JFP **22** (4–5): 439–476, 2012. © Cambridge University Press 2012
doi:10.1017/S0956796812000275

Fortifying macros

RYAN CULPEPPER*
School of Computing, Salt Lake City, Utah, USA
(*e-mail:* `ryanc@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.

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.

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

* Part of this research was conducted while the author was a PhD student at Northeastern University.

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—macros only extend the definition and expression forms of the language, and a macro name must occur in operator position or alone, like a variable reference—extensions to the language easily compose. Since extensions are anchored to names, extensions can be managed by controlling the scope of their names. Language extensions can be organized into modular libraries. These syntax libraries can contribute to the *target* language of new macros; for example, the *match* pattern-matching form expands into a use of the **cond** conditional form, which itself expands into a nested sequence of **if** forms. Syntactic extensions can also be used to enrich the *implementation* language of new macros, allowing programmers to express syntax transformations in novel ways. These two forms of extension allow programmers to construct towers of languages one layer at a time (Queinnec, 1996; Flatt, 2002).

Introducing new language elements via macros has long been a standard element of every Lisper's and Schemer's repertoire. Racket (Flatt & PLT, 2010), formerly PLT Scheme, is a descendant 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. Syntax errors should directly reflect the programmer's mistakes. In particular, a syntax error should refer to terms that the programmer wrote, not the product of several rounds of rewriting; and furthermore, the mistake should be reported in terms and concepts documented by the language. Languages built with macros are held to the same standards as Racket itself.

Sadly, existing systems make it surprisingly hard to produce easy-to-understand macros that properly validate their syntax. They force the programmer to mingle the declarative specification of syntax and semantics with highly detailed validation code. Without validation, however, macros are not true abstractions. Instead, erroneous terms flow through the parsing process until they eventually trip over constraint checks at the lower levels of the language tower. Low-level checking, in turn, yields incoherent error messages referring to terms the programmer did not write and concepts that do not apply to the programmer's code. In short, such macros are sources of confusion and distraction, not seamless linguistic abstractions.

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.

The remainder of this paper is organized as follows. Section 2 introduces Racket syntax. Section 3 discusses how macros are expressed and shows that current methods of error checking are cumbersome and verbose. Section 4 introduces our new macro system by showing how it solves the problems raised in Section 3. Section 5 explains the model our system uses to identify and report syntax errors. Section 6 defines the expressive pattern language our macro system is built on. Section 7 briefly discusses

Term	*z*	::=	*datum* | *id* | (*z*$_1$. *z*$_2$)
Datum	*datum*	::=	() | number | string | boolean | keyword
Identifier	*id*		
Definition	*def*	::=	(**define** *id expr*)
		|	(**define** (*id* . *formals*) *def** *expr*$^+$)
		|	(**define-syntax** *id expr*)
		|	(**define-syntax** (*id* . *formals*) *def** *expr*$^+$)
		|	...
Expression	*expr*	::=	number | string | boolean | (**quote** *term*) | #'*term*
		|	*id* | (**λ** *formals def** *expr*$^+$) | (*expr expr**) | ...
Formals	*formals*	::=	() | *id* | (*id* . *formals*) | ([*id expr*] . *formals*)
		|	(*keyword id* . *formals*) | (*keyword* [*id expr*] . *formals*)

Fig. 1. Racket syntax.

our implementation, and Section 8 discusses our experiences using it. Section 9 discusses related work.

2 Preliminaries

Racket programs are written with *terms* consisting primarily from atoms and pairs.[1] Figure 1 presents the simple subset of Racket's syntax we use in this paper. A Racket program's concrete syntax is *read* into tree-structured representations of terms called *syntax objects*; macro expansion and compilation manipulate syntax objects, not raw program text or token streams. We use the term *parsing* to refer to a macro's process of validating and destructuring its input syntax; this paper is primarily concerned with that process.

Atoms include identifiers such as **define**, **λ**, *xyz*, +, and *string→number* (written `define`, λ, `xyz`, +, and `string->number`, respectively). Atoms also include numbers like 5, −7.2, and 3+4i; strings like `"hello"`, `"goodbye"` and `"I am the walrus"`; the booleans #t (true) and #f (false); and keywords such as #:super and #:property. Note that keywords are distinct from identifiers; we refer to significant identifiers such as **define** and **λ** not as "keywords" but instead as "special forms."

Compound terms are formed using "dotted pairs," although the dots seldom appear explicitly. Racket treats the term (*x y z*) as equivalent to (*x* . (*y* . (*z* . ())));
the latter notation makes the structure of the term explicit. Although the dotted notation occurs rarely in programs, it is frequently useful in syntax specifications. For example, consider the syntax of function definitions (the second **define** case) and the specification of *formals*. If the formals "list" ends in an identifier rather than (), the identifier is a "rest" parameter that gets bound to a list of all leftover arguments. Some examples of function definitions are

[1] The same is also true of other Lisp dialects. For the rest of the paper, we do not comment on whether a property of Racket is shared with other dialects unless the point is particularly relevant.

```
(define (greet) (display "hello"))
(define (twice x) (+ x x))
(define (max #:compare [compare >=] x0 . xs)
  (define (max2 a b) (if (compare a b) a b))
  (foldr max2 x0 xs))
```

Square brackets are treated as equivalent to parentheses; their use is purely a matter of convention and readability.

The specifications of *def* and *expr* are both open-ended, because they can be extended using macros. A macro does not declare that it specifically extends *def* or *expr*; macro expansion is performed in both contexts, and if a macro produces a definition in an expression context, Racket signals an error. Macros are defined using **define-syntax**, and the #'*term* expression form interprets *term* as a syntax template and uses it to construct a syntax object. Macro definitions and syntax templates are discussed in greater detail in Section 3. In contrast to *def* and *expr*, *formals* is not extensible; macro expansion occurs *only* in *def* and *expr* contexts.

3 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) = ((λ (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 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)))
```

The backquote character (**quasiquote**) introduces an S-expression template; the comma (**unquote**) is an escape that inserts the value of the following expression into the template, and the comma and at-sign combination (**unquote-splicing**) is an escape that includes the contents of the list computed by the following expression.

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 write sloppy macros like these unless their tools make it easy to write robust ones.

One such tool is the Macro-By-Example (MBE) notation by Kohlbecker & 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. When a pattern matches, its *syntax pattern variables* are bound to the corresponding sub-terms of the macro occurrence. These sub-terms are substituted in for occurrences of pattern variables in the corresponding template, and the substituted template is the macro's 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 ...)]))
```

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 sub-terms with homogeneous structure—ellipses can be considered as an S-expression version of the familiar "Kleene star" operator. Homogeneous sequences occur frequently in S-expression syntax. Often the elements of sequences have non-trivial structure, such as the binding pairs associating **let**-bound variables with their values in the code above.

Ellipses are also used in syntax templates to generate sequences of similar terms based on a single template fragment. The repetition must be "driven" by pattern variables occurring under ellipses in the pattern. For example, given the pattern for **let** above, the template (*var* ...) is legal, because *var* occurs before an ellipsis in the pattern, but (*body* ...) would be illegal. The repeated fragment need not be a single pattern variable; the template ((12 *var body*) ...) is also legal; it produces a term for each *var* match, and the value of *body* is included in each result. A pattern variable that occurs under ellipses in the pattern must occur under ellipses in the template; for example, the simple template *rhs* would be illegal.

Ellipses may be nested. The *ellipsis depth* of a pattern variable is the nesting level of ellipses around it. In the **let** macro above, *var* and *rhs* have a depth of 1 and *body* has a depth of 0. In the pattern ((*x* ...) ...), which would match a sequence of sequences of terms, *x* has depth 2. Syntax templates are statically checked to make sure that pattern variables are used at ellipsis depths consistent with the corresponding patterns. See Kohlbecker & Wand (1987) for details on template checking and transcription.

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. In both cases **let** inspects the syntax, successfully matches it against its pattern, and produces an invalid **λ** expression. Then **λ**, implemented as a primitive syntactic form by a careful compiler writer, signals an error. For example, Racket reports "**λ**: duplicate identifier in: *x*" for the first term. Chez Scheme (Dybvig, 2010) reports "invalid parameter list in (**λ** ((*x* *y*)) (*g* *x* *y*))" 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 full solution. Macros should report errors on their own terms, not in terms of their expansions.

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 mischaracterizes the legal syntax of **let**.

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 result 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 2 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 expression, and a result

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

Fig. 2. **let** with guards.

```
(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 ...))]))
```

Fig. 3. **let** with hand-coded error checking.

expression. Pattern variables are used via syntax templates, which are terms marked with a #' prefix; templates are instantiated by substituting in the pattern variables' values. A macro's result expression is typically a syntax template or a computation that ultimately returns a template's value.

The macro in Figure 2 has a guard expression that uses auxiliary syntax templates to refer to just the **let** form's variables: (*andmap identifier?* (*syntax→list* #'(*var* ...))) checks that each "variable" is truly an identifier, and (*not* (*check-duplicate* #'(*var* ...))) ensures that no identifier occurs more than once. The names *andmap*, *identifier?*, *syntax→list*, and *check-duplicate* all refer to functions in Racket's standard library.

Guard expressions suffice to prevent macros from accepting invalid syntax, but they suffer from two flaws. First, since guard expressions are separated from result 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.

To provide precise error explanations, explicit error checking is necessary, as shown in Figure 3. Of the 10 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

```
(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
```

Fig. 4. Syntax of **define-struct.**

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 leaving the nuances of error reporting aside, 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.

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 4. It has several keyword options, which can occur in any order. The #:transparent and #:inspector keywords control how structure values can be inspected via reflection. The #:mutable option makes the fields mutable. The #:property option allows structure types to set behavior such as how they are printed, compared for equality, and so on. Different keywords come with different numbers of arguments, e.g., #:mutable takes none and #:property takes two.

Parsing a **define-struct** form gracefully is simply beyond the capabilities of MBE's pattern language, which only supports 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 ("——" represents elided code):

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

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

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

Fig. 5. Syntax of **syntax-parse**.

In the actual implementation of **define-struct**, the parsing of keyword options alone takes over 100 lines of code. In comparison, the specification of the same syntax in our new system is an order of magnitude shorter.

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.

4 The design of syntax-parse

Our system, dubbed **syntax-parse**, features three significant improvements over MBE:

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

Using **syntax-parse**, a programmer writes declarative specifications of a macro's syntax; these specifications are used to validate a macro's input syntax. If validation succeeds, the syntax has also been parsed and the components bound to pattern variables to use for code generation. If validation fails, **syntax-parse** automatically synthesizes a syntax error, using annotations from the syntax specifications to craft a comprehensible error message.

The declarative framework also accommodates explicit side-condition checks. Side-condition failures are handled just the same as pattern matching failures; a single notion of *progress* determines which failures are mentioned in the final error report. The syntax classes of our system are similar to non-terminals in traditional grammars, but the addition of side conditions enables the disciplined interleaving of pattern-based specifications and hand-coded checks, resulting in more flexibility than a pure grammar-based system.

This section illustrates the design of **syntax-parse** with a series of examples based on the **let** example. The subsequent sections elaborate on the error-reporting model and the specification pattern language.

4.1 Validating syntax

The syntax of **syntax-parse**—specified in Figure 5—is similar to that of **syntax-case**. It matches a syntax object—a representation of a term—against a sequence of clauses. Each clause consists of a pattern, a sequence of auxiliary side clauses, and a result expression. A #:fail-when clause represents a side condition. A #:with clause matches a pattern against a computed term, binding additional pattern variables; an example of a #:with clause appears in Section 4.2.

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 basic shape of the original specification, not the side conditions.

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, so it does not appear in the template.

If the #:fail-when clause's condition expression evaluates to a true value,[2] parsing fails with the given message: "duplicate variable name"; otherwise, the side-condition check passes. Furthermore, if the value produced by the condition expression is a syntax object, that syntax is included as the specific site of the failure. The *check-duplicate* function returns either an identifier—the first duplicate, if one is found—or false, so it is well-suited for use in a #:fail-when check.

In short, side conditions differ from guard expressions in that the failures they generate carry information describing the reasons for the failure. Section 5 explains in greater detail how #:fail-when is superior to explicitly raising an error.

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

```
> (let ([x 1] [[x] 2]) (h x))
let: duplicate variable name in: x
> (let ([[(x y)] (f 7)]) (g x y))
let: expected identifier in: (x y)
```

[2] In Racket, any value other than #f (false) is considered a true value.

The boxes indicate 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 does not provide good error messages. Here is an example that is missing a pair of parentheses:

```
> (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.

4.2 Defining syntax classes

Syntax classes—in particular, their *descriptions*—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." It has a single variant pattern. The pattern variables *var* and *rhs* have moved out of the main pattern into the syntax class, and they are 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 structures. 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 *b*, *var*, and *rhs*—nested attributes such as *b.var* are not exported by a syntax class. The latter two 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.

The macro can now be written as follows:

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

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 though *bs* itself does not occur within ellipses in the macro's pattern.

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

```
> (let ([x] 5) (add1 x))
let: expected binding pair in: x
> (let [17])
let: expected sequence of binding pairs in: 17
```

The next section explains how **syntax-parse** selects errors to report and how descriptions are incorporated into error messages.

5 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 pattern-matching failures (including side-condition failures) to report as the error or errors. Second, it pinpoints the faulty term and uses text from the syntax specifications to explain the fault or describe the expected class of syntax. This section explains the error selection and explanation processes first via examples, then with a semantic model of backtracking that forms the basis for the pattern semantics of Section 6.

5.1 Error selection

Pattern variable annotations and side conditions serve a dual role in our system. As seen, they 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-checking code of Figure 3. A macro can have multiple clauses, and a syntax class can have multiple variants. When there are multiple alternatives, all of them must be attempted before an error is raised.

To illustrate this process, we must introduce choice into our running example. As it happens, Racket inherits Scheme's **let** syntax, which has another variant 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 ([loop (λ (bs.var ...) body)]) (loop 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. These two patterns happen to be mutually exclusive, so the order of the clauses is irrelevant, but in general the clauses are tried in order.

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 useful error message.

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 set 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. Frequently there is a unique maximum, resulting in a single error explanation. Otherwise, multiple explanations are presented.

Figure 6 defines our notion of progress. The structure of terms influences the structure of progress: a progress string is a sequence of progress steps, and 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 written left to right, so for example the second

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

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

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

Fig. 6. Progress.

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 sub-term. 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 side-condition check—in contrast to a pattern-matching check—to occur *after* traversal of the term. We encode this relative ordering by adding a LATE progress step. Thus, while the named-**let** case fails with the progress string REST · FIRST in the example above, the normal-**let** case fails with REST · FIRST · LATE, which is greater progress, and so we report only the duplicate variable name error.

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.

5.2 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 and the specific term where parsing failed.

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. In general, a pattern is ineffable unless it is a datum or literal or has an explicit description or message attached to it. An example of an ineffable pattern is the following:

(*var:identifier rhs:expr*)

If a term such as *x* 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 like each of these. We have chosen not to generate such messages automatically for two reasons: first, they do not scale well to complicated 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 5.1 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:

	Term	Pattern
1	*x*	(*var*:*identifier* *rhs*:*expr*)
2	*x*	*b*:*binding*
3	(*x* 1)	(*b*:*binding* ...) with side constraint
4	(*x* 1)	*bs*:*distinct-bindings*
5	(**let** (*x* 1) (*add1* *x*))	(**let** *bs*:*distinct-bindings* *body*:*expr*)

The first, third, and fifth frames contain ineffable patterns, shown in grey above. We discard the ineffable patterns and extract the descriptions from the syntax classes in the patterns that remain. The result is the following *expectation stack*:

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. A programming environment might permit interactive exploration of the full expectation stack.

When error selection returns a single failure with maximum progress, the expectation stack of that failure is used directly to generate the error message. When multiple maximal failures exist, however, the expectation stacks must be combined. In the simple case, the failures are unrelated, as in the following example:

(**let** $\boxed{5}$)

This use of **let** has two maximal failures, both with progress Rest · First. The first failure corresponds to the named-**let** pattern; it has the following expectation stack:

expected identifier, given 5

The second failure corresponds to the normal-**let** pattern, and it has the following expectation stack:

expected sequence of binding pairs, given 5

The two failures are distinct; neither expectation stack is an extension of the other. Consequently, we form the error message by concatenating the error messages corresponding to the separate failures separated by "or," yielding

expected identifier or expected sequence of binding pairs

Failures are considered distinct even if their expectation stacks share a common tail; in fact, this is common. Consider the following syntax class:

```
(define-syntax-class mutable-option
  #:description "mutability option"
  (pattern #:mutable)
  (pattern #:immutable))
```

The two patterns consist of literal data, so descriptions are automatically generated for them. Thus attempting to parse *x* as a *mutable-option* produces the following two failures:

expected the literal #:mutable, given *x*
expected mutability option, given *x*

and

expected the literal #:immutable, given x
expected mutability option, given x

Despite the shared context, the specific cause of the failure can be summarized by these two possibilities:

expected the literal #:mutable or the literal #:immutable

Of course, a detailed view of the error would likely merge the two failures, making the shared context explicit.

When one of the failures is an extension of the other, the more specific failure is discarded, since it is subsumed by the more general error. For example, consider the following syntax class:

```
(define-syntax-class id-maybe-default
  #:description "identifier with optional default"
  (pattern var:identifier)
  (pattern (var:identifier default:expr)))
```

Attempting to parse the term 5 as a *id-maybe-default* yields two failures with the same progress, ϵ. The first pattern fails with the expectation stack

expected identifier, got 5
expected identifier with optional default, got 5

There are two frames: one for the *id-maybe-default* syntax class and one for the *identifier*-annotated pattern variable. In contrast, the second pattern is ineffable, so its expectation stack is simply

expected identifier with optional default, got 5

If we naively combined the two failures as though they were unrelated, we would produce the error message

expected identifier or identifier with optional default, got 5

But that is redundant; "identifier" is included in "identifier with optional default." There is no benefit in mentioning both a syntax class and one of its specific variants, so we trim the expectation stack to its common tail and report

expected identifier with optional default, got 5

Such pairs of failures, one extending the other, occur often with ellipsis patterns where the base case is a literal () pattern. Handling them naively would result in spurious "... or expected the literal ()" error message fragments.

5.3 Errors and backtracking

We model the error selection algorithm using a monad based on well-known backtracking monads (Hughes, 1995) but specialized to carry failure information. We refer to this monad as the RF ("remember failures") monad.

One representation of the traditional backtracking monad is a list (or stream) of successes; failure is represented as the empty list. In contrast, the corresponding representation of the RF monad is a list of successes together with a set of maximal failures. In the context of pattern matching, a success contains a substitution mapping attributes to their values, and a failure contains a progress string and an expectation stack. Failures are ordered by progress string; only the maximal failures are retained.

This monad differs from the familiar list monad in the way it handles failures. It differs from the exception monad in that *we remember our failures even when we succeed*. We defer explaining our choice of backtracking model to Section 6.5, since it is influenced by the patterns we support.

6 Syntax patterns

The power of **syntax-parse** comes from its expressive pattern language, an extension of the syntax patterns of MBE. Sections 4 and 5 introduced some features of our pattern language. This section describes the pattern language in more detail, including additional pattern forms that enable **syntax-parse** to handle the variety of syntax found in real Racket macros.

The full pattern language is large, partly because it is designed to handle the complexity of Racket syntax. One example is the syntax of **define-struct**'s keyword arguments (Figure 4). The keyword arguments can occur in any order; some of them comprise multiple terms; many of them can occur at most once; and some pairs of keyword arguments are mutually exclusive. Keyword arguments and other uses of constrained repetitions are common enough in Racket that **syntax-parse** is obligated to support them.

There are also redundancies in **syntax-parse**. Some are essentially syntactic sugar; for example, there are pattern equivalents to the #:fail-when and #:with side clauses, and there is a pattern that has the same effect as a syntax class's #:description declaration. There are also partial overlaps between features. For example, some syntaxes can be specified equally well as explicitly recursive syntax classes or as an ellipsis pattern. But neither feature subsumes the other: ellipses patterns cannot handle arbitrary recursive structure, whereas recursive syntax classes can, but ellipsis patterns do allow declarative specification of repetition constraints, which would have to be added to a recursive syntax class using parameters and explicit checks.

There are four categories of patterns: single-term (S), action (A), head (H), and ellipsis-head (EH). We present the pattern language in seven groups, however, organized by common purposes. For each group, we present the syntax of the new pattern forms and explain how they are used. Some examples in early sections rely on ellipsis patterns, which are not explained in detail until Section 6.6; however, the high-level explanation of ellipses from Section 3 should be sufficient to understand the examples.

Single-term patterns	S	::=	x
		|	x :*stxclass*
		|	*datum*
		|	(S . S)
		|	(**~var** x (*stxclass* *expr**))
		|	(**~literal** x)
Expressions	*expr*	::=	· · ·
		|	(**syntax-parse** *expr* [S *sclause** *expr*]*)
Side-clause	*sclause*	::=	#:fail-when *expr* *expr*
		|	#:with S *expr*
Definitions	*def*	::=	· · ·
		|	(**define-syntax-class** *stxclass* *desc*$^?$ *variant**)
		|	(**define-syntax-class** (*stxclass* x^*) *desc*$^?$ *variant**)
Description	*desc*	::=	#:description *expr*
Variant	*variant*	::=	(**pattern** S *side-clause**)

Fig. 7. Basic single-term patterns.

6.1 Basic patterns and syntax classes

The most fundamental kind of pattern is the *single-term pattern*, which specifies a set of individual terms. The clauses of a **syntax-parse** form contain single-term patterns, and the variants of a syntax class are single-term patterns.

Figure 7 presents the basic single-term patterns. The first three variants—pattern variables, annotated pattern variables, pair patterns—are familiar from Section 4. So are datum patterns, in the form of (), which ends compound patterns.[3] Numbers, booleans, strings, and keywords can also be used as datum patterns that match themselves.

A pattern consisting of a single pattern variable binds that pattern variable at ellipsis depth 0. A pattern variable annotated with a syntax class additionally binds *nested attributes*: the attributes of the syntax class prefixed with the pattern variable name. The *attributes* of a pattern are its pattern variables together with their nested attributes.

The attributes of a syntax class are the pattern variables (not including nested attributes) that its patterns have in common. Taking only the non-nested attributes means that a syntax class's attributes can be computed independent of any other syntax class definition, which simplifies the handling of forward references and mutual recursion. Syntax class definitions have both a compile-time part (the attributes bound) and a run-time part (the parser). The compilation of a syntax class's run-time part is delayed until the compile-time parts of all syntax classes in the enclosing module have been gathered.

A pair pattern propagates the attributes from its first sub-pattern to its second sub-pattern. This propagation allows expressions (such as **~var**[4] pattern arguments)

[3] Recall that the notation (x y z) is shorthand for (x . (y . (z . ()))).
[4] All pattern forms start with a tilde (~). This convention makes it easy to distinguish pattern form names from pattern variables.

to refer to attributes bound by previous patterns. Pair patterns, as well as most other composite patterns, require the attributes of their components to be disjoint.

The **~literal** pattern form recognizes identifiers that refer to the same binding as the given literal identifier, as determined by the standard *free-identifier=?* function (Dybvig *et al.*, 1993). A **~literal** pattern binds no attributes. Here is a simplified version of Racket's **cond** form, which uses a literal pattern to recognize a terminal **else** clause, together with an example use.

```
(define-syntax (cond stx)
  (syntax-parse stx
    [(cond [test result] . more) #'(if test result (cond . more))]
    [(cond [(~literal else) result]) #'result]
    [(cond) #'(void)]))
```

```
(cond [(< the-number 10) 'little]
      [else 'big])
```

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:

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

The upper bound is inserted into the check message using the *format* procedure.

Note that a "syntax class application" in a **~var** pattern is not an expression, and a syntax class is not a value, since the attributes bound by a pattern must be known statically.

We can give an alternative definition of *distinct-bindings* via a syntax class parameterized over the identifiers that have already been seen. Figure 8 shows the alternative definition and the auxiliaries *bindings-excluding* and *identifier-except*. 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). Finally, the *identifier-except* syntax class is a trivial combination of *identifier* and a side condition on the *seen* parameter. The *bound-identifier=?* function (Dybvig *et al.*, 1993) tests whether two identifiers would be equivalent as binding occurrences—it is the same notion of equivalence used by *check-duplicate*.

```
(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 (identifier-except 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 (identifier-except seen)
  (pattern x:identifier
           #:fail-when (for/or ([id seen]) (bound-identifier=? #'x id))
                       "duplicate variable name"))
```

Fig. 8. Parameterized syntax classes.

The alternative definition of *distinct-bindings* accepts the same terms as the definition from Section 4.2, but it reports errors differently. 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. Thus they report different errors for the following term:

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

The first version reports "expected binding pair: [*x y z*]" while the alternative in Figure 8 reports "duplicate variable name: *a*." In general, we prefer to report structural errors before context-sensitive errors, and so we prefer the original version. It is also simpler.

6.2 Combining patterns

The **~and** and **~or** pattern forms (Figure 9) add the ability to logically combine patterns.

The **~and** form provides a way of analyzing a term multiple ways. Matching and binding are done left to right within an **~and** pattern, so later sub-patterns can rely on earlier ones, including using their pattern variables. One use of an **~and** pattern is to parse a term while also giving a name to the whole term, typically to extract its source location or use the whole term in a debugging message.

Another use of **~and** patterns is to specify constraints on the parts based on information from the whole; the components of the **~and** form act as separate parsing passes, where information is passed from the first, shallow pass to subsequent, deeper passes. In the parlance of attribute grammars, this use corresponds to an inherited attribute that depends on a synthesized attribute.

```
Single-term patterns  S  ::= ...
                           |  (~and S+)
                           |  (~or S+)
```

Fig. 9. Combining patterns.

For example, consider a **digraph** macro that constructs a directed graph based on node declarations consisting of a node name and the names of its outgoing neighbor nodes. Each neighbor name must be "bound" by some node declaration; so (**digraph** [*a* (*b*)] [*b* (*a* *b*)]) is legal but (**digraph** [*a* (*b*)]) is not. That can be expressed as follows:

```
(define-syntax (digraph stx)
  (syntax-parse stx
    [(~and (_ [left:identifier (_:identifier ...)] ...)
           (_ [_ ((~var right (identifier-in (syntax→list #'(left ...))))) ...)] ...))
     #'——]))
```

The *identifier-in* syntax class is like *identifier-except* (Figure 8) but with the side condition negated.

The **~or** form represents choice; 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 #f (false) for absent attributes. Using **attribute**, a programmer can check if it is safe to use an attribute in a template. Here is a 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)]))
```

Pragmatically, it is usually preferable to specify such syntax using syntax classes with multiple variants. Missing attributes can be bound using #:with clauses, eliminating the need for a condition expression where the attribute is used:

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

Conversely, a multi-variant syntax class can be desugared into a single-variant syntax classes by combining the variants with an **~or** pattern.

```
Single-term patterns  S  ::= ...
                           | (~describe expr S)
```

Fig. 10. Controlling error messages.

```
Action patterns       A  ::= (~fail condition message)
                           | (~parse S expr)
                           | (~late A)
Single-term patterns  S  ::= ···
                           | (~and S {S|A}*)
```

Fig. 11. Action patterns.

6.3 Attaching descriptions to patterns

A **~describe** pattern (Figure 10) attaches a description to its sub-pattern by adding a frame onto the expectation stack of any failure that occurs in its sub-pattern. The frame consists of the given description and the term that is currently being parsed. Thus if a failure occurs and there is no other description closer to the source of the error, the description is used to explain the failure.

A syntax class's description can be desugared into a **~describe** pattern wrapped around its variant patterns. Recall the *binding* and *distinct-bindings* syntax class definitions from Section 4.2; 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 turn, *distinct-bindings* could be inlined into the **let** macro itself if only we could desugar the #:with clause away.

6.4 Action patterns

The action patterns of Figure 11 do not describe syntax *per se*; instead, they affect the parsing process without consuming input. The **~fail** form provides a way of explicitly causing a match failure; **~parse** allows the programmer to divert matching from the current input to a computed term; and **~late** affects the ordering of failures. They constitute the pattern equivalents of the #:fail-when and #:with side-clauses.

A **~fail** pattern performs an explicitly-coded check. As with #:fail-when, if the condition evaluates to a syntax value, it is added to the failure as the specific term that caused the error. Recall from Section 5.1 that #:fail-when side-conditions are performed as "late" checks; that is, their failures get an additional LATE progress step. In contrast, **~fail** performs "early" checks—no automatic LATE step. Early and late checks might alternatively be called "pre-order" and "post-order" checks. The *identifier* syntax class performs its test as an early check:

```
(define-syntax-class identifier
  #:description "identifier"
  (pattern (~and x (~fail (not (identifier? #'x)) #f))))
```

If the message is #f instead of a string, the fail pattern is ineffable, so the **~fail** check in *identifier* always uses the syntax class's description instead.

The **~late** form turns enclosed checks into late checks. In fact, the #:fail-when keyword option used in *distinct-bindings* (Section 4.2) is just shorthand for a combination of **~late** and **~fail** joined to the pattern via **~and**, which is extended to accept action patterns:

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

Note that 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 **~parse** form evaluates its sub-expression and matches it against the given pattern. It corresponds to a #:with clause. For example, the *field-declaration* syntax class from Section 6.1) is equivalent to the following:

```
(define-syntax-class field-declaration
  (pattern (~and field:identifier
                 (~late (~parse internal #'field))))
  (pattern [internal:identifier field:identifier]))
```

The **~parse** form requires an update to our notion of progress from Section 5.1. Now the matching process is no longer limited to sub-terms of the original term; it may involve arbitrary computed terms. We consider progress into one computed term to be unrelated to progress into another. Consequently, we change progress from a total order to a partial order by adding a new kind of progress step, BASE(*stx*):

$$\text{Progress Step}\quad ps\quad ::=\quad \text{FIRST}\ \mid\ \text{REST}\ \mid\ \text{LATE}\ \mid\ \text{BASE}(stx)$$

A BASE progress step is equal to another BASE progress step containing the same term; it is incomparable with any other progress step. The rules for error reporting stay the same: select the set of failures with *maximal* progress.

6.5 Head patterns

The patterns we have introduced so far cannot directly represent syntactic elements that do not consist of a single term, such as optional terms and keyword arguments. Consider a **test-case** macro, specified as it might be in the macro's documentation:

```
(test-case maybe-around-clause body-expr)
where   maybe-around-clause  ::=  #:around proc-expr
                               |  ϵ
```

Head patterns	H	::=	(**~seq** . L)
		|	(**~and** H $\{H \mid A\}^*$)
		|	(**~or** H^+)
		|	(**~describe** *expr* H)
List patterns	L	::=	()
		|	(S . L)
		|	(H . L)
Single-term patterns	S	::=	⋯
		|	(H . S)
Definitions	*def*	::=	⋯
		|	(**define-splicing-syntax-class** *stxclass* $desc^?$ $variant_H^*$)
		|	(**define-splicing-syntax-class** (*stxclass* x^*) $desc^?$ $variant_H^*$)
Head variant	$variant_H$	::=	(**pattern** H $side\text{-}clause^*$)

Fig. 12. Head patterns.

The *maybe-around-clause* non-terminal does not represent a single term; it represents either zero or two terms appearing in sequence before the test case's body expressions. Here are some example uses of **test-case**:

```
(test-case (check (= (+ 2 2) 4)))
(test-case #:around call-with-setup (check ——))
```

We cannot represent *maybe-around-clause* as a syntax class using single-term patterns. In particular, the following definition is wrong:

```
(define-syntax-class maybe-around-clause
  (pattern (#:around proc:expr))
  (pattern ()))
```

It accepts terms such as (#:around *call-with-setup*) and (), but we need a kind of pattern that when given a term like

```
(#:around call-with-setup (check ——))
```

matches the prefix consisting of the two-term sequence #:around *call-with-setup* and tells its context to continue matching the remainder: ((*check* ——)).

In other words, we need a mechanism for dealing with the contents of a term as a *sequence* and matching sub-*sequences* instead of sub-*terms*.

We introduce *head patterns*, described in Figure 12, as a new category of patterns that describe sequences of terms. The primary head pattern form is **~seq**, which is followed by a proper list pattern (L)—a syntactic restriction of single-term patterns that match only proper lists. The **~seq** form matches sequences of terms that, if parenthesized as a single compound term, would match L. For example, (**~seq** *x:identifier y:expr*) matches a sequence of two terms: an identifier followed by an expression. Contrast that pattern with (*x:identifier y:expr*), which matches *a single compound term* containing an identifier followed by an expression.

Since **syntax-parse** matches single terms, head patterns are ultimately used by combining them with a single-term tail pattern: (H . S). The result is 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 pattern. The prefix sequence may be empty; in fact, specifying optional syntactic elements is one of the primary uses of head patterns.

For example, the pattern ((**~seq** *x* *y*) *z:identifier*) matches the term (1 2 *c*) because the term can be split into the two-term prefix sequence 1 2 matching (**~seq** *x* *y*) and the suffix (*c*) matching (*z:identifier*).

By itself, **~seq** adds nothing to the expressiveness of syntax patterns. After all, ((**~seq** *x* *y*) *z:identifier*) is just equivalent to (*x* *y* *z:identifier*). The utility of **~seq** patterns is that they can be combined via **~or** to express multiple alternative prefixes. Several other pattern forms, including **~and** and **~describe**, are also extended with head-pattern versions. The overloading is resolved as follows: if an **~and**, **~or**, or **~describe** form has any head patterns as components, it is a head pattern; otherwise, it is a single-term pattern.

Here is one way to specify the syntax of **test-case** using head patterns:

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

As with single-term patterns, a syntax class is usually preferable to an explicit **~or** head pattern. Head patterns are encapsulated as *splicing syntax classes*. The optional #:around keyword argument could be extracted thus:

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

Compare that with the original informal specification:

$$\begin{array}{rcl} \textit{maybe-around-clause} & ::= & \texttt{\#:around}\ \textit{proc-expr} \\ & | & \epsilon \end{array}$$

The goal of **syntax-parse** is to enable programmers to specify syntax in a form consistent with the syntax's informal documentation.[5]

Head patterns are not intrinsically tied to keywords. We could describe the syntax of **let**, accommodating both the named-**let** and normal-**let** variants, with the following pattern:

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

The decision whether to have separate **syntax-parse** clauses or collapse them into a single clause using head patterns depends on how many optional elements there are and how similar the code generation templates are. In the case of **let**, we prefer to keep the two cases separate.

[5] Racket's documentation system (Flatt *et al.*, 2009) favors non-terminals with explicit ϵ productions over the ? operator; our implementation of **syntax-parse** does support an analogue of ?, but we find that splicing syntax classes are usually preferable.

Head patterns motivate our choice of model for backtracking with failure. Recall from Section 5.3 that the RF monad can be represented as a list of successes together with a set of failures. Why not use the simpler model where a result is *either* a list of successes *or* a failure set, rather than containing both? After all, it would seem that 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. Consider the following example, a version of **test-case** that supports optional names:

```
(define-splicing-syntax-class maybe-name
  (pattern (~seq #:name name:identifier))
  (pattern (~seq)
           #:with name #'anonymous))
(define-syntax (test-case stx)
  (syntax-case stx
    [(test-case n:maybe-name body:expr) ——]))
(test-case #:name "bob" (check ——))
```

In the use of the *test* macro, the programmer has mistakenly provided a string instead of an identifier. Nevertheless, *maybe-name* succeeds—by matching zero terms and leaving #:name and what follows for the macro's main pattern to fail on. In the "either-or" model, that success means that the failures *maybe-name* encountered before vacuously succeeding are lost, and the system would report "expected expression, got #:name"—a misleading message. In the our backtracking model, however, *maybe-name*'s success is accompanied by its previous failure, which overrules the macro pattern's failure, and the system reports the correct error: "expected identifier, got "bob"."

6.6 Ellipsis-head patterns

The head patterns of Section 6.5 support the specification of syntactic elements like keyword arguments and optional fragments—but only one at a time. Recall the syntax of **define-struct** from Figure 4. It has multiple keyword arguments that can occur in any order. Some of the keyword arguments can occur at most once; others can occur multiple times. Some pairs of keyword arguments are mutually exclusive. Head patterns alone are insufficient for specifying **define-struct**'s syntax.

Ellipsis-head patterns, shown in Figure 13, are the final ingredient necessary to specify syntax like **define-struct**. An ellipsis-head pattern may have multiple alternatives combined with **~or**, where each alternative is a head pattern. It accepts sequences consisting of some number of instances of the alternatives joined together. Each alternative may be annotated with a *repetition constraint* (**~optional** or **~once**) that restricts the number of times that alternative may appear in the sequence. If a repetition constraint is violated, an error message is synthesized using its *name* argument.

The alternatives of an ellipsis pattern are handled differently from other uses of **~or**. Instead of absent values accruing for every alternative that is not chosen, only

```
Ellipsis patterns        EH  ::=  (~or EH⁺)
                              |   (~once H #:name expr)
                              |   (~optional H #:name expr)
                              |   H
Single-term patterns     S   ::=  ···
                              |   (EH ... . S)
List patterns            L   ::=  ···
                              |   (EH ... . L)
```

Fig. 13. Ellipsis-head patterns.

```
Single-term patterns  S  ::=  ···
                          |   (~commit S)
                          |   (~delimit-cut S)
Head patterns         H  ::=  ···
                          |   (~commit H)
                          |   (~delimit-cut H)
Action patterns       A  ::=  ···
                          |   ~!
```

Fig. 14. Control patterns.

the chosen alternative accrues attribute values. For example, when the term (1 *a* 2 *b* *c*) is matched against the pattern ((**~or** *x:identifier* *y:number*) . . .), *x* is bound to (*a* *b* *c*) and *y* to (1 2).

The following pattern specifies the syntax of **define-struct**:

```
(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))
  . . . )
```

After the fields come the keyword options in any order. Keywords and their arguments are grouped together with **~seq**. The options that can occur at most once are wrapped with **~optional** repetition constraints. 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 within a single **~optional** constraint.

6.7 Backtracking control patterns

Our final group of patterns, shown in Figure 14, controls **syntax-parse**'s pattern matching, which is fully backtracking by default. To demonstrate, the following macro partitions a list of natural numbers into two lists with equal sums:

```
(define-syntax (balance nats)
  (syntax-parse nats
    [((~or a:nat b:nat) ...)
     #:fail-when (not (= (apply + (syntax→datum #'(a ...)))
                         (apply + (syntax→datum #'(b ...)))))
                 "sums differ"
     #'(quote ((a ...) (b ...)))]))
```

The **~or** pattern can succeed in two ways for each number in the sequence; it can match it as *a* or as *b*. That is, it represents a *choice point*. Since alternatives are tried in order, it always assigns the number to *a* initially. But after the complete list has been scanned, the side condition check triggers a failure if *a* and *b* are unbalanced. The failure causes the matching process to backtrack and reconsider the last choice it made; now it assigns the last number to *b* instead and tries the side condition check again. The pattern matcher will continue to backtrack until it finds a balanced partitioning or until it exhausts every possible assignment.

```
> (balance 1 2 3 4 5 6 7)
'((1 2 4 7) (3 5 6))
```

In practice, no macro would ever use backtracking this way, but this example demonstrates the power—and hints at the cost—of unrestricted backtracking.

The final class of patterns, *backtracking control patterns*, modifies the way backtracking works. The **~commit** form discards choice points created for its subpattern; and the **~!** ("cut") form discards choice points created within the enclosing **~delimit-cut** form.

The canonical example for **~!** (cut) is to drop alternatives once an identifying marker or tag has been found. Consider the following task: write a function that classifies forms as either core definitions or expressions and validates the syntax of the definitions. Any form starting with **define** or **define-syntax** is a definition; anything else is an expression. A core definition contains an identifier and an expression; the function abbreviation is not allowed. Here is a first attempt using **syntax-parse**:

```
(define (kind-of-form stx)
  (syntax-parse stx
    [((~literal define) var:identifier rhs:expr)
     'definition]
    [((~literal define-syntax) var:identifier rhs:expr)
     'definition]
    [e 'expression]))
```

The problem with this function is that it confuses ill-formed definitions with expressions. Given a bad term such as (**define** *x*), the first pattern fails, but the final pattern succeeds. One solution is to make the final pattern disjoint from the others by adding in their negations; however, this can cause the size of the final pattern to explode. It also makes the final pattern fragile: it must be updated if other cases are added or removed. Instead, once a pattern's distinguishing feature is seen, we drop the choice points for all of the other patterns:

```
(define (kind-of-form stx)
  (syntax-parse stx
    [((~and (~literal define) ~!) var:identifier rhs:expr)
     'definition]
    [((~and (~literal define-syntax) ~!) var:identifier rhs:expr)
     'definition]
    [e 'expression]))
```

The task of simultaneously classifying and validating forms is common in Racket macros that partially expand their sub-forms. Racket's **class** macro (Flatt *et al.*, 2006), for example, performs partial expansion to distinguish method definitions, field definitions, construction-time expressions, and other declarations.

A **~delimit-cut** form limits the effect of **~!**; only choice points creating within the enclosing **~delimit-cut** form are discarded. The clauses of a **syntax-parse** expression and the patterns of a syntax class body are implicitly wrapped in **~delimit-cut**.

A **~commit** form is a restricted form of cut that allows a pattern to match in only one way; a subsequent failure cannot backtrack to within the **~commit** form (but may backtrack to before it). The behavior of **~commit** is described by the following equation:

$$(\textbf{\~{}commit}\ S) = (\textbf{\~{}delimit-cut}\ (\textbf{\~{}and}\ S\ \textbf{\~{}!}))$$

One use of **~commit** is as an optimization hint; the programmer uses **~commit** to declare that the pattern should match in at most one way, so there is no need to keep around choice points that will never be used. Another use of **~commit** is in multi-pass parsing. For example, in the following pattern:

(**~and** S_1 S_2)

any failures accumulated by S_1 before it succeeds are retained during the parsing of S_2, and if they have high enough progress, they can drown out failures from S_2. The solution is to commit to the successful parse of S_1 before proceeding to S_2:

(**~and** (**~commit** S_1) S_2)

An alternative solution is to use a #:with clause or to use **~late** and **~parse** directly. In that case, failures from the first pattern are not discarded, but they have less progress than failures from the second pattern.

7 Implementation

The implementation of **syntax-parse** uses a variant of the two-continuation representation of the backtracking monad (Wand & Vaillancourt, 2004), adapted to carry failure information. Substitutions are usually represented implicitly using Racket's lexical environment rather than as an explicit data structure.

We have not yet added traditional pattern-matching optimizations to **syntax-parse**, but we plan to do so. Traditional optimizations must be adapted, however, to accommodate progress tracking. For example, exit optimization (Le Fessant & Maranget, 2001) may not skip a clause that cannot succeed if the clause may fail with greater progress than the exiting clause. For example, consider the following pattern:

```
(~or (~and (a b) C)
     d:identifier)
```

If the pattern indicated by C is reached, the pattern (a b) has already matched, so the term cannot possibly match *d:identifier*. Thus if error information were discounted, the failure continuation passed to C could skip the second alternative entirely. But **syntax-parse** cannot perform that optimization without proving that C fails with strictly greater progress than *d:identifier* does. If C is a **~fail** pattern, for example, the optimization would be unjustified.

Our performance goals to date have been to make **syntax-parse** fast enough to be usable, and we have succeeded: Racket programmers have enthusiastically embraced **syntax-parse**. For macro programmers, achieving acceptable performance with **syntax-parse** consists mainly of avoiding pathological backtracking such as the **balance** macro represents (Section 6.7) and occasionally restructuring patterns or using backtracking control patterns to prune choice points.

Programmers must be aware of the backtracking nature of pattern matching and the performance characteristics of the syntax classes they use. It is not a goal of **syntax-parse** to limit the complexity of pattern matching (to linear or quadratic, for example). Rather, our goal is to help programmers to write clean specifications that perform well.

8 Case studies

Racket has included **syntax-parse** since August 2009. Users of **syntax-parse** confirm that **syntax-parse** makes it easy to write macros for complex syntax. Reformulating existing macros with **syntax-parse** can cut parsing code by several factors without loss in quality in error reporting. 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.

8.1 A notation for loops

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.

Expression	*expr*	::=	· · ·
		|	(**loop** *lclause* · · ·)
Loop clause	*lclause*	::=	(**initial** (*vars* *init* $step^?$ $test^?$) · · ·)
		|	(**for** *x* **in** *lst*)
		|	(**for** *x* **in-vector** *vec*)
		|	(**incr** *i* **from** *init* **to** *final* **by** *step*)
		|	*bclause*
		|	· · ·
Body clause	*bclause*	::=	(**when** *expr*)
		|	(**do** $expr_1$ · · ·)
		|	· · ·
Variable(s)	*vars*	::=	*id* | (*id* · · ·)

Fig. 15. Subset of **loop** syntax.

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.

Figure 15 shows a small subset of the **loop** grammar. The full package provides over 20 loop clause forms in addition to the ones listed above. Programmers can also define new loop forms via macros using the package's toolkit of binding and control forms.

The original **loop** macro performs little error checking; in 31 exported macros there are only 3 syntax validation checks plus a handful of internal sanity checks. The exported macros consist of the **loop** macro itself plus 31 macros in continuation-passing style (Hilsdale & Friedman, 2000) implementing the loop clause forms such as **for** and **do**.

A loop-clause form such as **for** is implemented by a macro named *loop-clause/for*; the name is chosen to reduce contention for short names. The **loop** macro rewrites the abbreviated loop-clause names to the long form, except that programmers can write the long form in parentheses, e.g., ((*loop-clause/for*) *x* **in** *xs*), to avoid the rewriting. The code to recognize and rewrite both cases is duplicated, since **for** enforces the same protocol for its auxiliaries: **in-vector** becomes **for-clause/in-vector**.

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** clause forms:

```
(define-syntax-class (loopkw prefix)
  #:description "loop clause name or parenthesized long-form loop clause name"
  #:attributes (kw kw.macro)
  (pattern x:id
           #:with kw:cps-macro (format-id #'x "~a/~a" prefix #'x))
  (pattern (kw:cps-macro)))
```

The #:attributes option explicitly lists the syntax class's attributes; it is necessary when re-exporting nested attributes. There are two patterns, one for the abbreviated identifier and one for the parenthesized long form. The *format-id* function creates an identifier using a format string.

Macros written in portable Scheme often use continuation-passing style to support modular and extensible sub-forms, such as the loop clause syntax, since standard Scheme—unlike Racket—has no mechanism for eagerly expanding sub-forms. Each type of loop clause is implemented by a CPS macro that transforms the loop clause into the **loop** package's internal representation, then applies the "continuation" macro to the result.

Macros in continuation-passing style 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 (*loop-clause/for* (*x* **in** *xs*) *k kargs*) to accommodate the **loop** macro's continuation. Errors in the programmer's use of **for** should be reported in terms of the original syntax, not the rewritten syntax. Consequently, 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. We transformed the other 11 CPS macros 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. The registration part is easily accomplished by creating new macro-definition forms; **syntax-parse** is not necessary for that step. Performing the checks normally involves calling the *syntax-local-value* function to examine the value statically bound to the macro name. This check is nicely encapsulated in a syntax class; even better, we were able to use the built-in *static* syntax class to handle the *syntax-local-value* lookup and perform the predicate check.

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. A few forms, such as **initial**, have more structured syntax, so we define syntax classes for their sub-terms, including a shared syntax class *var/vars* that accepts a single variable or a parenthesized group of variables.

The original version of the **loop** macro consists of 1840 lines of code, not counting comments and empty lines. The implementation of the loop clause 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 47 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 the 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

```
(define-tokens value-tokens (NUM))
(define-empty-tokens op-tokens (+ − * / LPAREN RPAREN EOF))
(define calc-lexer ——)

(define calc
  (parser
    (start exp)
    (end EOF)
    (tokens value-tokens op-tokens)
    (error (λ (tok? name val) (error 'calc "bad token: ~s" name)))
    (precs (left − +) (left * /))
    (grammar
      (exp [(NUM) $1]
           [(exp + exp) (+ $1 $3)]
           [(exp − exp) (− $1 $3)]
           [(exp * exp) (* $1 $3)]
           [(exp / exp) (/ $1 $3)]
           [(LPAREN exp RPAREN) $2]))))

(calc (calc-lexer "1+3*5+1")) ;; ⇒ 17
```

Fig. 16. Example of **parser** macro.

code shrank, *despite much improved error handling*, due to simplifications enabled by **syntax-parse**.

8.2 An LALR(1) parser generator

The **parser** macro (Owens *et al.*, 2004) implements a parser generator for LALR(1) grammars. The macro takes 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. Figure 16 shows an example parser, a minimal arithmetic expression parser that calculates instead of producing an abstract syntax tree.

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, the **tokens** clause, which imports terminal descriptions, and the **start** clause, which specifies the start symbol or symbols. 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. Some clauses depend on others. For example, the productions in the **grammar** clause depend on the terminals imported by the **tokens** clause, and the **start** symbols clause depends on the non-terminals defined in the **grammar** clause.

The original version used a loop and mutable state to recognize clauses; different clauses were validated and parsed at various points later in the macro's processing.

The new version consists of two well-defined passes. The first pass checks for missing and duplicate clauses using our improved ellipsis patterns and examines each mandatory clause just enough to extract the information needed to validate the other clauses in the second pass. For example, the **grammar** clause is inspected just enough to determine what non-terminal names are defined, but the productions are not checked. The second pass uses syntax classes parameterized over the results gathered from the first pass to perform full validation.

The original version of **parser** explicitly detects 39 different syntax errors beyond those caught by MBE-style patterns. Repetition constraints (**~once** and **~optional**) on the different clause variants cover 13 of the original errors plus a few that the original macro failed to check. Pattern variable annotations cover 11 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 5.2); **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 forms.

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.

9 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 system's head patterns can be used to define grammars similar to those supported by traditional token-stream parsers. As it stands, **syntax-parse** is not a viable substitute for a traditional parser generator, but the addition of look-ahead and memoization, particularly to splicing syntax classes, would give it the power and efficiency of packrat parsers for comparable grammars. Look-ahead is trivial to add; memoization would be more difficult.

Our side conditions seem distantly related to disambiguation filters (van den Brand *et al.*, 2002), but the differences are significant. Side conditions can decide whether to reject productions based on arbitrary Racket computations, whereas disambiguation filters are specified like grammar productions with an attached filter action. On the other hand, disambiguation filters support actions other than production rejection: such as prioritizing one parse tree ahead of another. Side conditions, in contrast, cannot change **syntax-parse**'s strict left-to-right exploration of alternatives.

The ordering of parse failures is 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 & Felleisen, 2004) and evolved to encompass binding information and hygienic expansion (Herman & Wand, 2008). These systems, however, are incapable of fortifying a broad range of widely used macro idioms, and they do not address the issues of error feedback or of modular syntax specification addressed by our system.

10 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. Through our experience with **syntax-parse**, 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

I am grateful to my advisor, Matthias Felleisen, for posing the problem and guiding me in my search for a solution; Matthias was a co-author of the ICFP version of this paper. I also thank Matthew Flatt, Guy Steele, Sam Tobin-Hochstadt, and Jon Rafkind for feedback on the design and implementation of **syntax-parse**.

References

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

Despeyroux, T. (October 1995) Logical programming and error recovery. *Industrial Applications of Prolog*.

Dybvig, R. K. (2010) Chez scheme version 8 users's guide. *Cadence Research Systems*.

Dybvig, R. K., Hieb, R. & Bruggeman, C. (1993) Syntactic abstraction in Scheme. *Lisp Symb. Comput*. **5**(4), 295–326.

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

Flatt, M. (2002) Composable and compilable macros: You want it when? In *International Conference on Functional Programming*, pp. 72–83.

Flatt, M., Barzilay, E. & Findler, R. B. (2009) Scribble: Closing the book on ad hoc documentation tools. In *International Conference on Functional Programming*, pp. 109–120.

Flatt, M., Findler, R. B. & Felleisen, M. (November 2006) Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems*, pp. 270–289.

Flatt, M. & PLT. (January 2010) *Reference: Racket*. Tech. Rep. PLT [online]. Accessed July 2012. Available at: http://racket-lang.org/tr1/.

Ford, B. (September 2002) *Packrat Parsing: A Practical Linear-time Algorithm with Backtracking*. MPhil thesis, Massachusetts Institute of Technology.

Herman, D. & Wand, M. (March 2008) A theory of hygienic macros. In *European Symposium on Programming*, pp. 48–62.

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

Hughes, J. (1995) The design of a pretty-printing library. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques: Tutorial Text*. London, UK: Springer-Verlag, pp. 53–96.

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

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

Le Fessant, F. & Maranget, L. (2001) Optimizing pattern matching. In *International Conference on Functional Programming*, pp. 26–37.

Owens, S., Flatt, M., Shivers, O. & McMullan, B. (September 2004) Lexer and parser generators in Scheme. In *Workshop on Scheme and Functional Programming*, pp. 41–52.

Queinnec, C. (1996) Macroexpansion reflective tower. *In Proceedings of the Reflection'96 Conference*, pp. 93–104.

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

Sperber, M., Dybvig, R. K., Flatt, M., van Straaten, A., Findler, R. & Matthews, J. (2009) Revised^6 report of the algorithmic language Scheme. *J. Funct. Program.* **19**(S1), 1–301.

van den Brand, M. G. J., Scheerder, J., Vinju, J. J. & Visser, E. (2002) Disambiguation filters for scannerless generalized LR parsers. In *Compiler Construction*. Horspool, N. (ed). Springer-Verlag, pp. 143–158.

Wand, M. & Vaillancourt, D. (2004) Relating models of backtracking. In *International Conference on Functional Programming*, pp. 54–65.