermore, abstractions that focus on raising syntax errors would not address the other purpose of guards, the selection among multiple valid alternatives.

Even ignoring the nuances of error reporting, some syntax is simply difficult to parse with MBE patterns. Macro writers cope in two ways: either they compromise on the user's convenience with simplified syntax or they hand-code the parser.

```
(define-struct struct (field ...) option ...)

    where struct, field are identifiers
          option ::= #:mutable
                  |  #:super super-struct-expr
                  |  #:inspector inspector-expr
                  |  #:property property-expr value-expr
                  |  #:transparent
```

Figure 3. The syntax of **define-struct**

Keyword arguments are one kind of syntax difficult to parse using MBE patterns. An example of a keyword-enhanced macro is Racket's **define-struct** form, whose grammar is specified in figure 3. It has several keyword options, which can occur in any order. The #:transparent and #:inspector keywords control when structure values can be inspected via reflection. The #:mutable option makes the fields mutable; the #:property option allows structure types to override behavior such as how they are printed; and so on. Different keywords come with different numbers of arguments, e.g., #:mutable has none and #:property takes two.

Parsing a **define-struct** form gracefully is simply beyond the capabilities of MBE's pattern language, which focuses 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 unworkable. The alternative is, again, to move part of the parsing into the transformer code. The macro writer sketches the rough structure of the syntax in broad strokes with a pattern, then fills in the details with procedural parsing code:

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

In the actual implementation of **define-struct**, the parsing of keyword options alone takes over one hundred lines of code. In comparison, when formulated in our new system this code is shortened by an order of magnitude.

In summary, MBE offers weak syntax patterns, forcing the programmer to move the work of validation and error-reporting into guards and transformers. Furthermore, guard expressions accept or reject entire clauses, and rejection comes without information as to why the guard failed. Finally, MBE lacks the vocabulary to describe a broad range of important syntaxes. Our new domain-specific language for macros eliminates these problems.

3. The design of syntax-parse

Our system, dubbed **syntax-parse**, uses a domain-specific language to support parsing, validation, and error reporting. It features three significant improvements over MBE:

- an expressive language of syntax patterns, including pattern variables annotated with the classes of syntax they can match;
- a facility for defining new syntax classes as abstractions over syntax patterns; and
- a matching algorithm that tracks progress to rank and report failures and a notion of failure that carries error information.

Furthermore, guard expressions are replaced with *side conditions*, which provide rejection messages.

The syntax classes of our new system serve a role similar to that of non-terminals in traditional grammars. Their addition allows the disciplined interleaving of declarative specifications and hand-coded checks.

This section illustrates the design of **syntax-parse** with a series of examples based on the **let** example.

```
(syntax-parse stx-expr [pattern side-clause ... expr] ...)

    where side-clause ::= #:fail-when cond-expr msg-expr
                       |  #:with pattern stx-expr
```

Figure 4. Syntax of `syntax-parse`

3.1 Validating syntax

The syntax of **syntax-parse**—specified in figure 4—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
    [(let ([var rhs] ...) body)
     #'((λ (var ...) body) rhs ...)]))
```

It enforces only the two side conditions in the original specification.

To this skeleton we add the constraint that every term labeled *var* must be an identifier. Likewise, *rhs* and *body* are annotated to indicate that they are expressions. For our purposes, an expression is any term other than a keyword. The final constraint, that the identifiers are unique, is expressed as a side condition using a #:fail-when clause. Here is the revised macro:

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

Note that a syntax class annotation such as *expr* is not part of the pattern variable name, and it does not appear in the template.

The call to *check-duplicate* acts as a condition; if it is false, failure is averted and control flows to the template expression. But if it returns any other value, parsing fails with a "duplicate variable name" message; furthermore, if the condition value is a syntax object—that is, the representation of a term—that syntax is included as the specific site of the failure. In short, side conditions differ from guard expressions in that the failures they generate carry information describing the reasons for the failure.

At this point, our **let** macro properly validates its syntax. It catches the misuses earlier and reports the following errors:

> (**let** ([*x* 1] [$\boxed{x}$ 2]) (*h x*))
let: duplicate variable name in: *x*
> (**let** ([$\boxed{(x\ y)}$ (*f* 7)]) (*g x y*))
let: expected identifier in: (*x y*)

The boxes indicates the specific location of the problem; the DrRacket programming environment highlights these terms in red in addition to printing the error message.

For some misuses, **let** still doesn't provide good error messages. Here is an example that is missing a pair of parentheses:

> $\boxed{\textbf{(let } (x\ 5)\ (add1\ x))}$
let: bad syntax

Our **let** macro rejects this misuse with a generic error message. To get better error messages, the macro writer must supply **syntax-parse** with additional information.

3.2 Defining syntax classes

Syntax classes form the basis of **syntax-parse**'s error-reporting mechanism. Defining a syntax class for binding pairs gives **syntax-parse** the vocabulary to explain a new class of errors. The syntax of binding pairs is defined as a syntax class thus:

```
(define-syntax-class binding
  #:description "binding pair"
  (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"
     #'((λ (b.var ...) body) b.rhs ...)]))
```

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**, such as **let**∗ and **letrec**.

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"
  (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*, the *var* and *rhs* attributes of *distinct-bindings* have an *ellipsis depth* of 1, so *bs.var* and *bs.rhs* can be used within ellipses in the macro's template, even if *bs* does not occur within ellipses in the macro's pattern:

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

Now that we have specified the syntax of *binding* and *distinct-bindings*, **syntax-parse** can use them to generate good error message for additional misuses of **let**:

> (**let** ($\boxed{x}$ 5) (*add1 x*))
let: expected binding pair in: *x*
> (**let** $\boxed{17}$)
let: expected sequence of binding pairs in: 17

The next section explains how **syntax-parse** generates error messages and how defining syntax classes affects error reporting.

4. Reporting errors

The **syntax-parse** system uses the declarative specification of a macro's syntax to report errors in macro uses. The task of reporting errors is factored into two steps. First, the matching algorithm selects the most appropriate error to report. Second, it reports the error by pinpointing the faulty term and describing the fault or stating the expected class of syntax.

4.1 Error selection

Pattern variable annotations and side conditions serve a dual role in our system. As seen, pattern variable annotations and side conditions allow **syntax-parse** to validate syntax. When validation fails, **syntax-parse** reports the specific site and cause of the failure. But annotations and side conditions do not simply behave like the error checks of figure 2. A macro can have multiple clauses, and a syntax class can have multiple variants. If there are multiple choices, all of them must be attempted before an error is raised and explained.

To illustrate this process, we must introduce choice into our running example. Serendipitously, Racket inherits Scheme's **let** syntax, which has another variant—a so-called "named **let**"—that specifies a name for the implicit procedure. This notation provides a handy loop-like syntax. For example, the following program determines whether the majority of numbers in a list are positive:

```
(define (mostly-positive? nums)
  (let loop ([nums nums] [pos 0] [non 0])
    (cond [(empty? nums) (> pos non)]
          [(positive? (first nums))
           (loop (rest nums) (+ 1 pos) non)]
          [else (loop (rest nums) pos (+ 1 non))])))
```

Implementing the new variant of **let** is as simple as adding another clause to the macro:

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

The macro uses the annotations to pick the applicable pattern; it chooses named-**let** if the first argument is an identifier and normal-**let** if it is a binding list. It happens that the two patterns are mutually exclusive, so the order of the clauses is irrelevant.

The use of annotations to select the matching clause must be reconciled with the role of annotations in error reporting. An annotation rejection during pattern-matching clearly cannot 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 **syntax-parse** clauses or multiple variants of a syntax class definition, they are tried in order. When an alternative fails, **syntax-parse** records the failure and backtracks to the next alternative. As alternatives are tried, **syntax-parse** accumulates a list of failures, and each failure contains a measure of the *matching progress* made. If the whole matching process 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 error explanation. Otherwise, the maximal failures are combined.

$$\begin{array}{lll} \text{Progress} & \pi & ::= \ ps^* \\ \text{Progress Step} & ps & ::= \ \textsc{First} \ | \ \textsc{Rest} \ | \ \textsc{Late} \end{array}$$

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

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

Figure 5. Progress

Figure 5 defines our notion of progress as sequences of progress steps. The progress steps FIRST and REST indicate the first and rest of a compound term, respectively. Parsing is performed left to right; if the parser is looking at the rest of a compound term, the first part must have been parsed successfully. 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 named-**let** clause fails at the second sub-term with the progress string REST · FIRST:

(**let** $\boxed{\text{([}a\text{ 1] [2 }b\text{])}}$ (∗ *a b*))

The normal-**let** clause, however, fails deeper within the second argument, at REST · FIRST · REST · FIRST · FIRST:

(**let** ([*a* 1] [$\boxed{2}$ *b*]) (∗ *a b*))

This second sequence denotes strictly more progress than REST · FIRST. Thus, the second failure is selected, and the macro reports that it expected an identifier in place of 2.

Matching progress is not only a measure of position in a term. Consider the following example:

(**let** $\boxed{\text{([}x\text{ 1] [}x\text{ 2])}}$ (+ *x x*))

Both clauses agree on the faulty subterm. But this example is clearly closer to a use of normal-**let** rather than named-**let**. The faulty term matches the structure of *distinct-bindings*, just not the side condition.

Pragmatically, we consider a check for side conditions—in contrast to an annotation check—to occur *after* traversal of the term. A progress step dubbed LATE signals the failure of a side condition. Thus, while the named-**let** clause fails with the progress string REST · FIRST in the example above, the normal-**let** clause fails with REST · FIRST · LATE, which is greater progress than the first.

Sometimes multiple alternatives fail at the same place, e.g.,

> (**let** $\boxed{5}$)
let: expected identifier or sequence of binding pairs in: 5

Both clauses make the same amount of progress with this term: REST · FIRST. As a result, both failures are selected, and the error message includes both descriptions.

4.2 Error messages

In addition to progress, a failure contains a message that indicates the nature of the error and the term where the failure occurred. A typical error message is

let: expected binding pair in: *x*

This message consists of the macro's expectations (a binding pair) and the specific term where parsing failed (*x*).

A syntax error should identify the faulty term and concisely explain what was expected. It should not recapitulate the macro's documentation; rather, the error message should make locating the appropriate documentation easy, e.g., via links and references. Consequently, **syntax-parse** produces messages from a limited set of ingredients. It automatically synthesizes messages for literal and datum patterns; for example, the pattern 5 yields the message "expected the literal 5." As a special case, it also knows how to report when a compound term has too many sub-terms. The only other ingredients it uses are provided by the macro developer: descriptions and side-condition messages.

In particular, **syntax-parse** does not synthesize messages to describe compound patterns. We call such patterns and the failures they cause "ineffable"; our system cannot generate explanations for them. An example is 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 a compound term consisting of an identifier and an expression." Another is "expected (identifier expr)." In practice, macro writers occasionally write error messages of both forms. We have chosen not to generate such messages automatically for two reasons: first, they do not scale well to large or sophisticated patterns; and second, we consider such messages misguided.

Generating messages from patterns is feasible when the patterns are 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. *Error reporting should be based on documented concepts, not implementation details.*

When a compound pattern such as the one above fails, the pattern's context is searched and the nearest enclosing description is used to report the error. Consider the following misuse of **let**:

```
(let (x 1) (add1 x))
```

The error selection algorithm from section 4.1 determines that the most specific failure arose trying to match *x* against the pattern (*var:identifier rhs:expr*). Here is the full context of the failure:

- matching *x* against (*var:identifier rhs:expr*) failed
- while matching *x* against *b:binding*
- while matching (*x* 1) against *bs:distinct-bindings*
- while matching (**let** (*x* 1) (*add1 x*)) against the complex pattern (**let** *bs:distinct-bindings body:expr*)

The first and fourth frames contain ineffable patterns. Discarding them and rephrasing the expected syntax gives us the following context:

- expected binding pair, given *x*
- expected sequence of binding pairs, given (*x* 1)

The message and term of the first frame are used to formulate the error message "**let**: expected binding pair in: *x*" because it is the closest one.

5. Syntax patterns

The power of **syntax-parse** is due to its expressive pattern language, an extension of the syntax patterns of MBE. Sections 3 and 4 have introduced some features of our pattern language. This section describes additional pattern forms that, in our experience, increase the expressive power of our system to the level necessary for developing real syntax specifications.

$$
\begin{array}{rrl}
\text{Patterns } S & ::= & x \\
 & | & x : class \\
 & | & (S \;.\; S) \\
 & | & (S \;\ldots\; .\; S) \\
 & | & datum \\
 & | & (\tilde{}\textbf{literal}\; x) \\
 & | & (\tilde{}\textbf{var}\; x\; (class\; e^*)) \\
 & | & (\tilde{}\textbf{and}\; S^+) \\
 & | & (\tilde{}\textbf{or}\; S^+) \\
 & | & (\tilde{}\textbf{describe}\; expr\; S)
\end{array}
$$

Figure 6. Single-term patterns

5.1 Single-term patterns

Figure 6 describes the syntax of syntax patterns, specifically *single-term patterns*, the kind of pattern that specifies sets of single terms. The first four—pattern variables, annotated pattern variables, pair patterns, and ellipsis patterns—appear in section 3. So do datum patterns, in the form of (), which ends compound patterns.[1] In general, data like numbers, booleans, and strings can be used as patterns that match themselves. The **˜literal** pattern form[2] recognizes identifiers that have the same binding as the enclosed identifier; this

[1] The notation (*a b*) is shorthand for (*a* . (*b* . ())).

[2] All pattern keywords start with a tilde (˜).

```
(define-syntax-class distinct-bindings
  #:description "sequence of binding pairs"
  (pattern (~var bs (bindings-excluding '()))
           #:with (var ...) (bs.var ...)
           #:with (rhs ...) (bs.rhs ...)))

;; seen is a list of identifiers
(define-syntax-class (bindings-excluding seen)
  (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 ...)))

;; seen is a list of identifiers
(define-syntax-class (id-excluding seen)
  (pattern x:identifier
           #:fail-when (for/or ([id seen])
                         (bound-identifier=? #'x id))
                       "duplicate variable name"))
```

Figure 7. Parameterized syntax classes

is the standard notion of identifier equality in hygienic macro systems [Dybvig et al. 1993].

A **˜var** pattern constrains a pattern variable to a syntax class. The colon notation is a shorthand for parameterless syntax classes; e.g., *x:identifier* is short for (**˜var** *x* (*identifier*)). When the syntax class takes parameters, the explicit **˜var** notation is required.

A syntax class's parameters may be used in its sub-expressions, including its description and any of its side conditions. For example, here is a syntax class that recognizes literal natural numbers less than some upper bound:

```
;; ex.: the pattern (~var n (nat< 10)) matches
;; any literal natural number less than 10
(define-syntax-class (nat< bound)
  #:description (format "natural number < ~s" bound)
  (pattern n:nat
    #:fail-when (not (< (syntax→datum #'n) bound))
         (format "got a number ~s or greater" bound)))
```

Notice how the upper bound is inserted into both the description and the check message using the *format* procedure.

We can use parameterized syntax classes to give an alternative definition of *distinct-bindings*, via a syntax class parameterized over the identifiers that have already been seen. Figure 7 shows the alternative definition and the auxiliaries *bindings-excluding* and *id-excluding*. The pattern *bindings-excluding* syntax class accepts sequences of distinct bindings but also requires that the bound names not occur in *seen*. Consider *bindings-excluding*'s second pattern; *var0* must be an identifier not in *seen*, and the identifier bound to *var0* is added to the blacklisted identifiers for the rest of the binding sequence. Note that *var0* is in scope in the argument to *bindings-excluding*. Since patterns are matched left to right, pattern variable binding also runs left to right, following the principle of scope being determined by control dominance [Shivers 2005].

While it *accepts* the same terms, this alternative definition of *distinct-bindings* reports errors differently from the one in section 3.2. 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 thus report different errors for the following term:

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

In such cases, the macro writer must decide the most suitable order of validation.

The **~and** pattern form provides a way of analyzing a term multiple ways. Matching order and binding go left to right within an **~and** pattern, so later sub-patterns can rely on earlier ones.

The **~or** form matches if any of its sub-patterns match. Unlike in many pattern matching systems, where disjunction, if it is supported at all, requires that the disjuncts bind the same pattern variables, **~or** patterns are more flexible. An **~or** pattern binds the *union* of its disjuncts' attributes, and those attributes that do not occur in the matching disjunct are marked "absent."

It is illegal to use an absent attribute in a syntax template, so **syntax-parse** provides the *attribute* form, which accesses the value of the attribute, returning *false* for absent attributes. Using *attribute*, a programmer can check whether it is safe to use an attribute in a template. 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
    [(~or field:identifier [internal:identifier field:identifier])
     (make-field (if (attribute internal) #'internal #'field)
                 #'field)]))
```

Some 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
  (pattern field:id
           #:with internal #'field)
  (pattern [internal:id field:id]))
```

The final pattern form, **~describe**, pushes a new description *d* onto the matching context of its sub-pattern. Hence, if a failure occurs and if there is no other description closer to the source of the error, the description *d* is used to explain the failure.

There is no difference between a description attached to a syntax class and one given via **~describe**. Recall the *binding* and *distinct-bindings* syntax class definitions from section 3.2; the *binding* syntax class could be inlined into *distinct-bindings* as follows:

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

In fact, *distinct-bindings* could be inlined into the **let** macro itself using **~describe** and action patterns.

$$
\begin{array}{llcl}
\text{Action patterns} & A & ::= & (\textbf{~parse}\ S\ \mathit{expr}) \\
 & & | & (\textbf{~fail}\ \mathit{condition}\ \mathit{message}) \\
 & & | & (\textbf{~late}\ A) \\
\text{Patterns} & S & ::= & \cdots \\
 & & | & (\textbf{~and}\ S\ \{S|A\}^*)
\end{array}
$$

Figure 8. Action patterns

5.2 Action patterns

The action patterns of figure 8 do not describe syntax; instead, they affect the parsing process without consuming input. The **~parse** form allows the programmer to divert matching from the current input to a computed term; **~fail** provides a way of explicitly causing a match failure; and **~late** affects the ordering of failures.

The **~parse** form evaluates its sub-expression and matches it against the given pattern. One use for the **~parse** form is to bind default values within an **~or** pattern, avoiding the need for explicit *attribute* checks later. Recall *parse-field-declaration*. Here *internal* is bound in both alternatives, simplifying the result template:

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

This example also shows the use of **~and** to sequence an action pattern after a single-term pattern. Since **~and** propagates attributes bound in each of its sub-patterns to subsequent sub-patterns, **~and** can be used to parse a term and then perform actions depending on the contents of the term.

The **~fail** patterns allow programmers to perform side-constraint checks. Additionally, if the condition evaluates to a syntax value, it is added to the failure as the specific term that caused the error.

By default, **~fail** performs early checks. For example, the *identifier* syntax class performs its test as an early check:

```
(define-syntax-class identifier
  (pattern (~and x (~fail (not (identifier? #'x)) no-msg))))
```

The **~late** form turns enclosed checks into late checks. In fact, the #:fail-when keyword option used in *distinct-bindings* is just shorthand for a combination of **~late** and **~fail**:

```
(define-syntax-class distinct-bindings
  #:description "sequence of distinct bindings"
  (pattern (~and (b:binding ...)
                 (~late (~fail (check-duplicate #'(b.var ...))
                               "duplicate variable name")))))
```

5.3 Head patterns

The patterns of Sections 5.1 and 5.2 do not provide the power needed to parse macros like **define-struct** from figure 3. There are elements of **define-struct**'s syntax that comprise multiple consecutive terms, but single-term patterns describe only single terms, and action patterns do not describe terms at all. An occurrence of the super option, for example, consists of two adjacent terms: the keyword #:super followed by an expression, e.g.,

```
(define-struct point (x y) #:super geometry #:mutable)
```

No single-term pattern describes the inspector option. In particular, the pattern (#:super *sup:expr*) does not, because #:super and its argument do not appear as a separate parenthesized term, such as (#:super *geometry*).

$$
\begin{array}{llcl}
\text{Head patterns} & H & ::= & (\textbf{~seq}\ .\ L) \\
 & & | & (\textbf{~and}\ H\ \{H|A\}^*) \\
 & & | & (\textbf{~or}\ H^+) \\
 & & | & (\textbf{~describe}\ \mathit{expr}\ H) \\
 & & | & S \\
\text{List pattern} & L & ::= & () \\
 & & | & (S\ .\ L) \\
 & & | & (H\ .\ L) \\
 & & | & (H\ ...\ .\ L) \\
\text{Patterns} & S & ::= & \cdots \\
 & & | & (H\ .\ S) \\
 & & | & (H\ ...\ .\ S)
\end{array}
$$

Figure 9. Head patterns

Our solution is to introduce the *head patterns* of figure 9, 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:identifier* ... *y:expr*) matches a sequence of any number of identifiers followed by one expression. Contrast that pattern with (*x:identifier* ... *y:expr*), which matches *a single compound term* containing a sequence of identifiers followed by an expression.

A head pattern may be combined with a normal single-term pattern to form a single-term pattern. The combined pattern matches a term by attempting to split it into a prefix sequence of terms that matches the head pattern and a suffix term that matches the tail. The term need not be a compound term if the prefix can be empty. For example, the pattern ((**~seq** *x y z*) *w:identifier* ...) matches the term (1 2 3 *a b*) 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:identifier* ...). Of course, ((**~seq** *x y z*) *w:identifier* ...) is equivalent to (*x y z w:identifier* ...). The **~seq** pattern is useful primarily when combined with other pattern forms, such as **~and** and **~or**, as in macros with optional keyword arguments:

```
(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 **let**, accommodating both normal-**let** and named-**let** syntax, with the following pattern:

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

Splicing syntax classes encapsulate head patterns. Each of its variants is a head pattern (H), most often a **~seq** pattern, although other kinds of head pattern are possible. The optional #:around keyword argument could be extracted thus:

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

A pattern variable annotated with a splicing syntax class can represent multiple terms. In this example, *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 patterns	EH	$::=$	(**~or** EH^+)
		$\mid$	(**~once** H #:name *expr*)
		$\mid$	(**~optional** H #:name *expr*)
		$\mid$	H
Patterns	S	$::=$	$\cdots$
		$\mid$	(EH ... . S)
List patterns	L	$::=$	$\cdots$
		$\mid$	(EH ... . L)

Figure 10. Ellipsis-head patterns

5.4 Ellipsis-head patterns

Ellipsis-head patterns—specified in figure 5.4 are the final ingredient necessary to specify syntax like the keyword options of **define-struct**. An ellipsis-head pattern may have multiple alternatives combined with **~or**; each alternative is a head pattern. It specifies

```
(define-struct name:identifier (field:identifier ...)
  (~or (~optional (~seq #:mutable) #:name "mutable clause")
       (~optional (~seq #:super super-expr) #:name "super clause")
       (~optional (~or (~seq #:inspector inspector-expr)
                       (~seq #:transparent))
                  #:name "inspector or transparent clause")
       (~seq #:property pkey:expr pval:expr))
  ...))
```

Figure 11. syntax-parse pattern for **define-struct**

sequences consisting of some number of instances of the alternatives joined together. An alternative may be annotated with one of two *repetition constraint* 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 provide enough power to specify **define-struct**'s syntax. Figure 11 shows the complete pattern. After the fields come the keyword options, in any order. Keywords and their arguments are grouped together with **~seq** patterns. Many 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.

6. Semantics

The **syntax-parse** matching algorithm is based on two principles:

- Errors are selected from all failures based on progress.
- Errors are described using explicitly-provided descriptions.

This section presents the semantics of pattern matching in **syntax-parse** and explains how it implements the two principles. The error selection algorithm is represented by a backtracking monad with a notion of failure that incorporates matching progress. The error description principle is implemented by the semantic functions, which propagate error descriptions as an inherited attribute.

6.1 Tracking failure

We model backtracking with failure information with a "single-elimination" monad, a variant of well-known backtracking monads [Hughes 1995]. A single-elimination (SE) sequence consists of a finite list of successes (a_i) terminated by at most one failure (ϕ):

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

The monad is parameterized by the type of success elements; see below. The sequences of successes may be empty. For simplicity we always include the failure and use • to represent "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 •) consists of a progress (π) together with a set of reasons (ℓ). Each reason consists of a term and a message. When sequences are combined, their failures are joined: (1) the failure with the greatest progress (see figure 5) is selected; (2) if they have the same progress, their message sets are combined. The identity element is •; it is considered to have less progress than any other failure. Failure is a bounded join-semilattice with least element •.

Figure 12 defines the monad's operations, including unit, bind (written $\star$), and disjoin (written $\hat{\ }$). The unit operation creates a

$$
\begin{array}{llll}
\text{SE(A)} & se & ::= \langle a_1, \cdots, a_n; \phi\rangle & \text{where } a_i \in A \\
\text{Failure} & \phi & ::= \bullet \mid \textsc{Fail}(\pi, \{\ell_1, \cdots, \ell_n\}) & \\
\text{Progress} & \pi & ::= \epsilon \mid \pi \cdot \textsc{First} \mid \pi \cdot \textsc{Rest} \mid \pi \cdot \textsc{Late} & \\
\text{Reason} & \ell & ::= (z, msg) & \\
\text{Message} & msg & &
\end{array}
$$

$$
\begin{array}{rcl}
\text{unit}(a) & = & \langle a; \bullet\rangle \\
\text{fail}(\pi, \ell) & = & \langle\, ; \textsc{Fail}(\pi, \{\ell\})\rangle \\
\langle a_1, \cdots, a_n; \phi\rangle \star f & = & f(a_1) \mathbin{\hat{}} \cdots \mathbin{\hat{}} f(a_n) \mathbin{\hat{}} \langle\, ; \phi\rangle
\end{array}
$$

$$
\begin{array}{rl}
& \langle a_1, \cdots, a_k; \phi_1\rangle \mathbin{\hat{}} \langle a_{k+1}, \cdots, a_n; \phi_2\rangle \\
= & \langle a_1, \cdots, a_k, a_{k+1}, \cdots, a_n; \phi_1 \vee \phi_2\rangle
\end{array}
$$

Figure 12. Single-elimination sequences and operations

sequence of one success and no failure. Disjoin (ˆ) concatenates successes and joins (∨) the failures, and bind (⋆) applies a function to all successes in a sequence and combines the resulting sequences with the original failure. This monad is similar to the standard list monad except for the way it handles failures.

One might expect to use the simpler model of a list of successes *or* a failure. After all, if a pattern succeeds, backtracking typically occurs only 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. Furthermore, our choice has two advantages over the seemingly simpler model. First, ranking failures purely by progress is compelling and easy for programmers to understand. Second, this monad corresponds neatly to a two-continuation implementation [Wand and Vaillancourt 2004].

6.2 Domains and signatures

We explain pattern matching on a core version of the pattern language. The colon shorthand for annotated pattern variables is desugared into the ˜**var** form. Similarly, all datum patterns are given as explicit ˜**datum** patterns. All ˜**and** and ˜**or** patterns are converted to have exactly two sub-patterns; ˜**and** patterns must be left-associated so that any action patterns in the original ˜**and** pattern occur as second sub-patterns of the desugared ˜**and** patterns. The disjuncts of core ˜**or** patterns all bind the same attributes; additional bindings via ˜**and** and ˜**parse** are added as necessary to make "absent" attributes explicit.

We generalize the repetition constraint forms ˜**optional** and ˜**once** 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. We omit the ellipsis nesting depth of attributes; it is a static property and as such easy to compute separately.

Syntax classes take a single parameter and references to syntax classes are updated accordingly. The syntax class's variants are combined into a single ˜**or** pattern, which is wrapped with a ˜**describe** pattern holding the syntax class's description.

Finally, we assume an eval function for evaluating expressions. The environment of evaluation is a substitution with mappings for attributes encountered previously in the pattern matching process. For simplicity, we do not model the environment corresponding to the program context. It would be easy but tedious to add.

Figure 13 defines the additional domains and operations used by the semantics as well as the signatures of the denotation functions. Terms consist of atoms and "dotted pairs" of terms. Parsing success is represented by a substitution σ mapping names to terms. Substitutions are combined by the $\sqcup$ operator, which produces a substitution with the union of the two arguments' attribute bindings. We overload the combination operator notation; when the right-hand side is a SE-sequence, it indicates that the left-hand substitution is combined with every substitution in the sequence.

$$
\begin{array}{lll}
\text{Term} & z & ::= x \mid \mathit{datum} \mid \texttt{()} \mid \texttt{(}z_1 \;\texttt{.}\; z_2\texttt{)} \\
\text{Substitution} & \sigma, \rho & ::= \{x_1 \mapsto z_n, \cdots, x_n \mapsto z_n\}
\end{array}
$$

$$
\begin{array}{rcl}
\sigma \sqcup \langle \sigma_1, \cdots, \sigma_n; \phi\rangle & = & \langle \sigma \sqcup \sigma_1, \cdots, \sigma \sqcup \sigma_n; \phi\rangle \\
\mathcal{S}\llbracket S\rrbracket^{\rho}_{\Delta} z\pi\ell & : & \text{SE(Substitution)} \\
\mathcal{A}\llbracket A\rrbracket^{\rho}_{\Delta} \pi\ell & : & \text{SE(Substitution)} \\
\mathcal{H}\llbracket H\rrbracket^{\rho}_{\Delta} z\pi\ell & : & \text{SE(Substitution, Term, Progress)}
\end{array}
$$

Figure 13. Domains, operations, signatures for pattern semantics

The pattern denotation functions are parameterized over a set of syntax definitions Δ and a substitution ρ from patterns already matched. In addition to the appropriate patterns, the denotation functions take up to three additional arguments: a term (z) to parse, a progress string (π), and a failure reason (ℓ). The term and progress arguments change as the matching algorithm descends into the term. The term argument is not needed, however, for action patterns. The reason argument represents the closest enclosing description; it changes when matching passes into a ˜**describe** form.

Each of the pattern denotation function returns a SE-sequence representing successes and failure. The $\mathcal{S}$ and $\mathcal{A}$ functions return sequences whose success elements are substitutions. The $\mathcal{H}$ function additionally includes terms and progress strings, which indicate where to resume matching.

$$
\begin{array}{l}
\mathcal{S}\llbracket(\tilde{}\,\textbf{var}\ x)\rrbracket^{\rho}_{\Delta} z\pi\ell \\
\;= \text{unit}(\{x \mapsto z\}) \\
\mathcal{S}\llbracket(\tilde{}\,\textbf{var}\ x\ (c_{\mathrm{S}}\ e))\rrbracket^{\rho}_{\Delta} z\pi\ell \\
\;= \mathcal{S}\llbracket S\rrbracket^{\{y \mapsto \text{eval}(e,\rho)\}}_{\Delta} z\pi\ell \star \lambda\sigma.\ \text{pfx}(x, \sigma) \sqcup \text{unit}(\{x \mapsto z\}) \\
\qquad \text{where } \{c_{\mathrm{S}}(y) = S\} \in \Delta \\
\mathcal{S}\llbracket(\tilde{}\,\textbf{datum}\ d)\rrbracket^{\rho}_{\Delta} z\pi\ell \\
\;= \begin{cases} \text{unit}(\emptyset) & \text{when } z = d \\ \text{fail}(\pi, (z, \text{“expected } d\text{”})) & \text{otherwise} \end{cases} \\
\mathcal{S}\llbracket(S_1\ .\ S_2)\rrbracket^{\rho}_{\Delta} z\pi\ell \\
\;= \begin{cases} \mathcal{S}\llbracket S_1\rrbracket^{\rho}_{\Delta} z_1 (\pi\cdot\textsc{First})\ell \\ \star\ \lambda\sigma.\ \sigma \sqcup \mathcal{S}\llbracket S_2\rrbracket^{\rho\sqcup\sigma}_{\Delta} z_2 (\pi\cdot\textsc{Rest})\ell \\ \qquad \text{when } z = \texttt{(}z_1\ \texttt{.}\ z_2\texttt{)} \\ \text{fail}(\pi, \ell) \quad \text{otherwise} \end{cases} \\
\mathcal{S}\llbracket(\tilde{}\,\textbf{and}\ S_1\ S_2)\rrbracket^{\rho}_{\Delta} z\pi\ell \\
\;= \mathcal{S}\llbracket S_1\rrbracket^{\rho}_{\Delta} z\pi\ell \star \lambda\sigma.\ \sigma \sqcup \mathcal{S}\llbracket S_2\rrbracket^{\rho\sqcup\sigma}_{\Delta} z\pi\ell \\
\mathcal{S}\llbracket(\tilde{}\,\textbf{and}\ S_1\ A_2)\rrbracket^{\rho}_{\Delta} z\pi\ell \\
\;= \mathcal{S}\llbracket S_1\rrbracket^{\rho}_{\Delta} z\pi\ell \star \lambda\sigma.\ \sigma \sqcup \mathcal{A}\llbracket A_2\rrbracket^{\rho\sqcup\sigma}_{\Delta} \pi\ell \\
\mathcal{S}\llbracket(\tilde{}\,\textbf{or}\ S_1\ S_2)\rrbracket^{\rho}_{\Delta} z\pi\ell \\
\;= \mathcal{S}\llbracket S_1\rrbracket^{\rho}_{\Delta} z\pi\ell \mathbin{\hat{}} \mathcal{S}\llbracket S_2\rrbracket^{\rho}_{\Delta} z\pi\ell \\
\mathcal{S}\llbracket(\tilde{}\,\textbf{describe}\ e\ S)\rrbracket^{\rho}_{\Delta} z\pi\ell \\
\;= \mathcal{S}\llbracket S\rrbracket^{\rho}_{\Delta} z\pi(z, \mathit{eval}(\rho, e)) \\
\mathcal{S}\llbracket(H_1\ .\ S_2)\rrbracket^{\rho}_{\Delta} z\pi\ell \\
\;= \mathcal{H}\llbracket H_1\rrbracket^{\rho}_{\Delta} z\pi\ell \star \lambda(\sigma, z', \pi').\ \sigma \sqcup \mathcal{S}\llbracket S_2\rrbracket^{\rho\sqcup\sigma}_{\Delta} z'\pi'\ell
\end{array}
$$

Figure 14. Semantics of S-patterns

6.3 Meaning

A **syntax-parse** expression has the following form:

(**syntax-parse** *stx* [S_1 *rhs*$_1$] … [S_n *rhs*$_n$])

The meaning of the **syntax-parse** expression is defined via the following denotation:

$$
\begin{array}{l}
\mathcal{S}\llbracket S\rrbracket^{\emptyset}_{\Delta} z\epsilon\ell \\
\quad \text{where } \textit{result} \text{ is fresh with respect to } S, \Delta \\
\qquad\quad S = (\text{~}\textbf{or}\ (\text{~}\textbf{and}\ S_1\ (\text{~}\textbf{parse}\ \textit{result}\ \textit{rhs}_1)) \cdots \\
\qquad\qquad\qquad (\text{~}\textbf{and}\ S_n\ (\text{~}\textbf{parse}\ \textit{result}\ \textit{rhs}_n))) \\
\qquad\quad z = \text{eval}(\textit{stx}, \emptyset) \\
\qquad\quad \ell = (z, \text{“bad syntax”})
\end{array}
$$

If the sequence contains at least one substitution, the result of the **syntax-parse** expression is the *result* attribute of the first substitution in the sequence. Otherwise, the **syntax-parse** expression fails with an error message derived from the SE-sequence's failure.

Figure 14 shows the denotations of single-term patterns. A variable pattern always matches, and it produces a substitution mapping the pattern variable to the input term. A class pattern matches according to the pattern recorded in the syntax class environment Δ. The resulting substitutions' attributes are prefixed (pfx) with the pattern variable, and the pattern variable binding itself is added.

When a **~datum** pattern fails, it synthesizes an error message based on the expected datum. The other pattern variants use the inherited error reason (ℓ), which represents the closest enclosing description around the pattern. That is, it represents the nearest "explainable" frame in the matching context.

The pair, head, and **~and** patterns propagate the success substitutions from their first sub-patterns to their second sub-patterns. This allows expressions within patterns to refer to attributes bound by previous patterns. Head patterns also produce a term and progress string in addition to each success substitution; the term and progress indicate where to resume matching.

$$
\begin{array}{l}
\mathcal{A}\llbracket(\text{~}\textbf{parse}\ S\ e)\rrbracket^{\rho}_{\Delta}\pi\ell \\
\ = \ \mathcal{S}\llbracket S\rrbracket^{\rho}_{\Delta}(\text{eval}(e,\rho))(\pi\cdot\text{L}\small{\text{ATE}})\ell \\
\mathcal{A}\llbracket(\text{~}\textbf{fail}\ e_{cond}\ e_{msg})\rrbracket^{\rho}_{\Delta}\pi\ell \\
\ = \ \begin{cases} \text{fail}(\pi, (v, \text{eval}(e_{msg},\rho))) \\ \qquad \text{if } v \text{ is a true value, where } v = \text{eval}(e_{cond},\rho) \\ \text{unit}(\emptyset) \qquad \text{otherwise} \end{cases} \\
\mathcal{A}\llbracket(\text{~}\textbf{late}\ A)\rrbracket^{\rho}_{\Delta}\pi\ell \\
\ = \ \mathcal{A}\llbracket A\rrbracket^{\rho}_{\Delta}(\pi\cdot\text{L}\small{\text{ATE}})\ell
\end{array}
$$

Figure 15. Semantics of A-patterns

Action patterns, unlike other kinds of patterns, do not depend on the term being matched. Like single-term patterns, however, they produce records. Figure 15 displays the denotations of action patterns. The **~parse** pattern evaluates its sub-expression to a term and matches that term against the sub-pattern. The **~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. The **~late** form extends the progress string, marking the enclosed pattern as a late check.

A **~seq** pattern matches a sequence of terms if the embedded list pattern would match the compound term consisting 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, (**~end-of-head**), that sneaks the additional information into the substitution. 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 (take). We convert the second conjunct from a head pattern to a single-term pattern and use it to match the new term.

We omit the semantics of ellipsis patterns. It is similar to the semantics of head patterns, but an ellipsis-head pattern additionally

$$
\begin{array}{l}
\mathcal{H}\llbracket(\text{~}\textbf{seq}\ .\ L)\rrbracket^{\rho}_{\Delta}z\pi\ell \\
\ = \ \mathcal{S}\llbracket S\rrbracket^{\rho}_{\Delta}z\pi\ell \star \lambda\sigma.\ (\sigma - \{\texttt{pr},\texttt{term}\}, \sigma(\texttt{pr}), \sigma(\texttt{term})) \\
\qquad\quad \text{where } S = \text{rewrite-L}(L) \\
\mathcal{S}\llbracket(\text{~}\textbf{end-of-head})\rrbracket^{\rho}_{\Delta}z\pi\ell \\
\ = \ \text{unit}(\{\texttt{pr} = \pi, \texttt{term} = z\}) \\
\mathcal{H}\llbracket(\text{~}\textbf{and}\ H_1\ H_2)\rrbracket^{\rho}_{\Delta}z\pi\ell \\
\ = \ \mathcal{H}\llbracket H_1\rrbracket^{\rho}_{\Delta}z\pi\ell \star \lambda(\sigma, z', \pi').\ \sigma \sqcup \mathcal{S}\llbracket S_2\rrbracket^{\rho}_{\Delta}(\text{take}(z,\pi,\pi'))\pi\ell \\
\qquad\quad \text{where } S_2 = (H_2\ .\ ()) \\
\mathcal{H}\llbracket(\text{~}\textbf{or}\ H_1\ H_2)\rrbracket^{\rho}_{\Delta}z\pi\ell \\
\ = \ \mathcal{H}\llbracket H_1\rrbracket^{\rho}_{\Delta}z\pi\ell\ \hat{}\ \mathcal{H}\llbracket H_2\rrbracket^{\rho}_{\Delta}z\pi\ell \\
\mathcal{H}\llbracket(\text{~}\textbf{var}\ x\ (c_{\text{H}}\ e))\rrbracket^{\rho}_{\Delta}z\pi\ell \\
\ = \ \mathcal{H}\llbracket H\rrbracket^{\{y\mapsto\text{eval}(e,\rho)\}}_{\Delta}z\pi\ell \star \text{f} \\
\qquad\quad \text{where } \{c_{\text{H}}(y) = H\} \in \Delta \\
\qquad\qquad\quad \text{f}(\sigma, z', \pi') = \text{unit}(\text{g}(\sigma,\pi'), z', \pi') \\
\qquad\qquad\quad \text{g}(\sigma,\pi') = \{x \mapsto \text{take}(z,\pi,\pi')\} \sqcup \text{pfx}(x,\sigma)
\end{array}
$$

`pr`, `term` do not appear in the pattern

$$
\begin{array}{rcl}
\text{rewrite-L}(()) & = & (\text{~}\textbf{end-of-head}) \\
\text{rewrite-L}((S_1\ .\ L_2)) & = & (S_1\ .\ \text{rewrite-L}(L_2)) \\
\text{rewrite-L}((H_1\ .\ L_2)) & = & (H_1\ .\ \text{rewrite-L}(L_2)) \\
\text{rewrite-L}((EH_1\ \ldots\ .\ L_2)) & = & (EH_1\ \ldots\ .\ \text{rewrite-L}(L_2))
\end{array}
$$

Figure 16. Semantics of H-patterns

yields a repetition environment mapping a **~between** form to the number of times it has occurred in the sequence so far. A **~between** form's lower bound is checked when matching proceeds to the tail; its upper bound is checked on every iteration of the head pattern.

6.4 Implementation

The implementation of **syntax-parse** uses a two-continuation representation of the backtracking monad. The success continuation is represented as an expression where possible, so that substitutions are represented in Racket's environment rather than as a data structure. Thus, the code is similar to the backtracking-automaton method of compiling pattern matching. We have not yet attempted to add known pattern-matching optimizations to our implementation but plan on doing so. Optimizations must be adapted to accommodate progress tracking. For example, exit optimization [Fessant and Maranget 2001] may not skip a clause that cannot succeed if the clause may fail with greater progress than the exiting clause.

7. Case studies

Racket has included **syntax-parse** for one year. Reformulating existing macros with **syntax-parse** can cut parsing code by several factors without loss in quality in error reporting. Users confirm that **syntax-parse** makes it easy to write macros for complex syntax. The primary benefit, however, is increased clarity and robustness.

This section presents two case studies illustrating applications of **syntax-parse**. The case studies are chosen from a large series to span the spectrum of robustness; the first case study initially performed almost no error checking, whereas the second case study checked errors aggressively. Each case study starts with a purpose statement, followed by an analysis of the difference in behavior and a comparison of the two pieces of code.

7.1 Case: loop

The **loop** macro [Shivers 2005] allows programmers to express a wide range of iteration constructs via *loop clauses*. The **loop** macro is an ideal case study because the existing implementation performs almost no error-checking, and its author makes the following claim:

> It is frequently the case with robust, industrial-strength software systems for error-handling code to dominate the line counts; the loop package is no different. Adding the code to provide careful syntax checking and clear error messages is tedious but straightforward implementation work.
>
> Olin Shivers, 2005

In other words, adding error-checking to the **loop** macro is expected to *double* the size of the code. Using **syntax-parse** we can do better.

The original **loop** macro performs little error checking; in thirty-two exported macros there are only three syntax validation checks plus a handful of internal sanity checks. The exported macros consist of the **loop** macro itself plus thirty-one CPS macros [Hilsdale and Friedman 2000] for loop clauses such as **for** and **do**.

CPS macros pose challenges for generating good error messages because the macro's syntax differs from the syntax apparent to the user due to the CPS protocol. When the programmer writes (**for** *x* **in** *xs*), the **loop** macro rewrites it as (**for** (*x* **in** *xs*) *k kargs*) to accommodate the macro's continuation. Errors in the programmer's use of **for** should be reported in terms of the original syntax, not the rewritten syntax. We accomplish this by parsing the syntax in two passes. We parse the CPS-level syntax and reconstruct the original term, and then we parse that term. Twenty of the CPS macros are expressed using **define-simple-syntax**, a simplified version of **define-syntax**. We changed **define-simple-syntax** to automatically rewrite these macros' patterns to perform two-stage parsing; we also changed them to use **syntax-parse** internally so that the simple macros could use annotations and the other features of our system. The other eleven CPS macros were transformed by hand.

Another hazard of CPS macros is inadvertent transfer of control to a macro that does not use the CPS protocol, resulting in incoherent errors or unexpected behavior. In Racket, this problem can be prevented by registering CPS macros and checking their applications. We use a syntax class to recognize registered CPS macros.

Once the concrete syntax is separated from the CPS-introduced syntax, validating it is fairly simple. Many of the loop forms take only expressions, so validation is trivial. Some of the loop forms require *identifier* annotations or simple side conditions. The **initial** and **bind** loop forms have more structured syntax, so we define syntax classes for their sub-terms, including a shared syntax class *var/vars*; it represents a single variable or a group of variables.

A loop-clause keyword such as **for** is implemented by a macro named **loop-keyword/for**; the name is chosen to reduce contention for short names. The **loop** macro rewrites the loop-clause keywords, except that programmers can write the long form in parentheses, e.g., ((**loop-keyword/for**) *x* **in** *xs*), to avoid the rewriting. The code to recognize and rewrite both cases and is duplicated, since **for** enforces the same protocol for its auxiliaries: **in** becomes **for-clause/in**. In the **syntax-parse** version, we define a *loopkw* syntax class that does the rewriting automatically. The syntax class is parameterized so it can handle both **loop** and **for** keywords.

The original version of the **loop** macro consists of 1840 lines of code, not counting comments and empty lines. The implementation of the loop keyword macros takes 387 lines; the rest includes the implementation of its various intermediate languages and scope inference for loop-bound variables. The **syntax-parse** version is 1887 lines, an increase of forty-seven lines. The increase is due to the new version of **define-simple-syntax**. Overall, the increase is 12% of the size of the main body of the macros and merely 2.6% of entire code, which falls far short of the 100% increase predicted by the package's highly experienced author. Aside from the new helper macro, the parsing code shrank, *despite much improved error handling*, due to simplifications enabled by **syntax-parse**.

7.2 Case: parser

The **parser** macro [Owens et al. 2004] implements a parser generator for LALR(1) grammars. The macro a grammar description and a few configuration options, and it generates a table-driven parser or a list of parsers, if multiple start symbols are given. The **parser** case study represents macros with aggressive, hand-coded error reporting. The macro checks both shallow properties as well as context-dependent constraints.

The **parser** macro takes a sequence of clauses specifying different aspects of the parser. Some clauses are mandatory, such as the **grammar** clause, which contains the list of productions, and the **tokens** clause, which imports terminal descriptions. Others are optional, such as the **debug** clause, which specifies a file name where the table descriptions should be printed. In all, there are ten clauses, five mandatory and five optional, and they can occur in any order.

The original version used a loop and mutable state to recognize clauses; different clauses were parsed at various points later in the macro's processing. The new version uses our improved ellipses patterns in two well-defined passes to resolve dependencies between clauses. For example, the productions in the **grammar** clause depend on the terminals imported by the **tokens** clause. The second pass involves syntax classes parameterized over the results gathered from the first pass.

The original version of **parser** explicitly detects thirty-nine different syntax errors beyond those caught by MBE-style patterns. Repetition constraints (**~once** and **~optional**) on the different clause variants cover thirteen of the original errors plus a few that the original macro failed to check. Pattern variable annotations cover eleven of the original errors, including simple checks such as "Debugging filename must be a string" as well as context-dependent errors such as "Start symbol not defined as a non-terminal." The latter kind of error is handled by a syntax class that is parameterized over the declared non-terminals. Side-condition checks cover eight errors—such as "duplicate non-terminal definition"— with the use of `#:fail-when`.

The remaining seven checks performed by the original macro belong to catch-all clauses that explain what valid syntax looks like for the given clause or sub-form. Five of the catch-all checks cover specific kinds of sub-forms, such as "Grammar must be of the form (**grammar** (*non-terminal productions* ...) ...)." In a few cases the message is outdated; programmers who revised the **parser** macro failed to update the error message. In the **syntax-parse** version each of these sub-forms is represented as a syntax class, which automatically acts as a local catch-all according to our error message generation algorithm (section 4.2); **syntax-parse** reports the syntax class's description rather than reciting the macro's documentation. (A macro writer could put the same information in the syntax class description, if they wanted to.) The final two checks are catch-alls for parser clauses and the parser form itself. These are implemented using **~fail** and patterns crafted to catch clauses that do not match other clause keywords.

In most cases the error messages are rephrased according to **syntax-parse** conventions. For example, where the original macro reported "Multiple grammar declarations," the new macro uses "too many occurrences of grammar clause"; and where the original macro reported "End token must be a symbol," the new macro produces the terser message "expected declared terminal name."

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(1) 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. In the new version, parsing and processing took a total of 378 lines of code, consisting of 124 lines for parsing (25 for the main macro pattern and 99 for syntax class definitions) and 254 lines for processing.

By reasoning that the lines dedicated to processing should be roughly equivalent in both versions, we estimate 300 lines for processing in the original version, leaving 270 for parsing. Thus the **syntax-parse** version requires less than half the number of lines of code for parsing, and the new parsing code consists of modular, declarative specifications. The error reporting remains of comparable quality.

8. Related work

Other backtracking parsers, such as packrat parsers [Ford 2002], also employ the technique of tracking and ordering failures. Unlike shift/reduce parsers, which enjoy the viable-prefix property, packrat parsers cannot immediately recognize when an input stream becomes nonviable—that is, where the error occurs. Instead, they maintain a high-water mark, the failure that occurs furthest into the input along all branches explored so far. While these string parsers can represent progress as the number of characters or tokens consumed, **syntax-parse** uses a notion of progress based on syntax tree traversal.

Our ordering of parse failures is also similar to the work of Despeyroux [1995] on partial proofs in logic programming. 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 original rules are incomparable; only the maximal proofs are returned. In contrast to the order of that system, which is indifferent to the system's rules and propositions, our system uses the pragmatics of parsing syntax to define the order.

Another line of research in macro specifications began with static checking of syntactic structure [Culpepper and Felleisen 2004] and evolved to encompass binding information and hygienic expansion [Herman and Wand 2008]. These systems, however, are incapable of fortifying 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.

9. Conclusion

Our case studies, our other experiences, and reports from other programmers confirm that **syntax-parse** makes it easy to write easy-to-understand, robust macros. Overall **syntax-parse** macros take less effort to formulate than comparable macros in MBE-based systems such as **syntax-case** and **syntax-rules** or even plain Lisp-style macros. Also in contrast to other macro systems, the **syntax-parse** style is distinctively declarative, closely resembling grammatical specification with side conditions. Best of all, these language extensions are translated into implementations that comprehensively validate all the constraints and that report errors at the proper level of abstraction. Even though **syntax-parse** has been available for less than a year, it has become clear that it improves on MBE-style macros to the same degree—or perhaps a larger one—that MBE improved over Lisp-style macros.

Acknowledgments We are grateful to Matthew Flatt, Guy Steele, Sam Tobin-Hochstadt, and Jon Rafkind for feedback on the design and implementation of **syntax-parse**.

References

Cadence Research Systems. *Chez* Scheme Reference Manual, 1994.

R. Culpepper and M. Felleisen. Taming macros. In *International Conference on Generative Programming and Component Engineering*, pages 225–243, 2004.

T. Despeyroux. Logical programming and error recovery. In *Industrial Applications of Prolog*, Oct. 1995.

R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, Dec. 1993.

F. L. Fessant and L. Maranget. Optimizing pattern matching. In *International Conference on Functional Programming*, pages 26–37, 2001.

R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

M. Flatt and PLT. Reference: Racket. Technical report, PLT Inc., January 2010. `http://racket-lang.org/tr1/`.

B. Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, Sept. 2002.

D. Herman and M. Wand. A theory of hygienic macros. In *European Symposium on Programming*, pages 48–62, Mar. 2008.

E. Hilsdale and D. P. Friedman. Writing macros in continuation-passing style. In *Workshop on Scheme and Functional Programming*, pages 53–59, 2000.

J. 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.

E. E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Symposium on Principles of Programming Languages*, pages 77–84, 1987.

P. J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101, 1965.

S. Owens, M. Flatt, O. Shivers, and B. McMullan. Lexer and parser generators in Scheme. In *Workshop on Scheme and Functional Programming*, pages 41–52, Sept. 2004.

O. Shivers. The anatomy of a loop: a story of scope and control. In *International Conference on Functional Programming*, pages 2–14, 2005.

M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews. Revised^6 report of the algorithmic language Scheme. *Journal of Functional Programming*, 19(S1):1–301, Aug. 2009.

M. Wand and D. Vaillancourt. Relating models of backtracking. In *International Conference on Functional Programming*, pages 54–65, 2004.