# Honu: A Syntactically Extensible Language

Jon Rafkind
University of Utah
rafkind@cs.utah.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu

## Abstract

Honu is a new language that fuses traditional algebraic notation (e.g., infix binary operators) with Scheme-style language extensibility. A key element of Honu's 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. We present the design of Honu, explain its parsing and macro-extension algorithm, and show example syntactic extensions.

## 1. Introduction

An extensible programming language accommodates additions to the language without requiring those additions to be adopted by a standardization committee, approved by a core set of implementors, or imposed on all users of the language. Whether for domain-specific languages or improved general-purpose constructs, extensible languages offer the promise of accelerating language design, leading to clearer and more correct programs by narrowing the gap between an idea and its expression as a program.

As appealing as the idea sounds, only the Lisp family of languages has so far made extensibility work well enough to be widely embraced by its users. The line of work on extensible syntax runs from early Lisp days, through Scheme to better support composable macros (Kohlbecker et al. 1986), and through Racket to support language variants as radical as static types (Flatt et al. 2012). This success in the Lisp family of languages has been surprisingly difficult to replicate in non-parenthetical syntaxes, however.

If language extensibility is not constrained to parentheses, then it seems natural to design an extension mechanism that accommodates as many grammar extensions as possible. SugarJ (Erdweg et al. 2011), for example, leverages SDF's (Heering et al. 1989) support for composable grammars to allow about as much flexibility as current parsing technology can manage. This flexibility opens the door to a range of complex grammar composition and ambiguity problems, however. From a Lisp perspective, programmers may end up worrying about technical details of character-by-character parsing, instead of designing new expressive forms.

In this paper, we offer *Honu* as an example in the middle ground between the syntactic minimalism of Lisp and maximal grammatical freedom. Our immediate goal is to produce a syntax that is more natural for many programmers than Lisp notation—most notably, using infix notation for operators—but that is similarly easy for programmers to extend. We explicitly trade expressiveness for syntactic simplicity. More generally, we suggest that a language has its own syntactic style, and successful extensions leverage that consistency rather than subverting it. Syntactic consistency is especially obvious in Lisp-style languages, but curly-brace languages have their own conventions that can be exploited and preserved by extensions to the language.

To support infix operators and syntax unconstrained by parentheses, Honu adds a precedence-based parser to a Lisp-like parsing pipeline. Since the job of this parsing stage is to turn a relatively flat sequence of terms into a Lisp-like syntax tree, we call it *enforestation*. Enforestation is not merely a preprocessing of program text; it is integrated into the macro-expansion machinery so that it obeys and leverages binding information to support hygiene, macro-generating macros, and local macro binding—facilities that have proven important for building expressive and composable language extensions in Lisp, Scheme, and Racket.

[Copyright notice will appear here once 'preprint' option is removed.]

## 2. Honu 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.

### 2.1 Honu 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 / (2 * a), b / (2 * a)]
9    }
10 }
```

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; for example, `f(x)[y]` could either access 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.

### 2.2 Honu Macros

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

```
macro ⟨name⟩ ( ⟨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]

One simple use of macros is to remove boilerplate. For example, suppose we 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
  }
}
```

We 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, we might instead 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 this 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* (Felleisen and Culpepper 2010). 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, we could define a `parabola` macro to generate parabolic formulas, and then we 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 our 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 `gener-`

[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 (Flatt 2002).

ate_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.

### 2.3 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"
```

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

The Honu pattern form binds a new syntax class:

pattern ⟨*name*⟩ ( ⟨*literals*⟩ ) { ⟨*pattern*⟩ }

A pattern form is similar to a macro without an expansion ⟨*body*⟩. Pattern variables in ⟨*pattern*⟩ turn into sub-pattern 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 }
```

we 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 cond's template, 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.

### 2.4 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 ⟨*name*⟩ ⟨*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 ⟨*name*⟩ ⟨*prec*⟩ ⟨*assoc*⟩ ⟨*binary transform*⟩

unary_operator ⟨*name*⟩ ⟨*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.

As a example binary operator, we 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:

```
binary_operator raise 10 'left
  function (left, right){
    syntax(pow(left, right))
  }
```

The precedence level of raise is 10, and it associates to the left. Since 10 is higher than the precedence of + and -, the expression

```
2 + 8 raise 2 - 1
```

raises 8 to the second power before adding 2 and subtracting 1.

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

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

We 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 5 'left lg 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 certain parsing tasks with binding resolution, as we discuss in the next section.

## 3. 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 we describe Honu parsing in part by analogy to Scheme, but the middle layer is unique to Honu.

### 3.1 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-sequence⟩ )
                  |  ( ⟨expression⟩ )
                  |  ⟨expression⟩ [ ⟨expression⟩ ]
                  |  [ ⟨comma-sequence⟩ ]
                  |  [ ⟨expression⟩ : ⟨expression⟩ = ⟨expression⟩ ]
                  |  { ⟨sequence⟩ }
                  |  ⟨form identifier⟩ ⟨term⟩*
⟨comma-sequence⟩ ::= ⟨expression⟩ [,] ⟨comma-sequence⟩
                  |  ⟨expression⟩
⟨sequence⟩       ::= ⟨expression⟩ [;] ⟨sequence⟩
                  |  ⟨expression⟩
```

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

- Unary and infix binary operations are supported through the extensible ⟨*unary operator*⟩ and ⟨*binary operator*⟩ productions.
- The ⟨*expression*⟩ `(` ⟨*comma-sequence*⟩ `)` 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-sequence*⟩ `]` 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 semi-colons does not prevent their use in different ways within a new syntactic form.

### 3.2 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 indicating 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.

In the case of Scheme, the reader's grouping of nested ⟨*term*⟩s via parentheses is another crucial ingredient toward making language extensions compose, since parentheses consistently delimit syntactic forms independent from other aspects of parsing the term. In Honu, the reader's matching of parentheses plays a lesser role, since Honu supports infix operators and other syntactic forms that are not fully parenthesized.

### 3.3 Enforestation

To handle infix syntax, the Honu parser relies on an *enforestation* phase that converts a relatively flat sequence of ⟨*term*⟩s into a more Scheme-like tree of nested expressions. Enforestation handles operator precedence and the relatively delimiter-free nature of Honu syntax, and it is macro-extensible. 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⟩ ::= ...
```

| $\langle$*tree term*$\rangle$

Enforestation is driven by an `enforest` function that extracts the first expression from an input stream of $\langle$*term*$\rangle$s. The `enforest` function incorporates aspects of the precedence parsing algorithm by Pratt (1973) to keep track of infix operator parsing and precedence. Specifically, the `enforest` function has the following contract:

$$\texttt{enforest} : \langle \textit{term} \rangle^* \quad (\langle \textit{tree term} \rangle \to \langle \textit{tree term} \rangle) \quad \langle \textit{prec} \rangle \\ \to (\langle \textit{tree term} \rangle, \langle \textit{term} \rangle^*)$$

The arguments to `enforest` are as follows:

- *input* — a list of $\langle$*term*$\rangle$s for the input stream;
- *combine* — a function that takes a $\langle$*tree term*$\rangle$ for an expression and produces the result $\langle$*tree term*$\rangle$; 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.

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 $\langle$*expression*$\rangle$ with the remainder terms of the input stream.

If the *input* starts with a tree term, then `enforest` effectively checks for a match with one of the $\langle$*expression*$\rangle$ grammar productions that themselves start with $\langle$*expression*$\rangle$. Let *init* be the tree term that starts *input*, and the second element of *input* determines how `enforest` proceeds:

- If the second term in *input* is an identifier that is bound as a binary operator, then `enforest` tail-calls itself, dropping the first two terms of *input*. The new *combine* and *precedence* for the recursive call depend on how the found operator's precedence compares to the current *precedence*.

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

  If the found operator's precedence is greater than the current *precedence*, then a new *combine* function takes a *right* and applies the operator's transformer to *init* and *right*; the result is then passed to the original *combine*. Meanwhile, *precedence* becomes the found operator's precedence.
- If the second term in *input* is a parenthesized sequence of terms, then the input is parsed as a function call. The parenthesized sequence is processed through recursive calls to `enforest`, checking for the optional comma after each expression, until the subsequence is exhausted. Each recursive call to `enforest` is given the remainder of the parenthesized terms, the identity function for *combine*, and zero for *predecence*.

  The *init* and the tree terms from recursive calls are grouped into a function-call tree term, which is supplied to *combine* to produce the `enforest` result. The remaining terms in *input* (after the *init* and the parenthesized term) are also returned.
- If the second term in *input* is a square-bracketed sequence of terms, then the input is parsed as an array reference. The bracketed sequence is processed through a recursive call to `enforest` with the identity function for *combine* and zero for *precedence*. A single call to `enforest` must exhaust the subsequence (or else a syntax error is reported).

  The bracketed expression is combined with *init* into a array-reference tree term, which is supplied to *combine* to produce the result. The remaining terms of *input* (after *init* and the bracketed term) are also returned.
- Otherwise, the second term in *input* is not a binary operator, parenthesized sequence, or square-bracketed sequence; `enforest` applies *combine* to *init* and returns the result. The rest of *input* (after *init*) is also returned.

Otherwise, the first term in *input* is not a tree term, and the `enforest` function proceeds by case analysis of the term:

- If the *input* starts with a term that can be used as a literal, such as a number or string, then the literal is turned into a tree term and passed to *combine*, whose result becomes the result of `enforest` along with the rest of *input*.
- If the *input* starts with an identifier that is bound as a declared variable, then the identifier is turned into a tree term for the variable reference and passed to *combine*, whose result becomes the result of `enforest` along with the rest of *input*.
- If the *input* starts with an identifier that is bound as a unary operator, then `enforest` tail-calls itself with the rest of *input*. A new *combine* accepts a term *next* and passes it to the unary operator's transformer; the result of the transformer is passed on to the original *combine*. Meanwhile, *precedence* for the recursive call is the maximum of the current *precedence* and the operator's precedence.
- If the *input* starts with a parenthesized or square-bracketed sequence of terms, then the parenthesized sequence is enforested through recursive calls to `enforest`. In the case of parentheses, the result must be a single tree term, and it is returned. In the case of square brackets, the results are packaged into a list-creating tree term.
- If the *input* starts with a curly-braced subsequence of terms, the terms are packaged as-is into a tree term representing a block expression. Further enforestation is deferred until it is triggered by `parse`; see section 3.5.
- If the *input* starts with a $\langle$*form identifier*$\rangle$ that is bound to a macro, then the macro is applied.

  In its most primitive form, a macro transformer has the contract

  $$\langle \textit{term} \rangle^* \to (\langle \textit{term} \rangle, \langle \textit{term} \rangle^*)$$

  That is, it takes the *input* sequence and produces a single term for the expansion of the macro, plus the remainder of the input stream that is not consumed by the macro.

  The `enforest` function accepts the results from the macro transformer and then tail-calls itself recursively, flattening the result ($\langle$*term*$\rangle$, $\langle$*term*$\rangle$*) tuple back to a $\langle$*term*$\rangle$* sequence for *input*, and keeping the *combine* and *precedence* intact. Normally, the initial $\langle$*term*$\rangle$ result from a macro is parenthesized, so that it is kept intact as an expression during further enforestation—in particular, `syntax` produces a parenthesized term—although low-level macros are not currently constrained to that behavior.

  A macro transformer can itself call `enforest` to tease out subexpressions from the input stream. Normally, recursive calls to `enforest` are triggered through the use of the `expression` syntax class, as we discuss in section 3.4.
- If the *input* starts with a $\langle$*form identifier*$\rangle$ that is bound to a primitive syntactic form, then parsing proceeds the same as for a

macro, but the transformer for a primitive form always produces a tree term.

The `function` form acts as a Honu macro when it is used as a declaration, and it expands to a `var` declaration with a `function` expression in that case. When `function` is used as an expression (i.e., an anonymous function), then it instead produces an expression tree term.

As an example, with the input

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

enforestation starts with the entire sequence of terms, the identity function, and zero:

```
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. We 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:

```
enforest(2 * 3 - f (10), combine1, 1)
  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)
```

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 `*` before chaining to the current *combine* function:

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

The current input sequence once again begins with a literal:

```
enforest(<literal: 3> - f (10), combine2, 2)
```

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. We abbreviate this new tree term:

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

Since the current precedence is back to 0, the precedence of `-` is now higher than the current precedence:

```
enforest(f (10), combine3, 1)
 where combine3(t) = <bin: -, <expr: 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)` immediately produces `<literal: 10>` for the argument sequence, so that the non-nested `enforest` continues as

```
enforest(<call: <id: f>, <literal: 10>>, combine3, 1)
```

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>>>
```

### 3.4 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 2.2) has the shape

```
macro ⟨name⟩ ( ⟨literals⟩ ) { ⟨pattern⟩ } { ⟨body⟩ }
```

The `macro` form generates a low-level macro by compiling the ⟨*pattern*⟩ 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.

### 3.5 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 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 (Sperber 2011) and Racket (Flatt et al. 2012), 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 (Flatt et al. 2012). 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 (Findler and Felleisen 2002) 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`.

### 3.5.1 Parsing Algorithm

The contract of the Honu `parse` function is

`parse` : $\langle term\rangle*$ $\langle bindings\rangle$ $\rightarrow$ $\langle AST\rangle*$

That is, `parse` takes a sequence of $\langle term\rangle$s and produces a sequence of $\langle AST\rangle$ records that can be interpreted. Initially, `parse` is called with an empty mapping for its $\langle bindings\rangle$ 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` : $\langle term\rangle*$ $\langle bindings\rangle$ $\rightarrow$ $(\langle tree\ term\rangle*, \langle bindings\rangle)$
`parse2` : $\langle tree\ term\rangle*$ $\langle bindings\rangle$ $\rightarrow$ $\langle AST\rangle*$

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

The `parse1` function takes *input* as the $\langle term\rangle$ 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 $\langle bindings\rangle$ 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.

### 3.5.2 Parsing Example

As an example, consider the following sequence in an environment where identifiers such as `macro` are bound as usual:

```
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 $\langle terms\rangle$ 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 we 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 expand2 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 "at ~a dx ~a\n" and f(<literal: 12>) sequences 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.

### 3.5.3 Parsing as Expansion

For completeness, we 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 (Flatt et al. 2012).

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.

## 4. Extended Example

Using Honu macros, we can build a class system on top of a primitive form for defining records. 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, we 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 = datum_to_syntax(name_stx, 'this, name_stx)
20   var meth_names = syntax_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 12–16 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 as in Barzilay et al. (2011) 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:

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

The `with_syntax` form at line 23 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 26–35 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, which we omit for space, but we show the `call` macro for calling a method:

```
39 macro call(){ object:expression
40               name:identifier(arg:expression ...) } {
41   var name_stx = first(syntax_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.

## 5. Related Work

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.

Honu macro definitions integrate with the parser without having to specify grammar-related details. Related systems, such as SugarJ (Erdweg et al. 2011), Xoc (Cox et al. 2008), and Polyglot (Nystrom et al. 2003) 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 (Tomita 1985) parser that enables them to extend the the class of tokens, which allows a natural embedding of domain-specific languages. Ometa (Warth and Piumarta 2007) and Xtc (Grimm 2006) 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.

Some macro systems resort to AST constructors for macro expansions instead of templates based on concrete syntax. Maya (Baker 2001) fits the AST-constructor category. Template Haskell (Jones and Sheard 2002), SugarJ, and the Java Syntax Extender (Bachrach and Playford 2001) include support for working with concrete syntax, but they also expose a set of abstract syntax tree constructors for more complex transformations. Caml4p (de Rauglaudre 2007) 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 (Weise and Crew 1993) 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 (Kelsey et al. 1998), Dylan (Shalit 1998), and Fortress (Rafkind 2009), can express many simple macros, but they are cumbersome to use for complex transformations.

ZL (Atkinson et al. 2010) is like Honu in that it relies on Lisp-like read and parsing phases, it generalizes those to non-parenthesized 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 (Heering et al. 1989) supports macro-like implementations of languages as separate from the problem of parsing. Systems built with Stratego can use SDF (Heering et al. 1989) 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.

Many systems implement some form of modularity for syntactic extension. Both SDF and Xoc (Cox et al. 2008) 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.

Multi-stage allows programs to generate and optimize code at run-time for specific sets of data. Mython (Riehl 2009), MetaOcaml (Calcagno et al. 2003), LMS (Rompf and Odersky 2010) 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 (Culpepper and Tobin-Hochstadt 2010) implements compile-time optimizations using the Racket macro system.

**Acknowledgment:** Thanks to Ryan Culpepper and Kevin Atkinson for feedback on the design and presentation of Honu.

## Bibliography

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

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

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

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

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.

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.

Ryan Culpepper and Sam Tobin-Hochstadt. Design and Implementation of Typed Scheme. *Higher Order and Symbolic Computation*, 2010.

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

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.

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

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

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

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/`

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

J. Heering, P.R H. Hendricks, P. Klint, and J. Rekers. The syntax definition formalism sdf reference manual. SIGPLAN Not., 24(11);43-75, 1989.

J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF—reference manual—. *SIGPLAN Not.* 24(11), pp. 43–75, 1989.

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

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

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proc. Lisp and Functional Programming*, pp. 151–181, 1986.

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.

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

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

Jonathan Riehl. Language embedding and optimization in mython. In *Proc. DLS 2009. pp.39∼48*, 2009.

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.

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

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

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

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

Daniel Weise and Roger Crew. Programmable syntax macros. In *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation*, 1993.