Honu

Version 6.1

August 1, 2014

Honu is a language with Java-like syntax built on top of Racket. Honu's main goal is to support syntactic abstraction mechanisms similar to Racket. Currently, Honu is a prototype and may change without notice.

`#lang honu` package: `honu`

Contents

1 Get started

To use Honu in a module, write the following line at the top of the file.

```
#lang honu
```

You can use Honu at the REPL on the command line by invoking racket like so

```
racket -Iq honu
```

2 Reader

2.1 Tokens

The Honu reader, honu-read, will tokenize the input stream according to the following regular expressions.

- Identifiers are [a-zA-Z_?][a-zA-Z_?0-9]*
- Strings are "[^"]*"
- Numbers are \d+(\.\d+)?
- And the following tokens + = * / - ^ || | && <= >= <- < > ! :: := : ; ‘ ’ . , ( ) { } [ ]

Comments can be written for a single line or in block form. Use **#** or **//** for a line comment and **/* */** for block comments. Block comments can be nested.

```
# i am a comment
// i am also a comment
/* start of a comment /* with an inner comment */ end of first com-
ment */
```

2.2 Structure

```
(require honu/core/read)
```
package: honu

After tokenization a Honu program will be converted into a tree with minimal structure. Enclosing tokens will be grouped into a single object represented as an s-expression. Enclosing tokens are pairs of (), {}, and [].

Consider the following stream of tokens

```
x ( 5 + 2 )
```

This will be converted into

```
(x (#%parens 5 + 2))
```

{} will be converted to (#%braces ...) and [] will be conveted to (#%brackets ...)

```
(honu-read port) → any
  port : port?
```

Read an s-expression from the given port.

```
(honu-read-syntax name port) → any
  name : any
  port : port?
```

Read a syntax object from the given port.

```
(honu-lexer port) → (list position-token?)
  port : port?
```

Tokenize a port into a stream of honu tokens.

3 Parsing

Honu is parsed using an algorithm based primarily on operator precedence. The main focus of the operator precedence algorithm is to support infix operators. In short, the algorithm operates in the following way

- 1. parse an expression
- 2. check for a binary operator. if one is found then continue to step 3 otherwise return the expression from step 1 immediately.
- 3. parse another expression
- 4. check for a binary operator. if one is found then check if its precedence is higher than the operator found in step 2, and if so then continue parsing from step 3. if the precedence is lower or an operator is not found then build an infix expression from the left hand expression from step 1, the binary operator in step 2, and the right hand expression in step 3.

Parsing will maintain the following registers

- **left** - a function that takes the right hand side of an expression and returns the infix expression by combining the left hand side and the operator.
- **current** - the current right hand side
- **precedence** - represents the current precedence level
- **stream** - stream of tokens to parse

This algorithm is illustrated with the following example. Consider the raw stream of tokens

```
1 + 2 * 3 - 9
```

left	current	precedence	stream
`(lambda (x) x)`	`#f`	`0`	`1 + 2 * 3 - 9`
`(lambda (x) x)`	`1`	`0`	`+ 2 * 3 - 9`
`(lambda (x) #'(+ 1 x))`	`#f`	`1`	`2 * 3 - 9`
`(lambda (x) #'(+ 1 x))`	`2`	`1`	`* 3 - 9`

```
(lambda (x) (left #'(* 2 x)))      2     2          3 - 9

(lambda (x) (left #'(* 2 x)))      3     2          - 9

(lambda (x) #'(- (+ 1 (* 2 3)) x)) #f    1          9

(lambda (x) #'(- (+ 1 (* 2 3)) x)) 9     1
```

When the stream of tokens is empty the **current** register is passed as an argument to the **left** function which ultimately produces the expression

```
(- (+ 1 (* 2 3)) 9)
```

In this example `+` and `-` both have a precedence of 1 while `*` has a precedence of 2. Currently, precedences can be any number that can be compared with `<=`.

The example takes some liberties with respect to how the actual implementation works. In particular the binary operators are syntax transformers that accept the left and right hand expressions as parameters and return new syntax objects. Also when the `*` operator is parsed the **left** function for `+` is nested inside the new function for `*`.

An *expression* can be one of the following

- **datum** - number, string, or symbol.

  ```
  5
  ```

- **macro** - a symbol bound to a syntax transformer.

  ```
  cond x = 5: true, else: false
  ```

- **stop** - a symbol which immediately ends the current expression. these are currently , ; :
- **lambda expression** - an identifier followed by `(id ...)` followed by a block of code in braces.

  ```
  add(x, y){ x + y }
  ```

- **function application** - an expression followed by `(arg ...)`.

```
f(2, 2)
```

- **list comprehension** -

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

- **block of code** - a series of expressions wrapped in braces.
- **expression grouping** - any expression inside a set of parenthesis

```
(1 + 1) * 2
```

4 Macros in Honu

4.1 Honu syntax

A good concrete syntax for honu macros is still under development.

4.2 Low Level Racket Interface

A Honu macro can be defined in Racket using `define-honu-syntax`.

```
(define-honu-syntax name function)
```

Defines *name* to be a honu macro that uses *function* as the syntax transformer. *function* should accept two parameters, the first is the syntax tree that follows the macro name in the current input and the second related to the current context but for now is not used.

function should return 3 values using `values`.

- a new syntax object that corresponds to the computation performed by the macro
- the rest of the input syntax that is to be parsed
- a boolean, `#t` or `#f`, that tells the parser whether or not to immediately return the current expression or to continue parsing.

Macro's should use `syntax-parse` to pattern match on their input although this is not strictly necessary. Honu provides the syntax class `honu-expression` from `honu/core/parse2` that will re-invoke the honu parser and return a single expression. The result of using `honu-expression` can be accessed with the `result` attribute.

The definition of the `for` form for Honu.

```
(define-honu-syntax honu-for
  (lambda (code context)
    (syntax-parse code #:literal-sets (cruft)
#:literals (honu-= honu-in)
      [(_ iterator:id honu-= start:honu-expression
          honu-to end:honu-expression
          honu-do body:honu-expression .
          rest)
       (values
       #'(for ([iterator
```

```
         (in-range
          start.result
          end.result)])
  body.result)
  #'rest
  #t)]
[(_ iterator:id honu-in
    stuff:honu-expression
    honu-do
    body:honu-expression
    .
    rest)
  (values
 #'(for ([iterator stuff.result])
     body.result)
 #'rest
 #t)]))))
```

5 Language

`var` is a macro that defines a new variable.

```
var x = 1
```

`for` is a macro that is similar to Racket's `for`.

```
for id = expression to expression do expression
```

```
for id in expression do expression
```

6 Examples

```
// A for loop that iterates between two bounds.
for x = 1 + 5 to 10 do
  printf("x is ~a\n" x)

// Similar to above but shows a block of expressions in the body
for x = 1 to 10 do {
 var y = x + 1;
 printf("x ~a y ~a\n", x, y)
}

// A for loop that iterates over a list of numbers
for x in [1, 2, 3] do {
 printf("x ~a\n", x);
}
```