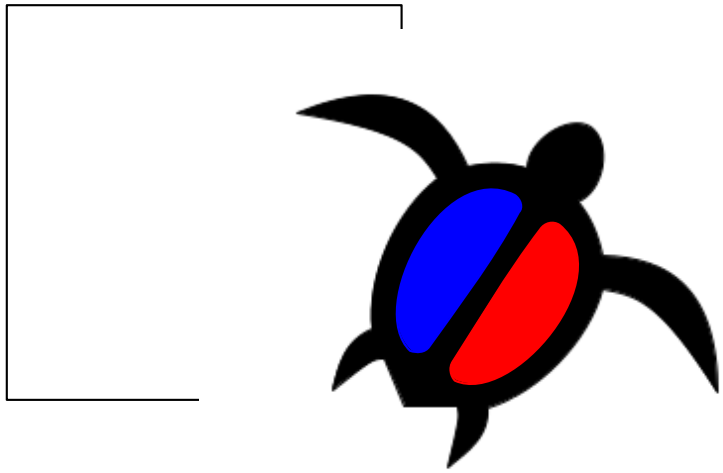


Honu

Jon Rafkind

Honu is

unconstrained **infix** **extensible**

Honu is

unconstrained **infix** **extensible**

Macros are

hygienic **procedural** **composable**

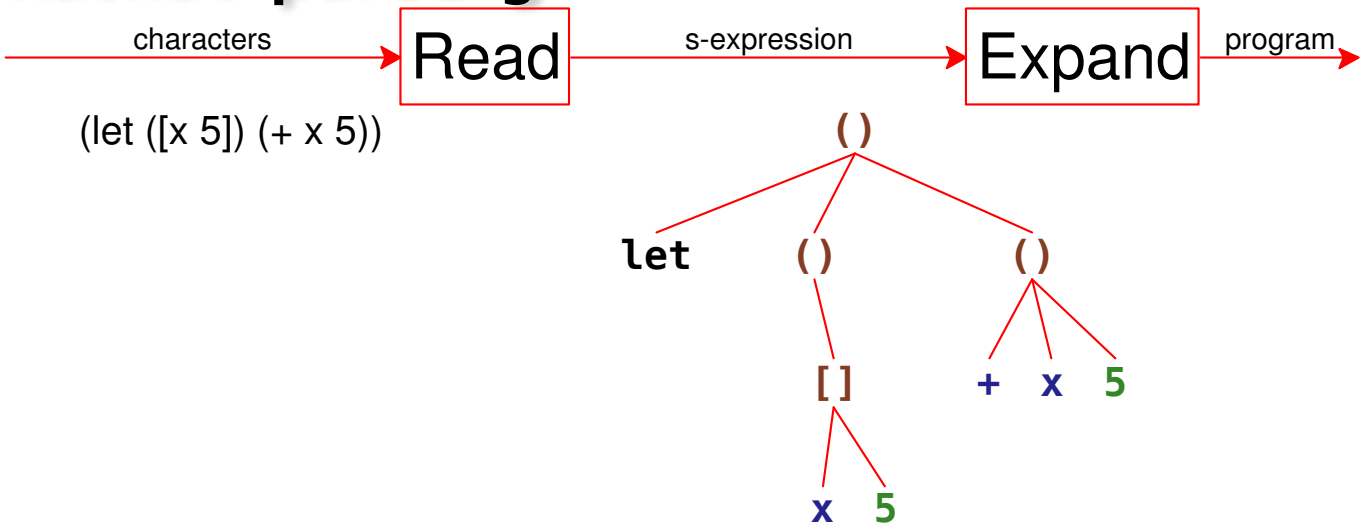
Infix syntax

```
#lang honu

function loadImages(path, files){
  binary_operator + 1 'left string_append
  [make_bitmap(path + file): file = files]
}

loadImages(root, ["a.png", "b.png", "c.png"])
```

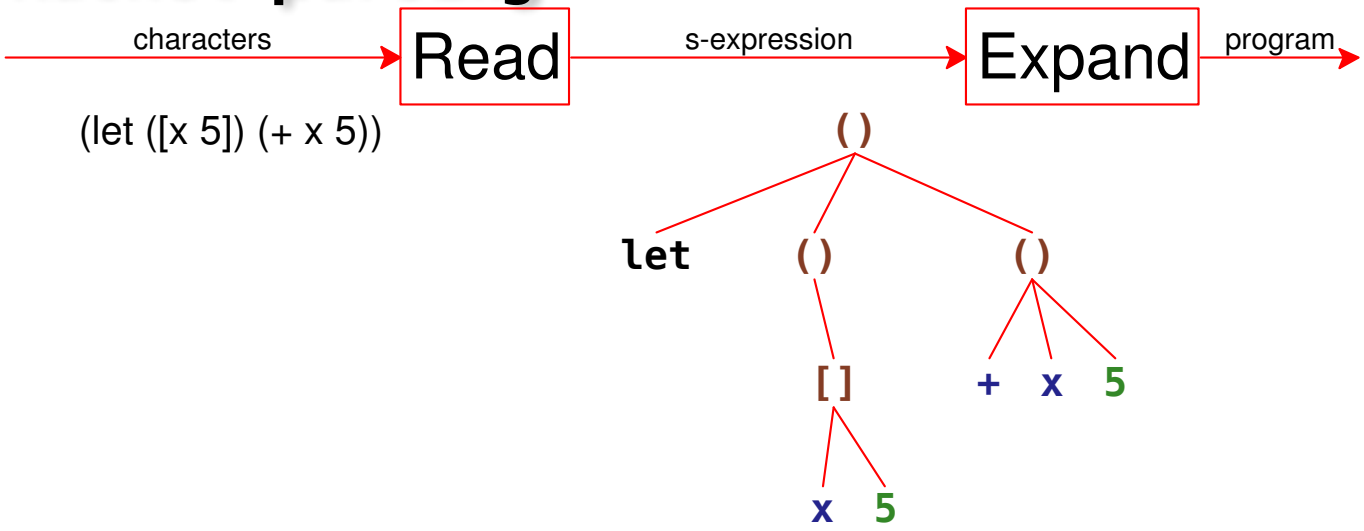
XML as a macro

```
#lang honu

macro xml ...

var data = xml
    <record>
      <name> { get_name(person) } </name>
      <age> { get_age(person) } </age>
      <address> { get_address(person) } </address>
    </record>
```

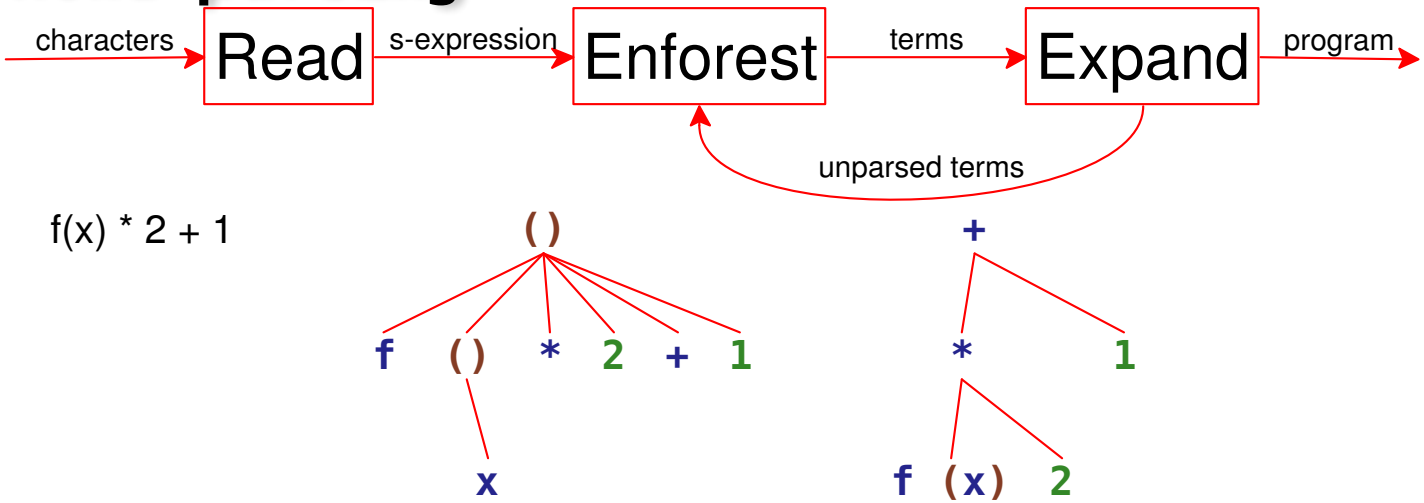
Racket parsing



Racket parsing



Honu parsing



Honu expansion

Enforest

```
function c_to_f(temp){
  temp * 9/5 + 32
}
printf("temperature ~a", c_to_f(40))
```

Honu expansion

Enforest

```
function c_to_f(temp){
  temp * 9/5 + 32
}
printf("temperature ~a", c_to_f(40))
```

Expand

<function>, `printf("temperature ~a", c_to_f(40))`

Honu expansion

Enforest

```
function c_to_f(temp){
  temp * 9/5 + 32
}
printf("temperature ~a", c_to_f(40))
```

Expand

<function>, `printf("temperature ~a", c_to_f(40))`

Enforest

```
printf("temperature ~a", c_to_f(40))
```

Honu expansion

Enforest

```
function c_to_f(temp){
  temp * 9/5 + 32
}
printf("temperature ~a", c_to_f(40))
```

Expand

<function>, `printf("temperature ~a", c_to_f(40))`

Enforest

```
printf("temperature ~a", c_to_f(40))
```

Done

<function>, <call>

Honu grammar

Expression := <literal> | <identifier>
| <unary> <expression>
| <expression> <unary>
| <expression> <binary> <expression>
| <expression> (<expression, ...)
| (<expression>)
| <expression> [<expression>]
| [<expression> , ...]
| [<expression> : <expresion> = <expression>]
| { <sequence> ... }
| <identifier> <term> ...

Honu grammar

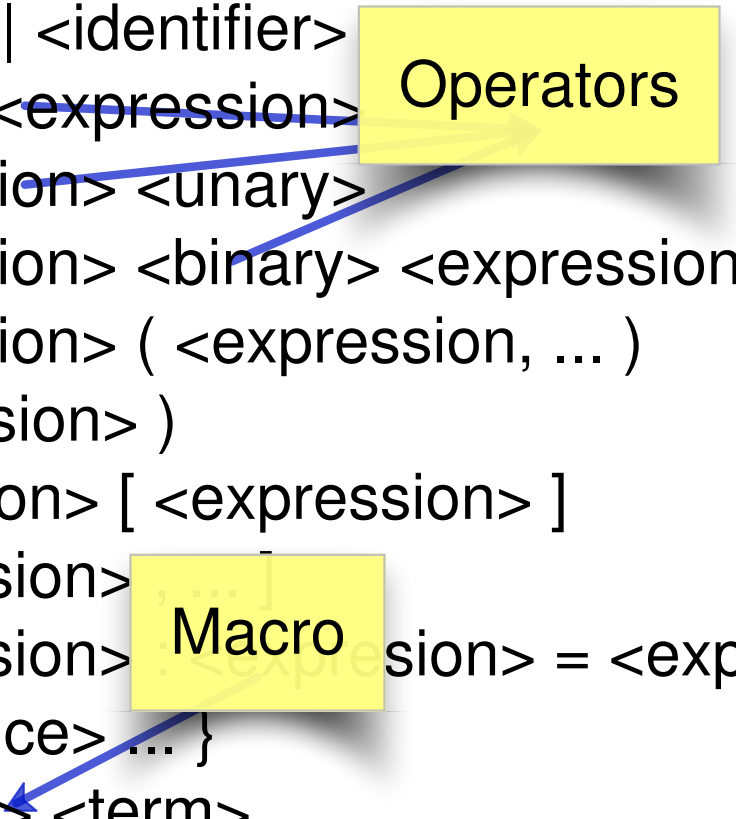
Expression := <literal> | <identifier>
| <unary> <expression>
| <expression> <unary>
| <expression> <binary> <expression>
| <expression> (<expression, ...)
| (<expression>)
| <expression> [<expression>]
| [<expression> , ...]
| [<expression> : <expression> = <expression>]
| { <sequence> ... }
| <identifier> <term> ...

Function call

List comprehension

Honu grammar

Expression := <literal> | <identifier>
| <unary> <expression>
| <expression> <unary>
| <expression> <binary> <expression>
| <expression> (<expression, ...)
| (<expression>)
| <expression> [<expression>]
| [<expression> , ...]
| [<expression> : <expression> = <expression>]
| { <sequence> ... }
| <identifier> <term> ...



Macros

Definition

```
macro name(literal ...){ pattern ... }{
    body ...
}
```

Use

```
macro withCloser(=){name:id = e:expression { body ... }}{
   syntax({
     var name = e
     body ...
     name.close()
   })
}
```

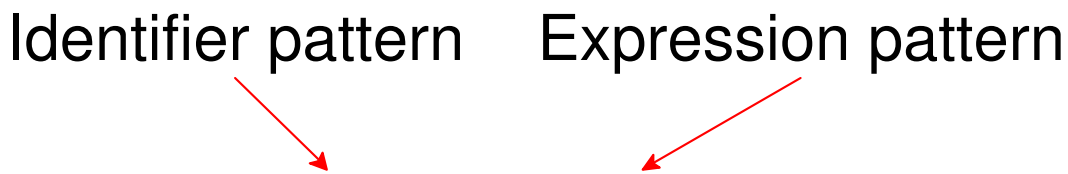
In action

```
withCloser d = getDatabase() {
   ...
}
printf("Done")
```

Patterns

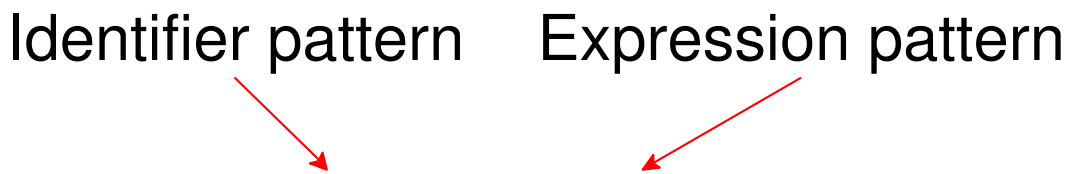
```
macro withCloser(){ name:id = e:expression { body ...  } }{
  syntax({
    var name = e
    body ...
    name.close()
  })
}
```

Patterns




```
macro withCloser(){ name:id = e:expression { body ...  } }{
  syntax({
    var name = e
    body ...
    name.close()
  })
}
```

Patterns




```
macro withCloser(){ name:id = e:expression { body ...  } }{
  syntax({
    var name = e
    body ...
    name.close()
  })
}
```

```
expression = enforest(terms)
```

Patterns

```
pattern letvar(=){name:id = expr:expression}
```

Patterns

```
pattern letvar(=){name:id = expr:expression}

Definition
macro let(){ v:letvar ... { body ... } }{
    /* ..implementation.. */
}
```

Patterns

```
pattern letvar(=){name:id = expr:expression}
```

Definition

```
macro let(){ v:letvar ... { body ... } }{
  syntax({
    $ var v_name = v_expr $ ...
    body ...
  })
}
```

Use

```
let day = getDay()
    month = getMonth() {
  printf("~a\n", is_christmas(day, month))
}
```

Racket Honu Bridge

```
(require honu/core/api honu)

(define-honu-macro xml
  (lambda (stx)

                                  ))
```

Racket Honu Bridge

```
(require honu/core/api honu)

(define-honu-macro xml
  (lambda (stx)
    (syntax-parse stx
      [(..stuff.. . rest)
       (values (racket-syntax ..new-stx..)
               #'rest #t)])))
```

Operators

Definition

```
binary_operator logbase 4 'left
          function(left right){
            log(left) / log(right)
          }
```

Use

```
2 * 12 logbase 4 + 8
```

Infix enforest

```
macro call(){ object:id . method:id(arg:expression ...)}{}

1 + call plane.passengers() * 5
```

Infix enforest

```
macro call(){ object:id . method:id(arg:expression ...)}{}

1 + call plane.passengers() * 5

1

Parse State
```

Infix enforest

```
macro call(){ object:id . method:id(arg:expression ...)}{}

1 + call plane.passengers() * 5

1 +

Parse State
a = function (right){ mkOp(+, 1, right) }
```

Infix enforest

```
macro call(){ object:id . method:id(arg:expression ...)}{}

1 + call plane.passengers() * 5

1 + call plane.passengers()

Parse State
a = function (right){ mkOp(+, 1, right) }
```

Infix enforest

```
macro call(){ object:id . method:id(arg:expression ...)}{}

1 + call plane.passengers() * 5

1 + call plane.passengers() *

Parse State
a = function (right){ mkOp(+, 1, right) }
b = function (right){ mkOp(*, <call>, right) }
```

Infix enforest

```
macro call(){ object:id . method:id(arg:expression ...)}{}

1 + call plane.passengers() * 5

1 + call plane.passengers() * 5

Parse State
a = function (right){ mkOp(+, 1, right) }
b = function (right){ mkOp(*, <call>, right) }
a(b(5))
```

`#lang honu`

Thank you