Honu: A syntactically extensible language

Jon Rafkind[0] and Matthew Flatt[1]

University of Utah rafkind@cs.utah.edu mflatt@cs.utah.edu

Abstract. Honu is a new language that contains a system for extending its syntax with an interface built on concrete syntax. Honu combines an existing hygienic macro system with a novel use of a precedence parser to achieve a syntax that is algol-like while maintaining the power of s-expression based macros. We demonstrate how to build the parser and connect it to the underlying macro system.

1 Introduction

Syntactically extensible languages offer many advantages over languages whose syntax is static. The connection between an expression and the programmer's intention can be made much clearer by inventing new syntactic forms that provide an intuitive interface while implementing the desired behavior. Such forms have the additional benefit of being indistinguishable from built-in forms.

Scheme and Racket have have embraced syntactic abstraction mechanisms because their native syntax, S-expressions, remove a major implementation hurdle. S-expressions cleanly delineate forms from one another while also avoiding the problem of parsing infix expressions. Due to the simplicity of the grammar of s-expressions, a large amount of research has been done towards creating powerful syntactic abstraction systems for languages that use s-expression as their syntax.

One of the major results of such research is that syntactic abstraction systems can work with concrete syntax. That is, the input and output of a syntactic abstraction mechanism is the syntax from the same language that is being used. Syntactic abstraction systems that use concrete syntax generally either provide only a rule based mechanism to transform syntax or are limited to s-expressions or both. Rule based approaches lack a great deal of flexibility that procedural transformations have such as the ability to produce new names for bindings. Procedural systems, such as `syntax-case`, also allow the language to be introspective by analyzing sub-forms and inducing macro expansion on them.

An alternative design is to reify the abstract syntax tree from the implementation language into objects in the language being extended. Special operations, typically AST constructors, are then provided from the implementation language to allow new syntactic forms to be created. This paradigm has two major deficiencies: firstly forms for which no AST node exists, such as another macro, cannot be produced and secondly using AST constructors quickly becomes unwieldy.

It is fairly clear that choosing the first design is preferable, however, many popular languages today do not use s-expressions and thus end up transforming abstract syntax

trees because trees are a convenient program representation for compilers to deal with. The main difficulty in implementing a macro system for languages with rich syntax is integrating macro expansion with parsing which traditional parsing technology did not account for.

We present the design of a parser that integrates macro processing and preserves key aspects of powerful syntactic abstraction systems. In particular our macro system is hygienic, operates with concrete syntax, has procedural transformations, and is composable. Our key insight into why existing parsing technology can be extended with macros is that many of the forms in a language can be implemented as macros and the parser only has to deal with a very small set of core syntactic forms.

2 Honu

Our main goal is to develop a language that can use relatively free form syntax and supports language extensions that are common to Scheme and Racket. The result is a language called Honu which is a single step away from s-expression based language extension mechanisms on the path towards supporting languages with rich syntax.

Other lines of research have focused on supporting extension mechanisms for languages with rich syntax by allowing modification of the syntax's lexical structure. In this way they hope to support many different kinds of language extension. We are interested in supporting less dynamic extensions while remaining flexible and sound.

A key distinction in Lisp-like languages is the separation between reading raw input and expanding syntax trees. Honu leverages this idea to enforce a consistent view of syntax after the reading phase similar to how Lisp languages create s-expressions while leaving a large amount of work left to the expander. While the expansion phase of Honu and Lisp languages is similar, Honu has the additional challenge of parsing infix expressions for which our solution is to integrate precedence parsing into the expansion layer.

Aside from parsing, Honu achieves two of its goals, hygiene and procedural transformations, partly due to the underlying macro system in Racket. Honu simply has to invoke the proper Racket API's to gain hygiene and expand into Racket syntax to gain access to the metacircularity implemented by interpreter. The other two goals, composability and using concrete syntax as the main interface, result from designing a flexible parser.

Beyond integrating infix parsing into the expander, the remaining key challenge is developing a pattern and template notation for destructuring and constructing syntax respectively. We build on **syntax-parse** (Culpepper 2010) but it remains a work in progress.

3 Background

Extensible language mechanisms have come in many forms. Systems such as Xoc (Kohler 2008), Xtc (Grimm 2004), Stratego (Rekers 1989), Caml4p (Rauglaudre 2003), and Rscheme (Kolby 2002) have developed compiler frameworks that expose core aspects

of the system. Programmers must learn intricate details about how those systems work to utilize the provided API's even though often times the given API hides many implementation details.

Systems that provide syntactic abstraction can broadly be separated into term rewriting systems and fully programmable systems. Term rewriting systems such as those found in R5RS Scheme (Rees 1998), Myans (Baker 2001) for Java, and Dylan (Shalit 1998) have enjoyed reasonable success but are limited in their power. Racket and other systems that implement Dybvig's syntax-case (Dybvig 2007) allow macros to compute arbitrary expressions.

The Java Syntax Extender (Playford 2001) provides programmable macros while also allowing new syntax to be written concretely. While this achieves many of the goals we hope for, the JSE was not designed with hygiene at the outset. They hypothesized that hygiene could be later added to the system but no additions have been forthcoming. Furthermore JSE restricts macro invocation to places where it can determine the syntactic extent by forcing where delimiters can be placed.

An important piece of macro systems that use concrete syntax is the ability to construct new syntax using templates. MS2 (Crew 1993) was an early system that incorporated Lisp's quasiquote into a templating system for C. However MS2 does not have the facilities to expand syntax that correspond to infix syntax or any other complex scheme. Template Haskell (Sheard 2002) is a more recent system that also has problems using purely concrete syntax for templates. Some syntaxes can be written solely using an expression similar to quasi-quote while others need to be manually constructed out of AST nodes.

Improvements in parsing technology are slowly merging into macro systems. GLR (Tomita 1985) and PEG (Ford 2002) are relatively recent parsing formalisms that are easier to use than traditional parsers. GLR parsers are similar to LR parsers but will return possibly more than one parse tree instead of failing to generate an ambiguous LR parser. GLR grammars can also be combined with other GLR grammars in a composable way. **Xoc** is a recent attempt at using GLR for language extension. PEG parsers are also composable and continue to be developed towards having an extensible grammar. **Ometa** has added new features to the PEG formalism that make it a good fit for future syntactic extension.

Extensible compilers, such as Xoc and Polyglot (Myers 2003), have been developed to give programmers an easier path to extend a base language but are overly complex for the kinds of problems that syntactic abstraction usually solves. The key ingredient to building a usable macro system is maintaining a strong link with the language being extended. Meta languages and extra API's, while powerful, hinder the understanding of the system as a whole and eventually fall into disuse.

4 Design

The design goals of the system are to support syntactic transformations which use concrete syntax. More generally the details of the parsing implementation must be hidden as much as possible from the programmer so as not to burden them with arcane details. There are probably multiple parsing algorithms that can be modified to support

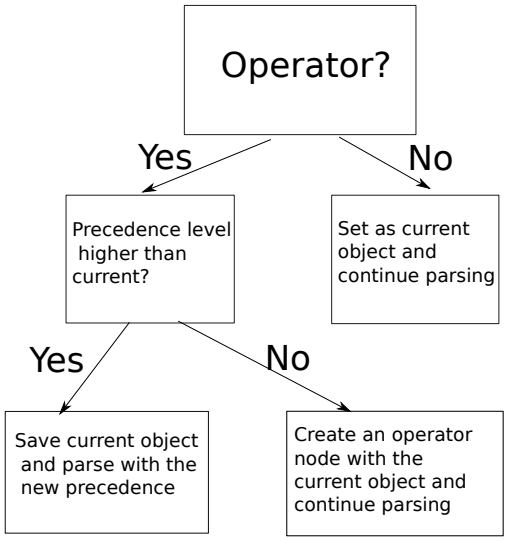
syntactic extensions but the complexity of the algorithm can quickly lead to complex designs.

Our choice of a parsing algorithm is motivated by the simplicity with which macro invocation can be weaved in. Algorithms based on LR do not lend themselves to dynamic modification while parsing because their domain is in parsing syntax performantly. Performing dynamic lookups while parsing requires the algorithm to be as unconstrained as possible. A further challenge is allowing macros to have unlimited extent and communicate the unparsed part of the input stream back to the parser.

In this view we have found that operator precedence parsing is a simple algorithm that allows us to use dynamic introspection on the syntax being parsed. Operator precedence parsing is convenient because it does not require a global table of operators to extract the current precedence and associativity level. Moreover, the only forms the precedence parser needs to recognize are expressions separated by operators and terminating syntaxes. Macros are merely a subset of expressions that are invoked when the parser discovers a binding to one.

4.1 Precedence parsing

A precedence parser keeps track of four pieces of data while parsing: the current parsed object, the current precedence level, a function for combining parsed objects on the left and right side of an operator, and the rest of the input to be parsed. For each new token that is seen by the parser a small decision tree is invoked.



Arbitrary rules can be added to the decision procedure that recognize more forms. Of course, we have added a rule to recognize tokens bound to macros and invoke them on the rest of the stream.

4.2 Honu Parsing

The result of parsing a Honu program is a lower level executable syntax, namely that of the Racket implementation. Expanding directly to Racket syntax has the advantage that existing Racket code can be easily shared with Honu programs. Racket macros can be used directly in Honu programs in limited circumstances or indirectly with wrappers that act as gateways between Honu syntax and Racket syntax.

The first step in parsing a Honu program is invoking the reader functionality that Racket provides. This stage performs two jobs: lexing the input and creating a tree structure out of certain patterns. Honu uses traditional lexing technology to produce a flat list of tokens which are then fed into the tree analyzer. Syntax trees are made out of enclosing pairs, such as parentheses and braces, while the rest of the syntax is left alone. This minimal amount of restructuring facilitates the later stages. The result is a

syntax object which is a data structure similar to an abstract syntax tree node except a syntax object is the only node available. A syntax object can contain either an atomic piece of data or a list of syntax objects.

The second stage of parsing involves interleaving precedence parsing with macro expansion. The decision procedure we use is the same as above but with an additional rule for invoking macros.

If the token is bound to a macro then pass the entire input to the macro and expect three return values: a parsed syntax object, the unparsed part of the input, and a boolean value which tells the parser to treat the macro as a terminating syntax.

The parser is able to distinguish macros from other tokens by checking their compile time value. Racket allows bindings to be created that have a value during the time when the program is being compiled which includes the time when parsing occurs. This is accomplished by alternatively parsing a form, checking if it defines a compile time value, and if so updating the compile time environment with that value. Before the parser parses the unconsumed input stream it first outputs the low level Racket syntax which the Racket macro system will analyze.

Operators are handled in a similar fashion. The precedence and associativity levels of an operator are stored in the compile time value that the operator is bound to. The parser cooperates with the definition of operators in a way that lets it extract the precedence and associativity levels. New operators can be defined in the same file being parsed using the same technique that is used for macros. In this way the existence of an operator is local to its definition instead of being added to a global table.

Other forms have been added to the Honu parser as conveniences that could not otherwise have been implemented as macros. In particular, list projections and list constructors are implemented directly by analyzing forms surrounded by brackets.

When the parser can match syntax of the following form it becomes a projection.

```
[expression: identifier <- expression]
```

Which can be used as follows

```
[x + 1: x <- [1, 2, 3]]
```

becomes the list [2, 3, 4].

5 Macro Interface

A Honu macro is defined as a function that accepts the entire input stream, performs pattern matching on it, and finally returns some concrete syntax as well as some other data. The type signature of a macro is as follows

macro :: syntax -> (syntax, syntax, boolean)

Input is destructured by the macro using pattern matching. Pattern matching is accomplished using Scheme notation combined with syntactic classes. Syntactic classes act similar to non-terminals in BNF's but in Honu are first class objects that can be defined freely. Honu currently defines one syntactic class that macros can use, **expression**, which acts as a gateway between patterns and the Honu parser. In this way macros can invoke other macros without knowing about their existence simply by calling out to the parser.

Here is example pattern that would match two expressions with an equals sign between them.

```
e1:expression = e2:expression
```

A syntactic class is merely given by appending a colon and the name of the class after some identifier.

The matched input is then transformed by the body of the macro into arbitrary syntactic forms. The **syntax** form provides a way to create concrete syntax from a template and pattern variables following the traditional procedure of Scheme's **syntax-case**. Crucially, the concrete syntax created by the macro can be anything that the parser can ultimately understand. Macros do not have to cooperate with each other in special ways to compose.

6 Examples

6.1 Operators

Operators can be defined as syntactic transformations or as plain functions. In both cases the operator contains additional information that communicates precedence and associativity to the parser.

An assignment operator is a good candidate for a syntactic transformation as opposed to a normal function because the left hand side needs to be mutated directly.

```
(define-honu-operator/syntax := 0.0001 'left
  (lambda (left right)
    (with-syntax ([left left]
                  [right right])
      #'(set! left right))))
```

The precedence is given by 0.0001 and the associativity by the symbol **'left**.

A more traditional operator can be defined with a simpler form

```
(define-binary-operator :: 0.1 'right cons)
```

Which works in the expected way when multiple operators are chained together.

```
1 :: 2 :: 3 :: null;
```

will parse as

```
1 :: (2 :: (3 :: null));
```

6.2 Macros

A simple for loop can be written like so

```
for x = 5 to 10 do {
  printf("~a\n", x)
}
```

for exemplifies a typical use case for macros in that it binds an identifier and matches arbitrary expressions to pattern variables. The implementation of **for** in Racket is given as follows.

```
1. (define-honu-syntax honu-for
2.   (lambda (code)
3.     (syntax-parse code #:literal-sets (literals)
4.       [(_ iterator:id = start:honu-expression
5.           to end:honu-expression
6.           do body:honu-expression .
7.           rest)
8.        (values
9.          #'(for ([iterator (in-range start.result end.result)])
10.             body.result)
11.         #'rest
12.         #t)])))
```

Identifiers followed by a colon and another identifier are pattern variables that match the syntactic class given by the right hand side identifier. **honu-expression** is a built-in syntactic class that re-invokes the `parse` method.

Lines 9 and 10 produce the resulting syntax from this macro in terms of Racket functionality. Honu syntax can also be written at this point.

The **rest** pattern variable on line 11 contains the part of the input stream that was not consumed by this macro. The parser will continue to parse the rest of the input stream if it is able. In this case the true boolean value on line 12 will notify the parser to stop parsing the current expression and return the completed form.

Macros can be written in Honu itself but the syntax is currently more limited than the Racket counterpart. Honu macros explicitly give their pattern as part of their definition and can then perform arbitrary work on the matched patterns. The general form of a macro in Honu is

```
macro Name (Literals ...) {Pattern ...} {Body ...}
```

In the following example we define a macro **add1** that can be composed with the **for** macro.

```
macro add1 () {x:expression} {syntax(x_result + 1)}

for z = 1 to add1 5 * 2 do
  printf("z is ~a\n", z)
```

Note that the **for** macro only required an **expression** after the **to** keyword. This expression happens to be a macro which itself wants to parse an expression. The **add1**

macro could have matched a more complex form and the **for** macro would not have been affected in any way.

An example with more complex parsing requirments is **cond**.

```
1. cond
2.   x = 1: x + 1,
3.   x = 2: x + 9;
```

Lines 2 and 3 are clauses of the **cond** block. Each clause is implemented simply as

```
check:expression : body:expression maybe_comma
```

This pattern is packaged up into a syntactic class so that the **cond** pattern becomes

```
cond clause:cond_clause ...
```

7 Future Work

An obviously important step in creating a usable macro system is being able to define new macros in the same language that is being extended. A syntax for defining new macros has been implemented but more work is needed to make it as powerful and convenient as Racket level macros. In particular, macro templates need to support syntax projections that are common to languages with infix syntax such as expanding to a list of comma delimited expressions and inserting an infix operator between two expressions.

Macros that bind identifiers to nested macros affect how the rest of the form is parsed. For instance

```
macro double {x:id stuff:expression} {
   syntax{
     macro x {left:expression : right:expression} => left + right;
     x stuff;
   }
 }
```

The macro **double** expects a simple expression to follow an identifier, but the inner macro bound by **x** expects two expressions with a colon between them. The outer macro will not have consumed the sort of input that **x** wants and so **x** will fail.

The solution is either for the outer macro to explicitly consume syntax that it knows the inner macro will require or for the outer macro to match an arbitrary amount of syntax inside some enclosing form and pass that directly to the inner macro.

```
macro double {x:id {any ...}} {
  syntax{
    macro x {left:expression : right:expression} => left + right;
    x any ...;
  }
 }
```

8 Conclusion

Honu is a proof of concept language that replaces the need for complex parsing systems with power macro technology. While Honu cannot be used to create arbitrary syntactic extensions, such as embedding XML syntax, it can be used to create new forms. Honu continues in the long tradition of Lisp like languages that allow domain specific languages to be created in a safe manner.

Other languages can be implemented inside Honu's macro system but only as long as they follow the rule that forms start with a macro. Haskell's *where* clause would not be a suitable candidate for the Honu macro system because the parser would not be able to find the *where* token until after it had parsed the expression that was supposed to have a new binding in it.

Smaller domain specific languages, such as SQL, should be suitable to embed. It is clear that programmers would benefit from the ability to write hygienically safe macros in the languages they already use and Honu is an effort towards lowering that barrier.

Bibliography

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

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

Matthias Felleisen, Ryan Culpepper. Fortifying Macros. In *Proc. ICFP*, 2010.

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

Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. In *Proc. ICFP*, 2002.

Robert Grimm. Xtc. 2004.

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

Donovan Kolby. Extensible Language Implementation. PhD dissertation, University of Texas at Austin, 2002.

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

Jonathan Bachrach, Keith Playford. Java Syntax Extender. In *Proc. 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2001.

Daniel de Rauglaudre. Camlp4. 2003. `http://caml.inria.fr/pub/old_caml_site/camlp4/index.html`

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

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

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

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

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