

Ioke
☺
☹

Seph





Ola Bini

computational metalinguist

ola.bini@gmail.com

<http://olabini.com/blog>

Ioke



**How expressive can you
make a language if you
disregard performance?**

arbitrarily sized numbers

operator shuffling

comprehensions

decimal numbers

macros

reflection/introspection

symbols

homoiconicity

messages

ranges

dictionaries

methods

lists

mixins

booleans

pairs

literals

pervasive documentation

conditions

dynamic typing

sets

multi-vm

hooks

regular expressions

strong typing

aspects

syntax

dynamic places

lexical blocks

closures

generalized assignment

tuples

ratios

destructuring

enumerables

icheck

java integration

sequences

prototype based OO

blank slate

Basics

Dynamic, strong typing

Prototype based OO

Everything is message passing

JVM is home, CLR is adopted home

Common Lisp style condition system

Runtime macros

More syntax than Lisp - less than most other languages

Sugared syntax

`; this is a comment`

`x = 42`

`y = (x * 3 / 15.3) + 3 / 5`

`Text sayHello = method("hello #{self}" println. "^^^" println)
"Ola" sayHello`

`something = [42, 55, 16]
something[10] = "hello"`

`something first asText length println`

Desugared/canonical syntax

; this is a comment

=(x, 42)

=(y, x *(3) /(15.3)) +(3 /(5))

Text =(sayHello, method("hello #{self}" println. "^^^" println))
"Ola" sayHello

=(something, [(42, 55, 16)])
something []=(10, "hello")

something first asText length println

Assignment is a message send



```
=(x, 42)
```

```
=(y, x *(3) /(15.3)) +(3 /(5))
```

```
Text =(sayHello, method("hello #{self}" println. "^^^" println))  
"Ola" sayHello
```

```
=(something, [(42, 55, 16)])  
something []=(10, "hello")
```

```
something first asText length println
```

Operators are also messages
(and shuffled for precedence)

`=(x, 42)`

`=(y, x *(3) /(15.3)) +(3 /(5))`

`Text =(sayHello, method("hello #{self}" println. "^^^" println))`
`"Ola" sayHello`

`=(something, [(42, 55, 16)])`
`something []=(10, "hello")`

`something first asText length println`

List creation, aref and aset are messages too

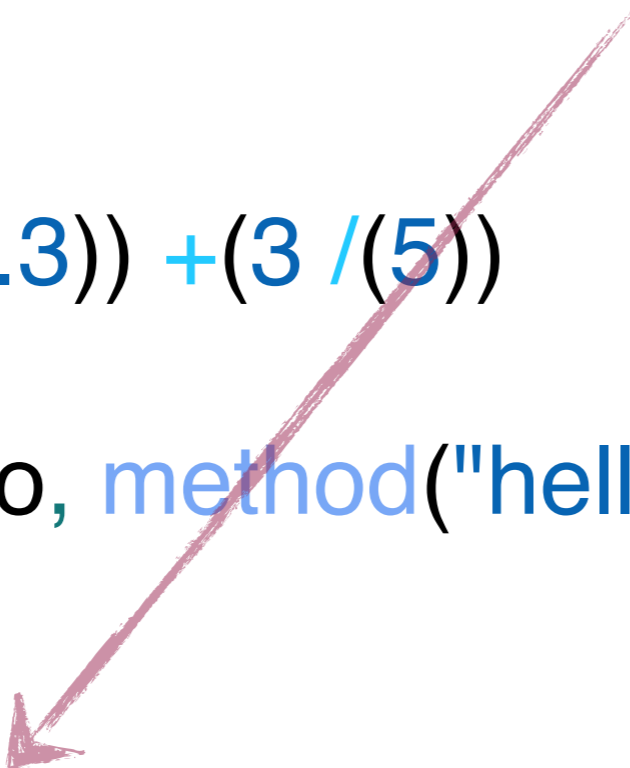
```
=(x, 42)
```

```
=(y, x *(3) /(15.3)) +(3 /(5))
```

```
Text =(sayHello, method("hello #{self}" println. "^^^" println))  
"Ola" sayHello
```

```
=(something, [(42, 55, 16)])  
something []=(10, "hello")
```

```
something first asText length println
```



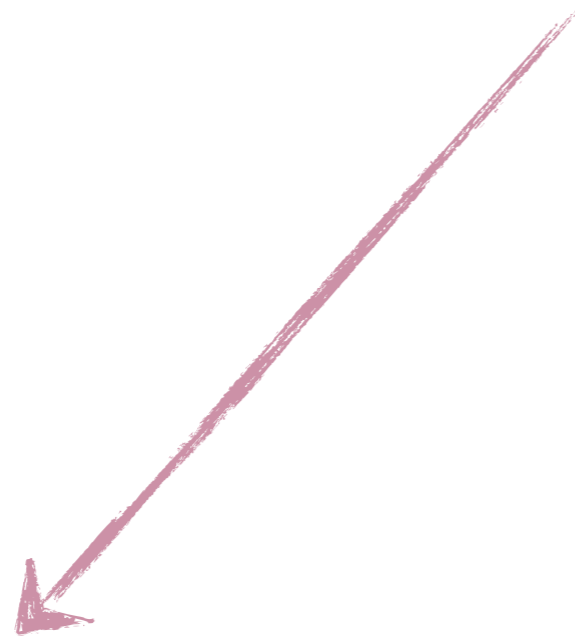
Internal syntax

`=(x, internal:createNumber(42))`

`=(y, x *(internal:createNumber(3)) /(internal:createDecimal(15.3))) +(internal:createNumber(3) /(internal:createNumber(5)))`

`internal:concatenateText("hello ", self, "") println`
`internal:createText("Ola") sayHello`

Creation of literals is a message



```
=(x, internal:createNumber(42))
```

```
=(y, x *(internal:createNumber(3)) /(internal:createDecimal(15.3))) +(
  internal:createNumber(3) /(internal:createNumber(5)))
```

```
internal:concatenateText("hello ", self, "") println
internal:createText("Ola") sayHello
```

Creation of literals is a message



```
=(x, internal:createNumber(42))
```

```
=(y, x *(internal:createNumber(3)) /(internal:createDecimal(15.3))) +(
  internal:createNumber(3) /(internal:createNumber(5)))
```

```
internal:concatenateText("hello ", self, "") println
internal:createText("Ola") sayHello
```

Creation of literals is a message

`=(x, internal:createNumber(42))`

`=(y, x *(internal:createNumber(3)) /(internal:createDecimal(15.3))) +(internal:createNumber(3) /(internal:createNumber(5)))`

`internal:concatenateText("hello ", self, "") println`
`internal:createText("Ola") sayHello`

Code folding: Enumerable#find

[false, nil, 42, false] find \Rightarrow 42

(1..100) map(* 2) find(> 42) \Rightarrow 44

["bar", "foo", "foo bar"] find(\sim #/.../) \Rightarrow "foo bar"

(1..100) find(x, x * x * x > 100) \Rightarrow 5

Code folding: Enumerable#some

[false, nil, 42, false] some \Rightarrow 42

["bar", "foo", "foo bar"] some(\sim `#!/.../`) \Rightarrow Regexp Match

["bar", "foo", "foo bar"] some(s, s[5]) \Rightarrow 97

```

Mixins Enumerable find = macro(
  len = call arguments length
  if(len == 0,
    self each(n, if(cell(:n), return(it))),

    if(len == 1,
      theCode = call arguments first
      self each(n,
        if(theCode evaluateOn(call ground, cell(:n)),
          return(cell(:n)))))

    lexicalCode = LexicalBlock createFrom(call arguments, call ground)
    self each(n,
      if(lexicalCode call(cell(:n)),
        return(cell(:n))))))
  nil)

```

Mixins Enumerable some = macro(
len = call arguments length
if(len == 0,
self each(n, if(cell(:n), return(it))),

if(len == 1,
theCode = call arguments first
self each(n,
if(theCode evaluateOn(call ground, cell(:n)),
return(it))),

lexicalCode = LexicalBlock createFrom(call arguments, call ground)
self each(n,
if(lexicalCode call(cell(:n)),
return(it))))))

false)

Mixins Enumerable find = enumerableDefaultMethod(

```
  ·,  
  if(cell(:x),  
    return(cell(:n))),  
  nil)
```

Mixins Enumerable some = enumerableDefaultMethod(

```
  ·,  
  if(cell(:x),  
    return(it)),  
  false)
```

```

let(enumerableDefaultMethod,
  dsyntax(
    [docstr, initCode, repCode, returnCode]

    "(dmacro(` docstr,
      []
      'initCode
      self each(n, x = cell(:n). 'repCode)
      'returnCode,

      [theCode]
      'initCode
      self each(n,
        x = theCode evaluateOn(call ground, cell(:n)). 'repCode)
      'returnCode,

      [argName, theCode]
      'initCode
      lexicalCode = LexicalBlock createFrom(list(argName, theCode), call ground)
      self each(n, x = lexicalCode call(cell(:n)). 'repCode)
      'returnCode))),

```

DSL example: parser combinator

```
simple = IParse Parser(  
  digit    = 0..9  
  letter   = ("a".."z") | ("A".."Z")  
  id       = letter (letter | digit)*  
  id2      = letter* (letter | digit)  
  number   = 1..9 digit*  
  primary  = "(" expr ")" | number | id  
  term     = primary ("*" | "/" ) term | primary  
  expr     = term ("+" | "-" ) expr | term  
  and     = expr "and" expr | expr  
)
```

First pass conversion

```
simple = IParse Parser(  
  digit    = 0 ..(9)  
  letter   = ("a" ..("z")) | ("A" ..("Z"))  
  id       = 'letter + (('letter |('digit)) * ())  
  id2      = 'letter * () + ('letter |('digit))  
  number   = 1 ..(9) + ('digit * ())  
  primary  = "(" + ('expr) + (")") | ('number) | ('id)  
  term     = 'primary + ("*" | ("/")) + ('term) | ('primary)  
  expr     = 'term + ("+" | ("-")) + ('expr) | ('term)  
  and      = 'expr + ("and") + ('expr) | ('expr)  
)
```

Creating parsers from literals

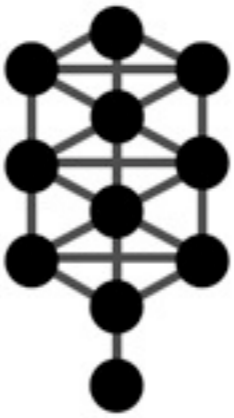
```
ParserContext = Origin mimic do(  
  internal:createText = method(raw,  
    IParse TextParser with(context: self, text: super(raw)))  
  internal:createNumber = method(raw,  
    IParse NumberParser with(context: self, number: super(raw)))  
)
```

```
ParserContext cell(:"") = dmacro([name]  
  IParse ParserParser with(context: self, name: name name)  
)
```

Combining parsers

```
BaseParser = Origin mimic do(  
  .. = method(other,  
    IParse RangeParser with(  
      context: context, start: self, end: other, inclusive: true))  
  
  | = method(other,  
    IParse OrParser with(context: context, first: self, second: other))  
  
  + = method(other,  
    IParse SequenceParser with(context: context, first: self, second: other))  
  
  cell(:"*") = method(  
    IParse RepeatingParser with(context: context, repeat: self))  
  
  cell(:"\"") = dmacro([name]  
    IParse ParserParser with(context: context, name: name name)))
```

Seph



**Take loke into the real
world**

Differential features vs luke

All objects are immutable (including the AST)

Local variables are mutable (including lexical scopes)

Light weight threads

Proper TCO

Clojure style concurrency primitives

Full numeric tower (with real numbers)

Stolen features/ideas


Light weight threads	Erlang
Proper TCO	Impl from Erjang
Persistent collections	Clojure
Refs/Vars/Agents/Atoms	Clojure
STM	Clojure
No global state module system	Newspeak
Numerical tower	Kawa/Scheme
Mostly everything else	loke

Accumulator (mutation)

accumulator: $\#(n, \#(i, n += i))$

Accumulator (mutation)

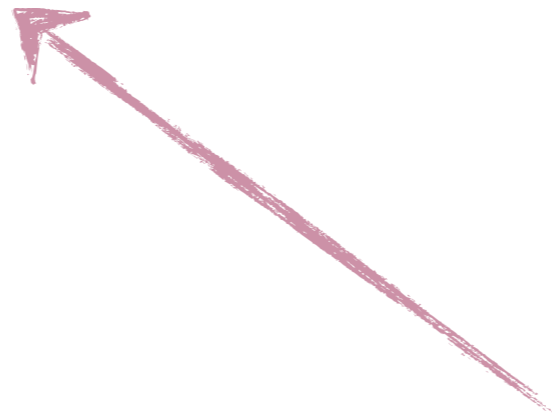
Creates abstraction,
both method and lexical closure



accumulator: `#(n, #(i, n += i))`

Accumulator (mutation)

accumulator: #(n, #(i, n += i))



Top level name followed by colon means
it's exported

Accumulator (light weight threads)

```
accumulatorp = #(n,  
  receive(  
    (p, i), val = n + i. p <- val. accumulatorp(val))),
```

```
accumulator: #(n,  
  p = -> accumulatorp(n)  
  #(i,  
    p <- (currentProcess, i)  
    receive(  
      v, v))))
```

Local method, only accessible
through lexical scoping




```
accumulatorp = #(n,  
  receive(  
    (p, i), val = n + i. p <- val. accumulatorp(val))),
```

```
accumulator: #(n,  
  p = -> accumulatorp(n)  
  #(i,  
    p <- (currentProcess, i)  
    receive(  
      v, v))))
```

Tail recursive call to itself

```
accumulatorp = #(n,  
  receive(  
    (p, i), val = n + i. p <- val. accumulatorp(val))),
```



```
accumulator: #(n,  
  p = -> accumulatorp(n)  
  #(i,  
    p <- (currentProcess, i)  
    receive(  
      v, v))))
```

Starts a new process,
saving the PID in p

```
accumulatorp = #(n,  
  receive(  
    (p, i), val = n + i. p <- val. accumulatorp(val))),
```

```
accumulator: #(n,  
  p = -> accumulatorp(n)  
  #(i,  
    p <- (currentProcess, i)  
    receive(  
      v, v))))
```

Sends a message to p

```
accumulatorp = #(n,  
  receive(  
    (p, i), val = n + i. p <- val. accumulatorp(val))),
```

```
accumulator: #(n,  
  p = -> accumulatorp(n)  
  #(i,  
    p <- (currentProcess, i)  
    receive(  
      v, v))))
```

Tail recursive sets

```
IntSet: Something with(  
  empty?: false,  
  adjoin: #(x, Adjoin with(s: self, obj: x)),  
  ∪: #(x, Union with(left: self, right: x)),
```

```
Adjoin: IntSet with(  
  contains?: #(y, obj == y || s contains?(y)),
```

```
Union: IntSet with(  
  empty?: #(left empty? && right empty?),  
  contains?: #(y, left contains?(y) || right contains?(y)),
```

```
Empty: IntSet with(  
  empty?: true,  
  contains?: #(_, false)),
```

```
IntegersMod: IntSet with(contains?: #(y, y % n == 0))
```

No global state / module system

Top level nests segments in new lexical contexts

Keyword syntax exports a name

A module user specifies imported name of module

This also allows for parameterized instances of modules

A module is just a simple Seph object

However, no Newspeak like “platform” object (for now)

Resources

<http://ioke.org>

<http://github.com/olabini/ioke>

#ioke (FreeNode)

ioke-language@googlegroups.com

<http://seph-lang.org> (soonish)

<http://github.com/olabini/seph> (soonish)

Questions?

OLA BINI

ThoughtWorks®

<http://olabini.com>
obini@thoughtworks.com

@olabini

loke vs Smalltalk

loke is prototype based

loke is file based and doesn't run in an image

loke has a radically different core and standard library

loke has both positional arguments

loke method names are selectors, without keyword arguments

loke vs Lisp

loke is single dispatch OO - all messages have a receiver

loke has syntax...

The loke AST is based around Messages, with prev and next pointers

Common Lisp has reader macros

loke macros are runtime - not load-time or compile-time

However, Common Lisp is probably the closest inspiration for some language and core features (ie conditions, format)

loke vs lo

loke doesn't have any concurrency features, lo has actors

lo is focused on embedding - loke is a language experiment

lo is fast...

loke is based on the JVM (and to a degree the CLR)

loke's core library is inspired by Ruby and quite different

loke has a CL style condition system

loke separates methods, macros and lexical blocks

loke has keyword method arguments

loke handles syntax slightly different

It seems mostly to be philosophy differences in API design

loke vs Ruby

loke has Macros

loke uses lazy sequences as basis for Enumerable

loke's surface syntax translates to canonical AST

loke has extremely simple semantics, Ruby not so much

`; this is a comment`

Simple assignment

`x = 42`



`y = (x * 3 / 15.3) + 3 / 5`

`Text sayHello = method("hello #{self}" println. "^^^" println)
"Ola" sayHello`

`something = [42, 55, 16]
something[10] = "hello"`

`something first asText length println`

`; this is a comment`

Not a float, a BigDecimal

`x = 42`

`y = (x * 3 / 15.3) + 3 / 5`

`Text sayHello = method("hello #{self}" println. "^^^" println)
"Ola" sayHello`

`something = [42, 55, 16]
something[10] = "hello"`

`something first asText length println`

; this is a comment

x = 42

y = (x * 3 / 15.3) + 3 / 5

Defining a method is
assignment

Text sayHello = method("hello #{self}" println. "^^^" println)
"Ola" sayHello

something = [42, 55, 16]

something[10] = "hello"

something first asText length println

; this is a comment

x = 42

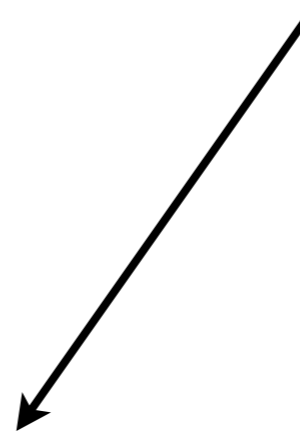
y = (x * 3 / 15.3) + 3 / 5

Text sayHello = method("hello **#{self}**" println. "^^^" println)
"Ola" sayHello

something = [42, 55, 16]
something[10] = "hello"

something first **asText** length **println**

String interpolation



`; this is a comment`

Period ends an expression

`x = 42`

`y = (x * 3 / 15.3) + 3 / 5`

`Text sayHello = method("hello #{self}" println. "^^^" println)
"Ola" sayHello`

`something = [42, 55, 16]
something[10] = "hello"`

`something first asText length println`

```
; this is a comment
```

```
x = 42
```

```
y = (x * 3 / 15.3) + 3 / 5
```

```
Text sayHello = method("hello #{self}" println. "^^^" println)  
"Ola" sayHello
```

```
something = [42, 55, 16] ← Creates a list  
something[10] = "hello"
```

```
something first asText length println
```