SYNTACTIC EXTENSION FOR LANGUAGES WITH IMPLICITLY DELIMITED AND INFIX SYNTAX

by

Jon Rafkind

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2013

Copyright © Jon Rafkind 2013

All Rights Reserved

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Jon Rafkind**

has been approved by the following supervisory committee members:

Matthew Flatt	, Chair	**2/25/2013** Date Approved
John Regehr	, Member	**2/12/2013** Date Approved
Matthew Might	, Member	**2/12/2013** Date Approved
Kent Dybvig	, Member	**3/1/2013** Date Approved
Sukyoung Ryu	, Member	**3/13/2013** Date Approved

and by **Alan Davis**, Chair of the Department of **School of Computing**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

I present the design of a parser that adds Scheme-style language extensibility to languages with implicitly delimited and infix syntax. A key element of my design is an *enforestation* parsing step, which converts a flat stream of tokens into an S-expression-like tree, in addition to the initial "read" phase of parsing and interleaved with the "macro-expand" phase.

My parser uses standard lexical scoping rules to communicate syntactic extensions to the parser. In this way extensions naturally compose locally as well as through module boundaries. I argue that this style of communication is better suited towards a useful extension system than tools not directly integrated with the compiler.

This dissertation explores the limits of this design in a new language called Honu. I use the extensiblity provided by Honu to develop useful language extensions such as LINQ and a parser generator. I also demonstrate the generality of the parsing techniques by applying them to Java and Python.

CONTENTS

ABSTRACT . . . . . **iii**

LIST OF FIGURES . . . . . **vi**

ACKNOWLEDGMENTS . . . . . **vii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Design Choices 2
1.2 Syntax Representation 2
1.3 Issues with Macros 3
1.4 Thesis Statement 3
1.5 Organization 4

2. PARSING LAYERS **5**

2.1 S-Expression Macros in Scheme 7
2.1.1 Expansion 7
2.1.2 Hygiene 10
2.2 Beyond S-expressions 11
2.2.1 Macros 16
2.2.2 Parsing Infrastructure 19

3. HONU **20**

3.1 Overview 20
3.1.1 Syntax 20
3.1.2 Honu Macros 21
3.1.3 Defining Syntax Classes 24
3.1.4 Honu Operators 25
3.2 Parsing Honu 27
3.2.1 Grammar 27
3.2.2 Reading 29
3.2.3 Enforestation 29
3.2.4 Macros and Patterns 35
3.2.5 Parsing 36
3.2.5.1 Parsing details. 37
3.2.5.2 Parsing example. 38
3.2.5.3 Parsing as expansion. 41
3.3 Extended Example 42
3.4 Honu Implementation 46
3.4.1 Syntax Patterns 50
3.4.2 Integration with Racket 52

3.4.3 Expression Phases 54

4. APPLICATIONS 55

4.1 XML 55
4.2 Linq 56
4.3 Parser Generator 60

5. EXTENSION FOR OTHER LANGUAGES 64

5.1 Extensible Java 64
5.2 Java Examples 68
5.3 Extensible Python 70
5.4 Python Examples 75

6. RELATED WORK 77

7. CONCLUSION 81

7.1 Contributions 81
7.2 Lessons 81
7.3 Future Work 82

APPENDIX 83

REFERENCES 93

LIST OF FIGURES

2.1 Scheme Expand 8
2.2 Scheme Parse 8
3.1 Enforestation 32
5.1 Java Expand 66
5.2 Python Expand 72

ACKNOWLEDGMENTS

I want to thank my advisor, Matthew Flatt, for his patience and acceptance of my ideas. I would also like to thank him and Ryan Culpepper for many technical discussions concerning hygiene.

I am grateful to my friends, Anton Burtsev and Anna Shestakova, who welcomed me into their family, and provided many alternative viewpoints to my own. I also appreciate the discussions and support from my other friends: Lu Zhao, Yang Chen, and Kevin Tew.

Lastly, I would like to thank my family for believing in me.

CHAPTER 1

INTRODUCTION

Languages constantly evolve by adding or changing features. This evolution is usually handled by a committee or benevolent dictator that goes through long processes to decide which features to add based on their importance and how well they integrate with the rest of the language. Features that are seen as too radical are often not adopted.

New features that require syntactic changes are likely candidates to be rejected. The core definition of a language's syntax frequently cannot be altered in a backwards compatible way. This leaves new features to be implemented in terms of existing syntax, which can quickly become unwieldy. Furthermore, separate features that would use the same syntax when designed in isolation require compromises in order to compose.

In the Lisp family of languages features that require syntactic changes are not seen as radical. New syntactic constructs can be seamlessly incorporated into the existing system through a syntactic extension mechanism. With this ability many new features do not require the consent of an authority, but rather can be implemented by users at their discretion. Syntactic extensibility is empowering for users and liberating for language maintainers.

Racket has taken the idea of building language features with syntactic extension farther than most other languages. Some of the more advanced systems built on top of the core Racket language include a class system, a pattern matcher, a functional reactive system, and a lazily evaluated language. These features are made possible due to Racket's rich syntactic extension mechanism. The class and pattern matching system are built as libraries that can be incorporated into any existing Racket program while the functional reactive system and lazy evaluated system are languages in their own right.

The success of extensibility in Scheme, Racket, and other languages in the Lisp family is due in part to their use of S-expressions for their concrete syntax. S-expressions simplify the job of the parser because they separate expressions in a predetermined way. The syntactic extension mechanisms in these languages operate directly on S-expressions. Languages with less uniform syntax than S-expressions typically use some other parsing technology, such as LR or PEG, to

convert the input concrete syntax into an abstract syntax tree. Scheme-style extensibility necessarily requires cooperation between the parser and the syntactic extension operators, but traditional parsing technologies have not been modified to handle this. An alternative strategy is to use the parsed abstract syntax tree as a starting point for Scheme style extensibility but the transformations applied to the abstract syntax tree have a different style relative to the transformations applied to concrete syntax.

1.1 Design Choices

Syntactic extensibility involves extending the parser, a component often thought as core to the compiler, and thus extending the compiler itself. Compiler extensions can take the form of liberal access to internal datastructures, or more commonly through some restricted API. A natural choice is to expose the language's grammar through an API. Changes to the grammar are then reflected in the execution of the parser. Grammar modifications are difficult if not impossible to compose, however, and so their utility is generally limited to extensions for an isolated use-case.

The syntactic extension API in Scheme evolved from a different landscape. Lisp is noteworthy for its claim that code is data. Code in Lisp is simply lists of symbols that can be programmatically constructed and evaluated at run-time through the `eval` function. The combination of the uniform nature of S-expressions and the ability to construct them programmatically resulted in a primitive extension system using the **fexpr** mechanism [25]. New forms in Lisp look similar to old forms, but were not implemented by the compiler. Instead, they were handled similarly to normal function calls. The dynamic nature of a **fexpr** prevented the compiler from making static assumptions about code, and so eventually the Lisp community moved towards a macro system with **defmacro** [35]. Scheme adopted a similar macro system due to its syntactic similarity to Lisp.

1.2 Syntax Representation

The protocol of the syntactic extension system in Scheme-style languages is based on concrete syntax. Macros can consume syntax based on pattern matching surface level input, and they can produce surface level output. Concrete syntax is a key factor in those systems being usable, because they provide a low barrier for macro writers to reason about the syntax of the language.

Many syntactic extension mechanisms for languages with infix syntax have been proposed that focus on transformations at the abstract syntax tree layer [22], although some include support for writing templates or pattern matchers in terms of concrete syntax [3, 32, 42]. Generally, the support for concrete syntax is limited to purely rewrite systems or syntactically intrusive methods that de-

limit macro invocations from the rest of the syntax in the language [38]. Procedural transformations can work with either abstract syntax trees or concrete syntax, but creating abstract syntax trees programmatically can be challenging. Concrete syntax is simpler for the macro writer to handle, but more complex for the language implementor to allow in transformations.

The choice between implementing a rewrite only extension system and one that allows for arbitrary computation is widely debated in the language community, but experience in Racket has shown that procedural extensions allow for a larger number of useful features to be built.

Transformation tools that operate at a layer lower than the concrete syntax impose a burden on the extension programmer because they must keep track of both the high level syntax and how it corresponds to the lower level details. When high level details are lost through the compilation process, extensions that only receive the low level representation may not compose very well. Users of libraries that contain extensions should not be tasked with workarounds for incompatible libraries, or else those libraries will fall into disuse.

1.3 Issues with Macros

Lisp macros are not hygienic; identifiers introduced by macros can clash with identifiers used at macro call sites. The issue of hygiene has been solved in Scheme by Dybvig [15] as well as others [24]. A second issue involves importing identifiers from external modules to be used in different contexts, either runtime or compile time. Flatt [18] introduced a notion of phases that distinguishes identifiers for the different contexts. Both of these issues play heavily into constructing a syntactic extension mechanism that is composable. Composability is an important property, because naturally extensions will be encapsulated in modules and imported into a context of an unknown nature.

A hygienic macro system allows macros to be composed with each other without any a-proiri knowledge. Hygiene ensures that two extensions will not break the lexical bindings of a program no matter how the extensions are intertwined. Hygiene can be implemented for rewrite-only extensions using local transformations, but extensions that can perform arbitrary computations require hygiene to be a fundamental design choice in the extension system itself.

1.4 Thesis Statement

Languages with an Algol-like syntax can be described as generally having constructs that are not explicitly delimited, matching delimiters for subexpressions, such as parenthesis or curly braces, infix operators, and constructs that start with a keyword. Non-Algol derived languages, like Ocaml

and Haskell, have a free form syntax relative to S-expressions but still share some properties such as sub-expressions and infix operators.

A language with implicitly delimited and infix syntax can provide a syntactic-extension mechanism that is hygienic, procedural, and composable.

There are two main challenges to implementing a syntactic extension system for languages with implicitly delimited and infix syntax. First, traditional parsing technology cannot be easily amended to support macros, and second, the free form nature of infix syntax creates additional engineering problems for macro writers.

Traditional parsing strategies for these types of languages generally consist of the same two steps: tokenization using regular expressions and conversion to an abstract syntax tree using a parsing algorithm specified by a grammar. Most parsing algorithms are concerned with efficiency and expressibility but not extensibility. Altering these algorithms to somehow become extensible is not a trivial matter. Languages that support syntactic extensibility, such as Racket, have designed parsers with extensibility as a design goal.

My goal is to apply Scheme-style syntactic extension to languages with implicitly delimited and infix syntax by interleaving macro expansion with the parsing process. In pursuit of this goal I have designed and implemented a language called *Honu*. I use Honu to demonstrate that modern languages have a different kind of syntactic consistency than S-expressions that can be exploited by a parsing strategy that is also useful for infix expressions. In addition to the description of Honu I show how my parsing strategy can be applied to other modern languages including Java and Python.

1.5 Organization

This rest of this dissertation is organized as follows. Chapter 2 introduces a parsing model that is capable of handling implicitly delimited and infix syntax. Chapter 3 implements a practical extensible language based on the parsing model with many useful features. Chapter 4 demonstrates interesting applications that can be built out of macros. Chapter 5 applies the parsing model to other mainstream languages and Chapter 6 describes related work.

CHAPTER 2

PARSING LAYERS

The general goal of parsing is to convert a raw stream of characters into a tree of nodes that represent the program. The most established paradigm involves matching a stream of tokens to a production in a grammar and outputting a node that represents the matched tokens. The LR and PEG parsing algorithms are examples of this method – although PEG usually matches characters instead of tokens – but the basic idea is the same. Once the entire stream of tokens has been matched the resulting tree is passed on to other phases of the compiler.

A common first step in parsing is tokenizing the raw characters to produce a uniform stream of tokens. Each token encapsulates some basic property of the syntax such as an identifier, string, or number. Scheme style languages go a step further and create a uniform tree, called an S-expression, out of the tokens based on matching delimiters such as parenthesis, braces, and brackets. The reason for this step is similar to the reason that tokenization is used in the first place: to allow the parser to deal with a higher level of abstraction than the raw character stream. In particular, Scheme-style languages utilize the uniform nature of the intermediate S-expression tree to lazily process branches. In Scheme, both the tokenization and S-expression formation steps are generally combined into a function called *read*.

The next step involves matching the tokens to a grammar specified in BNF. Many languages have unambiguous grammars such that a sequence of tokens can always be matched to a specific form regardless of the context in which those tokens appear. For example, in Java the expression `(x) 5` is always parsed as a type cast even if `x` is not a type. C requires a context sensitive parser that can determine which types are in scope so that it can resolve ambiguities when types can be confused with expressions. As an example, `(a) (b)` could be a type cast of the variable `b` to the type `a` if `a` is declared as a type, or the expression could be invoking the `a` function on the variable `b`.

Typically, there are at least two phases of parsing for languages with ambiguous forms like those in C. The first pass resolves core forms like `if`, `while`, and `for`, as well as handling expressions with operators. Ambiguous forms are left in an unresolved state for the second pass to fix. The

second pass builds up a table of identifiers and types in scope, and uses those to resolve expressions where types may occur.

The Scheme family of languages do not need traditional parsing technology such as LR to further reduce the S-expression tree. Instead, those kinds of languages are specified in such a way that traditional reductions are unnecessary. In particular, all function calls and special forms are prefix thus relieving much of the complexity of modern parsers. Prefix notation is a natural choice for S-expression-based languages even though S-expressions do not inherently require it. However, Scheme-style languages do contain forms that require a different kind of reduction. These forms are built out of macros: functions that translate S-expressions into other simpler syntactic forms, and are reduced through an *expand* function.

The *read* and *expand* steps in Scheme are similar to the first and second parsing steps in C. *Read* resolves enough program structure so that *expand* can easily process the result. Unlike C, however, much of the parsing decisions in Scheme are made in the *expand* step. In particular, *expand* reduces macro invocation forms by executing the macro and replacing the entire form with the output of the macro.

The *expand* step performs a number of tasks to ensure that macros are reduced in a hygienic manner. Macros are lexically bound to identifiers in the same way as any other kind of binding, and invocations are syntactically identical to special forms and function calls. Therefore, *expand* needs to be aware of the binding behavior of the program to precisely track the lifetime of a possible macro invocation. This tracking is achieved by adding rules to *expand* for core forms such as `lambda` and `define` that update the lexical environment appropriately.

Expand maintains an environment which maps identifiers to their binding information, either a plain lexical binding, macro, or special form, and updates the environment according to the binding behavior of the core forms. Subexpressions are processed lazily, thus allowing *expand* to register new bindings that can affect them. For example, in the following code as *expand* processes the `lambda` form in the body of the `let-syntax`, *expand* adds a mapping for the `a` argument that shadows the macro `a` bound by the `let-syntax` before processing the `lambda` body `(a 1 2)`.

```
(let-syntax ([a (lambda (stx) #'2)])
  ((lambda (a) (a 1 2)) +))
```

This code evaluates to `3`. If the argument to the inner `lambda` was a symbol other than `a`, but the body still used `a`, then the expression `(a 1 2)` would instead be a macro invocation that would replace the entire form with the result of the `a` macro, `2`.

```
(let-syntax ([a (lambda (stx) #'2)])
  ((lambda (b) (a 1 2)) +))
```

This version evaluates to 2.

By maintaining binding information *expand* is able to distinguish between macro invocations and core forms without the need for special syntax. This is the key reason that macros in Scheme can naturally extend the syntax of the language without imposing a syntactic burden on users.

This dissertation describes a new step called *enforest* that integrates with *expand* to handle languages with implicitly delimited and infix syntax. *Enforest* uses a parsing strategy that is capable of processing complex grammars, but augmented with support for macros. Crucially, *enforest* allows macros to be much less restricted than Scheme macros, as well as being integrated into syntactic forms such as infix expressions. The marriage of *expand* with *enforest* brings the power of Scheme-style macros to languages with implicitly delimited and infix syntax.

Enforest is described in more detail in Section 2.2, but first the following section describes expansion in Scheme.

2.1 S-Expression Macros in Scheme

Parsing and expansion are intertwined in Scheme. The parser's job is to convert core forms into an internal representation that the compiler or interpreter can process. Expansion reduces forms in the program to core forms by invoking macros produced by the parser. Expansion is progressively invoked on subexpressions of the program until every branch is fully expanded.

2.1.1 Expansion

Figures 2.1 and 2.2 together detail a simple substitution based expander modeled after Dybvig [15]. The *expand* function dispatches on the type of the output from the *match* [1] function, and updates the environment while *match* uses the environment to lookup binding information for identifiers. The *resolve* and *mark* procedures are related to hygienic concerns.

For the primitive binding form, `lambda`, *expand* updates the current environment with new bindings for the arguments and continues on with the subexpressions of the form. Other special forms, such as `if`, expand their subexpressions without affecting the current environment.

The parser recognizes macro definitions through `let-syntax` by creating a `syntax-binding` object. The expander then registers new macro definitions in the environment, but no trace of the

[1] Dybvig originally called this function *parse*, but we have renamed it to avoid a terminology clash in Chapter 3.

expand: Exp × Env → ExpandedExp

case *match* (exp, env) of:

variable(id) → variable(*resolve*(id))

function(id, exp1) → function(s, *expand*(*subst*(exp1, id, s), env'))
where env' = env[s = Variable] and s is fresh

application(exp1, exp2) → application(*expand*(exp1, env), *expand*(exp2, env))

if(exp1, exp2, exp3) → if(*expand*(exp1, env), *expand*(exp2, env), *expand*(exp3, env))

macro(id, exp1) → *expand*(*mark*(t(*mark*(exp1, m)), m), env)
where t=env(*resolve*(id)) and m is fresh

syntax-binding(id, exp1, exp2) → *expand*(*subst*(exp2, id, s), env[s = t])
where t = eval(*expand*(exp1, env)) and s is fresh

Figure 2.1: Scheme Expand

match: Exp × Env → ParsedExp

match(⟦id⟧, env) → variable(id)
if env(*resolve*(id)) = Variable

match(⟦(exp1 exp2)⟧, env) → application(exp1, exp2)
if exp1 != Sym

match(⟦(id exp)⟧, env) → application(id, exp)
if env(*resolve*(id)) = Variable

match(⟦(id exp)⟧, env) → macro(id, exp)
if env(*resolve*(id)) = Transformer

match(⟦if exp1 exp2 exp3⟧, env) → if(exp1, exp2, exp3)
if env(if) = Special

match(⟦(lambda id exp)⟧, env) → function(id, exp)
if env(lambda) = Special

match(⟦(let-syntax (id exp1) exp2)⟧, env) → syntax-binding(id, exp1, exp2)
if env(let-syntax) = Special

Figure 2.2: Scheme Parse

macro's definition syntax is left in the original program. Macro invocations are similar in that the parser recognizes macro applications, and the expander replaces the entire form.

The above model gives a flavor for how expansion works. The rest of the examples use the model from Racket that supports sequences of statements rather then just subexpressions. In particular, macro definitions use `define-syntax` instead of `let-syntax`.

As an example of expansion, consider the following code where `and` is a macro that expands into a collection of `if` expressions.

```
(if (and (> x 5) (< x 10)) 'in-range 'bad)
```

this code will first check the binding of `if` and discover that it resolves to a special form. The rest of the subexpressions are therefore expanded according to the rules for `if`. Next, the condition `(and (> x 5) (< x 10))` is expanded. The first element of this S-expression is `and`, which is bound to a macro, so the entire form is passed to the transformer for `and`. The expander pulls out the actual syntax transformer bound to `and` and applies it to a representation of the S-expression `(and (> x 5) (< x 10))`. The output of this application is the following code

```
(if (> x 5)
  (< x 10)
  #f)
```

The original expression `(and (> x 5) (< x 10))` is replaced by this new expression that is then further re-processed by expansion.

The syntax transformer for `and` is a function declared with `define-syntax` that accepts a single argument, which is the entire S-expression in which the `and` symbol appears.

```
(define-syntax and (lambda (and-syntax)
                     ...))
```

The goal of the `and` transformer is to convert the syntax arguments `(> x 5)` and `(< x 10)` into subexpressions of an `if` expression. In my simple example I can assume that `and` will only ever be used with two arguments, so the implementation of the transformer is simply

```
(lambda (and-syntax)
  (with-syntax ([part1 (get-syntax and-syntax 1)]
                [part2 (get-syntax and-syntax 2)])
    #'(if part1 part2 #f)))
```

The syntax `#'(if part1 part #f)` produces a representation of the S-expression `(if part1 part2 #f)` that is suitable for the expander to process. The result of expanding the original program is

```
(if (if (> x 5)
        (< x 10)
        #f)
    'in-range
    'bad)
```

This expression is entirely made up of core forms, so expansion completes.

2.1.2 Hygiene

An important goal of *expand* is to maintain the lexical properties of the program while transforming parts of the program tree. This property is commonly known as *hygiene* [24]. A hygienic transformation ensures that new bindings do not conflict in any way with bindings that were already a part of the tree. Transformations are of course allowed to create binding forms whose subexpressions are arguments to the macro, which has the potential to shadow already existing bindings in those subexpressions. In the simplified model above, hygiene is handled through the *mark* and *resolve* functions.

A common and simple example is `or`, which binds its first argument to a variable to prevent the argument from being evaluated multiple times. First, I demonstrate an unhygienic expansion of `or`.

```
(define-syntax (or stx)
  (with-syntax ([part1 (get-syntax stx 1)]
                [part2 (get-syntax stx 2)])
   #'(let ([t part1])
       (if t t part2))))

; Use of or
(or (> x 2) (< x -2))

; Expansion of or
(let ([t (> x 2)])
  (if t t (< x -2)))
```

The `let` expression is the expanded version of the `or` use. The `t` binding introduced by the `or` macro can shadow any `t` identifiers in the subexpressions in the new `if` expression.

```
(let ([t 1])
  (or (> t 2) (< t -2)))
```

Expands to

```
(let ([t 1])
  (let ([t (> t 2)])
    (if t t (< t -2))))
```

The second `let` expression will bind `t` to the value `#f`, so ultimately the `if` expression will try to evaluate `(< #f -2)` and cause a run-time exception.

A hygienic macro transformation solves this issue by renaming identifiers according to when they were originally bound. Assuming `let` is a core form the correct expansion of the above example is

```
(let ([t 1])
  (let ([t2 (> t 2)])
    (if t2 t2 (< t -2))))
```

In this expansion the binding of `t` introduced by the `let` expression in the `or` macro is renamed to `t2`, a unique identifier, to prevent it from binding `t` in the subexpressions of the `if`. Identifiers are renamed according to when they were created. Rather than using plain symbols identifiers, and all other S-expressions, are stored in a syntax object that contains binding information and other information useful for the renaming procedure. The interested reader is referred to Dybvig [8] for further details.

2.2 Beyond S-expressions

The *expand* function in Scheme properly maintains hygiene and tracks of the scopes of variables, but the *match* function is tied to prefix notation and expects a macro to consume the entire S-expression in which it appears. My key idea is that there is nothing inherent about the *match* function that limits it to S-expressions. Instead, I can integrate a parsing algorithm that can deal with infix notation and syntax normally found in other languages.

I replace the *match* function with another function called *enforest* that can handle implicitly delimited syntax. I do not place any restrictions on the parsing algorithm that *enforest* uses, although an operator precedence parser [27] is the easiest to implement.

A close relationship is maintained between *expand* and *enforest* similar to the one between *expand* and *match*. *Expand* recognizes primitive binding forms and maintains an environment that maps identifiers to binding information while *enforest* reduces forms in the language to primitive forms. Whereas the *expand* function in Scheme was responsible for invoking macros, instead *enforest* does this job. In Scheme the extent of a macro call is known beforehand because of the uniform nature of S-expressions; namely that when a macro binding appears at the head of an S-expression *match* recognizes a macro invocation, and uses the entire form as the argument to the macro. More free form syntaxes are not as uniform, and so the *match* function cannot know the extent of a macro.

Implicitly delimited syntaxes generally do have some level of uniformity that can be exploited in the same way as for Scheme. Many languages use parentheses, braces, and brackets among other things to delimit subexpressions. Scheme utilizes the fact that an S-expression encapsulates subexpressions to delay their expansion, which is useful for macros that create bindings within the scope of subexpressions. Therefore, I continue to use the *read* function with only minor changes to deal with the specifics of the syntax at hand.

The interplay between *enforest* and *expand* is demonstrated in the following example. First, I will define a simple language with conditionals, function calls, variable declarations, and function declarations and show how the functions *enforest* and *expand* work together to perform tasks that together parse the program. Then, I will add in macros to the language and show how the two functions need to be updated.

The grammar for the concrete syntax of the language is as follows

```
⟨program⟩ ::= ⟨expr⟩+
⟨expr⟩    ::= if ( ⟨expr⟩ ) { ⟨expr⟩* } else { ⟨expr⟩* }
            | ⟨id⟩ ( ⟨expr⟩* )
            | var ⟨id⟩ = ⟨expr⟩
            | function ⟨id⟩ ( ⟨id⟩* ) { ⟨expr⟩* }
            | ⟨id⟩ | ⟨number⟩ | ⟨string⟩
```

This language does not contain infix operators, so it is little more than the S-expression syntax of Scheme, but forms are not completely enclosed with delimiters so parsing this language still has some work to do.[2] Subexpressions are enclosed in matching delimiters, though, and so this language can pass through a *read* phase. The output of *read* is a sequence of ⟨*term*⟩ nodes.

```
⟨term⟩ ::= ⟨atom⟩
         | ( ⟨term⟩ ... )
         | [ ⟨term⟩ ... ]
         | { ⟨term⟩ ... }
⟨atom⟩ ::= symbol | number | string
```

[2]Infix notation, and how to parse it, is supported in the Honu language in Chapter 3.

The following program will be used as a running example to demonstrate the parsing methodology. This example computes the Fibonacci sequence.

```
function fibonacci(n){
  if (or(equals(n, 1), equals(n, 2))){
   1
  } else {
    var left = fibonacci(sub(n, 1))
    var right = fibonacci(sub(n, 2))
    add(left, right)
  }
}
```

This form is the concrete syntax tree. *Enforest* translates the concrete syntax tree into an abstract syntax tree made up of core forms that *expand* can process. The core forms that *expand* processes are

⟨*expand-ast*⟩ ::= if (⟨*expand-ast*⟩ , ⟨*expand-ast*⟩ , ⟨*expand-ast*⟩)
| call (⟨*id*⟩ , ⟨*expand-ast*⟩*)
| var (⟨*id*⟩ , ⟨*expand-ast*⟩)
| atom (⟨*atom*⟩)
| fun (⟨*id*⟩ , (⟨*id*⟩*) , ⟨*expand-ast*⟩)
| seq (⟨*expand-ast*⟩ , ⟨*expand-ast*⟩)
| unparsed (⟨*term*⟩ ...)
⟨*atom*⟩ ::= *symbol*
| *number*

The initial set of ⟨*term*⟩'s produced by *read* is wrapped in an unparsed node to inject the tree into the ⟨*expand-ast*⟩ structure. An ⟨*expand-ast*⟩ node can masquerade as a ⟨*term*⟩, which allows *enforest* to avoid redundant parsing. Only *enforest* can produce ⟨*expand-ast*⟩ nodes, however, other than the initial wrapping of the unparsed node.

⟨*term*⟩ ::= ...
| ⟨*expand-ast*⟩

The *expand* function is modeled as follows.

```
                    expand: expand-ast × Env  →  expand-ast × Env
case exp of:
                atom(a) → atom(resolve(a)), env if id = Symbol
                          otherwise atom(a)
             var(id, exp) → let v, _ = expand(exp, env) in
                            var(id, v), env[id = Variable]
        call(exp1, exp2) → let v1, _ = expand(exp1, env) in
                           let v2, _ = expand(exp2, env) in
                            call(v1, v2), env
         seq(exp1, exp2) → let v1, env1 = expand(exp1, env) in
                           let v2, env2 = expand(exp2, env1) in
                            seq(v1, v2), env2
   if(exp1, exp2, exp3) → let v1, _ = expand(exp1, env) in
                           let v2, _ = expand(exp2, env) in
                           let v3, _ = expand(exp3, env) in
                            if(v1, v2, v3), env
        fun(id, ids, exp) → let body, _ = expand(exp, env[ids = Variable]) in
                            fun(id, ids, body), env[id = Variable]
  unparsed(term ...) → let m = newmark() in
                       let parsed, rest = enforest(mark(term ..., m)) in
                       let exp1 = mark(parsed, m) in
                       let exp2 = mark(rest, m) in
                        expand(seq(exp1, exp2), env)
```

The rules for processing each production in the ⟨*expand-ast*⟩ grammar are as follows.

- A node starting with unparsed is passed to the *enforest* function. The output will be an ⟨*expand-ast*⟩ node and an unreduced ⟨*term*⟩ tree. The first value will be recursively expanded, and the second will be wrapped inside another unparsed node for further processing.
- A node starting with either var or function will create a new binding in the current lexical scope and parsing continues. The body of a function will be expanded in a new lexical scope.
- A node starting with call, seq, or if will have their arguments expanded recursively using the existing environment.
- An atom node containing an identifier is resolved, otherwise it remains an untouched piece of syntax.

The *enforest* function accepts a sequence of unparsed terms, represented by ⟨*term*⟩ below, parses a single expression into a parsed term and returns it and an unparsed tree.

```
enforest: term* × Env  →  expand-ast × term*
enforest(⟦atomic rest ...⟧, env) → atom(atomic), rest
enforest(⟦var id = term ...⟧, env) → let exp, rest = enforest(term ..., env) in
                                       var(id, exp), rest
                                     if env[var] = Special
enforest(id, ⟦(arg ...) rest ...⟧), env) → let e, _ = enforest(arg ..., env) in
                                            call(id, e), rest
enforest(⟦fun id(id ...){body ...} rest ...⟧), env) → fun(id, (id ...), unparsed(body ...)), rest
                                                      if env[function] = Special
enforest(⟦if (e1 ...){e2 ...} else {e3 ...} rest ...⟧, env) → let v1, _ = enforest(e1 ..., env) in
                                                              let v2, _ = enforest(e2 ..., env) in
                                                              let v3, _ = enforest(e3 ..., env) in
                                                                if(v1, v2, v3), rest
                                                              if env[if] = Special
```

The rule for `call` matches the result of calling *enforest* and then a term inside parenthesis. The rule for `function` matches the body without parsing it so that the function arguments can be bound by *expand* first.

I will demonstrate how *enforest* and *expand* work together to parse the Fibonacci example from above. Parsing functions appear in *black*, ⟨*expand-ast*⟩ nodes in `blue`, and unparsed ⟨*term*⟩ nodes in `red`.

```
expand(unparsed(function fibonacci(n){
                  if (or (equals (n , 1))
                         (equals (n , 2))){
                    1
                  } else {
                    var left = fibonacci(sub n , 1)
                    var right = fibonacci(sub n , 2)
                    add(left , right)
                  }
                }))
```

The rule for handling `unparsed` results in a call to *enforest*.

```
enforest(function fibonacci ...)
```

Enforest creates a function AST node and leaves the body untouched. There is no remaining syntax after the fibonacci function definition.

```
expand(fun(fibonacci, (n), unparsed(if (...) ...)))
```

The rule in *expand* for the `fun` AST node creates a new environment that maps `n` to a lexical variable, and continues to expand the body with this new environment.

```
expand(unparsed(if (...) ...))
```

Parsing the `if` expression is straightforward. The else expression is interesting only in that calling *enforest* on the body produces a parsed expression and an unparsed sequence of terms.

```
enforest(var left = fibonacci(sub n , 1)
         var right = fibonacci(sub n , 2)
         add(left , right))
```

The parsed expression is passed directly to *expand* while the unparsed terms are wrapped in a `unparsed` node, and passed to *expand*.

```
seq(
 expand(var(left, call(fibonacci, call(sub, n, 1))))
 expand(unparsed(var right = fibonacci(sub n , 2)
                 add(left , right))))
```

2.2.1 Macros

Adding macros to the language requires changing both the ⟨*expr*⟩ and ⟨*expand-ast*⟩ grammars. A new syntactic form for defining macros is added to the ⟨*expression*⟩ grammar that starts with the macro keyword and a new macro node is added to ⟨*expand-ast*⟩. The first ⟨*id*⟩ in the macro form declares the name of the macro, while the second is the name of the syntax argument used in the body of the macro. Macro invocations are added to the ⟨*expression*⟩ grammar as a form starting with a macro identifier, ⟨*id*⟩, followed by an arbitrary sequence of nodes.

```
⟨expr⟩        ::= macro ⟨id⟩ ⟨id⟩ { ⟨expr⟩* }
                | ⟨id⟩ ⟨term⟩*
                | ...
⟨expand-ast⟩  ::= macro ( ⟨id⟩ , ⟨id⟩ , ( ⟨expand-ast⟩ ) )
                | ...
```

The type of a macro is a function from syntax to a tuple of two pieces of syntax. The first piece in the tuple is the resulting form that the macro produced while the second piece is any remaining syntax from the input that was untouched.

$$\text{transformer: term}^* \rightarrow \text{term}^* \times \text{term}^*$$

The models for *expand* and *enforest* are updated according to the new productions in the grammars.

$$\textit{expand}\text{: expand-ast} \times \text{Env} \rightarrow \text{expand-ast} \times \text{Env}$$

```
case exp of:
  ...
  expand(macro(name, arg, body), env) → let transformer, _ = expand(body, compile-env[arg = Variable]) in
                                        null, env[id = Macro(transformer)]
```

enforest: term* × Env → expand-ast × term*
enforest(⟦macro name arg{ node ... } rest ...⟧, env) → macro(name, arg, unparsed(node ...)), rest
enforest(e rest ...) → e, rest
where e ∈ <*expand-ast*>
enforest(⟦id term ...⟧) → let t = *get-transformer*(id, env) in
let out, rest = t(term ...) in
unparsed(out), rest
if env[id] = Macro
...

The *expand* function applies itself recursively to the body of the macro using a compile-time environment instead of the normal environment. *Enforest* recognizes identifiers bound to macros and invokes their associated transformer on the entire input stream. The result of parsing a macro is the output of the macro transformer wrapped in a unparsed node. If the input is already an instance of ⟨*expand-ast*⟩ then it is returned as-is.

A macro receives all of the syntax ahead of it within its enclosing ⟨*term*⟩. This may include forms that will not be parsed by the macro. The unused portions are returned to the *enforest* function. A macro parses its input by using a combination of predicates and invoking the *enforest* function. Input that should be treated as a closed expression is parsed through recursive calls to *enforest*. Such expressions may contain macro calls themselves. The result of a macro should be a self-contained form that ultimately can be parsed into an ⟨*expand-ast*⟩ without any syntax remaining. For example, the following macro prints the time taken to evaluate its argument at runtime.

```
macro time stx {
  var arg, rest = enforest(stx)
  var result =
    syntax({
      var before = time()
      var out = arg
      var after = time()
      print("Took ~a", sub(after, before))
      out
    })
  result, rest
}

var answer = time fibonacci(82)
```

```
print("Answer is ~a\n", answer)
```

The `syntax` function produces a representation of the syntax given as its argument, in a similar manner to the `#'(...)` form in Scheme. The output from the `time` macro is the new piece of syntax bound to the `result` variable and the unparsed syntax bound to `rest` that is returned from the *enforest* function used inside the macro. Parsing the last two lines is demonstrated as follows.

```
expand(
 unparsed(var answer = time fibonacci(82)
          print("Answer is ~a\n", answer)))
```

Enforest parses the `var` form by recognizing the identifier `answer` then `=`, and finally sees that `time` is bound to a macro. The transformer associated with `time` is passed all of the syntax after the `time` identifier which includes the `print` expression.

The output of the `time` macro is a new piece of syntax and the unparsed terms from the input. The new syntax is a block containing calls to `time()` before and after the input expression. The remaining unparsed syntax is whatever is returned from the internal call to *enforest*. The result of parsing the `var` form is

```
seq(
 var answer(unparsed(var before = time()
                     var out = (call fibonacci 82)
                     var after = time()
                     print("Took ~a", sub(after, before))
                     out)),
 unparsed(print("Answer is ~a\n", answer)))
```

Now that I have introduced macros the issue of hygiene becomes important. All of the syntax passed to *enforest* by *expand* has a mark applied to it from the *mark* procedure and should interact safely with identifiers introduced by macros, similar to how Scheme expansion works. Marks differentiate symbols that have the same name but different lexical contexts. However, macros in this model can themselves call *enforest*, which could invoke further macros. To ensure the hygiene of the system, the syntax passed to *enforest* must similarly be marked. Instead of allowing macros to directly invoke *enforest* they instead invoke a function that marks the input, passes it to *enforest*, and then remarks the output. This function is called *safe_enforest* and is defined in the implementation.

```
(define (safe-enforest input)
  (define m (make-mark))
  (define-values (output rest)
                 (enforest (mark input m)))
  (values (mark output m) (mark rest m)))
```

The `time` macro from above would replace the call to `enforest` with this new `safe-enforest` function.

```
macro time stx {
  var arg, rest = safe-enforest(stx)
  ...
}
```

2.2.2 Parsing Infrastructure

The ability of *enforest* and *expand* to use the environment to classify identifiers in the program allows a wide range of features to be implemented. In the case of C, types can be differentiated from variables. Another good use case of this ability is to identify operators.

Traditional parsing technology cannot easily handle arbitrary operators. Usually operators are hard coded into the language's grammar. The key to allowing the user extend the set of operators is to allow the parser to recognize operators based on the current environment. In Chapter 3, I show how to use an operator precedence parser to allow arbitrary operators to be defined.

CHAPTER 3

HONU

Honu [28] is a prototype language that has a syntax reminiscent of Algol but incorporates Scheme-like syntactic extension through the use of enforestation. The design of a useable macro system for a language with implicitly delimited syntax and its implications are explored in this chapter.

3.1 Overview

Honu's syntax is similar to other languages that use curly braces and infix syntax, such as C and Javascript. Honu's macro support is similar to Scheme's, but the macro system is tailored to syntactic extensions that continue the basic Honu style, including support for declaring new infix operators.

All examples covered in the rest of the paper occur in an environment where identifiers such as `macro` are bound as usual.

3.1.1 Syntax

As an introduction to Honu syntax, the following Honu code declares a function to compute the roots of a quadratic equation.

```
1  function quadratic(a, b, c) {
2    var discriminant = sqr(b) - 4 * a * c
3    if (discriminant < 0) {
4      []
5    } else if (discriminant == 0) {
6      [-b / (2 * a)]
7    } else {
8      [(-b + discriminant) / (2 * a),
9       (-b - discriminant) / (2 * a)]
10   }
```

```
11  }
```

The function `quadratic` accepts three arguments and returns a list containing the roots of the formula, if any. Line 1 starts a function definition using `function`, which is similar to `function` in Javascript. Line 2 declares a lexically scoped variable named `discriminant`. Lines 4, 6, and 8 create lists containing zero, one, and two elements, respectively. Honu has no `return` form; instead, a function's result is the value of its last evaluated expression. In this case, lines 4, 6, and 8 are expressions that can produce the function's result.

As in Javascript, when `function` is used without a name, it creates a anonymous function. The declaration of `quadratic` in the example above is equivalent to

```
var quadratic = function(a, b, c) { .... }
```

Semicolons in Honu optionally delimit expressions. Typically, no semicolon is needed between expressions, because two expressions in a sequence usually do not parse as a single expression. Some expression sequences are ambiguous, however; for example, `f(x)[y]` could access either of the `y` element of the result of `f` applied to `x`, or it could be `f` applied to `x` followed by the creation of a list that contains `y`. In such ambiguous cases, Honu parses the sequence as a single expression, so a semicolon must be added if separate expressions are intended.

Curly braces create a block expression. Within a block, declarations can be mixed with expressions, as in the declaration of `discriminant` on line 2 of the example above. Declarations are treated the same as expressions by the parser up until the last step of parsing, in which case a declaration triggers a syntax error if it is not within a block or at the top level.

3.1.2 Honu Macros

The Honu `macro` form binds a ⟨*id*⟩ to a pattern-based macro:

`macro` ⟨*id*⟩ `(` ⟨*literals*⟩ `)` `{` ⟨*pattern*⟩ `}` `{` ⟨*body*⟩ `}`

The ⟨*pattern*⟩ part of a macro declaration consists of a mixture of concrete Honu syntax and variables that can bind to matching portions of a use of the macro. An identifier included in the ⟨*literals*⟩ set is treated as a syntactic literal in ⟨*pattern*⟩ instead of as a pattern variable, which means that a use of the macro must include the literal as it appears in the ⟨*pattern*⟩. The ⟨*body*⟩ of a macro declaration is an arbitrary Honu expression that computes a new syntactic form to replace the macro use.[1]

[1] The ⟨*body*⟩ of a macro is a compile-time expression, which is separated from the run-time phase in Honu in the same way as for Racket [18].

One simple use of macros is to remove boilerplate. For example, suppose I have a `derivative` function that computes the approximate derivative of a given function:

```
function derivative(f) {
  function (pt) {
    (f(pt + 0.001) - f(pt)) / 0.001
  }
}
```

I can use `derivative` directly on an anonymous function:

```
var df = derivative(function (x) { x * x - 5 * x + 8 })
df(10) // 15.099
```

If this pattern is common, however, I might provide a `D` syntactic form so that the example can be written as

```
var df = D x, x * x - 5 * x + 8
df(10) // 15.099
```

As a macro, `D` can manipulate the given identifier and expression at the syntactic level, putting them together with `function`:

```
macro D(){ z:id, math:expression } {
  syntax(derivative(function (z) { math }))
}
```

The pattern for the `D` macro is `z:id, math:expression`, which matches an identifier, then a comma, and finally an arbitrary expression. In the pattern, `z` and `math` are pattern variables, while `id` and `expression` are *syntax classes* [13]. Syntax classes play a role analogous to grammar productions, where macro declarations effectively extend `expression`. The syntax classes `id` and `expression` are predefined in Honu.

Although the ⟨*body*⟩ of a macro declaration can be arbitrary Honu code, it is often simply a `syntax` form. A `syntax` form wraps a *template*, which is a mixture of concrete syntax and uses of pattern variables. The result of a `syntax` form is a *syntax object*, which is a first-class value that represents an expression. Pattern variables in `syntax` are replaced with matches from the macro use to generate the result syntax object.

The expansion of `D` is a call to `derivative` with an anonymous function. The macro could be written equivalently as

```
macro D(){ z:id, math:expression } {
  syntax({
            function f(z) { math }
            derivative(f)
          })
}
```

which makes `D` expand to a block expression that binds a local `f` and passes `f` to `derivative`. Like Scheme macros, Honu macros are hygienic, so the local binding `f` does not shadow any `f` that might be used by the expression matched to `math`.

The `D` example highlights another key feature of the Honu macro system. Since the pattern for `math` uses the `expression` syntax class, `math` can be matched to the entire expression `x * x - 5 * x + 8` without requiring parentheses around the expression or around the use of `D`. Furthermore, when an expression is substituted into a template, its integrity is maintained in further parsing. For example, if the expression `1+1` was bound to the pattern variable `e` in `e * 2`, the resulting syntax object corresponds to `(1 + 1) * 2`, not `1 + (1 * 2)`.

Using `expression` not only makes `D` work right with infix operators, but it also makes it work with other macros. For example, I could define a `parabola` macro to generate parabolic formulas, and then I can use `parabola` with `D`:

```
macro parabola(){ x:id a:expression,
                        b:expression,
                        c:expression} {
  syntax(a * x * x + b * x + c)
}

var d = D x, parabola x 1, -5, 8
d(10) // 15.099
```

The ⟨*pattern*⟩ part of a macro declaration can use an ellipsis to match repetitions of a preceding sequence. The preceding sequence can be either a pattern variable or literal, or it can be multiple terms grouped by `$`. For example, the following `trace` macro prints each term followed by evaluating the expression.

```
macro trace(){ expr ... } {
```

```
  syntax($ printf("~a -> ~a\n", 'expr, expr) $ ...)
}
```

The ellipsis in the pattern causes the preceding `expr` to match a sequence of terms. In a template, `expr` must be followed by an ellipsis, either directly or as part of a group bracketed by `$` and followed by an ellipsis. In the case of `trace`, `expr` is inside a `$` group, which means that one `printf` call is generated for each `expr`.

All of my example macros so far immediately return a `syntax` template, but the full Honu language is available for a macro implementation. For example, an extended `trace` macro might statically compute an index for each of the expressions in its body and then use the index in the printed results:

```
macro ntrace(){ expr ... } {
  var exprs = syntax_to_list(syntax(expr ...))
  var indexes = generate_indices(exprs)
  with_syntax (idx ...) = indexes {
    syntax($ printf("~a -> ~a\n", idx, expr) $ ...)
  }
}
```

In this example, `syntax(expr ...)` generates a syntax object that holds a list of expressions, one for each `expr` match, and the Honu `syntax_to_list` function takes a syntax object that holds a sequence of terms and generates a plain list of terms. A `generate_indices` helper function (not shown) takes a list and produces a list with the same number of elements but containing integers counting from `1`. The `with_syntax` ⟨*pattern*⟩ `=` ⟨*expression*⟩ form binds pattern variables in ⟨*pattern*⟩ by matching against the syntax objects produced by ⟨*expression*⟩, which in this case binds `idx` as a pattern variable for a sequence of numbers. In the body of the `with_syntax` form, the `syntax` template uses both `expr` and `idx` to generate the expansion result.

3.1.3 Defining Syntax Classes

The syntax classes `id` and `expression` are predefined, but programmers can introduce new syntax classes. For example, to match uses of a `cond` form like

```
cond
  x < 3: "less than 3"
  x == 3: "3"
```

```
x > 3: "greater than 3"
```

I could start by describing the shape of an individual `cond` clause.

The Honu `pattern` form binds a new syntax class:

`pattern` ⟨*id*⟩ `(` ⟨*literals*⟩ `) {` ⟨*pattern*⟩ `}`

A `pattern` form is similar to a macro without an expansion ⟨*body*⟩. Pattern variables in ⟨*pattern*⟩ turn into subpattern names that extend a pattern variable whose class is ⟨*name*⟩.

For example, given the declaration of a `cond_clause` syntax class,

```
pattern cond_clause ()
  { check:expression : body:expression }
```

I can use `cond_clause` form pattern variables in the definition of a `cond` macro:

```
macro cond(){ first:cond_clause
              rest:cond_clause ... } {
  syntax(if (first_check) {
           first_body
         } $ else if (rest_check) {
           rest_body
         } $ ...)
}
```

Since `first` has the syntax class `cond_clause`, then it matches an expression–colon–expression sequence. In the template of `cond`, `first_check` accesses the first of those expressions, since `check` is the name given to the first expression match in the definition of `cond_clause`. Similarly, `first_body` accesses the second expression within the `first` match. The same is true for `rest`, but since `rest` is followed in the macro pattern with an ellipsis, it corresponds to a sequence of matches, so that `rest_check` and `rest_body` must be under an ellipsis in the macro template.

Pattern variables that are declared without an explicit syntax class are given a default class that matches a raw term: an atomic syntactic element, or a set of elements that are explicitly grouped with parentheses, square brackets, or curly braces.

3.1.4 Honu Operators

In addition to defining new macros that are triggered through a prefix keyword, Honu allows programmers to declare new binary and unary operators. Binary operators are always infix, while unary operators are prefix, and an operator can have both binary and unary behaviors.

The `operator` form declares a new operator:

`operator` ⟨*id*⟩ ⟨*prec*⟩ ⟨*assoc*⟩ ⟨*binary transform*⟩ ⟨*unary transform*⟩

The operator precedence ⟨*prec*⟩ is specified as a non-negative rational number, while the operator's associativity ⟨*assoc*⟩ is either `left` or `right`. The operator's ⟨*binary transform*⟩ is a function that is called during parsing when the operator is used in a binary position; the function receives two syntax objects for the operator's arguments, and it produces a syntax object for the operator application. Similarly, an operator's ⟨*unary transform*⟩ takes a single syntax object to produce an expression for the operator's unary application.

The `binary_operator` and `unary_operator` forms are shorthands for defining operators with only a ⟨*binary transform*⟩ or ⟨*unary transform*⟩, respectively:

`binary_operator` ⟨*id*⟩ ⟨*prec*⟩ ⟨*assoc*⟩ ⟨*binary transform*⟩

`unary_operator` ⟨*id*⟩ ⟨*prec*⟩ ⟨*unary transform*⟩

A unary operator is almost the same as a macro that has a single `expression` subform. The only difference between a macro and a unary operator is that the operator has a precedence level, which can affect the way that expressions using the operator are parsed. A macro effectively has a precedence level of 0. Thus, if `m` is defined as a macro, then `m 1 + 2` parses like `m (1 + 2)`, while if `m` is a unary operator with a higher precedence than +, `m 1 + 2` parses like `(m 1) + 2`. A unary operator makes a recursive call to parse with its precedence level but macros have no such requirement so unary operators cannot simply be transformed into macros.

As an example binary operator, I can define a `raise` operator that raises the value of the expression on the left-hand side to the value of the expression on the right-hand side:

```
1  binary_operator raise 10 left
2      function (left, right) {
3        syntax(pow(left, right))
4      }
```

The precedence level of `raise` is 10, and it associates to the left.

Naturally, newly declared infix operators can appear in subexpressions for a macro use:

```
var d = D x, x raise 4 + x raise 2 - 3
```

I can define another infix operator for logarithms and compose it with the `raise` operator. Assume that `make_log` generates an expression that takes the logarithm of the left-hand side using the base of the right-hand side:

```
binary_operator lg 5 left make_log

x raise 4 lg 3 + x raise 2 lg 5 - 3
```

Since `raise` has higher precedence than `lg`, and since both `raise` and `lg` have a higher precedence than the built-in `+` operator, the parser groups the example expression as

```
((x raise 4) lg 3) + ((x raise 2) lg 5) - 3
```

As the `raise` and `lg` examples illustrate, any identifier can be used as an operator. Honu does not distinguish between operator names and other identifiers, which means that `raise` can be an operator name and `+` can be a variable name. Furthermore, Honu has no reserved words and any binding—variable, operator, or syntactic form—can be shadowed. This flexible treatment of identifiers is enabled by the interleaving of parsing with binding resolution, as I discuss in the next section.

3.2 Parsing Honu

Honu parsing relies on three layers: a *reader* layer, an *enforestation* layer, and a *parsing* layer proper that drives enforestation, binding resolution, and macro expansion. The first and last layers are directly analogous to parsing layers in Lisp and Scheme, and so I describe Honu parsing in part by analogy to Scheme, but the middle layer is unique to Honu.

3.2.1 Grammar

A BNF grammar usually works well to describe the syntax of a language with a fixed syntax, such as Java. BNF is less helpful for a language like Scheme, whose syntax might be written as

⟨*expression*⟩ ::= ⟨*literal*⟩ | ⟨*identifier*⟩
| (⟨*expression*⟩ ⟨*expression*⟩*)
| (lambda (⟨*identifier*⟩*) ⟨*expression*⟩+)
| (if ⟨*expression*⟩ ⟨*expression*⟩ ⟨*expression*⟩)
| ...

but such a grammar would be only a rough approximation. Because Scheme's set of syntactic forms is extensible via macros, the true grammar at the level of expressions is closer to

⟨*expression*⟩ ::= ⟨*literal*⟩ | ⟨*identifier*⟩
| (⟨*expression*⟩ ⟨*expression*⟩*)
| (⟨*form identifier*⟩ ⟨*term*⟩*)

The (⟨*expression*⟩ ⟨*expression*⟩*) production captures the default case when the first term after a parenthesis is not an identifier that is bound to a syntactic term, in which case the expression is treated as a function call. Otherwise, the final (⟨*form identifier*⟩ ⟨*term*⟩*) production captures

uses of `lambda` and `if` as well as macro-defined extensions. Putting a `lambda` or `if` production would be misleading, because the name `lambda` or `if` can be shadowed or redefined by an enclosing expression; an enclosing term might even rewrite a nested `lambda` or `if` away. In exchange for the loss of BNF and a different notion of parsing, Scheme programmers gain an especially expressive, extensible, and composable notation.

The syntax of Honu is defined in a Scheme-like way, but with more default structure than Scheme's minimal scaffolding. The grammar of Honu is roughly as follows:

⟨*program*⟩ `::=` ⟨*sequence*⟩
⟨*expression*⟩ `::=` ⟨*literal*⟩ `|` ⟨*identifier*⟩
`|` ⟨*unary operator*⟩ ⟨*expression*⟩
`|` ⟨*expression*⟩ ⟨*binary operator*⟩ ⟨*expression*⟩
`|` ⟨*expression*⟩ `(` ⟨*comma-seq*⟩ `)`
`|` `(` ⟨*expression*⟩ `)`
`|` ⟨*expression*⟩ `[` ⟨*expression*⟩ `]`
`|` `[` ⟨*comma-seq*⟩ `]`
`|` `[` ⟨*expression*⟩ `:` ⟨*expression*⟩ `=` ⟨*expression*⟩ `]`
`|` `{` ⟨*sequence*⟩ `}`
`|` ⟨*form identifier*⟩ ⟨*term*⟩*
⟨*comma-seq*⟩ `::=` ⟨*expression*⟩ `[,]` ⟨*comma-seq*⟩
`|` ⟨*expression*⟩
⟨*sequence*⟩ `::=` ⟨*expression*⟩ `[;]` ⟨*sequence*⟩
`|` ⟨*expression*⟩

This grammar reflects a mid-point between Scheme-style syntax and traditional infix syntax:

- Prefix unary and infix binary operations are supported through the extensible ⟨*unary operator*⟩ and ⟨*binary operator*⟩ productions.
- The ⟨*expression*⟩ `(` ⟨*comma-seq*⟩ `)` production plays the same role as Scheme's default function-call production, but in traditional algebraic form.
- The `(` ⟨*expression*⟩ `)` production performs the traditional role of parenthesizing an expression to prevent surrounding operators with higher precedences from grouping with the constituent parts of the expression.
- The ⟨*expression*⟩ `[` ⟨*expression*⟩ `]` production provides a default interpretation of property or array access.
- The `[` ⟨*comma-seq*⟩ `]` production provides a default interpretation of square brackets without a preceding expression as a list creation mechanism.
- The `[` ⟨*expression*⟩ `:` ⟨*expression*⟩ `=` ⟨*expression*⟩ `]` production provides a default interpretation of square brackets with `:` and `=` as a list comprehension.
- The `{` ⟨*sequence*⟩ `}` production starts a new sequence of expressions that evaluates to the last expression in the block.

- Finally, the ⟨*form identifier*⟩ ⟨*term*⟩* production allows extensibility of the expression grammar.

In the same way that Scheme's default function-call interpretation of parentheses does not prevent parentheses from having other meanings in a syntactic form, Honu's default interpretation of parentheses, square brackets, curly braces, and semicolons does not prevent their use in different ways within a new syntactic form.

3.2.2 Reading

The Scheme grammar relies on an initial parsing pass by a *reader* to form ⟨*term*⟩s. The Scheme reader plays a role similar to token analysis for a language with a static grammar, in that it distinguishes numbers, identifiers, string, commas, parentheses, comments, etc. Instead of a linear sequence of tokens, however, the reader produces a tree of values by matching parentheses. Values between a pair of matching parentheses are grouped as a single term within the enclosing term. In Honu, square brackets and curly braces are distinguished from parentheses, but they similarly matched.

Ignoring the fine details of parsing numbers, strings, identifiers, and the like, the grammar recognized by the Honu reader is

⟨*term*⟩ ::= ⟨*number*⟩ | ⟨*string*⟩ | ⟨*identifier*⟩
| ⟨*comma*⟩ | ...
| (⟨*term*⟩*) | [⟨*term*⟩*] | { ⟨*term*⟩* }

For example, given the input

```
make(1, 2, 3)
```

the reader produces a sequence of two ⟨*term*⟩s: one for `make`, and another for the parentheses. The latter contains five nested ⟨*term*⟩s: `1`, a comma, `2`, a comma, and `3`.

In both Scheme and Honu, the parser consumes a ⟨*term*⟩ representation as produced by the reader, and it expands macros in the process of parsing ⟨*term*⟩s into ⟨*expression*⟩s. The ⟨*term*⟩s used during parsing need not have originated from the program source text, however; macros that are triggered during parsing can synthesize new ⟨*term*⟩s out of symbols, lists, and other literal values. The ease of synthesizing ⟨*term*⟩ representations—and the fact that they are merely ⟨*term*⟩s and not fully parsed ASTs—is key to the ease of syntactic extension in Scheme and Honu.

3.2.3 Enforestation

To handle infix syntax, the Honu parser uses an operator precedence algorithm in the *enforestation* phase to convert a relatively flat sequence of ⟨*term*⟩s into a more Scheme-like tree of

nested expressions. After a layer of enforestation, Scheme-like macro expansion takes over to handle binding, scope, and cooperation among syntactic forms. Enforestation and expansion are interleaved, which allows the enforestation process to be sensitive to bindings.

Enforestation extracts a sequence of terms produced by the reader to create a *tree term*, which is ultimately produced by a primitive syntactic form or one of the default productions of ⟨*expression*⟩, such as the function-call or list-comprehension production. Thus, the set of ⟨*tree term*⟩s effectively extends the ⟨*term*⟩ grammar although ⟨*tree term*⟩s are never produced by the reader:

⟨*term*⟩ ::= ...
| ⟨*tree term*⟩

Enforestation is driven by an `enforest` function that extracts the first expression from an input stream of ⟨*term*⟩s. The `enforest` function incorporates aspects of the precedence parsing algorithm by Pratt [27] to keep track of infix operator parsing and precedence. Specifically, `enforest` has the following contract:

`enforest` : ⟨*term*⟩* (⟨*tree term*⟩ → ⟨*tree term*⟩) ⟨*prec*⟩ ⟨*stack*⟩
→ (⟨*tree term*⟩, ⟨*term*⟩*)

The arguments to `enforest` are as follows:

- *input* — a list of ⟨*term*⟩s for the input stream;
- *combine* — a function that takes a ⟨*tree term*⟩ for an expression and produces the result ⟨*tree term*⟩; this argument is initially the identity function, but operator parsing leads to *combine* functions that close over operator transformers;
- *precedence* — an integer representing the precedence of the pending operator combination *combine*, which determines whether *combine* is used before or after any further binary operators that are discovered; this argument starts at `0`, which means that the initial *combine* is delayed until all operators are handled.
- *stack* — a stack of pairs containing a combine function and precedence level. Operators with a higher precedence level than the current precedence level push the current combine and precedence level on the stack. Conversely, operators with a lower precedence level pop the stack.

In addition, `enforest` is implicitly closed over a mapping from identifiers to macros, operators, primitive syntactic forms, and declared variables. The result of `enforest` is a tuple that pairs a tree term representing an ⟨*expression*⟩ with the remainder terms of the input stream.

The rules of enforestation are given in Figure 3.1.[2] If the first term is not a tree term or a special form then it is first converted into a tree term. Special forms include macros, operators, function calls, and bracketed sequences.

As an example, with the input

```
1+2*3-f(10)
```

enforestation starts with the entire sequence of terms, the identity function, a precedence of zero, and an empty stack:

```
enforest(1 + 2 * 3 - f (10), identity, 0, [])
```

The first term, an integer, is converted to a literal tree term, and then `enforest` recurs for the rest of the terms. I show a tree term in angle brackets:

```
enforest(<literal: 1> + 2 * 3 - f (10), identity, 0, [])
```

Since the input stream now starts with a tree term, `enforest` checks the second element of the stream, which is a binary operator with precedence 1. Enforestation therefore continues with a new *combine* function that takes a tree term for the operator's right-hand side and builds a tree term for the binary operation while the old combine function and precedence level are pushed onto the stack:

```
enforest(2 * 3 - f (10), combine1, 1, [(identity, 0)])
  where combine1(t) = <bin: +, <literal: 1>, t>
```

The first term of the new stream starts with 2, which is converted to a literal tree term:

```
enforest(<literal: 2> * 3 - f (10), combine1, 1,
         [(identity, 0)])
```

The leading tree term is again followed by a binary operator, this time with precedence 2. Since the precedence of the new operator is higher than the current precedence, a new *combine* function builds a binary-operation tree term for * while the *combine1* function and its precedence level are pushed onto the stack:

```
enforest(3 - f (10), combine2, 2,
         [(combine1, 1), (identity, 0)])
  where combine2(t) = <bin: *, <literal: 2>, t>
```

[2]The code for this model is listed in the Appendix.

enforest(*atom* $term_{rest}$ ..., *combine*, *prec*, *stack*)
= enforest(⟨literal: *atom*⟩ $term_{rest}$ ..., *combine*, *prec*, *stack*)

enforest(*identifier* $term_{rest}$ ..., *combine*, *prec*, *stack*)
= enforest((id: $identifier_{binding}$) $term_{rest}$ ...,
combine, *prec*, *stack*)
where (var: $identifier_{binding}$) = lookup(*identifier*)

enforest(*identifier* $term_{rest}$ ..., *combine*, *prec*,
[($combine_{stack}$, $prec_{stack}$) *stack*])
= enforest(*transformer*($term_{rest}$ ...),
$combine_{stack}$, $prec_{stack}$, *stack*)
where ⟨macro: *transformer*⟩ = lookup(*identifier*)

enforest($tree\text{-}term_{first}$ *identifier* $term_{rest}$ ...,
combine, *prec*, *stack*)
= enforest($term_{rest}$ ...,
`function(t){`⟨bin: *identifier*, $tree\text{-}term_{first}$, `t`⟩`}`,
$prec_{operator}$, [(*combine*, *prec*) *stack*])
where ⟨binop: $prec_{operator}$, *assoc*⟩ = lookup(*identifier*),
$prec_{operator}$ $>_{\text{assoc}}$ *prec*

enforest($tree\text{-}term_{first}$ *identifier* $term_{rest}$ ...,
combine, *prec*,
[($combine_{stack}$, $prec_{stack}$) *stack*])
= enforest(*combine*($tree\text{-}term_{first}$)
identifier $term_{rest}$ ...,
$combine_{stack}$, $prec_{stack}$, *stack*)
where ⟨binop: $prec_{operator}$, *assoc*⟩ = lookup(*identifier*),
$prec_{operator}$ $<_{\text{assoc}}$ *prec*

enforest($tree\text{-}term_{first}$ *identifier* $term_{rest}$ ...,
combine, *prec*, *stack*)
= enforest(`function(t){`identity(⟨un: *identifier*, `t`⟩)`}(`
$tree\text{-}term_{first}$)
$term_{rest}$ ...,
combine, *prec*, *stack*)
where ⟨unop: $prec_{operator}$, postfix⟩ = lookup(*identifier*),
$prec_{operator}$ $>_{\text{left}}$ *prec*

enforest(*identifier* $term_{rest}$ ..., *combine*, *prec*, *stack*)
= enforest($term_{rest}$ ...,
`function(t){`*combine*(⟨un: *identifier*, `t`⟩)`}`, $prec_{operator}$, *stack*)
where ⟨unop: $prec_{operator}$, prefix⟩ = lookup(*identifier*)

enforest((*term*$_{inside}$ ...) $term_{rest}$ ...,
combine, *prec*, *stack*)
= enforest($tree\text{-}term_{inside}$ $term_{rest}$ ..., *combine*, *prec*, *stack*)
where ($tree\text{-}term_{inside}$, ε) = enforest($term_{inside}$ ..., identity, 0, [])

Figure 3.1: Enforestation

enforest(*tree-term* (*term*$_{arg}$ ...) *term*$_{rest}$ ...,
combine, *prec*, *stack*)
= enforest(⟨call: *tree-term*, *tree-term*$_{arg}$, ...⟩ *term*$_{rest}$ ...,
combine, *prec*, *stack*)
where (*tree-term*$_{arg}$, ε) ... = enforest(*term*$_{arg}$, identity, 0, []) ...
enforest(*tree-term* [*term* ...] *term*$_{rest}$ ...,
combine, *prec*, *stack*)
= enforest(⟨arrayref: *tree-term*, *tree-term*$_{lookup}$⟩ *term*$_{rest}$ ...,
combine, *prec*, *stack*)
where (*tree-term*$_{lookup}$, ε) = enforest(*term* ..., identity, 0, [])
enforest([*term* ...] *term*$_{rest}$ ..., *combine*, *prec*, *stack*)
= enforest(⟨list: *term*, ...⟩ *term*$_{rest}$ ..., *combine*, *prec*, *stack*)
enforest({} *term*$_{rest}$ ..., *combine*, *prec*, *stack*)
= enforest(⟨block:⟩ *term*$_{rest}$ ..., *combine*, *prec*, *stack*)
enforest({*term* ...} *term*$_{rest}$ ..., *combine*, *prec*, *stack*)
= enforest(⟨block: *tree-term*, *tree-term*$_{block}$, ...⟩ *term*$_{rest}$ ...,
combine, *prec*, *stack*)
where (*tree-term*, *term*$_{unparsed}$ ...) = enforest(*term* ..., identity, 0, []),
(⟨block: *tree-term*$_{block}$, ...⟩, ε) = enforest((*term*$_{unparsed}$ ...), identity, 0, [])
enforest(*tree-term term*$_{rest}$ ..., *combine*, *prec*, [])
= (*combine*(*tree-term*), *term*$_{rest}$ ...)
enforest(*tree-term term*$_{rest}$ ..., *combine*, *prec*,
[(*combine*$_{stack}$, *prec*$_{stack}$) *stack*])
= enforest(*combine*(*tree-term*) *term*$_{rest}$ ...,
combine$_{stack}$, *prec*$_{stack}$, *stack*)

Figure 3.1: Continued

The current input sequence once again begins with a literal:

```
enforest(<literal: 3> - f (10), combine2, 2,
         [(combine1, 1), (identity, 0)])
```

The binary operator - has precedence 1, which is less than the current precedence. The current *combine* function is therefore applied to <literal: 3>, and the result becomes the new tree term at the start of the input. I abbreviate this new tree term:

```
enforest(<expr: 2*3> - f (10), combine1, 1,
                               [(identity, 0)])
  where <expr: 2*3> = <bin: *, <literal: 2>,
                               <literal: 3>>
```

Parsing continues by popping the combine function and precedence level from the stack. Since the precedence of - is the same as the current precedence and is left associative the *combine1* function is applied to the first tree term and another level of the stack is popped:

```
enforest(<expr: 1+2*3> - f (10), identity, 0, [])
```

The - operator is handled similarly to + at the start of parsing. The new *combine* function will create a subtraction expression from the current tree term at the start of the input and its argument:

```
enforest( f (10), combine3, 1, [(identity, 0)])
  where combine3(t) = <bin: -, exp<1+2*3>, t>
```

Assuming that f is bound as a variable, the current stream is enforested as a function-call tree term. In the process, a recursive call enforest(10, identity, 0, empty) immediately produces <literal: 10> for the argument sequence, so that the non-nested enforest continues as

```
enforest(<call: <id: f>, <literal: 10>>, combine3, 1,
         [(identity, 0)])
```

Since the input stream now contains only a tree term, it is passed to the current *combine* function, producing the result tree term:

```
<bin: -, <expr: 1+2*3>, <call: <id: f>, <literal: 10>>>
```

Finally, the input stream is exhausted so the identity combination function is popped from the stack and immediately applied to the tree term.

3.2.4 Macros and Patterns

From the perspective of `enforest`, a macro is a function that consumes a list of terms, but Honu programmers normally do not implement macros at this low level. Instead, Honu programmers write pattern-based macros using the `macro` form that (as noted in Section 3.1.2) has the shape

`macro` ⟨*id*⟩ `(` ⟨*literals*⟩ `)` `{` ⟨*pattern*⟩ `}` `{` ⟨*body*⟩ `}`

The `macro` form generates a low-level macro that returns a new sequence of terms and any unconsumed terms from its input. The ⟨*pattern*⟩ is compiled to a matching and destructuring function on an input sequence of terms. This generated matching function automatically partitions the sequence into the terms that are consumed by the macro and the leftover terms that follow the pattern match.

Literal identifiers and delimiters in ⟨*pattern*⟩ are matched to equivalent elements in the input sequence. A parenthesized sequence in ⟨*pattern*⟩ corresponds to matching a single parenthesized term whose subterms match the parenthesized pattern sequence, and so on. A pattern variable associated to a syntax class corresponds to calling a function associated with the syntax class to extract a match from the sequence plus the remainder of the sequence.

For example, the macro

```
macro parabola(){ x:id a:expression,
                      b:expression,
                      c:expression} {
  syntax(a * x * x + b * x + c)
}
```

expands to the low-level macro function

```
function(terms) {
  var x = first(terms)
  var [a_stx, after_a] = get_expression(rest(terms))
  check_equal(",", first(after_a))
  var [b_stx, after_b] = get_expression(rest(after_a))
  check_equal(",", first(after_b))
  var [c_stx, after_c] = get_expression(rest(after_b))
  // return new term plus remaining terms:
  [with_syntax a = a_stx, b = b_stx, c = c_stx {
```

```
      syntax(a * x * x + b * x + c)
    }, after_c]
}
```

The `get_expression` function associated to the `expression` syntax class is simply a call back into `enforest`:

```
function get_expression(terms) {
  enforest(terms, identity, 0)
}
```

New syntax classes declared with `pattern` associate the syntax class name with a function that similarly takes a term sequence and separates a matching part from the remainder, packaging the match so that its elements can be extracted by a use of the syntax class. In other words, the matching function associated with a syntax class is similar to the low-level implementation of a macro.

3.2.5 Parsing

Honu parsing repeatedly applies `enforest` on a top-level sequence of ⟨*term*⟩s, detecting and registering bindings along the way. For example, a `macro` declaration that appears at the top level must register a macro before later ⟨*term*⟩s are enforested, since the macro may be used within those later ⟨*term*⟩s.

Besides the simple case of registering a macro definition before its use, parsing must also handle mutually recursive definitions, such as mutually recursive functions. Mutual recursion is handled by delaying the parsing of curly-bracketed blocks (such as function bodies) until all of the declarations in the enclosing scope have been registered, which requires two passes through a given scope level. Multiple-pass parsing of declarations and expressions has been worked out in detail for macro expansion in Scheme [34] and Racket [19], and Honu parsing uses the same approach.

Honu not only delays parsing of blocks until the enclosing layer of scope is resolved, it even delays the enforestation of block contents. As a result, a macro can be defined after a function in which the macro is used. Along the same lines, a macro can be defined within a block, limiting the scope of the macro to the block and allowing the macro to expand to other identifiers that are bound within the block.

Flexible ordering and placement of macro bindings is crucial to the implementation of certain kinds of language extensions [19]. For example, consider a `cfun` form that supports macros with contracts:

```
cfun quadratic(num a, num b, num c) : listof num { .... }
```

The `cfun` form can provide precise blame tracking [17] by binding `quadratic` to a macro that passes information about the call site to the raw `quadratic` function. That is, the `cfun` macro expands to a combination of `function` and `macro` declarations. As long as macro declarations are allowed with the same placement and ordering rules as function declarations, then `cfun` can be used freely as a replacement for `function`.

The contract of the Honu `parse` function is

`parse` : ⟨*term*⟩* ⟨*bindings*⟩ → ⟨*AST*⟩*

That is, `parse` takes a sequence of ⟨*term*⟩s and produces a sequence of ⟨*AST*⟩ records that can be interpreted. Initially, `parse` is called with an empty mapping for its ⟨*bindings*⟩ argument, but nested uses of `parse` receive a mapping that reflects all lexically enclosing bindings.

Since `parse` requires two passes on its input, it is implemented in terms of a function for each pass, `parse1` and `parse2`:

`parse1` : ⟨*term*⟩* ⟨*bindings*⟩ → (⟨*tree term*⟩*, ⟨*bindings*⟩)
`parse2` : ⟨*tree term*⟩* ⟨*bindings*⟩ → ⟨*AST*⟩*

The `parse1` pass determines bindings for a scope, while `parse2` completes parsing of the scope using all of the bindings discovered by `parse1`.

3.2.5.1 Parsing details. The details of the `parse1` and `parse2` functions are given in this section.

The `parse1` function takes *input* as the ⟨*term*⟩ sequence and *bindings* as the bindings found so far. If *input* is empty, then `parse1` returns with an empty tree term sequence and the given *bindings*. Otherwise, `parse1` applies `enforest` to *input*, the identity function, and zero; more precisely, `parse1` applies an instance of `enforest` that is closed over *bindings*. The result from `enforest` is *form*, which is a tree term, and *rest*, which is the remainder of *input* that was not consumed to generate *form*. Expansion continues based on case analysis of *form*:

- If *form* is a `var` declaration of *identifier*, then a variable mapping for *identifier* is added to *bindings*, and `parse1` recurs with *rest*; when the recursive call returns, *form* is added to (the first part of) the recursion's result.

- If *form* is a `macro` or `pattern` declaration of *identifier*, then the macro or syntax class's low-level implementation is created and added to *bindings* as the binding of *identifier*. Generation of the low-level implementation may consult *bindings* to extract the implementations

of previously declared syntax classes. The `parse1` function then recurs with *rest* and the new *bindings*.

If `parse1` was called for the expansion of a module body, then an interpretable variant of *form* is preserved in case the macro is exported. Otherwise, *form* is no longer needed, since the macro or syntax-class implementation is recorded in the result *bindings*.

- If *form* is an expression, `parse1` recurs with *rest* and unchanged *bindings*; when the recursive call returns, *form* is added to (the first part of) the recursion's result.

The results from `parse1` are passed on to `parse2`. The `parse2` function maps each *form* in its input tree term to an AST:

- If *form* is a `var` declaration, the right-hand side of the declaration is parsed through a recursive call to `parse2`. The result is packaged into a variable-declaration AST node.
- If *form* is a `function` expression, the body is enforested and parsed by calling back to `parse`, passing along `parse2`'s ⟨*bindings*⟩ augmented with a variable binding for each function argument. The result is packaged into a function- or variable-declaration AST node.
- If *form* is a block expression, then `parse` is called for the block body in the same way as for a `function` body (but without argument variables), and the resulting ASTs are packaged into a single sequence AST node.
- If *form* is an identifier, then it must refer to a variable, since macro references are resolved by `enforest`. The identifier is compiled to a variable-reference AST.
- If *form* is a literal, then a literal AST node is produced.
- Otherwise, *form* is a compound expression, such as a function-call expression. Subexpressions are parsed by recursively calling `parse2`, and the resulting ASTs are combined into a suitable compound AST.

3.2.5.2 Parsing example. As an example, consider the following sequence:

```
macro info(at){ x:id, math:expression at point:expression } {
  syntax({
    var f = function(x) { math }
    printf("at ~a dx ~a\n", f(point))
```

```
  })
}

info x, x*x+2*x-1 at 12
```

Initially, this program corresponds to a sequence of ⟨*terms*⟩ starting with `macro`, `info`, and `(at)`. The first parsing step is to enforest one form, and enforestation defers to the primitive `macro`, which consumes the next four terms. The program after the first enforestation is roughly as follows, where I represent a tree term in angle brackets as before:

```
<macro declaration: info, ...>

info x, x*x+2*x-1 at 12
```

The macro-declaration tree term from `enforest` causes `parse1` to register the `info` macro in its *bindings*, then `parse1` continues with `enforest` starting with the `info` identifier. The `info` identifier is bound as a macro, and the macro's pattern triggers the following actions:

- it consumes the next `x` as an identifier;
- it consumes the comma as a literal;
- it starts enforesting the remaining terms, which succeeds with a tree term for `x*x+2*x-1`;
- it consumes `at` as a literal;
- starts enforesting the remaining terms as an expression, again, which succeeds with the tree term `<literal: 12>`.

Having collected matches for the macro's pattern variables, the `info` macro's body is evaluated to produce the expansion, so that the overall sequence becomes

```
{
  var f = function(x) { <expr: x*x+2*x-1> }
  printf("at ~a dx ~a\n", f(<literal: 12>))
}
```

Macro expansion of `info` did not produce a tree term, so `enforest` recurs. At this point, the default production for curly braces takes effect, so that the content of the curly braces is preserved in a block

tree term. The block is detected as the `enforest` result by `parse1`, which simply preserves it in the result tree term list. No further terms remain, so `parse1` completes with a single tree term for the block.

The `parse2` function receives the block, and it recursively parses the block. That is, `parse` is called to process the sequence

```
var f = function(x) { <expr: x*x+2*x-1> }
printf("at ~a dx ~a\n", f(<literal: 12>))
```

The first term, `var`, is bound to the primitive declaration form, which consumes `f` as an identifier, `=` as a literal, and then enforests the remaining terms as an expression.

The remaining terms begin with `function`, which is is the primitive syntactic form for functions. The primitive `function` form consumes the entire expression to produce a tree term representing a function. This tree term is produced as the enforestation that `var` demanded, so that `var` can produce a tree term representing the declaration of `f`. The block body is therefore to the point

```
<function declaration: f, <function: x,
                           <expr: x*x+2*x-1>>>
printf("at ~a dx ~a\n", f(<literal: 12>))
```

When `parse1` receives this function-declaration tree term, it registers `f` as a variable. Then `parse1` applies `enforest` on the terms starting with `printf`, which triggers the default function-call production since `printf` is bound as a variable. The function-call production causes enforestation of the arguments `"at ~a dx ~a\n"` and `f(<literal: 12>)` to a literal string and function-call tree term, respectively. The result of `parse1` is a sequence of two tree terms:

```
<function declaration: f, <function: x,
                           <expr: x*x+2*x-1>>>
<call: <var: printf>,
       <literal: "at ~a dx ~a\n">,
       <call <var: f>, <literal: 12>>>
```

The `parse2` phase at this level forces enforestation and parsing of the function body, which completes immediately, since the body is already a tree term. Parsing similarly produces an AST for the body in short order, which is folded into a AST for the function declaration. Finally, the function-call tree term is parsed into nested function-call ASTs.

3.2.5.3 Parsing as expansion. For completeness, I have described Honu parsing as a stand-alone and Honu-specific process. In fact, the Honu parser implementation leverages the existing macro-expansion machinery of Racket. For example, the Honu program

```
#lang honu
1+2
```

is converted via the Honu reader to

```
#lang racket
(honu-block 1 + 2)
```

The `honu-block` macro is implemented in terms of `enforest`:

```
(define-syntax (honu-block stx)
  (define terms (cdr (syntax->list stx)))
  (define-values (form rest) (enforest terms identity 0))
  (if (empty? rest)
      form
      #`(begin #,form (honu-block . #,rest))))
```

where `#‘` and `#,` are forms of `quasiquote` and `unquote` lifted to the realm of lexically scoped S-expressions.

The strategy of treating `enforest`’s first result as a Racket form works because `enforest` represents each tree term as a Racket S-expression. The tree term for a Honu `var` declaration is a Racket `define` form, function call and operator applications are represented as Racket function calls, and so on.

Expanding `honu-block` to another `honu-block` to handle further terms corresponds to the `parse1` recursion in the stand-alone description of Honu parsing. Delaying enforestation and parsing to `parse2` corresponds to using `honu-block` within a tree term; for example, the enforestation of

```
function(x) { D y, y*x }
```

is

```
(lambda (x) (honu-block D y |,| y * x))
```

When such a function appears in the right-hand side of a Racket-level declaration, Racket delays expansion of the function body until all declarations in the same scope are processed, which allows a macro definition of `D` to work even if it appears after the function.

Honu `macro` and `pattern` forms turn into Racket `define-syntax` forms, which introduce expansion-time bindings. The `enforest` function and pattern compilation can look up macro and syntax-class bindings using Racket’s facilities for accessing the expansion-time environment [19].

Besides providing an easy way to implement Honu parsing, building on Racket's macro expander means that the more general facilities of the expander can be made available to Honu programmers. In particular, Racket's compile-time reflection operations can be exposed to Honu macros, so that Honu macros can cooperate in the same ways as Racket macros to implement pattern matchers, class systems, type systems, and more.

3.3 Extended Example

I build a class system on top of a primitive form for defining records as an example of using Honu macros. Classes use a single inheritance hierarchy with the root being the class `object`. Each class has a single constructor whose parameters are given next to the class name, and method calls use `call`:

```
⟨class⟩      ::= class ⟨identifier⟩ ( ⟨identifier⟩* )
                 extends ⟨identifier⟩ ( ⟨identifier⟩* )
                 { ⟨field⟩* ⟨method⟩* }
⟨field⟩      ::= var ⟨identifier⟩ = ⟨expression⟩
⟨method⟩     ::= function ⟨identifier⟩ ( ⟨identifier⟩* ) { ⟨sequence⟩ }
⟨expression⟩ ::= ...
               | call ⟨expression⟩ ⟨identifier⟩ ( ⟨sequence⟩ )
               | this
```

For example, I can define a `fish` class whose instances start with a given weight, and a `picky_fish` subclass whose instances start with a fraction of food that they are willing to eat:

```
class fish(weight) extends object() {
  function eat_all(amt) { weight = weight + amt }
  function eat(amt) { call this eat_all(amt) }
  function get_weight() { weight }
}
class picky_fish(fraction) extends fish(weight) {
  function eat(amt) {
     call this eat_all(fraction * amt)
  }
}
var c = picky_fish(1/2, 5)
call c eat(8)
call c get_weight()
```

The `class` macro implementation relies on syntax classes for ⟨*field*⟩ and ⟨*method*⟩ declarations:

```
1  pattern field_clause (var =) {
2    var name:identifier = value:expression
3  }
4  pattern method_clause (function) {
5    function name:identifier(argument:identifier ...){
6      body ...
7    }
8  }
```

The `field_clause` syntax class uses `var` and `=` as literals, matching an identifier between them and an expression afterward. In the `method_class` syntax class, `function` is a literal; the body of a method does not have a syntax class, which means that it is left unparsed when the `method_clause` pattern is parsed. The `class` macro uses these syntax classes in its pattern:

```
9  macro extends(){ } { error("illegal use of keyword") }
10
11 macro class (function extends){
12    name:identifier(arg:identifier ...)
13    extends parent:identifier(parent_arg:identifier ...){
14      vars:field_clause ...
15      meths:method_clause ...
16   }
17 } {
18   var name_stx = first(syntax_to_list(syntax(name)))
19   var this_stx = to_syntax(name_stx, 'this, name_stx)
20   var meth_names = to_list(syntax(meths_name_x ...))
21   var function_names = [syntax_to_string(name):
22                         name = meth_names]
23   .... // continued below
24   }
```

Line 9 declares `extends` for use as a literal in the `class` macro, while a use of `extends` in an expression position triggers a syntax error. Lines 12–15 specify the pattern for uses of `class`. Line 18 extracts the class name as a syntax object, so that a `this` identifier on line 11 can be given (unhygienically) the same lexical context as the class name although a more robust solution

developed by Barzilay et al. [5] could have been used. Lines 20–22 extract the class's method names and converts them to strings.

Line 23 above continues as follows to build the expansion of a `class` form:

```
25  with_syntax this = this_stx,
26              (function_name ...) = function_names, {
27    syntax(var name = {
28      struct implementation{vtable, super, $ arg ,
29                            $ ... $ vars_name , $ ...}
30      function (arg ... parent_arg ...){
31        var vtable = mutable_hash()
32        $ hash_update(vtable, function_name,
33                      function(this, meths_argument ...){
34                        meths_body ... }) $ ...
35        implementation(vtable, parent(parent_arg ...),
36                       $ arg, $ ... $ vars_value , $ ...)
37      }
38    })
39  }
```

The `with_syntax` form at line 24 binds `this` as a pattern variable to the identifier syntax object in `this_stx`, and it binds `function_name` as a pattern variable for sequence of function names from `function_names`. Lines 28–36 implement the class. The result of the class macro is a constructor function bound to the name of the class. The constructor accepts the parameters declared next to the class name, as well as parameters declared next to the super class name; it instantiates a record containing the class parameters and an instance of the super class. The class's virtual method table is created by mapping each function name to a function that accepts the original method parameters as well as an extra `this` argument. Delaying the parsing of method bodies (in the `method_clause` pattern) ensures that `this` is in scope before the method body is parsed.

The `object` root class is defined directly as a function:

```
function object() {
  struct dummy{vtable, super}
  dummy(mutable_hash(), false)
}
```

Method calls rely on a simple `find_method` lookup function private to the user of the class system.

```
function find_method(object, name) {
  var vtable = object.vtable
  var method = hash_lookup(vtable, name, function (){ false })
  if (method == false) {
    if (object.super == false) {
      error("could not find method", name)
    } else {
      find_method(object.super, name)
    }
  } else {
    method
  }
}
```

The public version of `find_method` is the `call` macro.

```
39  macro call(){ object:expression
40                name:identifier(arg:expression ...) } {
41    var name_stx = first(to_list(syntax(name)))
42    with_syntax name_str = syntax_to_string(name_stx) {
43      syntax({
44        var target = object
45        var method = find_method(target, name_str)
46        method(target, arg ...)
47      })
48    }
49  }
```

The pattern on lines 39–40 matches an expression for the object whose method is being called, an identifier for the method name, and expressions for the arguments. The body of the macro converts the method name to a string on lines 41–42. The expansion of the macro on lines 43–47 is a block that binds `target` to the target object of the method call, finds the called method in the target object, and then calls the method passing along the target object as the first argument.

3.4 Honu Implementation

Honu is implemented as a language using Racket's extensible module system. Racket allows languages to be specified at the top of a file with a line that starts with `#lang` and then the name of a language; in the case of Honu it is simply

```
#lang honu
```

Languages in Racket have two main points of control to set up how programs behave. The first is specifying the read function, which turns raw characters into syntax objects. Honu implements a read function called `honu-read` that tokenizes its input using Java-like regular expressions for things like identifiers and numbers. The read function returns a syntax object which represents a balanced Honu tree consisting of matching parenthesis, brackets, and braces. Semicolons in Honu also produce a subtree by wrapping all the previous tokens until another semicolon within the current tree is seen. Delimiters, such as parenthesis and semicolons, are stored by tagging the tree with an identifier as the first token which relates the delimiter to the original input.

The surface syntax is parsed according to the following grammar.

⟨*term*⟩ ::= ⟨*atom*⟩
| (⟨*term*⟩*)
| { ⟨*term*⟩* }
| [⟨*term*⟩*]
| ⟨*term*⟩* ;

For example, matching parenthesis and braces are turned into `#%parens` and `#%braces`, respectively.

```
if (x > 5){
  foo(x)
}
```

Read will return a syntax object that looks as follows

```
(if (#%parens x > 5)
  (#%braces foo (#%parens x)))
```

Semicolons wrap all the previous tokens as long as there is no other semicolon has been seen.

```
if (x > 5){
  foo(x);
  bar(5);
}
```

Is read as

```
(if (#%parens x > 5)
  (#%braces
    (#%semicolon foo (#%parens x))
    (#%semicolon bar (#%parens 5))))
```

Although macros in Honu can technically pattern match on semicolons in subexpressions the purpose of the semicolon is to give users the ability to separate expressions regardless of the intentions of a macro. The normal use of a semicolon is to allow macros to parse an undetermined amount of syntax without consuming the entire input stream.

One use of this feature is with the `require` form. The `require` form is used to import modules into the current program's namespace. It is simpler for the `require` pattern to match everything and parse the result procedurally than to use utilities from the pattern matcher to do the same job.

```
macro require(){ form ... }{ ... }
```

The intended use of this form is to put a semicolon after the end of the forms that specify the modules to import.

```
require for_meta 1 racket/list, for_meta 0 racket/draw;
```

Honu uses the standard module defining facilities in Racket by defining the reader to use the `syntax/module-reader` language. The resulting syntax object produced by `honu-read` is wrapped in a Racket `module` form as well as a `module-begin` form.

```
(module filename base-language
  (#%module-begin forms ...))
```

The second point of control a Racket language has is defining `module-begin` as a macro that can rewrite the module as per its own rules. Typed Racket, for example, defines `module-begin` to expand all the subforms to find type declarations and compute types of all the expressions to ensure the program is well-typed. Honu's `module-begin` form is much simpler because Honu does not need to analyze the body of the module, but rather it just wraps the forms with a macro that invokes the enforest function to start the process of parsing Honu code.

```
(provide (rename-out [honu-module-begin #%module-begin]))
(define-syntax (honu-module-begin stx)
```

```
  (syntax-case stx ()
    [(_ forms ...)
     #'(#%module-begin (honu-unparsed-begin forms ...))]))
```

The original `#%module-begin` is used to wrap the new expression so that the language cooperates with the rest of the Racket machinery for compiling and executing the program. The `module-begin` form defined by Honu is named `honu-module-begin` and exported as `#%module-begin` which is required to integrate with the rest of the Racket system.

The `honu-unparsed-begin` macro calls enforest once and then wraps the output in `honu-unparsed-begin` again. If there are no arguments to `honu-unparsed-begin` then it returns a void expression thus terminating the Honu parsing process.

```
(define-syntax (honu-unparsed-begin stx)
  (syntax-case stx ()
    [(_ forms ...)
     (let-values ([(parsed unparsed) (enforest #'(forms ...))])
       (with-syntax ([parsed parsed]
                     [(unparsed ...) unparsed])
         #'(begin parsed (honu-unparsed-begin unparsed ...))))]
    [(_) #'(void)]))
```

The reason that `honu-unparsed-begin` returns a new syntax object instead of immediately calling `enforest` on the unparsed part of the input is that it gives the expander a chance to inspect the parsed form and add any new bindings to the current scope before moving on to subexpressions. That is, suppose I started with the following code which has already been processed by `honu-read`.

```
(honu-unparsed-begin
  var x = 1
  println(#%parens x * 2))
```

The first call to `enforest` will parse the **var** expression and return a Racket level expression for the definition of `x`.

```
(define x 1)
```

The unparsed part of the syntax will remain unchanged as

```
println(#%parens x * 2)
```

The output from the expander after processing `honu-unparsed-begin` once will thus be

```
(begin (define x 1) (honu-unparsed-begin println(#%parens x * 2)))
```

Racket's expander will process this S-expression from left to right so it will register a binding for `x` due to the `define` form before moving on to processing the `println` expression. Thus when Honu parses the `println` expression it will see that `x` is bound to a lexical variable.

The top-level module forms are not the only place where `honu-unparsed-begin` is used. Any time parsing of subexpressions is meant to be delayed they are wrapped in a `honu-unparsed-begin` form. Functions and macros with block bodies fall into this category. The following example defines a simple Honu function that will be parsed to a Racket `define` form. For simplicity I show the Honu code in its original form without the internal `#%parens` and `#%comma` identifiers but in reality `honu-unparsed-begin` would process the equivalent S-expression.

```
(honu-unparsed-begin
  function weight(mass, gravity){
    mass * gravity
  })
```

The enforest function recognizes the `function` identifier followed by another identifier, then parenthesis and block with braces which signifies a function definition. The output is a `define` form.

```
(define (weight mass gravity)
  (honu-unparsed-begin
     mass * gravity))
```

The body is wrapped with `honu-unparsed-begin` to allow the expander to register the arguments `mass` and `gravity` in the current lexical scope. Once the new variables have been registered the body is expanded as usual in a scope where `mass` and `gravity` refer to the arguments to the function.

The `honu-unparsed-begin` macro is internal to the Honu implementation so macro programmers cannot access it directly. Instead if they wish to delay parsing of subexpressions they can wrap them in a block consisting of opening and closing braces.

```
macro with_close(){ z:identifier in e:expression{ body ... } }{
  syntax(var z = e; { body ... } z.close() )
}

with_close a in get(){
  fill(a)
}
```

Braces by themselves are parsed by enforest as a Racket `let` expression whose body is wrapped with `honu-unparsed-begin`. The use of the macro above is rewritten as

```
var a = get();
{ fill(a) }
a.close()
```

The result of parsing `{fill(a)}` is

```
(let ()
  (honu-unparsed-begin fill(a)))
```

3.4.1 Syntax Patterns

Honu relies heavily on the syntax pattern matching facilities of `syntax/parse` developed by Culpepper [13]. The `syntax/parse` form implements syntax classes which relieve much of the tedium in using the enforest function in macros. It is also useful in implementing the enforest function itself as the enforest function relies on syntax classes to parse Honu.

Syntax classes are normally specified with patterns and whose return value is the implicit syntax matched by the class. For example, the following class matches a number, followed by an equals sign, followed by another number.

```
(define-syntax-class two-numbers
  [pattern a:number (~literal =) b:number])
```

This syntax class can then be used in any pattern that appears in a `syntax-parse` expression. The parser for enforest uses syntax classes to match Honu. For example in the following code the line with `var` in it is a pattern which matches a variable declaration.

```
(define (enforest input)
  (syntax-parse input
    [(var name:id = e:expression rest ...)
     (let () ...)]))
```

The `id` syntax class is built into `syntax/parse` and matches a single identifier. The `expression` syntax class is the same one used in Honu macros to match a single Honu expression. Unlike normal syntax classes, which are pattern-based, the `expression` syntax class is procedural so that it can recursively invoke enforest. The `expression` syntax class is implemented using a lower-level syntax class form that only requires it to behave according to the `syntax/parse` parsing protocol. In particular, a low-level syntax class must return a list consisting of two things, the number of consumed S-expressions and the matched syntax object. In the case that the syntax class fails to match it must call a failure function passed to it that signifies that the parser should skip that syntax class.

```
(define-primitive-splicing-syntax-class honu-expression
   #:attributes (result)
   (lambda (stx fail)
     (define mark (make-syntax-introducer))
     (define-values (parsed unparsed)
                    (enforest (mark stx)))
     (list (- (length stx) (length unparsed))
           (mark parsed))))
```

There are two other things to note in this code. The first is that syntax classes must statically declare the syntax attributes that can be used when a pattern variable uses the syntax class. In this case the keyword argument `#:attributes` declares one attribute, `result`. The second thing is that `honu-expression` is the point at which hygiene is enforced in the enforest layer. A new marking procedure is bound to `mark` and called on the input `stx`. The output is marked with the same procedure thus canceling any duplicate marks from the first time.

Pattern classes in Honu are implemented directly as syntax classes. The syntax patterns that pattern classes and macros use in Honu are converted into similarly looking `syntax/parse` patterns. The syntax matched by the syntax class `honu-expression` would result in exactly the original input syntax. The point of using `honu-expression`, though, is to execute the enforest function and get back an opaque result. Therefore all syntax classes have a `result` attribute which is bound

to the result of enforest if it is called. Honu patterns are rewritten so that the original pattern variable is bound to the result attribute so that they can be used as normal. For example a raw macro in Honu implemented in Racket would like so

```
(syntax-parse stx
  [(x:honu-expression)
   #'(x.result + 1)])
```

Where the user has to use `x.result` instead of the more natural `x`. Honu macros bind `x.result` to `x` which has the side affect of preventing programmers from accidentally using the raw syntax. The implemented macro looks as follows

```
(syntax-parse stx
  [(x:honu-expression)
   (with-syntax ([x #'x.result])
     #'(x + 1))])
```

3.4.2 Integration with Racket

Honu is parsed into a corresponding Racket program, and while parsing is occuring Honu syntax and Racket syntax are intermingled. To differentiate between them I use a syntax property and attach it to any syntax that has been processed by the enforest function.

```
(define honu-property 'honu-parsed)
(define (parsed-syntax stx)
  (syntax-property stx honu-property #t))
(define (parsed-syntax? stx)
  (syntax-property stx honu-property))
```

Only syntax objects that have the `honu-parsed` property will return `#t` when the `parsed-syntax?` predicate is called on them. This property is important to prevent the enforest function from trying to parse already parsed Honu expressions. For example in the following program the `a` macro expects to parse an expression but its argument is the output of the `b` macro that returns an already parsed expression.

```
macro b(){ x:expression } { syntax(x) }
macro a(){ z:expression } { syntax(1 + z) }
a b 2 + 3
```

The input to `a` will be the syntax object `(+ 2 3)`, which is not a Honu expression. To make it a Honu expression it is passed to `parsed-syntax`. The enforest function first checks if its input returns true for the predicate `parsed-syntax?` and if so returns it immediately.

```
(define (enforest input)
  (if (parsed-syntax? input)
    input
    (syntax-parse input
      ...)))
```

If the input syntax contains Racket code inside it then that code should be wrapped with a macro which will cause further enforestation to occur. Blocks, for example, are wrapped with `honu-unparsed-begin` which repeatedly calls enforest on the subexpressions until none are left.

Racket macros cannot be used directly in Honu code, which is unfortunate because there is a very large ecosystem of Racket code. To bridge the gap a Honu macro can be easily written in Racket and exported to Honu as long as the macro behaves according to the Honu parsing strategy.

A Honu macro must return two syntax objects, the new syntax result as computed by the macro and the unconsumed input. The only other salient point is that a Racket implementation should use `racket-syntax` instead of `syntax` to return a syntax object with the `parsed-syntax` property. A general recommendation is that macros use `syntax-parse` so that they can use the syntax class `honu-expression` instead of directly calling enforest. The following is an example of a Honu while loop macro implemented in Racket:

```
(define-honu-syntax honu-while
 (lambda (code)
  (syntax-parse code
   [(_ condition:honu-expression body:honu-body . rest)
   (values
    (racket-syntax (let loop ()
                    body.result
                    (when
                     condition.result
                     (loop))))
    #'rest)])))
```

The syntax class `honu-body` matches a sequence of expressions wrapped in a `#%brackets` delimeter.

3.4.3 Expression Phases

In Racket a macro can match a subexpression and potentially use that subexpression in an arbitrary phase. Any bindings for that subexpression will thus be resolved in the phase where the subexpression appears, regardless of what phase the macro invocation is used in.

```
(define-syntax-rule (at-phase-0 x)
  (begin x))
(define-syntax-rule (at-phase-1 x)
  (begin-for-syntax x))

(define x 1)
(at-phase-0 x)
(at-phase-1 x)
```

The call to `at-phase-0` will expand to `(begin x)` and `x` will resolve to the definition immediately above it. The call to `at-phase-1` will expand to `(begin-for-syntax x)` which will be expanded at phase 1 rather than phase 0 where the macro was first invoked. In this example there is no binding for `x` at phase 1 so a compilation error will occur. The point of this example is simply to show that Racket macros can determine what phase their subexpressions will be compiled in.

Due to the early parsing by enforest a Honu macro must be more up-front about which phase to parse in. The `honu-expression` syntax class parses in the same phase as the macro definition. To parse in one phase higher a default `honu-expression/phase+1` is provided which invokes the enforest function at a phase one higher than the macro definition.

CHAPTER 4

APPLICATIONS

I implement three example macros that demonstrate the power of the Honu macro system. First an inline XML macro that can escape to Honu expressions, second an implementation of Linq, and third a parser generator.

4.1 XML

The XML macro is the first example where the input is parsed with a more sophisticated strategy. In particular an XML node can contain instances of itself so the pattern class will reference itself to parse recursively.

The grammar for XML looks roughly as follows

⟨*node*⟩ `::= <` ⟨*identifier*⟩ `>` ⟨*node*⟩* `< /` ⟨*identifier*⟩ `>`
`|` ⟨*literal*⟩*

I can implement this grammar with the following pattern.

```
(define-splicing-syntax-class node
  [pattern (~seq < start:id > more:node ... < / end:id >)]
  [pattern (~seq plain:non-xml-id plain*:non-xml-id ...)])
```

Where `non-xml-id` is just an identifier but will not be confused with matching XML tokens such as `<`. The `node` pattern explicitly matches the identifiers `<`, `>`, and `/` which are already tokens in the Honu language.

With this pattern I can create simple XML expressions.

```
var dom = xml <html><body>Hello world</body></html>
```

I can extend this macro with the ability to insert Honu expressions as the content of XML nodes. A new clause is added to the `node` pattern that parses a Honu body.

```
(define-splicing-syntax-class node
  [pattern (~seq < start:id > more:node ... < / end:id >)]
```

```
  [pattern body:honu-body]
  [pattern (~seq plain:non-xml-id plain*:non-xml-id ...)])
```

A Honu body is any sequence of Honu code wrapped inside {} delimiters.

```
var dom = xml <html><body> { computeBody() } </body> </html>
```

Here I invoke the function computebody during the creation of this XML object. The **body** node will contain the result of the function call.

4.2 Linq

Linq is a domain specific language mainly used in C# that provides an SQL-like syntax for objects that implement an inteferface resembling a database. Data can be extracted from those objects using complex Linq queries that would otherwise be difficult to write just using method calls.

To illustrate, the following code is a Racket expression that selects all the XML nodes that are a descendant of the **Table1** node and have an age larger than 20, orders them lexicographically by their **familyName** element, and finally returns a pair consisting of the **familyName** element and the **address** element.

```
(define addresses
  (for/list ([add (sort (xml-descendents xml "Table1") string<?
                        #:key (lambda (element)
                                (xml-content element "familyName")))]
             #:when (> (xml-content element "age") 20))
    (list (xml-content add "familyName")
          (xml-content add "address"))))
```

Although this code is fairly readable, the Linq version is more straightforward both to program and to read.

```
var addresses = linq from add in xml->Descendants("Table1")
                     where add->Element("age") > 20
                     orderby add->Element("familyName")->Value()
                     select pair(add->Element("familyName")->Value(),
                                 add->Element("address")->Value())
```

Linq must be implemented as a macro because the syntax does not match any builtin Honu forms. Writing a parser for the Linq domain is a reasonable affair in Honu because of the free form nature of enforest. Macros that expand to Linq code can be implemented by a special form that defines a Linq-specific macro object and a pattern in the Linq parser that can invoke those objects.

The full grammar of Linq is

⟨*linq*⟩ ::= `linq` ⟨*from-clause*⟩ ⟨*query-body*⟩
⟨*query-body*⟩ ::= ⟨*body*⟩* ⟨*select-or-group-clause*⟩ [⟨*continuation*⟩]
⟨*body*⟩ ::= ⟨*from-clause*⟩
| ⟨*let-clause*⟩
| ⟨*where-clause*⟩
| ⟨*join-clause*⟩
| ⟨*join-into-clause*⟩
| ⟨*orderby-clause*⟩
⟨*from-clause*⟩ ::= `from` [⟨*type*⟩] ⟨*identifier*⟩ `in` ⟨*expression*⟩
⟨*let-clause*⟩ ::= `let` ⟨*identifier*⟩ `=` ⟨*expression*⟩
⟨*where-clause*⟩ ::= `where` ⟨*expression*⟩
⟨*join-clause*⟩ ::= `join` [⟨*type*⟩] ⟨*identifier*⟩ `in` ⟨*expression*⟩ `on` ⟨*expression*⟩ `equals` ⟨*expression*⟩
⟨*join-into-clause*⟩ ::= ⟨*join-clause*⟩ `into` ⟨*identifier*⟩
⟨*orderby-clause*⟩ ::= `orderby` ⟨*ordering*⟩*
⟨*ordering*⟩ ::= ⟨*expression*⟩ ⟨*direction*⟩
⟨*direction*⟩ ::= `ascending` | `descending`
⟨*select-or-group-clause*⟩ ::= ⟨*select-clause*⟩ | ⟨*group-clause*⟩
⟨*select-cluase*⟩ ::= `select` ⟨*expression*⟩
⟨*group-clause*⟩ ::= `group` ⟨*expression*⟩ `by` ⟨*expression*⟩
⟨*continuation*⟩ ::= `into` ⟨*identifier*⟩ ⟨*query-body*⟩

I can directly translate this BNF into a series of patterns.

```
pattern query_body(){ body:body ... s:select_or_group_cluase c:continuation? }

pattern body(){ e:from_clause } { e:let_clause }{ e:where_clause }{
  e:join_clause }{ e:join_into_clause }

pattern into_part(into){ into e:expression }

pattern from_clause(from, in){ from name:identifier in expr:expression
  into:into_part? }

pattern let_clause(let, =){ let name:identifier = expr:expression }

pattern where_clause(where){ where expr:expression }

pattern join_clause(join, in, on, equals){ join name:identifier in
  expr1:expression on expr2:expression equals expr3:expression }
```

```
pattern join_into_clause(join, in, on, equals, into){ join name:identifier in
  expr1:expression on expr2:expression equals expr3:expression into
  what:identifier }

pattern orderby_cluase(orderby){ orderby orders:order ... }

pattern order(ascending, descending){ ascending }{ descending }

pattern select_or_group_clause(){ select_clause }{ group_clause }

pattern select_clause(select){ e:expression }

pattern group_clause(group, by){ group expr1:expression by expr2:expression }

pattern continuation(into){ into id:identifier body:query_body }
```

With the appropriate literals defined as well.

```
var from = 0
var where = 0
...
```

The pattern matcher that macros use expect literal identifiers to be bound, and since the value of the literals will never be used it is immaterial what their value actually is. However, a literal can be bound as a macro that produces an error during its expansion at compile-time to prevent users from accidentally using the literal in expression position.

The Linq macro simply uses the `query_body` pattern as a starting point to parse Linq expressions.

```
macro linq(){ body:query_body }{ ... }
```

There are a number of ways to convert a Linq expression into a language expression. The implementation in C# is mostly a syntactic rewrite of each clause into a method call on an object that represents the Linq state. For example the C# Language Specification 3.1 shows the translation of a simple linq expression with a groupby clause.

```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Count() }
```

This is translated in two steps, first into

```
from g in
  from c in customers
  group c by c.Country
select new { Country = g.Key, CustCount = g.Count() }
```

And then into the final C# expression

```
customers.
GroupBy(c => c.Country).
Select(g => new { Country = g.Key, CustCount = g.Count() })
```

This example clearly demonstrates how the different parts of the Linq expression are intertwined. The variable g referenced after the **into** keyword is ultimately used in the select expression. Due to the nonlocal transformations that each pattern must undergo they cannot expand into syntax simply based on their immediate input. Instead the Linq fragments can be parsed and analyzed by the Linq macro which can deal with introducing variables for expressions that do not explicitly introduce them mention them.

Each Honu pattern therefore creates a tree that is easy for the Linq macro to deconstruct. For example the from clause will create a syntactic fragment based on its input that will be used later by the Linq macro.

```
pattern from_clause(from in into){ from name:identifier in expr:expression
out:into_part? }, {
  from(name, expr, out)
}
```

The syntax `from(name, expr, out)` is the result of this pattern. Note that `expr` uses the expression syntax class, and so any macros in `expr` will be expanded.

All the results from the patterns are bundled up in a sequence bound to the `body` pattern variable. The Linq macro can inspect these results to create the final expression that implements the Linq query.

```
macro linq(){ body:query_body }{
  var parts = syntax(body_body ...)
  compile_linq(parts)
}
```

`compile_linq` is a phase 1 function that analyzes the Linq fragments and produces an expression similar to the C# one above where SQL-like methods are called on a object representing a database.

4.3 Parser Generator

Parsers are a common tool required by sophisticated programs. Typically they are built by composing language features such as function parser combinators or an external program can generate the code based on a parser specification. Parsers built inside a language from datastructures and functions are interpreted by a parser engine which can be much less performant than generated code. The benefit of using existing language features to implement the parser is that existing abstractions can be used to improve code readability.

I can combine the two approaches by creating a macro that parses a BNF specification for a parser and generating a relatively performant backtracking PEG parser. The following BNF describes a simple arithemetic language where the operators have the usual precedence levels.

⟨*expression*⟩ ::= ⟨*number*⟩
| ⟨*expression*⟩ ⟨*operator*⟩ ⟨*expression*⟩
⟨*operator*⟩ ::= + | - | * | /

A non-left recursive version of this that properly respects the precedence of the operators is

```
peg start {
   start = { expr eof }
   expr = { expr2 ws expr1_rest }
   expr1_rest = { "+" ws expr2 ws expr1_rest }
              | { "-" ws expr2 ws expr1_rest }
              | { void }
   expr2 = { expr3 ws expr2_rest }
   expr2_rest = { "*" ws expr3 ws expr2_rest }
              | { "/" ws expr3 ws expr2_rest }
              | { void }
   ws = { " "* }
```

```
    expr3 = { number }
}
```

Each production is wrapped in braces to make pattern matching simpler for the parser generator but a more sophisticated generator could leave them off. The `peg` form is a macro where the immediately following identifier specifies the rule to start parsing with. The syntax of rules is straightforward, a rule name followed by an equals sign and a series of productions. This can be parsed simply enough using a pattern to match rules.

```
macro peg(){
  first:identifier {
   rule:rule ...
  }
} {
  syntax({
    $ rule_inner ... $ ...
    first
  })
}
```

The `peg` macro creates the rules as first-class values and returns the starting rule as its result.

```
pattern rule(=){
  name:identifier = first:production more:production_rest ...
}, {
  syntax(
    function name (data){
      first_inner ... (data) $ or more_production_inner ... (data) $ ...
                             or false
    })
  }
```

A rule is a named function that accepts one input which is the string to match. The rule matches a series of productions to the input in the order they are declared and returns the first nonfalse value. One thing to note here is the implicit `_inner` suffix on the pattern variables. The pattern variable

`first` contains a series of Honu terms that must be spliced into the final context to be parsed properly. The `_inner` suffixed version of the pattern variable has an ellipses depth of 1 which lets it be used with an ellipses in the template.

```
pattern production(){
{ element1:element_modifier elements:element_modifier ... }
}, {
  syntax(
   function (data){
     for element in [element1_inner ..., $ elements_inner ..., $ ...] do {
       if (data){
        data = element(data)
       } else {
       }
     }
     data
   })
}

pattern production_rest( | ){ | production:production }
```

A production consists of a series of elements that are themselves functions that return either the unconsumed input or false. If any of the elements do not match then the entire production fails and returns false. The first production in a rule occurs immediately after the equals sign but remaining productions are prefixed with a `|` symbol.

```
pattern element_modifier(*){ element:element * },{
syntax({
  var run = element_inner ...
  function (data){
    var out = run(data)
    var last = data
    while (out){
      last = out
      out = run(out)
```

```
    }
    last
  }
})
}
{ element:element }, { syntax(element_inner ...) }
```

An element can either be standalone or used with a modifier. In this case only one modifier is implemented for zero or more repetitions. Repetition is achieved by executing the element function on the data until it returns false at which point the last nonfalse value is returned.

Elements are the basic matchers of the parser. Each element is a function that performs some string comparison with the input and returns the part of the string that is larger than the element. For example the literal string element simply compares itself to the beginning of the input string and if they are equal then it returns a substring of the input starting from the length of the literal string. The eof element expects the input string to be empty while the void element matches nothing. A rule element is simply a variable reference to the function of the same name.

```
pattern element(eof void)
{ eof }, { syntax(function (data){ data == "" }) }
{ void }, { syntax(function (data){ data }) }
{ rule:identifier }, { syntax(rule) }
```

The remaining elements are elided.

The parser can be bound to a variable as normal and invoked on a string. Adding in AST constructors is more effort but does not fundamentally break the model.

```
var parser = peg start { ... }
parser("18 + 2 * 9 - 3")
```

CHAPTER 5

EXTENSION FOR OTHER LANGUAGES

The parsing methodology outlined in this dissertation works for other languages. The key is to incorporate a parsing strategy specific to the language in the enforest function and make modest modifications to *read* and *expand* for the binding forms.

I demonstrate how to modify the *read*, *expand*, and *enforest* functions for both Java and Python. These languages are not built as extensions on top of Honu, but rather they are complete systems themselves. The compilers for these languages would normally be implemented in the languages themselves but as prototypes they are implemented in Racket.

These prototypes only demonstrate how to parse the languages, they do not add meta-levels or hygiene.

5.1 Extensible Java

My prototype macro[1] system for Java is similar to Honu in that the *read* function is mostly the same and allows macros to be used at the statement level and in expressions. Unlike Honu the system for Java has three *enforest* functions rather than just one. The different *enforest* functions apply to different categories of productions from the Java grammar. Top level declarations are handled by the *enforest-top* function, class bodies are handled with *enforest-class*, and statements and expressions with *enforest-expression*. Statements and expressions are handled by the same *enforest* function to simplify the design and implementation of the prototype in that macros can be used as either statements or expressions instead of having two types of macros. All of the various *enforest* functions output some part of the Java AST, however, which is dealt with by a single *expand* function.

My prototype handles the following simplified Java BNF:

```
⟨top⟩          ::= package ⟨identifier⟩*
                 |  import ⟨identifier⟩*
                 |  public ⟨class⟩
⟨class⟩        ::= class ⟨identifier⟩ { ⟨class-body⟩ }
⟨class-body⟩   ::= ⟨visibility⟩ ⟨variable⟩
```

[1] The prototype can be found at http://github.com/kazzmir/java-macro

```
                |  ⟨visibility⟩ ⟨class⟩
                |  ⟨visibility⟩ [abstract] ⟨method⟩
⟨visibility⟩    ::= protected  |  private  |  public
⟨method⟩        ::= ⟨type⟩ ⟨identifier⟩ ( ⟨identifier⟩* ) { ⟨statement⟩* }
⟨type⟩          ::= ⟨identifier⟩ [[]]
⟨statement⟩     ::= if ( ⟨expression⟩ ) { ⟨statement⟩* }
                |  for ( ⟨type⟩ ⟨identifier⟩ : ⟨expression⟩ ) { ⟨statement⟩* }
                |  { ⟨statement⟩* }
                |  while ( ⟨expression⟩ ) ⟨statement⟩
                |  macro ⟨identifier⟩ ( ⟨identifier⟩* ) { ⟨term⟩* } { ⟨term⟩* }
                |  return ⟨expression⟩
                |  ⟨expression⟩
⟨expression⟩    ::= new ⟨type⟩ ( ⟨expression⟩* ) ⟨inline-class⟩
                |  ⟨expression⟩ ? ⟨expression⟩ : ⟨expression⟩
                |  ( ⟨type⟩ ) ⟨expression⟩
                |  ⟨expression⟩ . ⟨identifier⟩
                |  ⟨expression⟩ ⟨operator⟩ ⟨expression⟩
                |  ⟨expression⟩ ( ⟨expression⟩* )
⟨inline-class⟩  ::= { ⟨class-body⟩* }
```

The `top` production is handled by *enforest-top*, *class-body* by *enforest-class* and the rest of the productions by *enforest-expression*.

The undefined ⟨*term*⟩ nonterminal is a plain token produced by the *read* function.

The *expand* function processes a tree consisting of the following forms.

```
⟨form⟩ ::= java-unparsed-top ⟨form⟩*
       |  java-unparsed-class ⟨form⟩*
       |  java-unparsed-method ⟨form⟩*
       |  package ⟨identifier⟩*
       |  import ⟨identifier⟩*
       |  class ⟨identifier⟩ ⟨form⟩*
       |  var ⟨identifier⟩ ⟨type⟩ ⟨form⟩*
       |  constructor ⟨identifier⟩ ⟨form⟩*
       |  method ⟨identifier⟩ ⟨type⟩ ⟨form⟩*
       |  abstract-method ⟨identifier⟩ ⟨type⟩
       |  op ⟨identifier⟩ ⟨form⟩ ⟨form⟩
       |  unary-op ⟨identifier⟩ ⟨form⟩
       |  postfix-op ⟨identifier⟩ ⟨form⟩
       |  new ⟨type⟩ ⟨form⟩*
       |  new-class ⟨type⟩ ⟨form⟩* ⟨form⟩*
       |  macro ⟨identifier⟩ ⟨identifier⟩
```

The *expand* function accepts a form and an environment. The environment manages bindings by mapping symbols to descriptions of what they were bound to. There are three types of bindings: macros, variables, and types.

```
⟨binding⟩ ::= macro ⟨identifier⟩
          |  lexical
          |  type
```

The main purpose of *expand* is to update the environment based on the current node type. For example, when the `macro` node adds a binding for the name of the identifier whose value is a macro identifier that can be looked up by the *enforest* function to invoke a Java level macro. The `var` and `method` forms add new lexical bindings to the environment while the `class` form adds new types. The rules for all the nodes are given in Figure 5.1.

java-expand⟦(java-unparsed-top *node ...*), *environment*⟧
= ((enforest-top *node ...*) *environment*)
java-expand⟦(java-unparsed-class *node ...*), *environment*⟧
= ((enforest-class *node ...*) *environment*)
java-expand⟦(java-unparsed-method *node ...*), *environment*⟧
= ((enforest-expression *node ...*) *environment*)
java-expand⟦(package *identifier ...*), *environment*⟧
= ((package *identifier ...*) *environment*)
java-expand⟦(import *identifier ...* $identifier_{last}$), *environment*⟧
= ((import *identifier ...*) (add-type $identifier_{last}$ *environment*))
java-expand⟦(class *identifier node ...*), *environment*⟧
= ((class name java-expand⟦(*node ...*), *environment*⟧)
(add-type *identifier environment*))
java-expand⟦(var *identifier type node ...*), *environment*⟧
= ((var *identifier type* java-expand⟦(*node ...*), *environment*⟧)
(add-lexical *identifier environment*))
java-expand⟦(constructor *identifier node ...*), *environment*⟧
= ((constructor *identifier* java-expand⟦(*node ...*), *environment*⟧)
(add-lexical *identifier* $environment_{copy}$))
java-expand⟦(method *identifier type node ...*), *environment*⟧
= ((method *identifier type* java-expand⟦(*node ...*), *environment*⟧)
(add-lexical *identifier environment*))
java-expand⟦(abstract-method *identifier type*), *environment*⟧
= ((abstract-method *identifier type*)
(add-lexical *identifier environment*))
java-expand⟦(macro $identifier_{name}$ $identifier_{macro}$), *environment*⟧
= (() (add-macro $identifier_{name}$ $identifier_{macro}$ *environment*))

Figure 5.1: Java Expand

The tree output from the *read* function is injected into the AST by wrapping it with *java-unparsed-top*. This wrapped tree is then passed to *expand*. To illustrate I will show how the following code is parsed.

```
(java-unparsed-top
  public class Book{
    public int pages = 10 * 5;
    public int getPages(){
      return pages;
    }
  })
```

The *expand* function will invoke *enforest-top* which will parse a class according to the ⟨*class*⟩ production in the BNF. The result will be a class node in the AST where the body is wrapped with *java-unparsed-class*.

```
(class Book
  (java-unparsed-class
    public int pages = 10 * 5;
    public int getPages(){
      return pages;
    }))
```

Expand will process this node, and add a new binding to the environment between the identifier `Book` and `'type`. Then it will process the body, which consists of a *java-unparsed-class* node.

Member variable declarations and methods are then parsed into `var` and `method` nodes respectively. The body of the method is wrapped with a *java-unparsed-method* node to delay parsing of the inside so *expand* can register member variables and any function arguments in the environment.

```
(class Book
  (var int pages (op * 5 10))
  (method int getPages () (java-unparsed-method return pages;)))
```

Finally, *expand* will process the body of the `getPages` method resulting in the final AST:

```
(class Book
  (var int pages (op * 5 10))
  (method int getPages () (return pages)))
```

This resulting AST can be converted back to Java so it can be processed by the normal javac compiler or a new compiler can be written to directly parse this AST.

The *enforest-expression* function is the only *enforest* function that deals with macros. The other *enforest* functions could support macros as well but were left out to simplify the prototype. Java macros are invoked when the *enforest-expression* function finds a symbol that is mapped to a `macro` binding object. In the case that it is, the macro function is pulled out of the environment and invoked on the current input stream of terms. The output of the macro is repeatedly passed to *enforest-expression* until its syntax does not contain any new macros.

The environment also helps to parse Java expressions that contain types, such as casts and variable declarations. A variable declaration looks like

```
Type identifier;
```

Where the only difference between a type and an identifier is whether or not the type is bound to `'type` in the environment. Similarly if there is a type inside parenthesis followed by an expression then it is a cast. Other than cast expressions there are no other times where a type could be confused for an expression.

The Java grammar has many more operators, modifiers, and forms but these do not present a challenge to the parser. All binary, prefix unary, and postfix unary operators can be handled by the precedence parser. Modifiers for method and variable declarations are recognized directly by the parser so a modifier will not be confused with an expression or type. Other forms, such as the synchronized form, start with a keyword, and can therefore be handled directly by the parser or as a built-in macro form.

While I have opted not to allow the set of operators to be extended it would be straightforward to add.

5.2 Java Examples

The Java Syntax Extender [3] adds a new `foreach` statement with their macro system that works similarly to for loops in Java 5. The definition of `foreach` in my macro system is similar.

```
public void test(){
  macro foreach(in){ (T:type i:id in e:expression) body }{
    syntax(Iterator i = e.iterator();
           while (i.hasNext()){
             T element = (T) i.next();
```

```
            body
          })
  }

  foreach (File f in files.get()){
    println(f);
  }
}
```

Note that the pattern language supports the syntactic class `type` which, handles the complexities of generic types and arrays.

I can also add first class functions support to Java using a `lambda` keyword. Inline declarations of anonymous classes can be used to create closures in Java. I define an abstract `Lambda` class with one method for the interface to using the object. The macro instantiates the class and defines the method with the code given to the macro as an argument.

```
public class Test{
  public class Lambda{
    public abstract Object invoke(Object arg);
  }

  public void test(){
    macro lambda(){ (x:id) body }{
     syntax(new Lambda(){
             public Object invoke(Object x){
               body
             }
           })
   }

   Lambda l = lambda(x){ println(x); }
   l.invoke(12);
  }
}
```

If Java supported operating overloading, then the disguise as a normal function could be complete. Keyword macros cannot make invoking the procedure any more natural looking.

5.3 Extensible Python

My prototype implementation [2] for Python 2 can parse the entire language. The *enforest* and *expand* functions for Python work similar to those in Java, but the *read* function is adjusted to accomodate Python's whitespace rules. The amount of whitespace that precedes a line determines which block the line is a part of. For example, in the following code the indented print statements under the for loop make up the for loop's body.

```
for name in names:
    print "Name %s" % name
    print "Address %s" % address_of(name)
```

The *read* function converts this into the following tree

```
(for name in names :
  (%block
    (print "Name %s" % name)
    (print "Address %s" % address_of(%parens name))))
```

Enclosing tokens are treated in a similar way to Java and Honu's *read* function where a new token, in this case `%parens`, is inserted in from of tokens surrounded by a set of parenthesis.

The AST that expand processes corresponds directly with the grammar. As usual I add an additional form for unparsed Python code and a form for introducing macros.

⟨*form*⟩ ::= `assign` ⟨$form_{left}$⟩ ⟨$form_{right}$⟩
| `assert` ⟨*form*⟩*
| `binary-op` ⟨id_{op}⟩ ⟨$form_{left}$⟩ ⟨$form_{right}$⟩
| `unary-op` ⟨id_{op}⟩ ⟨*form*⟩
| `del` ⟨*form*⟩*
| `generator (` ⟨$id_{iterator}$⟩* `)` ⟨$form_{list}$⟩ ⟨$form_{result}$⟩
| `generator-if (` ⟨$id_{iterator}$⟩* `)` ⟨$form_{list}$⟩ ⟨$form_{result}$⟩ ⟨$form_{condition}$⟩
| `class` ⟨id_{name}⟩ ⟨id_{super}⟩ ⟨$form_{body}$⟩
| `raise` ⟨*form*⟩*
| `tuple` ⟨*form*⟩*
| `import-from` ⟨$id_{library}$⟩ ⟨$id_{imports}$⟩*
| `list-ref` ⟨$form_{list}$⟩ ⟨$form_{index}$⟩
| `list-splice` ⟨$form_{bottom}$⟩ [⟨$form_{top}$⟩]
| `global` ⟨*id*⟩
| `try` ⟨$form_{body}$⟩ ⟨$form_{excepts}$⟩*
| `except` ⟨id_{type}⟩ ⟨$id_{binding}$⟩ ⟨$form_{body}$⟩

[2] The implementation can be found at http://github.com/kazzmir/python-macros

```
|  for ⟨id_iterator⟩ ⟨form_list⟩ ⟨form_body⟩
|  if ⟨form_condition⟩ ⟨form_body⟩ ⟨form_elses⟩*
|  elif ⟨form_condition⟩ ⟨form_body⟩
|  else ⟨form_body⟩
|  call ⟨form_function⟩ ⟨form_args⟩*
|  dot ⟨form_left⟩ ⟨form_right⟩
|  import ⟨id⟩*
|  return ⟨form⟩
|  lambda ( ⟨id⟩* ) ⟨form⟩
|  def ( ⟨id⟩ ⟨id⟩* ) ⟨form⟩
|  macro ⟨id_name⟩ ( ⟨term_pattern⟩* ) ⟨form_body⟩
|  unparsed ⟨term_python⟩*
```

Python is dynamically typed so the only kinds of bindings needed are those for plain lexical bindings and macros. Environments map identifiers to these bindings.

```
⟨binding⟩ ::= macro ⟨identifier⟩
          |  lexical
```

Expand updates the bindings in the environment, and calls *enforest* when needed. The expand rules for the Python AST are given in Figure 5.2.

The `unparsed` form is the entry point to the *enforest* function. Python has two categories of syntax, statements and expressions, both of which are handled in the *enforest* function. I again use a precedence-style parser.

Parsing the Python grammar is mostly straightforward except for handling tuples. The parser immediately returns the current parsed expression upon seeing a comma token, but expressions can be separated by commas to form a tuple. This is to support argument lists in `def` and `lambda` forms as well has separating key value pairs in hashes. Instead of making comma a binary operator, I add special cases to expand when it sees a `unparsed` form. Specifically, `unparsed` will call *enforest* on a single expression and check if there is a comma in the unparsed stream. If a comma is present, then it must be a multiple variable assignment, so the parser knows to keep parsing and check for an `assign` form at which point it assembles all the individual left-hand expressions into a single `assign`.

The entire *enforest* function is roughly 590 lines of Racket code including helper functions. The *enforest* function accepts the stream of tokens and the current environment and performs pattern matching on the stream according to the grammar of Python.

```
(define (enforest input environment)
  (match input
    ...))
```

The following rule is a `racket/match` expression that matches a class statement with no super class.

```
python-expand⟦(unparsed node ...), environment⟧
 = (enforest(node, ...) environment)
python-expand⟦(import identifier ...), environment⟧
 = ((import identifier ...) environment)
python-expand⟦(from identifier_lib import identifier ...), environment⟧
 = ((from identifier_lib import identifier ...)
    environment)
python-expand⟦(assign node_left node_right), environment⟧
 = ((assign python-expand⟦node_left, environment⟧
            python-expand⟦node_right, environment⟧)
    (add-lexical environment node_left))
python-expand⟦(assert node), environment⟧
 = ((assert python-expand⟦node⟧) environment)
python-expand⟦(binary-op identifier node_left node_right), environment⟧
 = ((binary-op identifier
              python-expand⟦node_left, environment⟧
              python-expand⟦node_right, environment⟧)
    environment)
python-expand⟦(unary-op identifier node), environment⟧
 = ((unary-op identifier python-expand⟦node⟧)
    environment)
python-expand⟦(del node ...), environment⟧
 = ((del python-expand⟦node, environment⟧ ...)
    environment)
python-expand⟦(generator (identifier ...) node_list node_result), environment⟧
 = ((generator (identifier ...)
              python-expand⟦node_list, environment⟧
              python-expand⟦node-result, (add-lexical environment identifier ...)⟧)
    environment)
python-expand⟦(generator-if (identifier ...) node_list node_result node_condition), environment⟧
 = ((generator-if (identifier ...)
                python-expand⟦node_list, environment⟧
                python-expand⟦node_result, environment_generator⟧
                python-expand⟦node_condition, environment_generator⟧)
    environment_generator)
 where environment_generator = (term (add-lexical environment identifier ...))
```

Figure 5.2: Python Expand

python-expand⟦(class $identifier_{name}$ $identifier_{super}$ $node$), $environment$⟧
= ((class $identifier_{name}$ $identifier_{super}$
python-expand⟦$node$, $environment$⟧)
$environment$)
python-expand⟦(raise $node$ ...), $environment$⟧
= ((raise python-expand⟦$node$, $environment$⟧ ...)
$environment$)
python-expand⟦($node$, ...), $environment$⟧
= ((python-expand⟦$node$, $environment$⟧, ...)
$environment$)
python-expand⟦(import-from $identifier_{library}$ $identifier$ ...), $environment$⟧
= ((import-from $identifier_{library}$ $identifier$ ...)
$environment$)
python-expand⟦(list-ref $node_{list}$ $node_{index}$), $environment$⟧
= ((list-ref python-expand⟦$node_{list}$, $environment$⟧
python-expand⟦$node_{index}$, $environment$⟧)
$environment$)
python-expand⟦(list-splice $node_{bottom}$ $node_{top}$), $environment$⟧
= ((list-splice python-expand⟦$node_{bottom}$, $environment$⟧
python-expand⟦$node_{top}$, $environment$⟧)
$environment$)
python-expand⟦(global $identifier$), $environment$⟧
= ((global $identifier$)
(add-lexical $environment$ $identifier$))
python-expand⟦(try $node_{body}$ $node_{excepts}$ ...), $environment$⟧
= ((try python-expand⟦$node_{body}$, $environment$⟧
python-expand⟦$node_{excepts}$, $environment$⟧ ...)
$environment$)
python-expand⟦(except $identifier_{type}$ $identifier_{binding}$ $node_{body}$), $environment$⟧
= ((except $identifier_{type}$ $identifier_{binding}$
python-expand⟦$node_{body}$, (add-lexical $environment$ $identifier_{binding}$)⟧)
$environment$)
python-expand⟦(for $identifier_{iterator}$ $node_{list}$ $node_{body}$), $environment$⟧
= ((for $identifier_{iterator}$
python-expand⟦$node_{list}$, $environment$⟧
python-expand⟦$node_{body}$, (add-lexical $environment$ $identifier_{iterator}$)⟧)
$environment$)

Figure 5.2: Continued

python-expand⟦(if $node_{condition}$ $node_{body}$ $node_{elses}$ ...), *environment*⟧
= ((if python-expand⟦$node_{condition}$, *environment*⟧
python-expand⟦$node_{body}$, *environment*⟧
python-expand⟦$node_{elses}$, *environment*⟧ ...)
environment)
python-expand⟦(elif $node_{condition}$ $node_{body}$), *environment*⟧
= ((elif python-expand⟦$node_{condition}$, *environment*⟧
python-expand⟦$node_{body}$, *environment*⟧)
environment)
python-expand⟦(else $node_{body}$), *environment*⟧
= ((else python-expand⟦$node_{body}$, *environment*⟧)
environment)
python-expand⟦(call $node_{function}$ $node_{args}$ ...), *environment*⟧
= ((call python-expand⟦$node_{function}$, *environment*⟧
python-expand⟦$node_{args}$, *environment*⟧ ...)
environment)
python-expand⟦(dot $node_{left}$ $node_{right}$), *environment*⟧
= ((dot python-expand⟦node-left, *environment*⟧
python-expand⟦$node_{right}$, *environment*⟧)
environment)
python-expand⟦(import *identifier*), *environment*⟧
= ((import *identifier*) *environment*)
python-expand⟦(return *node*), *environment*⟧
= ((return python-expand⟦*node*, *environment*⟧) *environment*)
python-expand⟦(lambda (*identifier* ...) $node_{body}$), *environment*⟧
= ((lambda (*identifier* ...)
python-expand⟦$node_{body}$, (add-lexical *environment identifier* ...)⟧)
environment)
python-expand⟦(def ($identifier_{name}$ *identifier* ...) $node_{body}$), *environment*⟧
= ((def ($identifier_{name}$ *identifier* ...)
python-expand⟦$node_{body}$, (add-lexical *environment identifier* ...)⟧)
(add-lexical *environment* $identifier_{name}$))
python-expand⟦(macro $identifier_{name}$ (*token* ...) $node_{body}$), *environment*⟧
= (()
(add-macro *environment* $identifier_{name}$ (make-macro (*token* ...) $node_{body}$)))

Figure 5.2: Continued

```
[(list 'class (and name (? symbol?)) '%colon
               (list '%block body ...)
       rest ...)
  (define out (parsed `(class ,name () (unparsed ,@body))))
  (values out rest)]
```

This rule checks that the statement starts with `class`, then some identifier, then a colon, then a body wrapped inside a `%block` identifier. The tokens after the class form are stored in the `rest` variable and returned from the *enforest* function. The result of this rule is a class node according to the expand AST where the body is wrapped with a `unparsed` node so it can be left as-is. Rules for other Python productions act similarly.

The macro rule checks for an identifier bound as a Python macro, retrieves the transformer, and invokes it on the remaining part of the stream.

```
[(list (and macro (? (lambda (i)
                       (is-python-macro? i environment))))
               rest ...)
 (define transformer (python-macro-transformer
                       (environment-value environment macro)))
 (define-values (output unparsed)
               (transformer rest environment))
 (values (parse-all output environment) unparsed)]
```

The output of a macro is again reparsed until it becomes an AST node that expand can handle.

5.4 Python Examples

Python is highly flexible and can create useful abstractions through decorators and reflection but can still gain from a macro system. One simple example is adding a C-style switch statement which takes the places of repeated if-else blocks.

```
macro switch(case)( check:expr case condition:expr : body:expr ):
  return syntax(if check == condition:
                  body)
else (check:expr case condition:expr : body:expr case rest ...):
  return syntax(if check == condition:
```

```
                    body
                else:
                    switch check
                        case rest ...)

switch error:
  case FileNotFound: print "File not found",
  case OutOfSpace: print "Out of space"
```

CHAPTER 6

RELATED WORK

Macro systems have been researched heavily in the Lisp family of languages [6, 40]. Originally, Lisp supported syntactic extension through fexprs, which are functions whose arguments are unevaluated. The arguments to the fexpr correspond exactly to the concrete syntax because Lisp is a homoiconic language. Lisp is able to differentiate between fexprs and regular functions by tagging variables depending on how they are bound. The fexpr is executed as a regular function whose output is re-evaluated and can call eval on its arguments if needed.

Later, a macro system was added to Lisp with defmacro. Symbols bound to macro functions with defmacro were stored in a lexical environment. The Lisp compiler used an expansion step to process macros by examining the program top-down looking for s-expressions whose starting symbol was bound with defmacro and invoking their associated macro function on the s-expression. Naive macro expansion does not maintain lexical properties of the program properly. Kohlbecker [24] developed a hygienic macro expansion algorithm to make it more robust although it is quadratic in the number of expressions that it processes.

A linear macro expansion algorithm was first developed for Scheme with a restricted high level pattern based macro system by Clinger [11] and later improved to support procedural macros by Dybvig [15]. Procedural macros are conceptually invoked by the compiler and should not allow compile-time values to leak into run-time code. Flatt [18] devised a system that makes values for the different run-time phases of the program explicit. Honu builds directly on the advances made in this lineage in the Lisp world.

Enforestation is a technique that is used to build a macro system, one kind of syntactic extension mechanism. Other kinds of syntactic extension consist of modifying grammars and other low level aspects of parsers. Honu macro definitions integrate with the parser without having to specify grammar-related details. Related systems, such as SugarJ [16], Xoc [12], MetaLua [20] and Polyglot [26] require the user to specify which grammar productions to extend, which can be an additional burden for the programmer. Xoc and SugarJ use a GLR [37] parser that enables them to extend the the class of tokens, which allows a natural embedding of domain-specific languages.

MetaLua allows users to modify the lexer and parser but forces macro patterns to specify the syntactic class of all its pattern variables which prevents macros from binding use-site identifiers in expressions passed to the macro. Ometa [41] and Xtc [21] are similar in that they allow the user to extend how the raw characters are consumed, but they do not provide a macro system. Honu does not contain a mechanism for extending its lexical analysis of the raw input stream because Honu implicitly relies on guarantees from the reader about the structure of the program to perform macro expansion.

C++ templates are most successful language-extension mechanism outside of the Lisp tradition. Like Honu macros, C++ templates allow only constrained extensions of the language, since template invocations have a particular syntactic shape. Honu macros are more flexible than C++ templates, allowing extensions to the language that have the same look as built-in forms. In addition, because Honu macros can be written in Honu instead of using only pattern-matching constructs, complex extensions are easier to write and can give better syntax-error messages than in C++'s template language. C++'s support for operator overloading allows an indirect implementation of infix syntactic forms, but Honu allows more flexibility for infix operators, and Honu does not require an a priori distinction between operator names and other identifiers.

Converge [38] has metaprogramming facilities similar to C++ templates but allows for syntax values to flow between the compile time and runtime. Metaprograms in Converge are wrapped in special delimiters that notify the parser to evaluate the code inside the delimiters and use the resulting syntax object as the replacement for the metaprogram. Converge cannot create new syntactic forms using this facility, however. Composability of metaprograms is achieved using standard function composition.

Composability of macros is tightly correlated with the parsing strategy. Honu macros are highly composable because they are limited to forms that start with an identifier bound to a macro or be in operator position. Other systems that try to allow more general forms expose underlying parsing details when adding extensions. Systems based on LL, such as the one devised by Cardelli and Matthes [10], and LALR, such as Maya [4], have fundamental limits to the combinations of grammar extensions. PEG based systems are closed under union but force an ordering of productions which may be difficult to reason about.

Some macro systems resort to AST constructors for macro expansions instead of templates based on concrete syntax. Maya fits the AST-constructor category. Template Haskell [22], SugarJ, and the Java Syntax Extender [3] include support for working with concrete syntax, but they also expose a set of abstract syntax tree constructors for more complex transformations. Camlp4 [14]

is a preprocessor for Ocaml programs that can output concrete Ocaml syntax, but it cannot output syntax understood by a separate preprocessor, so syntax extensions are limited to a single level. MS2 [42] incorporates Lisp's quasiquote mechanism as a templating system for C, but MS2 does not include facilities to expand syntax that correspond to infix syntax or any other complex scheme.

Honu macros have the full power of the language to implement a macro transformation. Systems that only allow term rewriting, such as R5RS Scheme [23], Dylan [32], Arai and Wakita [1] and Fortress [29], can express many simple macros, but they are cumbersome to use for complex transformations.

ZL [2] is like Honu in that it relies on Lisp-like read and parsing phases, it generalizes those to non-parenthesiz-ed syntax, and its macros are expressed with arbitrary ZL code. Compared to Honu, macros in ZL are more limited in the forms they can accept, due to decisions made early on in the read phase. Specifically, arbitrary expressions cannot appears as subforms unless they are first parenthesized. ZL supports more flexible extensions by allowing additions to its initial parsing phase, which is similar to reader extension in Lisp or parsing extensions in SugarJ, while Honu allows more flexibility within the macro level.

Stratego [39] supports macro-like implementations of languages as separate from the problem of parsing. Systems built with Stratego can use SDF for parsing, and then separate Stratego transformations process the resulting AST. Transformations in Stratego are written in a language specific to the Stratego system and different from the source language being transformed, unlike Honu or other macro languages. Metaborg [7] and SugarJ use Stratego and SDF to add syntactic extensions using the concrete syntax of the host language. Metaborg is used purely in a preprocessing step, while SugarJ is integrated into the language in the same way as Honu.

Many systems implement some form of modularity for syntactic extension. Both SDF and Xoc [12] provide a way to compose modules which define grammar extensions. These systems have their own set of semantics that are different from the source language being extended. Honu uses its natural lexical semantics to control the scope of macros. Macros can be imported into modules and shadowed at any time thus macros do not impose a fundamental change into reasoning about a program.

Nemerle [33] provides many of the same features as Honu but requires macros to be put in a module separate from where the macro is used, because macros must be completely compiled before they can be used. Nemerle is thus unable to support locally-scoped macros, and it cannot bind identifiers from macro invocations to internal macros.

Multistage allows programs to generate and optimize code at run-time for specific sets of data. Mython [30], MetaOcaml [9], LMS [31] are frameworks that provide methods to optimize expressions by analyzing a representation of the source code. A similar technique can be achieved in Honu by wrapping expressions with a macro that analyzes its arguments and plays the role of a compiler by rewriting the expression to a semantically equivalent expression. Typed Racket [36] implements compile-time optimizations using the Racket macro system.

CHAPTER 7

CONCLUSION

I have presented the design and some use-cases of a syntactically extensible language with support for infix notation and implicitly delimited syntax. This system is demonstrably powerful and does not burden the user with parsing details.

Implementing this system requires using a nontraditional parsing technology which existing languages might be hesitant to attempt, but I have shown that it is possible to do for Java and Python. The benefits of syntactic extension to users are well worth the effort for language implementers who wish to grow their language.

7.1 Contributions

This dissertation extends the Scheme expansion model to support implicitily delimited and infix syntaxes with procedural, hygienic, and composable macros. This new model is then used as a basis to implement the Honu language, which contains realistic language features. Finally, I have shown how the model can be used to support other languages of a similar nature.

7.2 Lessons

One natural design to achieve syntactic extension is to allow the grammar to be modified by an input program. Arbitrary grammar extension is an attractive ideal, because it allows new syntactic constructs to be very liberal in their makeup, and to be mixed freely with the rest of the language. The main issue with modifying the grammar is composing extensions. The base grammar for a language is often meticulously groomed to not have any ambiguities in it. Adding new productions or changing existing ones can be very difficult if not outright impossible.

Operator precedence parsing is convenient for two reasons: first it is designed to handle languages with infix operators and secondly the underlying algorithm is simple enough to modify. PEG is another reasonable choice but due to backtracking a macro could be invoked but then a different parsing path chosen that would not have invoked the macro, thus forcing any side-effects of the macro to have to be undone. LR requires knowing in advance which rule the next token applies to, which does not mesh well with the arbitrary parsing that a macro can do.

New operators can be added to the operator precedence parser by attaching a property to an identifier and using a predicate in the parser to test for this property. Other parsing technology can be made to use predicates to test if an identifier is an operator but the precedence levels in PEG and LR is implemented by ordering the rules the operators appear in. Adding new operators necessarily consists of modifying the rules which can be difficult to do.

7.3 Future Work

Macros in Honu must start with a keyword, and thus are not as general as possible. There should be a way to augment *enforest* in such a way that it can scan ahead for a macro keyword while not breaking any hygienic concerns about syntax that comes before the keyword. Operators in Honu are already capable of manipulating syntax that appears before it, but the syntax before it must be parsed as an expression instead of subject to the operator's will.

The integration of the environment and parsing is the source of Honu's ability to interleave macro expansion with parsing. There are clearly similarities between the environment Honu keeps track of and the standard symbol table that a C parser uses. There is thus a potential to apply a Honu-style macro system to C which can simultaneously deal with C's ambiguous expressions.

Macro transformations in Honu are currently untyped. A macro can return any form, which has some associated type, in any context. A more robust system would prevent a macro from being used in a context where the resulting syntax has a type that does not fit. A simplistic solution would be to tag macros with the type of their expression, but that may not always be feasible. A more complex solution involves inspecting the template that the macro uses and inferring a type or somehow lazily computing one. Other macros systems have explored this area as well.

APPENDIX

This section contains an implementation of *enforest* shown in Figure 3.1 written in the Redex language. This code was successfully tested in Racket 5.3.3.

```
#lang racket

(require redex)
(provide (all-defined-out))

(define-language Honu
  [binary-operator + * ^ -]
  [unary-operator ! !!]
  [assoc left right]
  [atom integer]
  [prec integer]
  [combine identity
           [mkbin: binary-operator term]
           [mkun: unary-operator combine]]
  [stack empty [push combine prec stack]]
  [tree-term (literal: integer)
             (id: variable-not-otherwise-mentioned)
             (call: tree-term tree-term ...)
             (list: tree-term ...)
             (block: tree-term ...)
             (arrayref: tree-term tree-term)
             (un: unary-operator tree-term)
             (bin: binary-operator tree-term tree-term)]
  [term atom
        identifier
        (parens term ...)
```

```
        [squares term ...]
        {curlies term ...}
        tree-term]
  [identifier binary-operator
              unary-operator
              variable-not-otherwise-mentioned]

  [sequence (seq term ...)]
  [binding (macro: transformer)
           (binop: prec assoc)
           ;; precedence level and postfix boolean
           (unop: prec fixity)
           (var: identifier)]
  [fixity postfix prefix]
  [transformer a-transformer]

  [boolean #t #f])

;; A combine will either be the identity function or a binary transformer
;; A binary transformer comes with the left hand argument as part
;; of its definition. The right hand argument comes from the argument
;; to do-combine. 'do-combine' will create a tree-term with the binary
;; operator, left, and right-hand arguments. Then it will run the old
;; combine that was part of the binary transformer's definition
;; on the result.
(define-metafunction Honu
  do-combine : combine term -> term
  [(do-combine identity term) term]
  [(do-combine (mkbin: binary-operator term_left) term_right)
   (bin: binary-operator term_left term_right)]
  [(do-combine (mkun: unary-operator combine) term)
   (do-combine combine (un: unary-operator term))])
```

```
(define-metafunction Honu
  expand : transformer (seq term ...) -> (seq term ...)
  [(expand a-transformer (seq term ...)) (seq term ...)])

;; If the found operator's prec is greater than the current
;; prec, then a new do-combine function takes a right and
;; applies the operator's transformer to init and right; the result
;; is then passed to the original do-combine. Meanwhile, prec
;; becomes the found operator's prec.

;; If the found operator's prec is less or equal to the current
;; prec, then do-combine is applied to init to produce left; the
;; new do-combine function takes right and applies the operator's
;; binary transformer to left and right, while prec remains
;; unchanged.

(define-metafunction Honu
  higher : assoc prec prec -> boolean
  [(higher left prec_1 prec_2) #t
   (side-condition (> (term prec_1) (term prec_2)))]
  [(higher right prec_1 prec_2) #t
   (side-condition (>= (term prec_1) (term prec_2)))]
  [(higher assoc prec_1 prec_2) #f])

(define-metafunction Honu
  lower : assoc prec prec -> boolean
  [(lower left prec_1 prec_2) #f
   (side-condition (> (term prec_1) (term prec_2)))]
  [(lower right prec_1 prec_2) #f
   (side-condition (>= (term prec_1) (term prec_2)))]
  [(lower assoc prec_1 prec_2) #t])

(define-metafunction Honu
```

```
  lookup : identifier -> binding
  [(lookup +) (binop: 1 left)]
  [(lookup -) (binop: 1 left)]
  [(lookup ^) (binop: 3 right)]
  [(lookup *) (binop: 2 left)]
  [(lookup !) (unop: 5 prefix)]
  [(lookup !!) (unop: 3 postfix)]
  [(lookup identifier) (var: identifier)])

(define-metafunction Honu
  enforest : (seq term ...) combine prec stack ->
             (tuple tree-term (seq term ...))

  [(enforest (seq atom term_rest ...) combine prec stack)
   (enforest (seq (literal: atom) term_rest ...) combine prec stack)]

  [(enforest (seq identifier term_rest ...) combine prec stack)
   (enforest (seq (id: identifier_binding) term_rest ...)
             combine prec stack)
   (where (var: identifier_binding) (lookup identifier))]

  [(enforest (seq identifier term_rest ...) combine prec
             [push combine_stack prec_stack stack])
   (enforest (expand transformer (seq term_rest ...))
             combine_stack prec_stack stack)
   (where (macro: transformer) (lookup identifier))]

  [(enforest (seq tree-term_first identifier term_rest ...)
             combine prec stack)
   (enforest (seq term_rest ...)
             (mkbin: identifier tree-term_first)
             prec_operator [push combine prec stack])
   (where (binop: prec_operator assoc) (lookup identifier))
```

```
 (side-condition (term (higher assoc prec_operator prec)))]

[(enforest (seq tree-term_first identifier term_rest ...)
           combine prec
           [push combine_stack prec_stack stack])
 (enforest (seq (do-combine combine tree-term_first)
                identifier term_rest ...)
           combine_stack prec_stack stack)
 (where (binop: prec_operator assoc) (lookup identifier))
 (side-condition (term (lower assoc prec_operator prec)))]

[(enforest (seq tree-term_first identifier term_rest ...)
           combine prec stack)
 (enforest (seq (do-combine (mkun: identifier identity)
                            tree-term_first)
                term_rest ...)
           combine prec stack)
 (where (unop: prec_operator postfix) (lookup identifier))
 (side-condition (term (higher left prec_operator prec)))]

[(enforest (seq identifier term_rest ...) combine prec stack)
 (enforest (seq term_rest ...)
           (mkun: identifier combine) prec_operator stack)
 (where (unop: prec_operator prefix) (lookup identifier))]

[(enforest (seq (parens term_inside ...) term_rest ...)
           combine prec stack)
 (enforest (seq tree-term_inside term_rest ...) combine prec stack)
 (where (tuple tree-term_inside (seq))
        (enforest (seq term_inside ...) identity 0 empty))]

[(enforest (seq tree-term (parens term_arg ...) term_rest ...)
           combine prec stack)
```

```
 (enforest (seq (call: tree-term tree-term_arg ...) term_rest ...)
           combine prec stack)
 (where (seq (tuple tree-term_arg (seq)) ...)
        (seq (enforest (seq term_arg) identity 0 empty) ...))]

[(enforest (seq tree-term [squares term ...] term_rest ...)
           combine prec stack)
 (enforest (seq (arrayref: tree-term tree-term_lookup ) term_rest ...)
           combine prec stack)
 (where (tuple tree-term_lookup (seq))
        (enforest (seq term ...) identity 0 empty))]

[(enforest (seq [squares term ...] term_rest ...) combine prec stack)
 (enforest (seq (list: term ...) term_rest ...) combine prec stack)]

[(enforest (seq {curlies} term_rest ...) combine prec stack)
 (enforest (seq (block:) term_rest ...) combine prec stack)]

[(enforest (seq {curlies term ...} term_rest ...) combine prec stack)
 (enforest (seq (block: tree-term tree-term_block ...) term_rest ...)
           combine prec stack)
 (where (tuple tree-term (seq term_unparsed ...))
        (enforest (seq term ...) identity 0 empty))
 (where (tuple (block: tree-term_block ...) (seq))
        (enforest (seq {curlies term_unparsed ...}) identity 0 empty))]

[(enforest (seq tree-term term_rest ...) combine prec empty)
 (tuple (do-combine combine tree-term) (seq term_rest ...))]

[(enforest (seq tree-term term_rest ...) combine prec
           [push combine_stack prec_stack stack])
 (enforest (seq (do-combine combine tree-term) term_rest ...)
           combine_stack prec_stack stack)])
```

```
(define (parse input)
  (term (enforest (seq ,@input) identity 0 empty)))

(module+ test
  ;; combiners
  (test-equal
   (term (do-combine identity 5))
   (term 5))

  (test-equal
   (term (do-combine (mkbin: + (literal: 3)) (literal: 5)))
   (term (bin: + (literal: 3) (literal: 5))))

  ;; basic
  (test-equal
   (parse (term (5)))
   (term (tuple (literal: 5) (seq))))

  ;; binary precedence
  (test-equal
   (parse (term (1 + 5)))
   (term (tuple (bin: + (literal: 1) (literal: 5)) (seq))))

  (test-equal
   (parse (term (1 + 5 + 8)))
   (term (tuple (bin: + (bin: + (literal: 1)
                                (literal: 5)) (literal: 8)) (seq))))

  (test-equal
   (parse (term (1 + 5 * 8)))
   (term (tuple (bin: + (literal: 1)
                      (bin: * (literal: 5) (literal: 8))) (seq)))))
```

```
(test-equal
 (parse (term (1 + 5 ^ 8 * 4)))
 (term (tuple (bin: + (literal: 1)
                      (bin: *
                            (bin: ^ (literal: 5) (literal: 8))
                            (literal: 4)))
              (seq))))

;; unary operators
(test-equal
  (parse (term (! 7)))
  (term (tuple (un: ! (literal: 7)) (seq))))

(test-equal
 (parse (term (1 + 5 + ! 7)))
 (term (tuple (bin: +
                      (bin: + (literal: 1) (literal: 5))
                      (un: ! (literal: 7)))
              (seq))))

(test-equal
  (parse (term (8 !!)))
  (term (tuple (un: !! (literal: 8))
               (seq))))

(test-equal
  (parse (term (2 * 4 !!)))
  (term (tuple (bin: * (literal: 2)
                       (un: !! (literal: 4)))
               (seq))))

;; parens expressions
```

```
(test-equal
  (parse (term ((parens 5))))
  (term (tuple (literal: 5) (seq))))

(test-equal
  (parse (term ((parens 5) + 7)))
  (term (tuple (bin: + (literal: 5) (literal: 7)) (seq))))

(test-equal
  (parse (term ((parens 5 * 2) + 7)))
  (term (tuple (bin: +
                     (bin: * (literal: 5) (literal: 2))
                     (literal: 7))
               (seq))))

(test-equal
  (parse (term (1 * (parens 5 + 2) - 5)))
  (term (tuple (bin: -
                     (bin: *
                           (literal: 1)
                           (bin: + (literal: 5) (literal: 2)))
                     (literal: 5))
               (seq))))

;; function call
(test-equal
  (parse (term (f (parens 5))))
  (term (tuple (call: (id: f) (literal: 5))
               (seq))))

;; array lookup
(test-equal
  (parse (term (5 [squares 2 + 3]))))
```

```
  (term (tuple (arrayref: (literal: 5)
                          (bin: + (literal: 2) (literal: 3)))
               (seq))))

(test-equal
  (parse (term ({curlies 1 + 1})))
  (term (tuple (block: (bin: +
                             (literal: 1)
                             (literal: 1)))
               (seq))))

(test-equal
  (parse (term ({curlies 1 + 1
                         2 + 2})))
  (term (tuple (block:
                 (bin: + (literal: 1) (literal: 1))
                 (bin: + (literal: 2) (literal: 2)))
               (seq)))))
```

REFERENCES

[1] Hiroshi Arai and Ken Wakita. An implementation of a hygienic syntactic macro system for JavaScript: a preliminary report. In *Proc. Workshop on Self-Sustaining Systems*, 2010.

[2] Kevin Atkinson, Matthew Flatt, and Gary Lindstrom. ABI Compatibility Through a Customizable Language. In *Proc. Generative Programming and Component Engineering*, pp. 147–156, 2010.

[3] Jonathan Bachrach and Keith Playford. Java Syntax Extender. In *Proc. Object-Oriented, Programming, Systems, Languages, and Applications*, 2001.

[4] Jason Baker. Macros that Play: Migrating from Java to Myans. Masters dissertation, University of Utah, 2001.

[5] Eli Barzilay, Ryan Culpepper, and Matthew Flatt. Keeping it clean with syntax parameters. Workshop on Scheme and Functional Programming, 2011.

[6] Alan Bawden and Jonathan A. Rees. Syntactic Closures. In *Proc. ACM Conference on Lisp and Functional Programming*, 1988.

[7] Martin Bravenboor and Eelco Visser. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In *Proc. Object-Oriented, Programming, Systems, Languages, and Applications*, pp. 365–383, 2004.

[8] R. Kent Dybvig, Robert Hieb, Carl Bruggeman. Syntactic Abstraction in Scheme. Lisp and Symbolic Computation 5 pp 83–110, 1992.

[9] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Proc. Generative Programming and Component Engineering*, 2003.

[10] Luca Cardelli and Florian Matthes. Extensible syntax with lexical scoping. Technical report, Research Report 121, Digital SRC, 1994.

[11] William Clinger and Jonathan Rees. Macros that work. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

[12] Russ Cox, Tom Bergan, Austin T. Clements, Frans Kaashoek, and Eddie Kohler. Xoc, an extension-oriented compiler for systems programming. In *Proc. 13th Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[13] Ryan Culpepper and Matthias Felleisen. Fortifying Macros. In *Proc. ACM Intl. Conf. Functional Programming*, 2010.

[14] Daniel de Rauglaudre. Camlp4. 2007. `http://brion.inria.fr/gallium/index.php/Camlp4`

[15] R. Kent Dybvig. *Syntactic abstraction: the syntax-case expander*. Beautiful Code: Leading Programmers Explain How They Think. 407-428, 2007.

[16] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-Based Syntactic Language Extensibility. In *Proc. Object-Oriented, Programming, Systems, Languages, and Applications*, pp. 391–406, 2011.

[17] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 48–59, 2002.

[18] Matthew Flatt. Compilable and Composable Macros, You Want it *When?* In *Proc. ACM Intl. Conf. Functional Programming*, pp. 72–83, 2002.

[19] Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that Work Together: Compile-Time Bindings, Partial Expansion, and Definition Contexts. *Journal of Functional Programming* (to appear), 2012. `http://www.cs.utah.edu/plt/expmodel-6/`

[20] Fabien Fluetot and Laurence Tratt. Contrasting compile-time meta-programming in Metalua and Converge. Workshop on Dynamic Languages and Applications, 2007.

[21] Robert Grimm. Better extensibility through modular syntax. In *Proc. Programming Language Design and Implementation pp.38-51*, 2006.

[22] Simon Peyton Jones and Tim Sheard. Template metaprogramming for Haskell. In *Proc. Haskell Workshop, Pitssburgh, pp1-16*, 2002.

[23] Richard Kelsey, William Clinger, and Jonathan Rees (Ed.). R5RS. ACM SIGPLAN Notices, Vol. 33, No. 9. (1998), pp. 26-76., 1998.

[24] Eugene Kohlbecker. Syntactic Extensions in the Programming Language Lisp. PhD dissertation, Indiana University, 1986.

[25] J. McCarthy, R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, D. Park, and S. Russell. LISP I Programmer's Manual. Computation Center and Research Laboratory of Electronics, Massachusetts Institute of California, 1960.

[26] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proc. 12th International Conference on Compiler Construction. pp. 138-152*, 2003.

[27] Vaughan R. Pratt. Top down operator precedence. In *Proc. 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1973.

[28] Jon Rafkind and Matthew Flatt. Honu: Syntactic Extension for Algebraic Notation through Enforestation. In *Proc. Generative Programming and Component Engineering*, 2012.

[29] Ryan Culpepper, Sukyoung Ryu, Eric Allan, Janus Neilson, Jon Rafkind. Growing a Syntax. In *Proc. FOOL 2009*, 2009.

[30] Jonathan Riehl. Language embedding and optimization in mython. In *Proc. DLS 2009. pp.39-48*, 2009.

[31] Tiark Rompf and Martin Odersky. Lightweight Modular Staging: a Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proc. Generative Programming and Component Engineering*, pp. 127–136, 2010.

[32] Andrew Shalit. Dylan Reference Manual. 1998. `http://www.opendylan.org/books/drm/Title`

[33] Kamil Skalski, Michal Moskal, and Pawel Olszta. Meta-programming in Nemerle. In *Proc. Generative Programming and Component Engineering*, 2004.

[34] Michael Sperber (Ed.). *Revised^6 Report on the Algorithmic Language Scheme*. Cambridge University Press, 2011.

[35] Guy Steele. Common Lisp the Language, 2nd edition. Digital Press, 1994.

[36] Sam Tobin-Hochstadt and Matthias Felleisen. Design and Implementation of Typed Scheme. *Higher Order and Symbolic Computation*, 2010.

[37] Masaru Tomita. An efficient context-free parsing algorithm for natural languages. International Joint Conference on Artificial Intelligence. pp. 756–764., 1985.

[38] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. ACM Transactions on Programming Languages and Systems (TOPLAS), 30(6):1-40, 2008.

[39] Eelco Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9. In *Proc. Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, pp. 216–238, 2003.

[40] Mitchell Wand. The Theory of Fexprs is Trivial. 1998.

[41] Alessandro Warth and Ian Piumarta. Ometa: an Object-Oriented Language for Pattern Matching. In *Proc. Dynamic Languages Symposium*, 2007.

[42] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, 1993.