Rhombus: A New Spin on Macros without All the Parentheses

MATTHEW FLATT, University of Utah, USA
TAYLOR ALLRED, University of Utah, USA
NIA ANGLE, independent, USA
STEPHEN DE GABRIELLE, independent, UK
ROBERT BRUCE FINDLER, Northwestern University, USA
JACK FIRTH, independent, USA
KIRAN GOPINATHAN, National University of Singapore, Singapore
BEN GREENMAN, Brown University, USA
SIDDHARTHA KASIVAJHULA, independent, USA
ALEX KNAUTH, independent, USA
JAY MCCARTHY, Reach, USA
SAM PHILLIPS, independent, USA
SORAWEE PORNCHAROENWASE, University of Washington, USA
JENS AXEL SØGAARD, independent, Denmark
SAM TOBIN-HOCHSTADT, Indiana University, USA

Rhombus is a new language that is built on Racket. It offers the same kind of language extensibility as Racket itself, but using conventional (infix) notation. Although Rhombus is far from the first language to support Lisp-style macros without Lisp-style parentheses, Rhombus offers a novel synthesis of macro technology that is practical and expressive. A key element is the use of multiple binding spaces for context-specific sublanguages. For example, expressions and pattern-matching forms can use the same operators with different meanings and without creating conflicts. Context-sensitive bindings, in turn, facilitate a language design that reduces the notational distance between the core language and macro facilities. For example, repetitions can be defined and used in binding and expression contexts generally, which enables a smoother transition from programming to metaprogramming. Finally, since handling static information (such as types) is also a necessary part of growing macros beyond Lisp, Rhombus includes support in its expansion protocol for communicating static information among bindings and expressions. The Rhombus implementation demonstrates that all of these pieces can work together in a coherent and user-friendly language.

CCS Concepts: • **Software and its engineering → Extensible languages**.

Additional Key Words and Phrases: macros, infix syntax, binding spaces

Authors' addresses: Matthew Flatt, University of Utah, Salt Lake City, USA, mflatt@cs.utah.edu; Taylor Allred, University of Utah, Salt Lake City, USA, taylor.c.allred@utah.edu; Nia Angle, Concord, CA, USA, rokitna@hotmail.com; Stephen De Gabrielle, London, UK, spdegabrielle@gmail.com; Robert Bruce Findler, Northwestern University, Evanston, IL, USA, robby@cs.northwestern.edu; Jack Firth, Sunnyvale, CA, USA, jackhfirth@gmail.com; Kiran Gopinathan, National University of Singapore, kirang@comp.nus.sg.edu; Ben Greenman, Brown University, Providence, RI, USA, benjaminlgreenman@gmail.com; Siddhartha Kasivajhula, Oakland, CA, USA, sid@countvajhula.com; Alex Knauth, Williamsport, PA, USA, alexander@knauth.org; Jay McCarthy, Reach, USA, jay.mccarthy@gmail.com; Sam Phillips, Oakland, CA, USA, samdphillips@gmail.com; Sorawee Porncharoenwase, University of Washington, Seattle, USA, sorawee@cs.washington.edu; Jens Axel Søgaard, Skjern, Denmark, jensaxel@soegaard.net; Sam Tobin-Hochstadt, Indiana University, Bloomington, USA, samth@cs.indiana.edu.

2475-1421/2023/10-ART242
https://doi.org/10.1145/3622818

ACM Reference Format:
Matthew Flatt, Taylor Allred, Nia Angle, Stephen De Gabrielle, Robert Bruce Findler, Jack Firth, Kiran Gopinathan, Ben Greenman, Siddhartha Kasivajhula, Alex Knauth, Jay McCarthy, Sam Phillips, Sorawee Porncharoenwase, Jens Axel Søgaard, and Sam Tobin-Hochstadt. 2023. Rhombus: A New Spin on Macros without All the Parentheses. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 242 (October 2023), 30 pages. https://doi.org/10.1145/3622818

1 INTRODUCTION

Through decades of experience, the programming-languages community has discovered and refined ideas that should appear in most any language, including functional abstraction and lexically scoped variables. Beyond the basics, however, there are still more good ideas for programming constructs than can fit in any one language specification. Language extensibility, especially in the form of macros, helps to balance the competing goals of a manageable language size versus fit-to-purpose for a wide range of tasks. Even better, programmers can use macros to tailor a language to better match a specific domain (Felleisen et al. 2018).

The Lisp and Scheme communities have long championed macros, but historically, the message of macros has been difficult to detangle from Lisp's minimalistic, parenthesis-oriented notation. With macro systems included in newer languages like Scala, Rust, Elixir, and Lean, programmers are starting to see the concerns of core notation and extensibility detangled. Still, few would argue that the new batch of macro systems have achieved the expressiveness, fluidity, and central role of macros as they exist within the Lisp tradition. To some degree, the gap exists because macros are added on top of a core language, instead of designing the language to take advantage of macro extensibility from the start or to be a vehicle for defining the language's base forms.

Rhombus is a programmable programming language with conventional notation, designed from the start around macro extensibility. By "conventional notation," we mean that most any programmer will recognize arithmetic, function calls, indexed access, and field accesses based on precedents from algebra and Algol to JavaScript and Python. At the same time, by building on Racket's state-of-the-art facilities for extensibility, and by exposing and extending its facilities to work with conventional notation, Rhombus provides an especially rich toolbox for language construction.

A *combination* of previously explored ideas distinguish Rhombus from prior designs:

- Macro expansion uses an intermediate form called *shrubbery notation* that is analogous to S-expressions, but defers some grouping decisions to a macro-extensible parsing pass. For example, `f(1 + 2 * 3 > 4)` corresponds to a shrubbery form where `1 + 2 * 3 > 4` is a flat sequence of terms, but nested relative to `f`. Shrubbery's nesting puts a limit on the transformations that macros can perform, so programmers do not need to know every macro before making some sense of unfamiliar code.
- Pervasive pattern matching and repetition notation in the base language reduces the gap between everyday programming and metaprograming—in contrast to Scheme, which evolved to emphasize distinct macro-by-example constructs for syntactic extension. Specifically, repetition through ellipses is convenient for many programs that manipulate sequences, not just syntax sequences, and so Rhombus supports it more generally.
- Expansion integrates support for multiple *spaces*, which support different bindings for different contexts within a module, in contrast to the single-namespace approach of Scheme. Bindings and other contexts are macro-extensible in the same way as expression contexts. The operator `::`, for example, can have one meaning and expansion in an expression context and a different meaning and expansion in a binding context.

```
fun factorial(n):
  match n
  | 0: 1
  | _: n * factorial(n - 1)
def N = 10

> factorial(N)
3628800
```

```
fun
| factorial(0):
    1
| factorial(n):
    n * factorial(n - 1)

> factorial(N)
3628800
```

```
operator n!:
  ~stronger_than: + - * /
  match n
  | 0: 1
  | _: n * (n - 1)!

> N!
3628800
```

Fig. 1. Three ways of writing factorial

- Macro facilities support binding and propagating type-like static information, and macro expansion can be sensitive to that information. For example, a `use_static` declaration insists that every field or method selection with `.` is statically resolved to a field or method selector, excluding the possibility of a "message not understood" for that access.

Although Rhombus's design includes other elements, these properties are the ones that work together to extend the reach of macro extensions. The key precedents for Rhombus's design are Lisp macros (Hart 1963), Scheme hygienic macros (Kohlbecker et al. 1986; Clinger and Rees 1991; Dybvig et al. 1993), micros and macros (Krishnamurthi et al. 1999), Racket modules (Flatt 2002) and languages (Tobin-Hochstadt et al. 2011), scope sets (Flatt 2016), parsing via enforestation (Rafkind and Flatt 2012; Disney et al. 2014), syntax classes (Culpepper and Felleisen 2012), type systems as macros (Chang et al. 2017), and language support for DSL creation (Ballantyne et al. 2020).

2 RHOMBUS ESSENTIALS AND EXAMPLES

The goal of this section is not to provide a complete overview of Rhombus, but to introduce specific parts of Rhombus as needed for the rest of the paper, and also to sketch how those parts fit into the larger implementation picture—all to give a sense of where we're trying to go.

Rhombus is a whitespace-sensitive language, so line breaks and indentation in the examples are significant. We show read-eval-print-loop interactions where a leading `>` represents the prompt, but we omit the prompt for most definitions, since they would normally be written in a module and do not print a result. Also, we defer an explanation of `:` and `|` with their indentation rules until section 3.1—but because so much of the syntax is conventional, the examples in this section should be readable with just a little explanation.

2.1 Definitions

The left column of figure 1 shows a definition of a function `factorial` and a constant `N`, where the function uses `match` for pattern matching dispatch among clauses that each start with `|`, and `_` matches anything. The `fun` form also supports `|` cases directly, so the `factorial` could be written equivalently as shown in the middle column of figure 1.

The `def` and `fun` forms are part of the base Rhombus language, but they are implemented as macros over a primitive binding form. In the case of `fun`, the expansion has a function expression for the right-hand side. The `match` form similarly wraps a primitive conditional-binding protocol. Binding positions are macro-extensible, and many predefined binding forms implement patterns, which means that pattern matching is pervasively available in binding positions and not just part of `match`.[1]

[1]This is partly why `def` and `fun` are separate: to distinguish using a pattern constructor in a definition from binding a function name. For example, `fun Pair(x, y)` starts a definition of a function named `Pair`, while `def Pair(x, y)` starts the definition of `x` and `y` by pattern-matching a right-hand side that constructs a pair.

To define a prefix, infix, or postfix operator, use the `operator` form as shown in the right column of figure 1.[2] The `~stronger_than` declaration there gives `!` a higher precedence than basic arithmetic operators.[3] Absent other declarations, an expression that has `!` with some other operator, such as `==`, would require parentheses to disambiguate association. The keyword `~other` can be used in a precedence declaration to stand for all operators that are not otherwise mentioned. An operator's associativity can be specified with `~associativity` followed by `~left` (the default), `~right`, or `~none`. If parsing finds two operators with undeclared or incompatible precedence relationships, then it raises a syntax error asking for disambiguation.

Operator definitions can be local, and precedence relationships refer to bindings, not to symbolic operator names. Parsing a sequence of operators and operands based on precedence is part of macro expansion, so a precedence declaration ultimately must be attached to a macro. The `operator` macro is a shorthand for defining a function plus a macro to call to the function, which means that `operator` is a definition-generating and macro-generating macro.

2.2 Lists and Repetitions

List constructions and patterns are written with `[]` around comma-separated items. The `List.first` and `List.rest` functions offer one way to access the head and tail of a list.

```
fun
| sum([]): 0
| sum(ns): List.first(ns) + sum(List.rest(ns))

> sum([1, 2, 3])
6
```

A better approach is to use `...` in the `[]` binding form, which binds the preceding element pattern as a repetition. The `[]` expression form similarly supports `...` to reference a repetition. Binding and expression forms cooperate (Flatt et al. 2012) so that this variant generates essentially the same code as the previous version (that is, no new list is created for the recursive call):

```
fun
| sum([]): 0
| sum([n, m, ...]): n + sum([m, ...])
```

The `[]` form may appear hardwired into the language, but is treated as an implicit use of a macro that the Rhombus base language defines as an expression and pattern form to construct and match lists. The binding form recognizes `...`, and it converts the preceding pattern so that it binds repetitions. The expression form also recognizes `...`, and it treats the preceding form as a repetition context, which can refer to repetition bindings.[4]

Operators and the function-call form are defined so that they map over their arguments when they appear in repetition contexts:

```
> def [n, ...] = [1, 2, 3]
> [factorial(n), ..., n!, ...]
[1, 2, 6, 1, 2, 6]
> [n+1, ...]
[2, 3, 4]
```

[2] Shrubbery notation distinguishes operator tokens from identifier tokens, but `operator` allows either as an operator name, while the `fun` form expects an identifier for a function name.

[3] A `~` prefix converts an identifier to a keyword, which is never an expression or binding operator on its own.

[4] A reader may wonder whether it matters that `...` appears after the `,` instead of before. Putting `...` after `,` means that a list element is being repeated to create multiple elements, as opposed to a repetition that forms a single element of the list. This distinction is especially relevant in macro patterns and templates.

Mapping `+` over `n` works because literals like `1` act as repetitions of depth 0, and `...` replicates shallower repetitions to fill deeper ones. This treatment of nested and mixed repetitions, which was worked out in iterations of Scheme's macros-by-example (Kohlbecker and Wand 1987; Clinger and Rees 1991; Dybvig et al. 1993), is useful and expressive for many non-macro applications.[5]

2.3 Classes

A `class` declaration creates a new class and binds the class's name to a constructor procedure for expressions, a pattern form for bindings, and more:[6]

```
class Posn(x :: Int, y :: Int)

fun
| vector_sum([]): Posn(0, 0)
| vector_sum([p :: Posn, q, ...]):
    let Posn(xs, ys) = vector_sum([q, ...])
    Posn(p.x + xs, p.y + ys)

// or, equivalently
fun vector_sum([Posn(x, y), ...]):
  Posn(sum([x, ...]), sum([y, ...]))

> vector_sum([Posn(1, 2), Posn(3, 4), Posn(5, 6)])
Posn(9, 12)
```

The `::` binding operator expects an *annotation* afterward. Some annotations, like `Int`, are predefined, and `class` defines the class name also as an annotation. An annotation with `::` implies a run-time check to ensure that a value satisfies the annotation, and it adjusts the binding to propagate static information. As a result, for example, `p.x` can be statically resolved to an access of the `x` field of a `Posn`. Static information is itself implemented through the macro system, taking advantage of the fact that binding positions are macro-extensible and can expand to expansion-time definitions as well as definitions of run-time variables.

The `class` form connects a new datatype with many different facets of a program, including expression, binding, and static-information concerns. The resulting implementation complexity is tamed by implementing `class` as a macro that expands to many different definitions of more primitive forms, each generally handling a different facet of `class`'s role.

2.4 Syntax

A pair of single quotes `''` in Rhombus creates a *syntax object* (not a string), which is a representation of syntax that has nested term structure intact. Syntax-object printing reflects structure with a `«» ;` notation that is not whitespace-sensitive, which in the following example helps clarify that the content of a syntax object is not merely text:

```
> def noisy_identity_stx = 'fun (x):
                            println(x)
                            x'
> noisy_identity_stx
'fun (x):« println (x); x »'
```

In terms of structure, the syntax object preserves the fact `println(x)` is one group and `x` by itself is another, but that both pieces are in the block after `fun (x)`.

[5]Rhombus also supports a `&` prefix splicing operator in lists, maps, sets, and function arguments, both for patterns and constructions/calls, which is like a `*` prefix in Python or a `...` prefix in JavaScript.

[6]This example uses `let`, which is like `def`, but it binds names that are visible only later within its context. The `def` form tends to be more convenient at the top level, where exports and mutually recursive references are common, while `let` tends to be more convenient within a local block, because it allows shadowing by later `let`s.

The `$` prefix operator serves as an escape within a syntax quotation. When a `''`-quoted form is an expression, `$` escapes back to expression mode to compute a value that is substituted into the syntax-object template. When a `''`-quoted form is a binding pattern, `$` escapes back to binding mode for a nested pattern to match against a portion of an input syntax object.

```
> '[1, 2, 3].map($noisy_identity_stx)'
'[1, 2, 3] . map (fun (x):« println (x); x »)'

> match noisy_identity_stx
  | 'fun ($arg): $body': [arg, body]
['x', 'println (x); x']
```

A `...` can be used in syntax patterns and templates to bind and use repetitions. To make those patterns and templates more readable, `...` is implicitly escaped instead of being treated as literal; otherwise, `$` would be needed before each `...` that is intended as repetition.[7]

```
def '$f($arg, ...)' = 'expt(2, 5+5)'

> [f, arg, ...]
['expt', '2', '5 + 5']

> 'lazy_call($f, fun(): $arg, ...)'
'lazy_call (expt, fun ():« 2 », fun ():« 5 + 5 »)'
```

Figure 2 shows a metacircular interpreter for a small Rhombus-like language using syntax objects to represent programs, which is analogous to reusing S-expressions in Lisp. In a syntax pattern, annotated escapes like `x :: Identifier` and `x :: Int` resemble general binding annotations, but they are more precisely uses of *syntax classes* that are specific to syntax-pattern contexts. The interpreter's implementation also uses `{}` notation for constructing a map, the `++` operator for functional union of maps, `[]` for indexing a map with a key, and the `unwrap` method of a syntax object to extract a raw number or symbol. The point of this example is (1) to demonstrate how Rhombus includes all of the ingredients needed to make a compact metacircular interpreter, and one that is Lisp-like by working directly on syntax representations; and (2) to demonstrate how pervasive pattern matching and unified repetition notation are useful, especially in the `fun` case where they seamlessly span syntax matching, dictionary construction, and variadic functions.

2.5 Macros

Syntax objects work nicely for the interpreter in figure 2, but they are more typically used to implement macros. The `macro` form expects a pattern to match against a use of the macro, and then a block containing an immediate template to produce the result of macro expansion.

```
macro 'thunk: $body':
  'fun (): $body'

> def delayed_three = thunk: 1 + 2
> delayed_three()
3
```

More generally, macros can be implemented with arbitrary expansion-time code, which requires that `expr.macro` and expansion-time Rhombus are imported from `rhombus/meta`.[8] Figure 3 uses `expr.macro` to define `prims`, which locally defines `to_symbol` and `to_function` expansion-time functions. The `to_symbol` function normalizes an operator to an identifier using `unwrap_op` at expansion time, and `to_function` wraps an operator implementation as a two-argument function.

[7] A `$` or `...` with nothing before or after is treated as literal instead of an escape, so to write a literal `$` or `...` in a larger pattern or template, escape to an immediate quote using `$'$'` or `$'...'`.
[8] An `open` modifier makes all imported names available without the module name as a prefix.

```
fun interp(e, env :: Map):
  match e
  | 'fun ($x, ...): $e': fun (arg, ...):
                           interp(e, env ++ {x.unwrap(): arg, ...})
  | '$rator($rand, ...)': interp(rator, env)(interp(rand, env), ...)
  | '$(x :: Identifier)': env[x.unwrap()]
  | '($e)': interp(e, env)
  | '$(x :: Int)': x.unwrap()
  | '[$x, ...]': [interp(x, env), ...]
  | '$x ... $(op :: Operator) $y ...': env[op.unwrap_op()](interp('$x ...', env),
                                                           interp('$y ...', env))
  | 'block: let $x = $rhs; $body': interp('(fun ($x): $body)($rhs)', env)

def init_env:
  { 'cons'.unwrap(): List.cons,
    'first'.unwrap(): List.first,
    'rest'.unwrap(): List.rest,
    '+'.unwrap_op(): fun(x, y): x + y,
    '-'.unwrap_op(): fun(x, y): x - y,
    '*'.unwrap_op(): fun(x, y): x * y }

> interp('block:
            let swap = fun (x): [first(rest(x)), first(x)]
            first(swap([1, 2])) + 3',
         init_env)
5
```

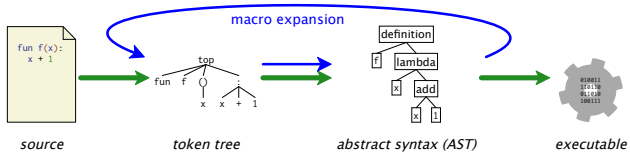
Fig. 2. Metacircular interpreter

```
import: rhombus/meta open

expr.macro 'prims { $name: $impl, ... }':
  fun | to_symbol(name :: Operator): name.unwrap_op()
      | to_symbol(name :: Identifier): name.unwrap()
  fun | to_function(impl :: Operator): 'fun(x, y): x $impl y'
      | to_function(impl): impl
  '{ '$(to_symbol(name))'.unwrap(): $(to_function(impl)), ... }'

def init_env = prims { cons: List.cons, first: List.first, rest: List.rest, +: +, -: -, *: * }
```

Fig. 3. A macro to simplify initial-environment construction

3 RHOMBUS SYNTAX AND REPRESENTATION

The Rhombus program-processing pipeline starts with source text, uses a *reader* to convert text into an intermediate *token tree* representation, parses that token tree into an AST, and finally compiles the AST to an executable:



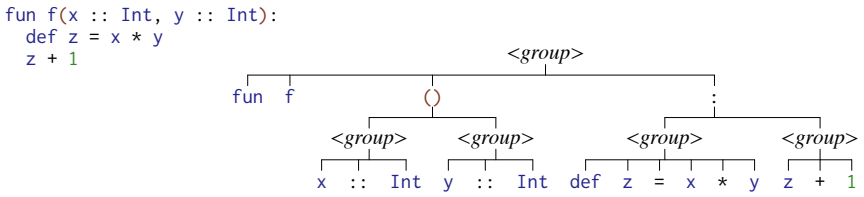


Fig. 4. Example shrubbery form and its parse tree

This is the same pipeline as used by Racket or most any dialect of Lisp, but Rhombus's reader converts text into *shrubbery* representation instead of an *S-expression* representation. Basic syntactic ingredients like `:`, indentation for blocks, and `|` for alternatives are not tied to Rhombus keywords, but are instead defined at the shrubbery layer. On top of that layer, in the same way that Lisp provides `quasiquote` and `unquote` to make working with S-expressions easier, Rhombus provides a generic set of pattern and template facilities that are tuned to the structure of shrubbery notation.

3.1 Shrubbery Notation

An idealized S-expression grammar of terms has just two productions: atoms (like numbers, booleans, identifiers, and strings) and a parenthesized sequence of terms.

$$\langle term \rangle \ ::= \ \langle atom \rangle \quad | \quad \texttt{(} \ \langle term \rangle^* \ \texttt{)}$$

This ideal sweeps many details under the rug, such as dot notation for raw pairs and the syntax of atoms, but it captures the feeling of programming with S-expressions: conceptual simplicity at the price of lots of parentheses. This approach has some appeal—enough that an alternative M-expression notation for Lisp famously never materialized (McCarthy 1978)—but S-expression notation also has drawbacks. Some characterize programming with S-expressions as "writing in ASTs," which is not the same "AST" as in the picture at the start of this section, but reflects the sense that programmers must write out a kind of parse tree with all grouping made explicit.

Many approaches to improving Lisp notation make use of whitespace or other delimiters as an alternative to parentheses, but aim for the same underlying S-expression structure. Those approaches include I-expressions (Möller 2003), Parendown (Angle 2017), the Scribble reader (Barzilay 2009), Sweet-expressions (Wheeler 2013), and Wisp (Babenhauserheide 2015). Another common adjustment is to add support for infix sections, including SIX in Gambit (Feeley 2019) or Curly-infix (Wheeler 2013); such extensions accommodate traditional arithmetic forms, but still in the context of an S-expression conversion, and not in a generally extensible way. Some Lisp and Scheme implementations, including Racket, allow `[]` and `{}` grouping as a synonym for parentheses. Clojure goes a step further, making `[]` and `{}` core parts of the grammar with meanings distinct from `()` in the base language.

Rhombus's approach is most like Clojure's in that it has a richer core notation, and Rhombus notation is even richer while still smaller and more general than a parsed AST. That richer base is only half of Rhombus's strategy, however. The other half is to reduce the amount of grouping that the base notation is expected to encode, and instead leave fine-grained grouping to an additional parsing pass. That second pass is integrated with macro expansion so that it can be extended by defining macros, including macros in nested scopes or through macro-generated macros. The base notation is called *shrubbery notation*, because it tends to have shallower nesting and grouping than S-expression trees. The parsing component that is interleaved with expansion, which finishes the construction of S-expression-like trees, is called *enforestation* (Rafkind and Flatt 2012).

The core shrubbery grammar resembles an S-expression grammar in that it has ⟨*term*⟩s and nesting under parentheses and other brackets, but it introduces an extra ⟨*group*⟩ layer. A ⟨*group*⟩ represents an unenforested sequence of ⟨*term*⟩s that optionally ends with a `:` block and/or `|` alternatives, where `:` and `|` create nesting in a way similar to parentheses. A ⟨*group*⟩ is never empty.

⟨*term*⟩ ::= ⟨*item*⟩ | ⟨*block*⟩ | ⟨*alts*⟩
⟨*item*⟩ ::= ⟨*atom*⟩ | `(` ⟨*group*⟩* `)` | `[` ⟨*group*⟩* `]` | `{` ⟨*group*⟩* `}` | `'` ⟨*group*⟩* `'`
⟨*group*⟩ ::= ⟨*item*⟩* ⟨*block*⟩? ⟨*alts*⟩? — *must be nonempty*
⟨*block*⟩ ::= `:` ⟨*group*⟩*
⟨*alts*⟩ ::= (`|` ⟨*group*⟩*)+

This grammar omits a description of how ⟨*group*⟩s are separated within a ⟨*group*⟩*, which is where line and indentation rules come into play, plus `;` and `,` separators. Figure 4 illustrates the parse tree for an example shrubbery form, which uses some of these rules:

- Each ⟨*group*⟩ either starts on a new line, or it is separated from the previous ⟨*group*⟩ by `;`. When a subsequent ⟨*group*⟩ in a sequence starts on a new line, it must be indented the same as the first ⟨*group*⟩ in the sequence.
- Within `()`, `[]`, and `{}` line and indentation rules still apply, but immediate groups must be separated by `,` (instead of `;`), even when a group starts on a new line. Line and indentation rules still apply within `'`, but the group separator is `;` and optional.
- If the first ⟨*group*⟩ after a `:` is on its own line, it must be indented more than the ⟨*group*⟩ that contains the `:`. If a `|` starts on a new line, it must be indented the same as the ⟨*group*⟩ that contains the `|`.
- For each subsequent `|` in an ⟨*alts*⟩, when it starts on a new line, it must line up with the first `|` in the ⟨*alts*⟩.

We leave full details to the Rhombus and shrubbery documentation. The details of the indentation rules, as well as the choice of indentation-sensitive parsing, are less important than (1) the approximate size and shape of the grammar that it encodes, and (2) having the grammar contain a small amount of nesting structure while prominently featuring ⟨*group*⟩ as a flat sequence of ⟨*term*⟩s.

3.2 Shrubbery Patterns and Templates

Armed with the concepts of *term*, *group*, and *block* from section 3.1, we can explain more details about syntax patterns and templates that section 2.4 glosses over. These details make patterns and templates convenient in practice.

A Rhombus syntax object is an example of *concrete syntax* (Aasa et al. 1988). A syntax object is a shrubbery form that is enriched with source-location and binding information at the term level. A syntax object can contain a single term, a multi-term group, or a multi-group sequence. For example, the syntax object `'1 + 2'` represents a single group with three terms: the integer `1`, the operator `+`, and the integer `2`. The syntax object `'f(1, 2)'` is a group with two terms, and the second term has two nested groups each with a single term: `1` and `2`, respectively. The syntax object `'print("hi"); print("bye")'` is a two-group sequence, where each group starts with the term `print`; the same syntax object (except for source locations) could be written with a newline instead of `;`. A term's source location sticks with the term when it is matched by a macro pattern and expanded into another context.

Syntax patterns in Rhombus match raw shrubbery forms, not parsed Rhombus expressions. In most contexts, an escape in a pattern is matched against a single term.

```
> match '1 + 2 + 3'
  | '$a + 3': "does not get here"
  | '$a + $b + 3': [a, b]
['1', '2']
```

In this example, a repetition match could be used, instead, to match any leading sequence of terms:

```
> match '1 + 2 + 3'
  | '$a ... + 3': [a, ...]
['1', '+', '2']
```

That repetition sequence can be put into a list, as above, or it can be put into a syntax object that represents a multi-term group:

```
> match '1 + 2 + 3'
  | '$a ... + 3': '$a ...'
'1 + 2'
```

When a `...` is the only term within a group that follows a group, then it matches repetitions of the preceding group, and escapes within the repeated group are bound as repetitions for corresponding matching parts.

```
> match '(1 + 2, 3 * 4, 5 - 6)'
  | '($n $op $m, ...)': [op, ...]
['+', '*', '-']
```

These rules would be enough to write Rhombus macro patterns, but working always at the term level can become tedious. In many cases, `...` repetitions would be needed to generalize terms to sequences. Rhombus streamlines pattern matching of syntax by adopting a few additional rules:

- The end of a group in a pattern is special. An escape in that position is allowed to match a sequence of terms without using `...`:

  ```
  > match '1 + 2 + 3'
    | '1 + $c': c
  '2 + 3'
  ```

- Along similar lines, if an escape is the only form within a block, then it is allowed to match a sequence of groups:

  ```
  > match 'thunk:
             def x = 1
             x + 1'
    | 'thunk: $body': body
  'def x = 1; x + 1'
  ```

- To insist on a single-term match, an escape can be annotated with the `Term` syntax class:

  ```
  > match 'thunk: def x = 1; x + 1'
    | 'thunk: $(body :: Term)': body
    | ~else: "no match"
  "no match"
  ```

  The `Group` syntax class similarly constrains a match to a single-group match, and it can only be used in an escape at the end of a group.

Rhombus programmers can define their own syntax classes, along the same lines as `define-syntax-class` and `define-splicing-syntax-class` in Racket (Culpepper and Felleisen 2012).

Templates are more permissive than patterns because an escape in any position automatically splices a multi-term group. The following example splices an unparsed sequence:

```
> match '1 + 2 + 3'
  | '1 + $c': '4 * $c'
'4 * 2 + 3'
```

Beware that the result `'4 * 2 + 3'` computes a different number than `'4 * (2 + 3)'` would. This is because match and substitution are faithful to the shrubbery structure, but not to an expression

```
import: rhombus/meta open

defn.macro 'datatype $(type :: Identifier)
            | $(variant :: Identifier)($field, ...)
            | ...':
  'interface $type
   class $variant($field, ...): implements $type
   ...'

datatype Type
| VarType(t :: String)
| ArrowType(dom :: Type, rng :: Type)

datatype Expr
| Lambda(id :: String, t :: Type, body :: Expr)
| App(rator :: Expr, rand :: Expr)
| Var(id :: String)
```

Fig. 5. Defining an ML-like datatype form on top of Rhombus's `interface` and `class` forms

structure that involves precedence for the `*` and `+` operators. Use the `expr_meta.Parsed` syntax class to parse a term sequence into an expression and prevent the splicing:

```
> match '1 + 2 + 3'
  | '1 + $(c :: expr_meta.Parsed)': '4 * $c'
'4 * #{(parsed #:rhombus/expr (+ (quote 2) (quote 3)))}'
```

The output here shows `#{}` and `parsed` within the result syntax object. The `#{}` wrapper is a shrubbery-level escape to S-expression notation, and a `parsed` S-expression is like an atom in that it is opaque to further shrubbery pattern matching. Meanwhile, the content of the `parsed` form is a Racket expression, because that's the meaning of parsing for Rhombus expressions. Different contexts have different parsed forms, but they are all represented as opaque S-expression objects.

Having to write `expr_meta.Parsed` in all macro definitions would be tedious and error-prone. As we will see in section 4.1, macro-definition forms automatically convert certain escapes into parsed-form escapes; the intent is that most macro authors will not need to deal with this detail.

4 MACROS AND EXPANSION

Shrubbery patterns and templates provide the starting point for Rhombus macros, which use those facilities at expansion time (a.k.a. compile time) to transform code. A macro-definition form like `macro` or `expr.macro` creates a bridge between a run-time context for macro uses and an expansion-time context for the macro's implementation. In this section, we show how macros are defined in Rhombus, and then we provide details about the implementation of macro expansion.

4.1 Defining Macros

The examples in section 2.5 demonstrate expression macros, but Rhombus additionally supports definition macros, binding macros, and several other forms. As an example, Figure 5 uses a definition macro (via `defn.macro` from `rhombus/meta`) to create an ML-like `datatype` form using Rhombus's `class` and `interface` forms. The `type` name and `variant` names are constrained to be identifiers via the `Identifier` syntax class. The `field` escape matches any sequence of terms, including a name with an annotation, because it is the only escape within its group.

The `type_of` function in figure 6 best reflects the notation of its domain with calls written as `type_of(env ⊢ expr)`. Figure 6 defines an `⊢` expression form to pair the environment and expression

```
expr.macro '$env ⊢ $expr': ~weaker_than ~other; '[$env, $expr]'
bind.macro '$env ⊢ $expr': '[$env :: Env, $expr :: Expr]'

expr.macro '$dom → $rng': 'ArrowType($dom, $rng)'
bind.macro '$dom → $rng': 'ArrowType($dom, $rng)'

fun | type_of(Γ ⊢ Lambda(id, t, body)):
        let body_type = type_of(Γ ++ {id: t} ⊢ body)
        t → body_type
    | type_of(Γ ⊢ App(rator, rand)):
        match type_of(Γ ⊢ rator)
        | dom → rng:
            if dom != type_of(Γ ⊢ rand):
            | error("actual and formal domain mismatch")
            | rng
        | ~else error("not a function")
    | type_of(Γ ⊢ Var(id)):
        Γ[id]
```

Fig. 6. Defining `type_of` with `⊢` and `→`

as a list, and it also defines `⊢` as a binding form so that the formal argument of `type_of` can be written with `⊢`. While we're at it, `→` is defined as an alias for `ArrowType`. The `⊢` and `→` expression forms could have been implemented more concisely using `operator`, but we use `expr.macro` here to focus on the kind of macro definition that `operator` generates.

In the definition of the `⊢` expression and binding forms, the `env` and `expr` escapes are intended to match multi-term expression and binding forms on either side of the `⊢`. Since that intent is the most common case, macro definition forms like `expr.macro` and `bind.macro` implicitly treat escapes as parsed terms when the overall pattern has the shape of a prefix, infix, or postfix operator pattern. That implicit treatment of escapes corresponds to annotating the escape with `expr_meta.Parsed`, `bind_meta.Parsed`, or whatever syntax class corresponds to the context.

Treating the left-hand side of the pattern as a parsed term turns out to be necessary for the overall parsing algorithm to discover an infix operator. For a right-hand side, Rhombus's parsing strategy allows more flexibility. For example, the `.` operator for field access is implemented as an infix macro that expects an expression on its left-hand side, but its right-hand side must be an identifier (not an expression) that is used as the name of a field. The pattern for `.` uses a syntax class to override the treatment of its right-hand side pattern:

```
expr.macro '$obj . $(field :: Identifier)':
  resolve_dot(obj, field)
```

This implementation relies on a `resolve_dot` expansion-time function to perform the main work of the `.` expansion and return a syntax object. Calling a helper function is allowed because, unlike `macro`, the `expr.macro` form allows an arbitrary expansion-time expression to implement a macro (in exchange for importing expansion-time bindings via `rhombus/meta` or similar).

The definition of `.` still does not use the most general form of a macro transformer. The most general form accepts all remaining terms of the enclosing group, consumes as many terms as it wants, and returns two values: the expansion result and any remaining terms. In that case, the macro is free in its choice of how to consume terms, but it can consume only terms from the remainder of the group; it cannot affect parsing in later parts of the token tree.

As an example of the most general infix macro form, the following `no_fail` macro catches an exception and conditionally replaces it with either the result of the expression after `no_fail` if

```
⟨document⟩ ::= (top ⟨group⟩ ...)
⟨group⟩    ::= (group ⟨term⟩ ... ⟨tail-term⟩)
⟨term⟩     ::= ⟨atom⟩ | (op ⟨symbol⟩)
             | (parens ⟨group⟩ ...) | (brackets ⟨group⟩ ...)
             | (braces ⟨group⟩ ...) | (quotes ⟨group⟩ ...)
⟨tail-term⟩ ::= ⟨term⟩ | ⟨block⟩ | (alts ⟨block⟩ ...)
⟨block⟩    ::= (block ⟨group⟩ ...)
```

Fig. 7. Representation of a shrubbery form as an S-expression form

one is supplied, or `#false` by default. To implement that choice, the macro accepts all terms after `no_fail` and detects the case that the sequence is empty, parsing the sequence as an expression if it is non-empty. The macro returns two syntax objects. The first is its expansion and the second is the (possibly-empty) unparsed tail of the sequence:

```
expr.macro '$expr no_fail $tail ...':
  ~weaker_than ~other
  match '$tail ...'
  | '': values('try: $expr; ~catch _: #false', '')
  | '$(rhs :: expr_meta.AfterInfixParsed('no_fail'))':
      values('try: $expr; ~catch _: $rhs',
             '$rhs.tail ...')
```

```
> 1/0 no_fail
#false
> fun divide(x, y): x/y no_fail "undefined for " +& x +& " and " +& y
> divide(1, 0)
"undefined for 1 and 0"
```

The second match case in `no_fail` uses the `expr_meta.AfterInfixParsed` syntax class, which is like `expr_meta.Parsed`, but it stops at an operator with lower precedence than a given one (`no_fail` in this case). The `expr_meta.AfterInfixParsed` syntax class delivers both the parsed expression as `rhs` and the remaining unparsed terms as a repetition `rhs.tail`. Shorthands that implicitly parse expressions or preserve tail terms can be implemented using this general form, and a postfix macro can be implemented as an "infix" macro that consumes no terms after the operator. A prefix macro has a similar general form, but without a parsed left-hand side.

4.2 Expansion and Enforestation Algorithm

Enforestation and macro expansion in Rhombus are driven by the Racket macro expander. Toward that end, the Rhombus implementation encodes shrubbery forms as S-expression forms using the grammar shown in figure 7. The encoding includes a `top` wrapper for a top-level sequence of ⟨*group*⟩s, includes an `op` form to distinguish operators from identifiers (since both are represented as symbols), and uses `block` for both a ⟨*block*⟩ and ⟨*alt*⟩. Here's an example of a shrubbery form and its S-expression encoding:

```
fun f(x :: Int, y :: Int):
  x * y + 1
```

```
(top (group fun f (parens (group x (op ::) Int)
                          (group y (op ::) Int))
           (block
            (group x (op *) y (op +) 1))))
```

The example illustrates that the S-expression encoding is too verbose for direct use, despite being a convenient vehicle to inherit Racket's hygiene, optimizing compiler, and IDE support (Findler et al. 2002; Flatt 2016; Flatt et al. 2019).

```
⟨term⟩ ::= ..... | ⟨tree⟩
⟨tree⟩ ::= (parsed ⟨expr⟩)
⟨expr⟩ ::= ⟨identifier⟩ | (⟨expr⟩ ...) | (lambda (⟨id⟩ ...) ⟨expr⟩)
         | ..... | (rhombus-expression ⟨group⟩)
```

Fig. 8. Extension of figure 7 to represent mixtures during expansion

Rhombus expansion involves a mixture of shrubberies and parsed trees. To support this mixture, figure 8 extends the grammar of ⟨*term*⟩ to include `(parsed` ⟨*expr*⟩`)`, where ⟨*expr*⟩ represents a Racket expression form; we use the non-terminal ⟨*tree*⟩ as a shorthand for `(parsed` ⟨*expr*⟩`)`. A Racket `rhombus-expression` form as an ⟨*expr*⟩, meanwhile, can wrap a ⟨*group*⟩ to return to Rhombus expansion of (S-expression encodings of) shrubbery forms. This mixture allows parsing of outer forms that produce definitions, which are handled by Racket's macro expander, to influence parsing of more nested forms that are delayed under `rhombus-expression`.

While Lisp-style macro expansion is driven by the initial term of a parenthesized sequence, ⟨*group*⟩ parsing in Rhombus needs a strategy that supports prefix, infix, and postfix operators. The remainder of this section describes a variant of *enforestation* (Rafkind and Flatt 2012) as realized for Rhombus as an expansion-time `enforest` function, where enforestation is a variant of Pratt precedence parsing (Pratt 1973). The `enforest` here is more general than the original presentation because it works beyond expression contexts and because it accommodates operators where the right-hand side is not parsed (as for the `.` operator).

The `enforest` function takes a sequence of ⟨*term*⟩s and returns a parsed ⟨*tree*⟩. More generally, `enforest` takes a sequence of ⟨*term*⟩s plus a current infix or prefix ⟨*name*⟩, so it can stop at infix names that have a lower precedence. The result from `enforest` is a parsed ⟨*tree*⟩ plus a sequence of ⟨*term*⟩s that remain to be parsed. The `rhombus-expression` Racket macro starts enforestation with a dummy ⟨*name*⟩ whose precedence is lower than all other names, so parsing will either consume all terms or fail due to an unbound name.

Figure 9 shows the essential cases of the `enforest` function. In the first case of `enforest`, a variable reference parses as itself. The next three cases use a `lookup` function that is supplied by the macro expander to access expansion-time values for macro bindings:

- When the input sequence starts with a name that is bound as a prefix macro, then the macro's ⟨*transformer*⟩ function is called. The function receives a sequence of ⟨*term*⟩s, and it returns a parsed ⟨*tree*⟩ plus a sequence of remaining ⟨*term*⟩s. Technically, there's no requirement that the remaining terms are a tail of the input sequence, but that's the intent. Enforestation recurs with the parsed ⟨*tree*⟩ and remaining ⟨*term*⟩s as the new sequence.
- When the input sequence starts with a parsed ⟨*tree*⟩ followed by a ⟨*name*⟩ that is bound as an infix macro, and when the name has a higher precedence than current name (as represented by the $>$ relation), then the infix macro's ⟨*transformer*⟩ function is called. The function receives the parsed ⟨*tree*⟩ plus the remaining ⟨*term*⟩s.
- When the input sequence starts with a parsed ⟨*tree*⟩ and a ⟨*name*⟩ that is bound as an infix macro but with lower precedence, then enforestation pauses. The returned ⟨*tree*⟩ is likely used as the parsed right-hand side for a prefix or infix transformation in progress.

The shorthands `Prefix` and `Infix` shown in figure 9 illustrate how a transformer for a prefix or infix macro can recur to `enforest` to parse a right-hand side. The shorthand constructors take a ⟨*transformer*⟩ that consumes a parsed right-hand form, instead of a term sequence, and the ⟨*transformer*⟩ is wrapped in a new function that combines it with a use of `enforest`. The ⟨*name*⟩ provided to `Prefix` and `Infix` is meant to be the same as the one bound to the macro, and the Rhombus forms that trigger this shorthand use the same name automatically. The most general protocol

variable

enforest⟦(⟨name⟩ ⟨term⟩ ...), ⟨name⟩$_{\text{after}}$⟧ ⇒ enforest⟦((parsed ⟨name⟩) ⟨term⟩ ...), ⟨name⟩$_{\text{after}}$⟧
where lookup⟦⟨name⟩⟧ ⇒ Variable

prefix

enforest⟦(⟨name⟩ ⟨term⟩ ...), ⟨name⟩$_{\text{after}}$⟧ ⇒ enforest⟦(⟨tree⟩ ⟨term⟩$_{\text{rest}}$ ...), ⟨name⟩$_{\text{after}}$⟧
where lookup⟦⟨name⟩⟧ ⇒ PrefixMacro(⟨transformer⟩),
and ⟨transformer⟩((⟨term⟩ ...)) ⇒ (⟨tree⟩ ⟨term⟩$_{\text{rest}}$ ...)

infix

enforest⟦(⟨tree⟩$_{\text{lhs}}$ ⟨name⟩ ⟨term⟩ ...), ⟨name⟩$_{\text{after}}$⟧ ⇒ enforest⟦(⟨tree⟩ ⟨term⟩$_{\text{rest}}$ ...), ⟨name⟩$_{\text{after}}$⟧
where lookup⟦⟨name⟩⟧ ⇒ InfixMacro(⟨transformer⟩),
⟨name⟩ > ⟨name⟩$_{\text{after}}$,
and ⟨transformer⟩(⟨tree⟩$_{\text{lhs}}$, (⟨term⟩ ...)) ⇒ (⟨tree⟩ ⟨term⟩$_{\text{rest}}$ ...)

enforest⟦(⟨tree⟩$_{\text{lhs}}$ ⟨name⟩ ⟨term⟩ ...), ⟨name⟩$_{\text{after}}$⟧ ⇒ (⟨tree⟩$_{\text{lhs}}$ ⟨name⟩ ⟨term⟩ ...)
where lookup⟦⟨name⟩⟧ ⇒ InfixMacro(⟨transformer⟩),
and ⟨name⟩ < ⟨name⟩$_{\text{after}}$

shorthands

Prefix(⟨transformer⟩, ⟨name⟩) = PrefixMacro(λ ((⟨term⟩ ...)).
enforest⟦(⟨term⟩ ...), ⟨name⟩⟧ ⇒ (⟨tree⟩$_{\text{rhs}}$ ⟨term⟩$_{\text{rest}}$ ...)
(⟨transformer⟩(⟨tree⟩$_{\text{rhs}}$) ⟨term⟩$_{\text{rest}}$ ...))

Infix(⟨transformer⟩, ⟨name⟩) = InfixMacro(λ (⟨tree⟩$_{\text{lhs}}$, (⟨term⟩ ...)).
enforest⟦(⟨term⟩ ...), ⟨name⟩⟧ ⇒ (⟨tree⟩$_{\text{rhs}}$ ⟨term⟩$_{\text{rest}}$ ...)
(⟨transformer⟩(⟨tree⟩$_{\text{lhs}}$, ⟨tree⟩$_{\text{rhs}}$) ⟨term⟩$_{\text{rest}}$ ...))

implicits

enforest⟦(⟨atom⟩ ⟨term⟩ ...), ⟨name⟩$_{\text{after}}$⟧ ⇒ enforest⟦(#%literal ⟨atom⟩ ⟨term⟩ ...), ⟨name⟩$_{\text{after}}$⟧
where ⟨atom⟩ is not a ⟨name⟩

enforest⟦((parens ⟨group⟩ ...) ⟨term⟩ ...), ⟨name⟩$_{\text{after}}$⟧
⇒ enforest⟦(#%parens (parens ⟨group⟩ ...) ⟨term⟩ ...), ⟨name⟩$_{\text{after}}$⟧

enforest⟦(⟨tree⟩ (parens ⟨group⟩ ...) ⟨term⟩ ...), ⟨name⟩$_{\text{after}}$⟧
⇒ enforest⟦(⟨tree⟩ #%call (parens ⟨group⟩ ...) ⟨term⟩ ...), ⟨name⟩$_{\text{after}}$⟧

.....

Fig. 9. Enforestation function implementation

for Rhombus macros corresponds to using PrefixMacro and InfixMacro directly, calling enforest indirectly through the `expr_meta.AfterPrefixParsed` or `expr_meta.AfterInfixParsed` syntax class.

The use of < and > for precedence comparisons suggests an order, but whether to apply an infix transformer depends only on the current and new ⟨*name*⟩s, so it can be any relation. As in Fortress (Steele et al. 2011), the comparison consults only information that is declared with one of the two names in reference to the other. That information includes both precedence and associativity, and the comparison reports an error if precedence and associativity information from the two names is inconsistent or inconclusive.

The remaining cases of enforest enable control over the meaning of a literal, a parenthesized term, a function-call form, and so on. The enforest function reifies the implicit form by inserting an explicit form name: `#%literal`, `#%parens`, `#%call`, or other implicit names (not shown) to cover the use of square brackets, curly braces, and quotes.[9] By default, these explicits consume the first term of a sequence, process it, and return the tail unchanged. If an explicit name is unbound, then the implementation of enforest in Rhombus substitutes a transformer that reports a specific error, which is more helpful than an unbound-identifier error.

[9] A `#%` prefix is generally allowed for shrubbery identifiers. The prefix is special only in that it connotes a name that is normally not written out, but explicit use of the name is also allowed.

5 SPACES

A Rhombus *space* represents a particular kind of program context, such as an expression context, binding context, or annotation context. A Rhombus module starts out in an expression context, and then some expression forms create other kinds of contexts, such as the binding context created for the left-hand side of `def` or the annotation context created on the right-hand side of `::`. Each space has its own sublanguage of forms that are implemented as macros specific to that space, but the same name can have a meaning in multiple spaces. For example, a class name like `Posn` works as a constructor in an expression context, as a pattern constructor in a binding context, and as an instance check in an annotation context.

Other languages similarly allow different bindings for different program contexts, such as expression versus pattern forms in Relit (Omar and Aldrich 2018) or extending different grammar productions in Fortress (Allen et al. 2009). Rhombus uses a different strategy where support for binding spaces is built directly into the representation of binding within a syntax object.

5.1 Space Scopes

Rhombus inherits Racket's handling of scope for macros, because it builds on Racket's macro expander and uses Racket syntax objects to implement Rhombus syntax objects. Through Racket's syntax objects, the definitions reached by an identifier are represented as a *set of scopes* (Flatt 2016), which generalizes normal lexical scope. Scope sets accommodate identifiers that are introduced by macro expansion, including identifiers that are put into mutually recursive definition contexts or transported from one module to another by expansion of an imported macro.

Rhombus takes further advantage of scope sets to implement spaces. Each space has a distinct *interned* scope, so the scope for a particular space is common to all modules and across all phases of expansion and evaluation. This shared scope allows different modules to cooperate by defining and referencing names in a way that is specific to the space. When a name is defined in a particular space, such as the repetition space, an interned scope is added to the defined name in addition to whatever other scopes the name has acquired through expansion. Similarly, when resolving a name in a particular sublanguage, an interned scope is first added to the name before attempting to find the definition through the Racket macro expander's lookup mechanism. Interned scopes were first added to Racket to implement Hackett (King 2018), which is a Haskell-like language that also needs to define names with different meanings in different sublanguages (e.g., types and expressions).

The extra scope explains how `⊢` can be bound both as an expression operator and binding operator in figure 6. The `expr.macro` form defines names in the expression space, while the `bind.macro` form defines names in the (Rhombus) binding space; the latter has an extra scope that the former does not, making the definitions distinct.

When a name is shadowed in some spaces but not others, there's a risk of breaking up sets of bindings that are meant to work together. As an example, imagine that the `::` operator is locally defined as an alias for `List.cons`, instead of checking a value against an annotation. If the expression and binding spaces have separate scopes, then the new `::` definition for expressions would be out-of-sync with `::` for binding contexts, where it would continue to associate a pattern with an annotation. To reduce this kind of mismatch, the Rhombus expression space does not use an extra scope. Shadowing `::` in the expression space—without also defining it in the binding space—has the effect of disabling `::` in the binding space, because a name with both the binding scope and the shadowing scope will have candidate definitions where neither scope set is a superset of the other, and so neither binding applies. This special treatment can work for only one space, but definitions in the expression space are by far the most common and (based on experience in earlier iterations of Rhombus) the most likely to create confusing mismatches. We experimented with

```
import:
  rhombus/meta open
  "orig.rhm" as orig

export:
  all_from(.orig): except_space bind
  ⊢: only_space bind

bind.macro '$env ⊢ $expr': ~weaker_than ~other; '$env orig.(⊢) $expr'
```

Fig. 10. Assuming that figure 6 is exported by `orig.rhm`, replace `⊢` for only the binding space

instead using compile-time multiple inheritance to enable implementation for multiple spaces and using per-space scopes as a secondary route to extensibility—an approach that minimizes mismatches, but proved tedious and inflexible in practice.

When a name is exported from a module using the `export` form, the name is exported in all spaces that have a definition. The `import` and `export` forms support modifiers that omit bindings from specific spaces or propagate bindings only from specific spaces. A module can thus fill in definitions for a name in new spaces, or it can replace definitions of a name in some spaces while re-exporting definitions for other spaces.

For example, the definition of the binding form in figure 6 lacks a precedence specification. Figure 10 demonstrates how to add one without modifying the source file (`"orig.rhm"`): `import` the original implementation, export a new implementation, and define the new `⊢` that includes a precedence declaration. Within `export`, the `only_space bind` modifier is not actually needed, since the only definition of `⊢` is the new one defined with `bind.macro`, while the original is accessed through the qualified path `orig.(⊢)`. The syntax `all_from(.orig)`, meanwhile, refers to all bindings provided by the module that is imported as `orig`. The space name `bind` is imported from `rhombus/meta`.

5.2 Defining New Spaces

Rhombus enables the creation of new spaces. Suppose that we want a form `rx()` for writing regular expressions, where `*` and `?` within `rx()` have their usual meaning for regular-expression notation, and strings always represent literal sequences. For example, `rx("."? "^"*)` should match an optional `.` followed by any number of `^`s, independent of whether `.` and `^` are also regular-expression operators. Matching can be implemented by using an existing Racket matcher for regular expressions, but that matcher works in terms of a string encoding like `"[.]?\\^*"`; the `rx()` form can expand to that encoding while proving a more naturally composable syntax. The module that defines `rx` can define all of the basic regular-expression operators, but we can also allow programmers to define their own operators.

The `rx` form should use a new space for regular expression operators like `*` and `?`. Selecting a new interned-scope identity is only one step of defining that new space, and the full definition includes several pieces, demonstrated in figure 11:

- A name `regexp` that is bound to the space, analogous to `bind` as used in figure 10 to refer to the space of binding operators. This name doubles as a path qualifier to access the macro-definition form that binds in the space, analogous to the `bind` prefix of `bind.macro`.
- A globally unique path for the space, `my/regexp/space`, which is used for interning the space's scope. This path could be derived automatically from the enclosing module's path, since the module path is globally unique, but currently it must be written explicitly.

- A name for the space's macro-definition form, `macro`, analogous to the `macro` part of `bind.macro`. This name is practically mandatory, because the space is useful only if it includes some way of adding definitions.
- Names for an expansion-time namespace prefix (`regexp_meta`) and for enforestation-triggering syntax classes (`Parsed`). The prefix is analogous to the `bind_meta` prefix in `bind_meta.Parsed`, and typical syntax classes correspond to `Parsed`, `AfterPrefixParsed`, and `AfterInfixParsed`. The `Parsed` name is practically mandatory, but the latter two can be omitted if no macros need to use the annotations. The syntax classes are bound in the expansion-time phase relative to the enclosing context, as opposed to the run-time phase, since they are intended for use by macros.

The `rx` expression macro in Figure 11 bridges to the new space by using the newly defined `regexp_meta.Parsed` syntax class. That syntax class has the effect of converting a regular expression into a string encoding that can be passed on to the `pregexp` function from Racket. Four uses of `regexp.macro` define four regular-expression operators that implement the conversion: a `#%literal` prefix operator that is implicitly applied to literals like strings, a `#%juxtapose` infix operator that is implicitly placed between expressions that are not separated by an infix operator, and the `*` and `?` postfix operators. The implementation of `#%literal` uses an existing literal-to-pattern conversion function from Racket.[10] The implementations of other operators manipulate the string form of subexpressions that are parsed as regular expressions, which means that they are represented as string syntax objects, and `unwrap` accesses the raw strings.

Exporting the `rx` expression form allows other modules to use it, but operators in the `regexp` space also need to be exported. Four regular expression operators are exported with the modifier `only_space regexp` to avoid re-exporting bindings from other spaces, such as the plain old `*` expression operator. Exporting `regexp` and `regexp_meta` provides access to `regexp.macro` and `regexp_meta.Parsed`, so other modules can define new regular-expression operators.

6 STATIC INFORMATION

By default, Rhombus is a dynamic language in the same sense as JavaScript or Python: field and method names using `.` are found dynamically at run time, array-like access with `[]` is dispatched dynamically to a suitable lookup function, and so on. This mode of operation can be convenient, but it can also create problems for performance and maintainability. In Lisp, a common fix is to replace the dynamic, procedure-based interface with a static, macro-based one (for example, (Bowman et al. 2015)). Rhombus adapts ideas from this line of work into the base language. Specifically, it builds on the binding protocol for type systems as macros (Chang et al. 2017) but without forcing an expansion order.

Static information is currently used in Rhombus to improve performance and reject simple mismatches. In the scope of a `use_static` declaration, uses of `.` or `[]` are rejected when they would trigger a dynamic lookup, and function calls are rejected when they have the wrong argument count or wrong keywords. Those features reflect one way of using static information, but Rhombus's approach lets us explore different points on the spectrum of types and static information. Exploring at the macro level offers a path that is flexible and incremental. Furthermore, exploration is not solely the province of the Rhombus base language, because Rhombus exposes all of the mechanisms needed to explore alternatives through new sets of cooperating macros and spaces. Section 7.2 describes an ML-style type checker implemented in this style.

[10] The use of `#{}` in `#%literal` escapes to S-expression notation, which is needed to refer to a name that is not a shrubbery identifier, because the name includes a hyphen.

```
space.enforest regexp:
  space_path my/regexp/space
  macro_definer macro
  meta_namespace regexp_meta:
    parse_syntax_class Parsed

expr.macro 'rx ($(pat :: regexp_meta.Parsed))':
  'pregexp($pat)'

regexp.macro '#%literal $(s :: String)':
  import lib("racket/base.rkt").#{regexp-quote}
  '$(#{regexp-quote}(s.unwrap()))'

regexp.macro '$left #%juxtapose $right':
  '$(left.unwrap() ++ right.unwrap())'

regexp.macro '$left *':
  ~stronger_than: #%juxtapose
  '$("(?:" ++ left.unwrap() ++ ")" ++ "*")'

regexp.macro '$left ?':
  ~stronger_than: #%juxtapose
  '$("(?:" ++ left.unwrap() ++ ")" ++ "?")'

export:
  rx
  only_space regexp: #%literal #%juxtapose * ?
  regexp
  meta: regexp_meta
```

Fig. 11. Implementing a new space for regular-expression operators and parsing

6.1 Binding Protocol

When a function argument is annotated with `:: List` or `:: Posn` as in section 2, the definition of the argument variable is paired with a definition of the same name in a static-information space, where the latter communicates that the variable's value is a list or an instance of the `Posn` class, respectively. When a binding `p :: Posn` is referenced in `p.x`, the `.` operator can consult static information associated with `p` to discover an efficient, `Posn`-specific accessor for `x`.

The propagation of static information from a function argument to the function body is not specific to function-argument handling, but instead built into the general protocol for expanding binding positions. The parsed representation of a binding has four parts:

- a set of names to be defined by the expansion in the expression space, each with a repetition depth, and each with static information to associate with the name;
- an expression to check whether a value in a designated input variable matches the binding, so that the next pattern-matching alternative (if any) can be tried;
- definitions that extract elements of a matched value by binding private intermediate variables; and
- a final set of definitions, possibly spanning multiple spaces and possibly using private variables that are introduced by the extraction step.

Splitting a binding form into these pieces allows binding forms to compose. For example, a `[Posn(x, y), ...]` binding pattern is a list pattern, but the list-pattern expansion needs information

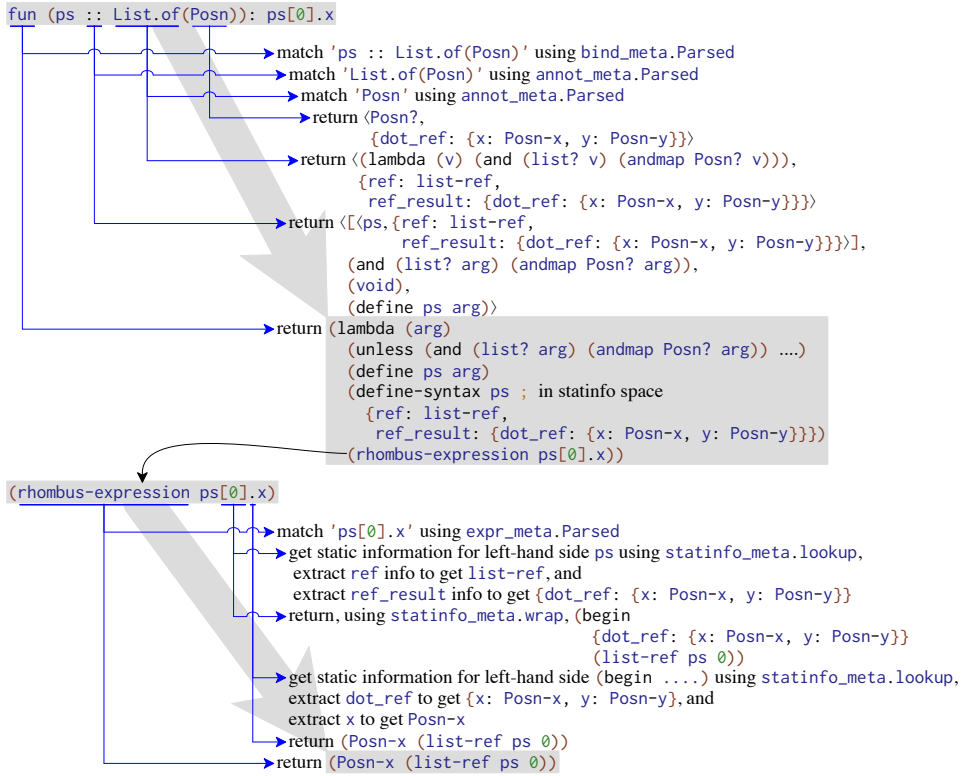


Fig. 12. Sketch of expansion steps, showing how static information is derived from annotations in a binding, then used and propagated by expressions. The right-hand column shows the steps that are taken, in order, with slight indentation to reflect nested expansion. The lines on the left identify the macro whose expansion performs the step. Rhombus-like map notation is used to represent key–value static information.

from the expansion of `Posn(x, y)` as a pattern to construct the overall list pattern's expansion. In particular, it needs to know that `Posn(x, y)` binds `x` and `y` at repetition depth 0 so that it can convert the associated values to actually bind `x` and `y` each as a repetition of depth 1. Similarly, the match-checking expression for the list expansion must detect a list argument and then map the match-checking expression from `Posn(x, y)` over the list. If the `Posn` class is defined with annotations on its fields, a `Posn(x, y)` pattern can propagate each field's static information to `x` and `y`, respectively. The `::` binding operator similarly gets static information from the annotation on its right-hand side to propagate to identifiers bound by its left-hand side.

An annotation's expansion, meanwhile, has two parts: a predicate that can be used (e.g., by matching) to determine whether a value satisfies the annotation, and static information that applies to any expression or definition that satisfies the annotation. Those two pieces are, again, composable to support annotation operators and constructors like `List.of()`, which takes another annotation and constructs a new one that corresponds to a list whose elements satisfy the given annotation.

The first half of figure 12 sketches the expansion steps of a binding with a `List.of(Posn)` annotation. The sketch shows how annotation information feeds into a binding's expansion, and then how a binding's expansion is incorporated into an expanded function body.

6.2 Downward and Upward Information

Static information on a defined name is a kind of "downward" information flow, because it goes from definitions to uses of the defined name that are typically later in the source text. Some situations benefit from an "upward" flow from a subexpression to an enclosing form. For example, if `ps` has static information from `ps :: List.of(Posn)`, then resolving `p[0].x` to an efficient access of the `x` field requires the expansion of the left-hand side `p[0]` to report the static information of `Posn`.

An expression macro communicates upward static information by expanding to a particular Racket expression form: `(begin (quote-syntax` ⟨*info*⟩`)) ⟨`*expr*`⟩)` where ⟨*info*⟩ is a packed form of static information and ⟨*expr*⟩ is the Racket expression that parsing would otherwise produce. The Rhombus helper function `statinfo_meta.wrap` performs this wrapping, while `statinfo_meta.lookup` extracts static information from either this pattern or from binding information (but the latter only when the given expression is an identifier). The second half of figure 12 illustrates in more detail the expansion a function body that contains `ps[0].x`.

Upward flows appear in bindings as well as expressions. For example, a binding of the form `[p, ...] :: List.of(Posn)` communicates information "upward" from the right-hand side of `::` into the binding expansion of the list pattern on the left. To support that kind of transfer, the first component of a binding expansion is not actually a list of names to be defined, each with its static information, but instead a function that takes upward information and *then* reports the names, each with static information that may incorporate information from the upward flow.

6.3 From Static Information to Types

In the limit, downward and upward flows can be combined with unification to implement general type checking and inference (see section 7.2). Interleaving partial inference and partial expansion can even enable type-directed macro expansion. The protocols for expansion and propagation can become complex, since the approach amounts to opening up the implementation of an expressive type system to fine-grained extension; the complexity can be addressed through a domain-specific language (Chang et al. 2017) or an expander that can pause a macro until enough information is available (Barrett et al. 2020). The Rhombus base language currently takes a more modest approach, demanding upward information only in limited contexts, and in particular not forcing the expansion of an expression within a block to propagate information outside the block (which avoids a swath of order-of-expansion problems, especially in recursive definition contexts).

Rhombus's protocol for static information parallels the flexibility of macros and spaces: instead of a single type system that all static information must inhabit, static information has a key–value form that enables different constructs to cooperate through any key that they both support. For example, information keyed by `call_result` is provided by the function-definition form, it is recognized by the function-call form, and any other macro is free to provide or use that key. When a shared vocabulary is absent or information does not match up, then information can be simply ignored, and expansion proceeds the same as if no information is available. At the same time, a macro is free to complain if it insists on information that it cannot find. Rhombus's `use_static` form redefines operators like `.` and `[]` to fail with a static error if they cannot statically resolve to specific accessors.

7 RHOMBUS PROGRAMS

Rhombus is used for building parts of Rhombus and its accompanying libraries, such as drawing and GUI bindings. The Rhombus documentation is written in a document-oriented variant of Rhombus based on Scribble (Flatt et al. 2009), and so is this paper. In an artifact accompanying this paper (Flatt et al. 2023), we provide a set of example programs: Runge-Kutta, which defines lazy streams and array operators; a "heatbugs" simulation and visualization, which has no macros,

but uses the GUI and drawing libraries; and a Rhombus embedding of Esterel (Berry and Gonthier 1992), which includes macros for Rhombus-style syntactic forms layered over a Racket run-time library. The artifact also has complete versions of the programs from the figures. We report here on two larger efforts: a video game framework (section 7.1) and a language for teaching (section 7.2).

7.1 Video Game Framework

One of the authors, while still a relative newcomer to the language, used Rhombus to implement a framework for video games.[11] The framework includes several internal DSLs implemented via macros: one to describe game asset bitmaps, one to describe world maps, one to express behavior trees, one to write dialog, and one to implement an entity-component system (ECS) layer. Figure 13 shows a few fragments of a demo game using the framework's DSLs.

Rhombus's notation made it an attractive implementation vehicle for the author of this project. Shrubbery notation provides the same benefits as S-expressions for writing DSLs without having to consider lower-level parsing concerns, but its rules for grouping and indentation closely match developer intuitions in a way that enables a more ergonomic syntax. For example, in the DSL for expressing dialog, notation similar to the industry standard Ink (Inkle 2023) was neatly expressed through simple Rhombus patterns and templates.

Other DSLs in the framework rely crucially on Rhombus's support for new definition forms, binding forms, and static information. For example, the framework's ECS layer provides a set of definition forms that implement an alternative to a conventional class system. The ECS allows programmers to declare datatypes and behaviors of objects independently and then separately compose them, a ubiquitous pattern in video games. When programmers define new data types in the ECS, the macros define an accompanying annotation form, and through Rhombus's propagation of static information, those annotations influence the behavior of some expression constructs. Specifically, when using a dot accessor after a variable that is annotated as an instance of an ECS datatype, the access is rewritten to an ECS lookup specialized to that datatype, rather than being interpreted as a plain Rhombus field access. The result is an ECS layer that is integrated into Rhombus as if it were a built-in language form.

7.2 Shplait Teaching Language

Shplait[12] is a language for use in an undergraduate course on programming language concepts. Students in the course implement a series of interpreters and type checkers in a functional style following Krishnamurthi (2006). Shplait closely resembles a subset of Rhombus, but it has a type system based on Standard ML, including type inference.

Shplait is implemented in Rhombus. Its expression forms are implemented as Rhombus expression macros, its definition forms are implemented as Rhombus defintion macros, and its type forms are defined in a Shplait-specific space that is created with `space.enforest`. Type information is communicated through Rhombus's framework for static information; expression and definition macros declare and consume type information, and they send constraints to a unification engine that is instantiated when a module is expanded. Polymorphism inference at the module level relies on a finishing pass that is triggered by a macro inserted at the end of a Shplait module.

Shplait is a successor to Plait,[13] which has the same constructs and type system but resembles Racket instead of Rhombus. We ported Plait's unification engine from Racket to Rhombus, but implemented the rest of Shplait fresh. Rhombus's features make the Shplait implementation more

[11] https://github.com/Gopiandcode/rhombus-in-the-rough
[12] https://docs.racket-lang.org/shplait/
[13] https://docs.racket-lang.org/plait/

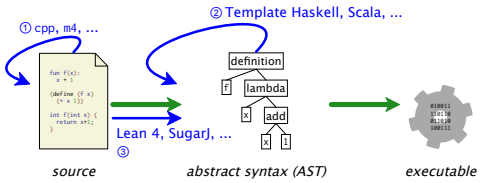
```
// behavior trees examples ------------------
declare_behaviour_state(target, patrol)
behaviour_tree attack_player(this) with state(target):
  all_of:
    check near_player(this) using state(target)
    perform do_attack_player(this, target)
....
behaviour_tree follow_patrol(this) with state(target, patrol):
  first_of:
    attack_player(this)
    follow_player(this)
    perform do_follow_patrol(this, patrol)

// dialog example --------------------------
dialog guard_dialog:
  guard_a: 1 "Halt! Who goes there?" || "Good ~a!" "day" || "Allo Allo..."
  1 "I'm a traveller, passing by" || "Hello again!" || "Just one more thing..."
  branch guard_a: "What do you want...."
  | "Nevermind...."
    ["The guard returns to his position."]
    ~> END
  | "What's up?"
    guard_a: "None-ya business."
    "I'll be on my way then..."
    ~> END

// ECS example ---------------------------
instance Bat with this:
  component position :~ Position = Vector2(280.,350.)
  component velocity :~ Velocity = Vector2(0.,0.)
  method is_aggroed():
    this.has_component(IsAggroedTo)
  method perform_attack():
    state.state := State.Attacking
  ....
```

Fig. 13. Some example uses of DSLs within a video game framework

modular and maintainable than the Plait implementation. Most of Plait lives in a 3.5k-line main module so that the core forms plus a monolithic type-checking traversal can see each other. The comparable part of Shplait is split across more than 40 modules that total only 3k lines of code, primarily because Shplait takes advantage of Rhombus spaces so that type-checking happens during macro expansion. The Rhombus static-information layer replaced the monolithic type-checking traversal.

8 RELATED WORK

A typical program processing pipeline starts with source text, parses to an abstract syntax tree (AST), then compiles the AST to executable code following the thick green arrows below:



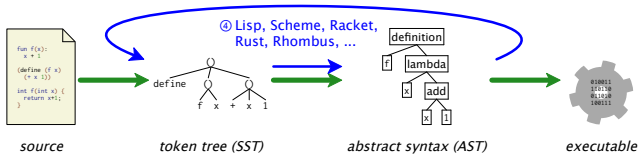
This diagram reflects the general problem of parsing and compilation, and there are many language workbenches to support implementation at that level, including Spoofax (Kats and Visser 2010), Rascal (Klint et al. 2009), and Xtext (Eysholdt and Behrens 2010). A system like Rascal, with its emphasis on pattern matching and term construction using concrete syntax, shares goals and capabilities with Rhombus. More apt comparisons to Rhombus, however, are *macro* systems: languages that allow the grammar of programs to be extended from within the program, as opposed to requiring changes in a toolchain. A language where syntactic extensions are imported from another module counts, as long as the import statement is within the program, instead of specified by changing the compilation command or adjusting a project configuration.

The most primitive forms of macros ①, such as `cpp` and `m4`, add a textual preprocessing step on the left of this picture, before the program is parsed into an AST. There are well-known problems with this approach. Its expressiveness is limited, and there is a potential mismatch between the macro processor's tokenization and scope and the target language's parsing and scope, which allows macros to perform transformations that do not respect the structure of the original program. Rhombus macros are not in this category (nor any of the above).

Instead of textual rewriting, many macro systems instead place expansion in the middle step ②, operating on the AST. Template Haskell (Jones and Sheard 2002), Scala (Burmako 2017; Scala 2023), OCaml PPX (OCaml 2023), and Elixir (McCord 2015) are examples of languages using this strategy. Since the text has been parsed into an AST, transformations naturally respect existing structure, and representing resolved references to bindings in the AST (as opposed to just variable names) provides basic scope management. Macro systems in the tradition of MetaML (Taha and Sheard 2000) and MacroML (Ganz et al. 2001) are also in this category, although at a less expressive point in the spectrum where macros cannot introduce new binding forms. Rhombus expansion is better characterized as operating on a *token tree*, instead of an AST, as introduced below.

Some languages allow extension at the step of parsing into an AST ③, usually through declarations that extend the parsing grammar while simultaneously rewriting the parsed terms to a more primitive form. This approach was used in compilation as far as back the 1960s (Leavenworth 1966) with similar work continuing into the 1990s (Cardelli et al. 1993), although more as part of a compiler than for use within a program. The macro systems of Fortress (Ryu 2009; Allen et al. 2009; Steele et al. 2011) and Maya (Baker and Hsieh 2002) follow this path with declarative extensions to different productions in a grammar, and Lean 4 (Ullrich and de Moura 2020) and Wyvern (Omar et al. 2014) are similar with programmer-extensible productions that are based on types. Fortress and Isabelle (Nipkow et al. 2002) support post-parsing, type-directed disambiguation of operators and grouping. Systems like SugarJ (Erdweg et al. 2011) support parsing extensions that are even more general and still compositional, based on parsing technologies like GLR (Tomita 1985). Structured editors like MPS (JetBrains 2003) or the hybrid approach of Eco (Diekmann and Tratt 2014) accomodate syntax composition by working with ASTs even while editing. These systems can offer greater syntactic choice than Rhombus enables, as discussed below, but we note some trade-offs there.

Lisp and related languages instead add an extra *reader* step in the program-processing pipeline to turn the text input into a semi-structured *token tree*, also known as a *skeleton syntax tree* (SST) (Bachrach and Playford 1999). The SST is an S-expression in the case of Lisp, but other choices are possible, such as Rust's `TokenTree`, Dylan and JSE's SST (Bachrach and Playford 1999, 2001), the Sweet.js reader's output (Disney et al. 2014), or Rhombus's shrubbery representation. Krishnamurthi (2006,page 9) characterizes this parsing strategy as *bicameral syntax*, because the initial SST layer is responsible for one layer of parsing, and then another level of parsing is built on that one, and programs must pass both levels.



Macro expansion ④ applies to the SST representation, instead of an AST. More precisely, expansion involves a mixture of SST terms with parsed terms, since the binding structure discovered by parsing is needed to define and apply macro transformers. Macro expansion therefore involves a back-and-forth between the SST representation and a parsed AST, with SSTs at the leaves in AST nodes that are still being parsed. With some macro systems, including Racket, a macro can force parsing of an expression subtree within its local context, in which case the SST and AST structures are further interleaved. Rhombus takes it a step further, allowing a subtree to be parsed in a designated space, such as the space of expressions, binding patterns, or class clauses.

8.1 Comparing AST and SST Extension

The choice of extension point—AST transformation ②, AST parsing ③, or SST transformation ④—affects the kinds of syntactic extensions that are easily accommodated. In a language that expands ASTs, new syntax constructs must fit the syntax of a core construct until expansion (where a function-call form is a typical choice), while an SST intermediate representation as in Rhombus offers more flexibility to new syntactic forms. SST and AST transformations both constrain extensions to fit a core set of parsing and grouping rules, while custom AST parsing rules at the text level allow more customization with arbitrary grammar productions, especially with a scannerless parser as in SugarJ. Local imports of AST grammar extensions are possible through lazy parsing, as demonstrated in Maya, but languages that allow custom AST parsing rules tend to require those rules at a coarser granularity, such as at the module level as in SugarJ. AST and SST transformers can more easily support local and nested extensions.

Rhombus embraces the constraints imposed by an SST representation, motivated by the same reasoning as for Honu (Rafkind and Flatt 2012), while taking advantage of SST flexibility to enable local and macro-generating macros. The underlying Racket machinery includes a `#lang` mechanism to support arbitrary text parsing at the module level, which is how Rhombus is implemented, but our focus here is on composable language extension using Rhombus's SST representation. Rhombus's parsing framework includes direct support for only prefix and infix operators, but an operator can be bound to a macro that gets control over all subtree terms after the operator, so mixfix forms are also possible. Such mixfix forms work to the degree that rules for internal delimiters can be expressed through infix-operator precedence relations, which is not as general as the longest-match rule of a parser like Lean's.

To make the comparison more concrete, here are some examples of extensions that are *not* supported by Rhombus:

- Anything that does not conform to shrubbery notation. Every new form must first conform to the grammar of shrubbery notation (section 3.1) before macro bindings are considered.
- A macro `m` that affects the parse of `g(c)` in the example `f(m, b); g(c)`. The impact of a macro `m` is limited for two reasons: parentheses around `m`, and the fact that `f(m, b)` is in a different group than `g(c)` in the enclosing sequence.
- A mixfix operator such as `_ +++ _ --- _` (where `_` indicates an argument) independent of other bindings for `+++`. That is, an "infix" `+++` binding could choose to recognize `---`, but that parsing has to be built into the sole binding for `+++` in a given context. In contrast, indepedent

extension would be possible in many systems that are based on extending a grammar, such as Fortress or Lean 4.

Directly comparing Rhombus to Honu (Rafkind and Flatt 2012), Rhombus's enforestation goes far beyond expressions with support for binding contexts, annotation contexts, and user-defined contexts. Additionally, Rhombus's pattern and template system for macros is more directly integrated with non-metaprogramming matching, including generalized support for `...` repetitions. Rhombus also adds a built-in system for static information. When it comes to expression enforestation, Rhombus's approach is essentially the same as Honu's, but with a small generalization to support macro-like infix operators such as `.` instead of requiring an infix operator to have a parsed expression on the right-hand side.

The goal of Rhombus's shrubbery notation is similar to the goal of Gel (Falcon and Cook 2009). Both are meant as a substrate for defining languages that use conventional notation, where the substrate itself does not impose a semantics or binding structure on the language. Shrubbery notation is more constrained; as its authors note, Gel accepts almost any input with balanced grouping symbols (such as parentheses). Shrubbery notation is more picky about identifiers, number, operators, and the delimiters between them, not to mention its newline and indentation requirements. Shrubbery's additional grouping structure puts it halfway between Gel and S-expressions and makes it more convenient for macro patterns and templates.

8.2 Comparing Choices on Expansion and Binding

Another dimension of extension is whether transformations are constrained to pattern-based rewrites, as in Dylan and older Scheme standards, or implemented by arbitrary functions that run at expansion time, as in conventional Lisp macros or Template Haskell. Rhombus supports macros that are implemented by arbitrary functions. These functions can be defined in the same module where they are used, and even within the same binding context where a macro is defined; the module system takes care of phase separation and managing compile-time state (Flatt 2002).

Lisp-style macros have traditionally only supported expansion in expression positions, as opposed to syntactic positions such as bindings, patterns, term construction, or type declarations. Some languages make the specific case of pattern matching extensible through macros or non-macro facilities (Wadler 1987; Martin et al. 2006; Syme et al. 2007; Omar and Aldrich 2018). In a sufficiently capable macro system, authors of new forms can enable extensibility of contexts that appear within those forms (Ballantyne et al. 2020; Dybvig et al. 1986), including forms for pattern matching (Tobin-Hochstadt 2011). Rhombus builds on the macro approach, generalizing the approach through spaces and using it in the base forms so that binding and other positions are pervasively extensible.

Finally, there's the question of whether macros respect and preserve scope, i.e., whether macros are *hygienic* (Kohlbecker et al. 1986; Adams 2015). Rhombus inherits Racket's support for hygienic expansion, as well as its hygiene-bending operations and its approach to definition contexts that can contain a mixture of macro definitions and uses (which fall outside of existing formal definitions of hygiene). Specifically, identifiers in the shrubbery representation are enriched with scope information that cooperates with parsing and macro expansion, allowing macro transformations to preserve binding relationships.

8.3 Expressiveness and Practice

Compared to Rhombus, other macro systems also support hygienic macros that are implemented with arbitrary functions, some allow macro-introduced definition forms, and some others even allow local macro definitions and macro-generating macros. Many other languages also allow operators to be defined with customized precedence and associativity. Rhombus's specific combination of ideas

is new, however, and intended to provide an especially fluid programming experience, matching the convenience and expressiveness that have made macros such an effective tool in Lisp environments.

Macros are already an important feature of the ecosystem in some non-Lisp languages, such as Scala, Rust, and Elixir, where libraries often provide macro-based interfaces. Macros in those languages are still not routinely used to create whole new languages as they are in Racket, however. Rust (2023) supports simple pattern-matching macros, and it provides a `TokenTree`-based interfaces for syntax manipulation at a much lower level, but it does not provide the smoother path that Racket offers to grow simple macros into complex ones. Scala (2023) supports macros as compile-time functions that receive and manipulate AST trees, and it includes pattern and template facilities for working with quoted code. However, its macros are limited to expression contexts; Racket macros are commonly used to generate definitions and imports, and control over definitions is cruicial for implementing new languages through macros. Elixir's support and culture of macros is closer in practice to Racket, reflected in part by the importance of the macro-based Phoenix (2023) framework for web applications. Elixir macros can introduce definition forms, and the macro system provides a form of hygiene. Elixir syntactic extensions manipulate parsed ASTs, which means that they have less control over parsing at that granularity, and its pattern matching and hygiene support are relatively limited.

We expect that Rhombus macros will preserve the kind of fluidity that Racket offers while reducing the gap between the notations that programmers would prefer to use and the ones that are easily expressed. Rhombus's macro system is equipped with enough expressive power to expand all of its own syntactic constructs down to a few λ-calculus-like constructs plus definition forms; it is currently defined in terms of Racket's core forms, but those same forms could just as well be expressed in Rhombus-native shape using shubbery notation. In short, Rhombus offers the first macro system that is uncompromising on both conventional notation and the ability to express its own constructs.

9 CONCLUSION

We have described a combination of ideas that together advance the frontier of macro-extensibility, especially for languages that do not look like Lisp. The combination not only supports infix parsing, but also extensibility for facets of the language such as binding positions and static information. As evidence that this combination is practical and effective, we offer the Rhombus implementation, which is available as a Racket package.

Rhombus is still a work in progress, but it is already a rich language, including forms for functions, classes, interfaces, modules, submodules, local namespaces, loops, list comprehensions, repetitions, pattern matching, exception handling, and more. This richness is possible because Rhombus is itself largely implemented as a set of macros—an idea whose value has been recognized since at least the 1960s, both inside and outside the Lisp community (Hart 1963; Leavenworth 1966). For layering and bootstrapping, many of the macros are written in Racket notation instead of Rhombus notation, but still using the Rhombus parsing and enforestation layer in the S-expression encoding. As a result, the base Rhombus constructs are easy to make extensible at the Rhombus level; the comprehension form, for example, is macro-extensible to support new iteration clauses as well as new collection targets. At the same time, the facilities that are used to build the base Rhombus languages are made available to Rhombus programmers through forms like `space.enforest`. This sense of Rhombus being implemented in itself ensures that the language has the same kind of flexibility and extensibility as a language in the Lisp family.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation. Matthew thanks Ilya Sergey and National University of Singapore for hosting his sabbatical; much of Rhombus's implementation took place in that productive environment. Thanks to Wing Hei Chan for technical corrections.

DATA-AVAILABILITY STATEMENT

Example programs and the current implementation of Rhombus at the time of writing are available as an artifact accompanying this paper (Flatt et al. 2023) as described in section 7.

REFERENCES

Annika Aasa, Kent Petersson, and Dan Synek. Concrete Syntax for Data Objects in Functional Languages. In *Proc. Lisp and Functional Programming*, 1988. doi:10.1145/62678.62688

Michael D. Adams. Towards the Essence of Hygiene. In *Proc. Principles of Programming Languages*, 2015. doi:10.1145/2775051.2677013

Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. Growing a Syntax. In *Proc. Foundations of Object-Oriented Languages*, 2009.

Nia Angle. Parendown. 2017. https://github.com/lathe/parendown-for-racket

Arne Babenhauserheide. SRFI-119: Wisp: Simpler Indentation-Sensitive Scheme. 2015. https://srfi.schemers.org/srfi-119/srfi-119.html

Jonathan Bachrach and Keith Playford. D-Expressions: Lisp Power, Dylan Style. 1999. https://people.csail.mit.edu/jrb/Projects/dexprs.pdf

Jonathan Bachrach and Keith Playford. The Java Syntactic Extender (JSE). In *Proc. Object-Oriented Programming, Systems, Languages and Applications*, 2001. doi:10.1145/504311.504285

Jason Baker and Wilson C. Hsieh. Maya: Multiple-Dispatch Syntax Extension in Java. In *Proc. Object-Oriented Programming, Systems, Languages and Applications*, 2002. doi:10.1145/512529.512562

Michael Ballantyne, Alexis King, and Matthias Felleisen. Macros for Domain-Specific Languages. In *Proc. Object-Oriented Programming, Systems, Languages and Applications*, 2020. doi:10.1145/3428297

Langston Barrett, David Thrane Christiansen, and Samuel Gélineau. Predictable Macros for Hindley-Milner (Extended Abstract). In *Proc. Workshop on Type-Driven Development*, 2020.

Eli Barzilay. The Scribble Reader: An Alternative to S-expressions for Textual Content. In *Proc. Scheme Workshop*, 2009.

Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19(2), 1992. doi:10.1016/0167-6423(92)90005-V

William J. Bowman, Swaha Miller, Vincent St{-}Amour, and R. Kent Dybvig. Profile-Guided Meta-Programming. In *Proc. Programming Language Design and Implementation*, 2015. doi:10.1145/2737924.2737990

Eugene Burmako. Unification of Compile-Time and Runtime Metaprogramming in Scala. Ph.D. dissertation, EPFL, 2017. doi:10.5075/epfl-thesis-7159

Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible Grammars for Language Specialization. In *Proc. Workshop on Database Programming Languages - Object Models and Languages*, 1993. doi:10.1007/978-1-4471-3564-7_2

Stephen Chang, Alex Knauth, and Ben Greenman. Type Systems as Macros. In *Proc. Principles of Programming Languages*, 2017. doi:10.1145/3093333.3009886

William Clinger and Jonathan Rees. Macros that Work. In *Proc. Principles of Programming Languages*, 1991. doi:10.1145/99583.99607

Ryan Culpepper and Matthias Felleisen. Fortifying Macros. *Journal of Functional Programming* 22(4-5), 2012. doi:10.1017/S0956796812000275

Lukas Diekmann and Laurence Tratt. Eco: A Language Composition Editor. In *Proc. Software Language Engineering*, 2014. doi:10.1007/978-3-319-11245-9_5

Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten Your JavaScript: Hygienic Macros for ES5. In *Proc. Dynamic Languages Symposium*, 2014. doi:10.1145/2775052.2661097

R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-Passing Style: Beyond Conventional Macros. In *Proc. Lisp and Functional Programming*, 1986. doi:10.1145/319838.319858

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. *Lisp and Symbolic Computation* 5(4), 1993. doi:10.1007/BF01806308

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-Based Syntactic Language Extensibility. In *Proc. Object-Oriented Programming, Systems, Languages and Applications*, 2011. doi:10.1145/2076021.2048099

Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster than the Quick and Dirty Way. In *Proc. 2010*, 2010. doi:10.1145/1869542.1869625

Jose Falcon and William R. Cook. Gel: A Generic Extensible Language. In *Proc. IFIP TC 2 Working Conference Domain-Specific Languages*, 2009. doi:10.1007/978-3-642-03034-5

Marc Feeley. Gambit: Scheme Infix Syntax Extension. 2019. http://www.iro.umontreal.ca/~gambit/doc/gambit.html#Scheme-infix-syntax-extension

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A Programmable Programming Language. *Communications of the ACM* 61(3), 2018. doi:10.1145/3127323

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a Programming Environment for Scheme. *Journal of Functional Programming* 12(2), 2002. doi:10.1017/S0956796801004208

Matthew Flatt. Compilable and Composable Macros: You Want it *When?* In *Proc. International Conference on Functional Programming*, 2002. doi:10.1145/583852.581486

Matthew Flatt. Binding as Sets of Scopes. In *Proc. Principles of Programming Languages*, 2016. doi:10.1145/2914770.2837620

Matthew Flatt, Taylor Allred, Nia Angle, Stephen De Gabrielle, Robert Bruce Findler, Jack Firth, Kiran Gopinathan, Ben Greenman, Alex Knauth, Siddhartha Kasivajhula, Jay McCarthy, Sam Phillips, Sorawee Porncharoenwase, Jens Axel Søgaard, and Sam Tobin-Hochstadt. Artifact for Rhombus: A New Spin on Macros without All the Parentheses. 2023. doi:10.1145/3580417

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. International Conference on Functional Programming*, 2009. doi:10.1145/1631687.1596569

Matthew Flatt, Ryan Culpepper, Robert Bruce Findler, and David Darais. Macros that Work Together: Compile-Time Bindings, Partial Expansion, and Definition Contexts. *Journal of Functional Programming* 22(2), pp. 181–216, 2012. doi:10.1017/S0956796812000093

Matthew Flatt, Caner Derici, R. Kent Dybvig, Andrew W. Keep, Gustavo E. Massaccesi, Sarah Spall, Sam Tobin-Hochstadt, and Jon Zeppieri. Rebuilding Racket on Chez Scheme (Experience Report). In *Proc. International Conference on Functional Programming*, 2019. doi:10.1145/3341642

Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. In *Proc. International Conference on Functional Programming*, 2001. doi:10.1145/507635.507646

Timothy P. Hart. MACRO Definitions for LISP. Massachusetts Institute of technology, AIM-057, 1963.

Inkle. Ink. 2023. https://www.inklestudios.com/ink/

JetBrains. MPS. 2003. https://www.jetbrains.com/mps/

Simon Peyton Jones and Tim Sheard. Template Meta-Programming for Haskell. In *Proc. Haskell Workshop*, 2002. doi:10.1145/581690.581691

Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proc. Object-Oriented Programming, Systems, Languages and Applications*, 2010. doi:10.1145/1932682.1869497

Alexis King. The Hackett Programming Language. 2018. https://lexi-lambda.github.io/hackett/

P. Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proc. Working Conference on Source Code Analysis and Manipulation*, 2009. doi:10.1109/SCAM.2009.28

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proc. Lisp and Functional Programming*, 1986. doi:10.1145/319838.319859

Eugene Kohlbecker and Mitch Wand. Macro-by-Example: Deriving Syntactic Transformations from Their Specifications. In *Proc. Principles of Programming Languages*, 1987. doi:10.1145/41625.41632

Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. 2006.

Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From Macros to Reusable Generative Programming. In *Proc. Generative Programming: Concepts and Experiences*, 1999. doi:10.1007/3-540-40048-6_9

B. M. Leavenworth. Syntax Macros and Extended Translation. *Communications of the ACM* 9(11), pp. 790–793, 1966. doi:10.1145/365876.365879

Odersky, Martin, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. École Polytechnique Fédérale de Lausanne, LAMP-REPORT-2006-001, 2006.

John McCarthy. History of LISP. *SIGPLAN Notices* 31(8), 1978. doi:10.1145/960118.808387

Chris McCord. *Metaprogramming Elixir: Write Less Code, Get More Done (and Have Fun!)*. O'Reilly, 2015.

Egil Möller. SRFI-49: Indentation-Sensitive Syntax. 2003. https://srfi.schemers.org/srfi-49/srfi-49.html

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a Proof Assistant for Higher-Order Logic*. Springer Science & Business Media, 2002. doi:10.1007/3-540-45949-9

OCaml. Preprocessors. 2023. https://ocaml.org/docs/metaprogramming

Cyrus Omar and Jonathan Aldrich. Reasonably Programmable Literal Notation. In *Proc. International Conference on Functional Programming*, 2018. doi:10.1145/3236801

Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely Composable Type-Specific Languages. In *Proc. European Conference on Object-Oriented Programming*, 2014. doi:10.1007/978-3-662-44202-9_5

Phoenix. Phoenix Framework. 2023. https://www.inklestudios.com/ink/

Vaughan R. Pratt. Top Down Operator Precedence. In *Proc. Principles of Programming Languages*, 1973. doi:10.1145/512927.512931

Jon Rafkind and Matthew Flatt. Honu: Syntactic Extension for Algebraic Notation through Enforestation. In *Proc. Generative Programming: Concepts and Experiences*, 2012. doi:10.1145/2371401.2371420

Rust. Macros. 2023. https://doc.rust-lang.org/reference/macros.html

Sukyoung Ryu. Parsing Fortress Syntax. In *Proc. Principles and Practice of Programming in Java*, 2009. doi:10.1145/1596655.1596667

Scala. Macros in Scala 3. 2023. https://docs.scala-lang.org/scala3/guides/macros/index.html

Guy L. Steele, Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, and Sukyoung Ryu. Fortress (Sun HPCS Language). Encyclopedia of Parallel Computing, 2011. doi:10.1007/978-0-387-09766-4_190

Don Syme, Gregory Neverov, and James Margetson. Extensible Pattern Matching Via a Lightweight Language Extension. In *Proc. International Conference on Functional Programming*, 2007. doi:10.1145/1291220.1291159

Walid Taha and Tim Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science* 248(1-2), pp. 211–242, 2000. doi:10.1145/258993.259019

Sam Tobin-Hochstadt. Extensible Pattern Matching in an Extensible Language. 2011. doi:10.48550/arXiv.1106.2578

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as Libraries. In *Proc. Programming Language Design and Implementation*, 2011. doi:10.1145/1993316.1993514

Masaru Tomita. An Efficient Context-Free Parsing Algorithm for Natural Languages. In *Proc. International Joint Conference on Artificial Intelligence*, 1985. doi:10.1145/362007.362035

Sebastian Ullrich and Leonardo de Moura. Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages. In *Proc. International Joint Conference on Automated Reasoning*, 2020. doi:10.1007/978-3-030-51054-1_10

Philip Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proc. Principles of Programming Languages*, 1987. doi:10.1145/41625.41653

David A. Wheeler. Readable Lisp S-expressions Project. 2013. https://readable.sourceforge.io/

Received 2023-04-14; accepted 2023-08-27