```
x00000001000d5688: rex.W test $0x1,%al
x00000001000d568c: jne    0x1000d56a6
x00000001000d5692: add    $0x2,%rax
x00000001000d5699: jno    0x1000d56cf
x00000001000d569f: sub    $0x2,%rax
x00000001000d56a6: mov    %rax,-0x10(%rbp)
x00000001000d56ad: movabs $0x2,%rax
x00000001000d56b7: mov    %rax,%rbx
x00000001000d56ba: mov    -0x10(%rbp),%rax
x00000001000d56c1: movabs $0x1000d5320,%r11
x00000001000d56cb: rex.WB callq *%r11
x00000001000d56ce: nop
x00000001000d56cf: mov    %rax,-0x10(%rbp)
x00000001000d56d6: mov    -0x10(%rbp),%r11
x00000001000d56dd: mov    %r11,-0x8(%rbp)
x00000001000d56e4: mov    %r11,%rax
```

@indutny

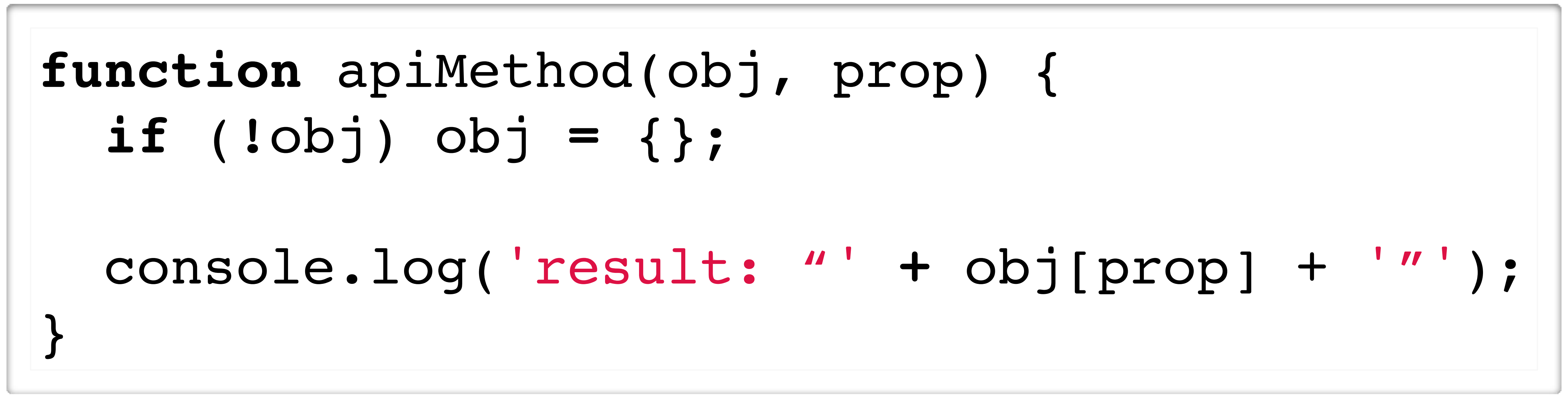
Twitter / Github / IRC

- Nodejitsu guy
- Node.js core team member
- Author of node-spdy module, Candor language and the parts of node’s debugger

Candor PL

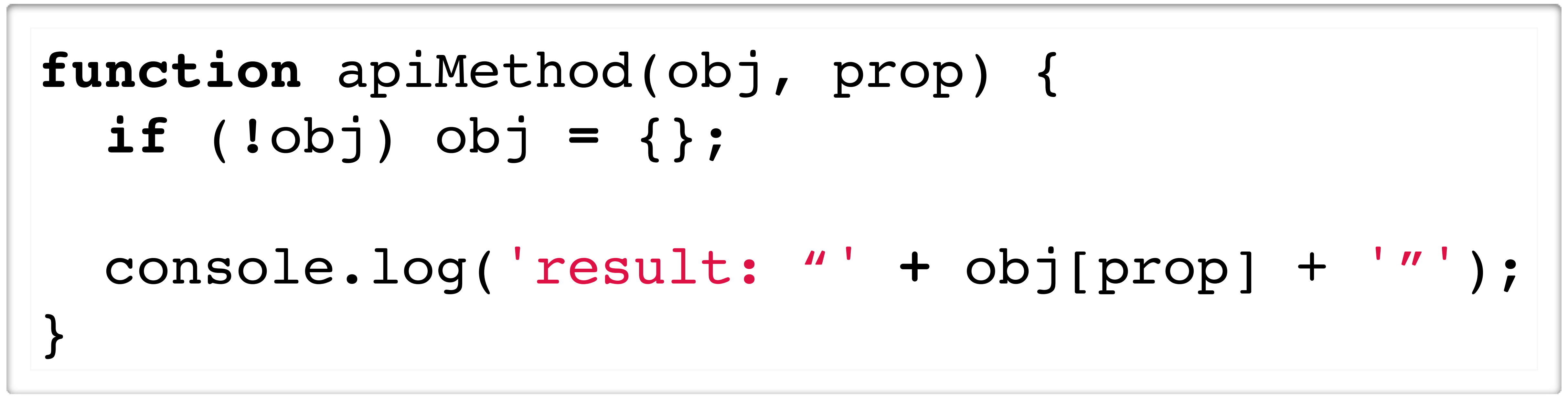
# The beginning

# Javascript



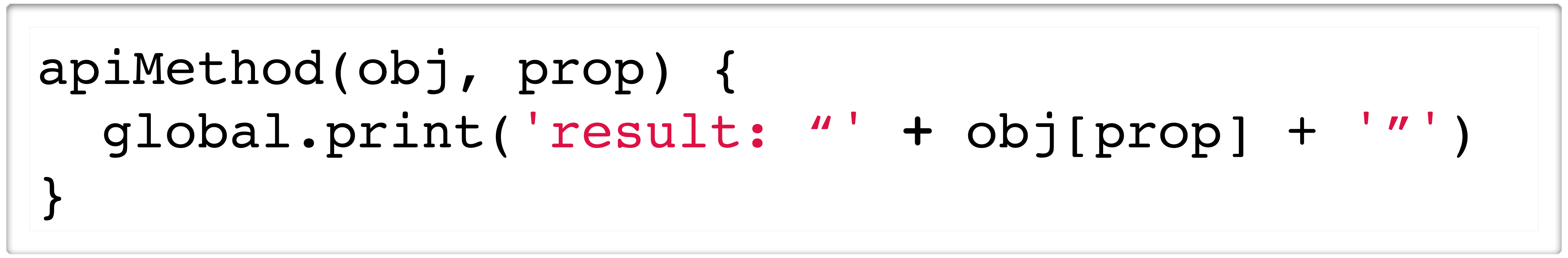
```
function apiMethod(obj, prop) {
  if (!obj) obj = {};

  console.log('result: “' + obj[prop] + '”');
}
```

## Javascript



```
function apiMethod(obj, prop) {
  if (!obj) obj = {};

  console.log('result: “' + obj[prop] + '”');
}
```

## Candor



```
apiMethod(obj, prop) {
  global.print('result: “' + obj[prop] + '”')
}
```

## Javascript

```
function apiMethod(obj, prop) {
  if (!obj) obj = {};

  console.log('result: “' + obj[prop] + '”');
}
```

## Candor

```
apiMethod(obj, prop) {
  global.print('result: “' + obj[prop] + '”')
}
```

## Outputs

```
apiMethod({}, 'wat?') // 'result: "undefined"'
```

```
apiMethod({}, 'wat?') // 'result: ""'
```

# Candor is different!

*And yes, it looks and feels like javascript*

- No semicolons, new-lines are significant
- No `**function**` keyword
- No exceptions
- No prototype-chains

# Candor is different!

*And yes, it looks and feels like javascript*

- No semicolons
- No `**function**` keyword
- No prototype-chains
- No exceptions
- **NO MORE GLOBAL LEAKS**

# What’s inside?

- Simple syntax, minimal amount of keywords and no reserved words!
- Compiler-friendly language, possibility of various optimizations
- ECMAScript-like semantics

# Why Candor?

# Why Candor?

*No, that's not about naming*

# ECMAScript is overcomplicated

Not at useful for server-side
as it was

Something different should be created

- No need in compatibility with legacy code
- Flexible specification
- OpenSource

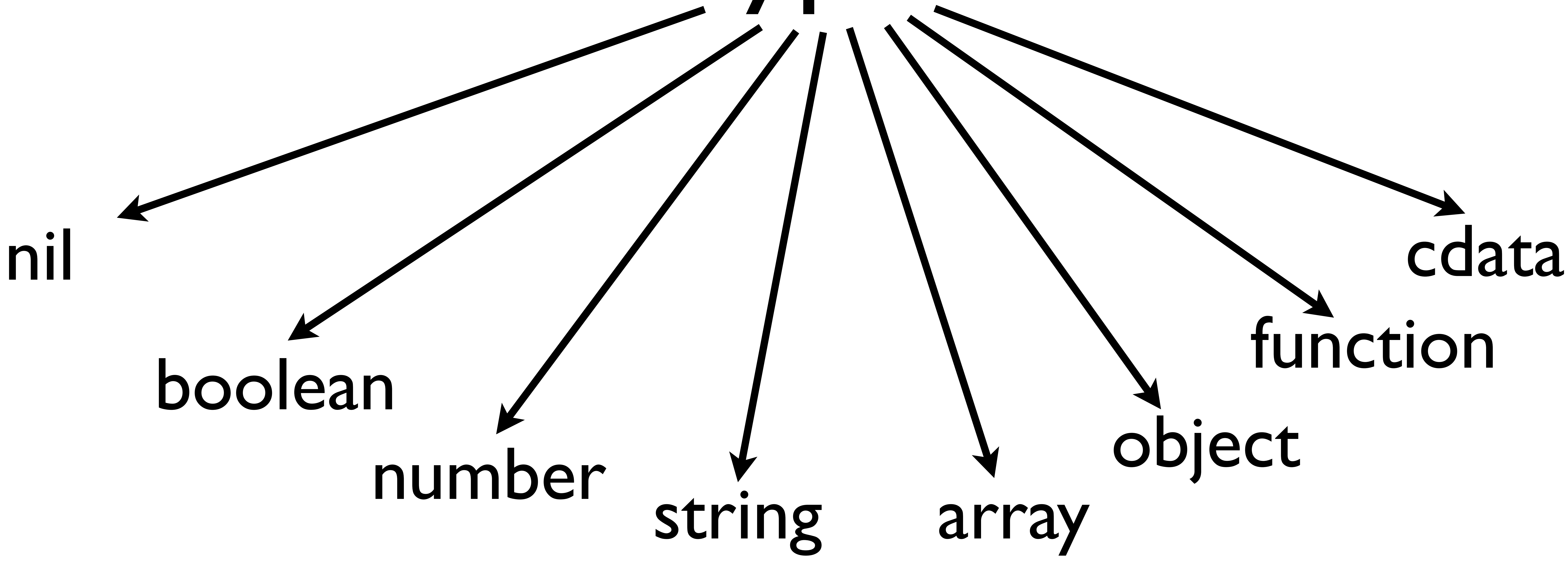
# Candor

- No need in compatibility with legacy code
- Flexible specification
- OpenSource

# Candor

## 1-minute syntax crash-course

# Types



nil

boolean

number

string

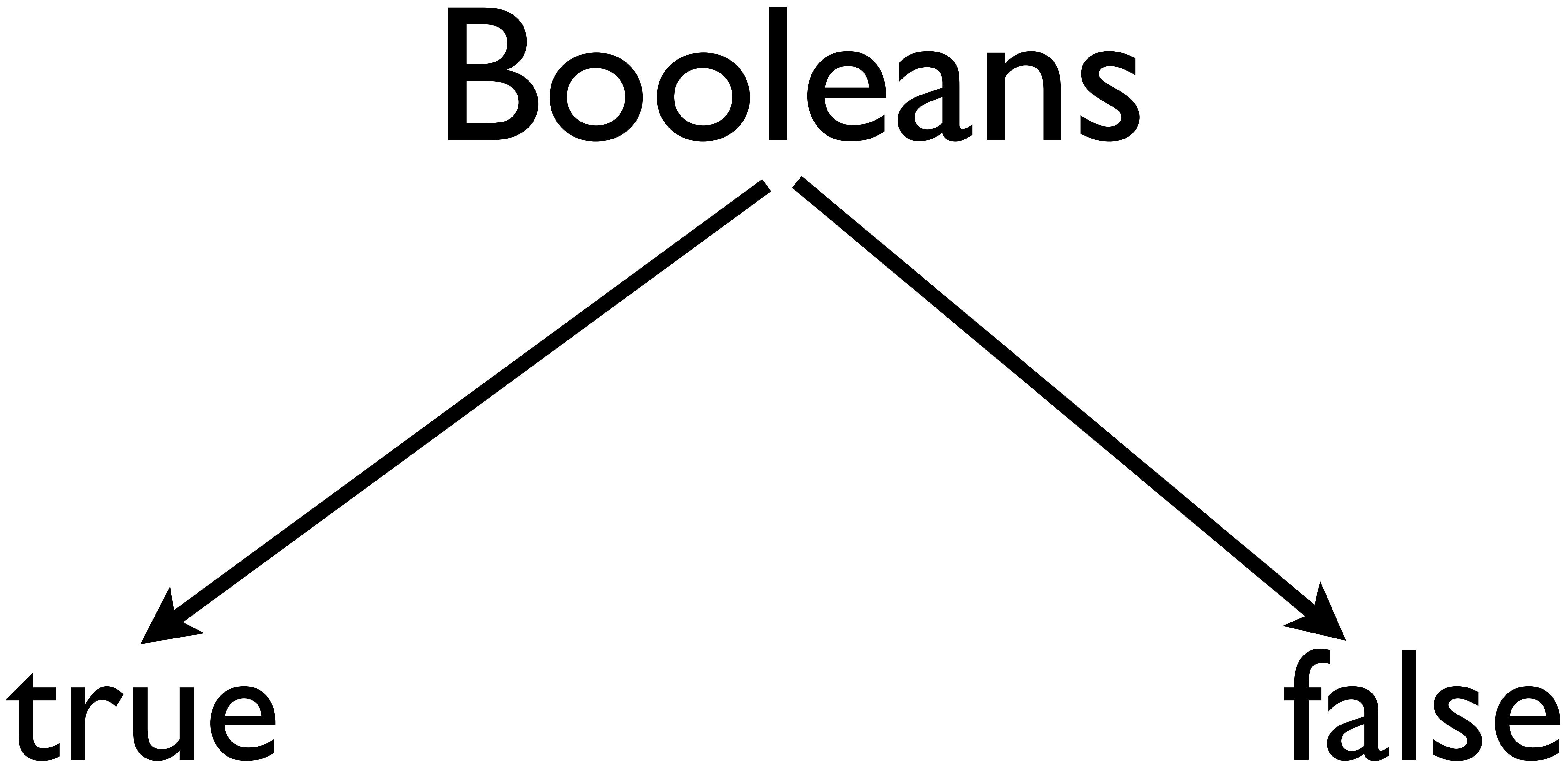
array

object

function

cdata

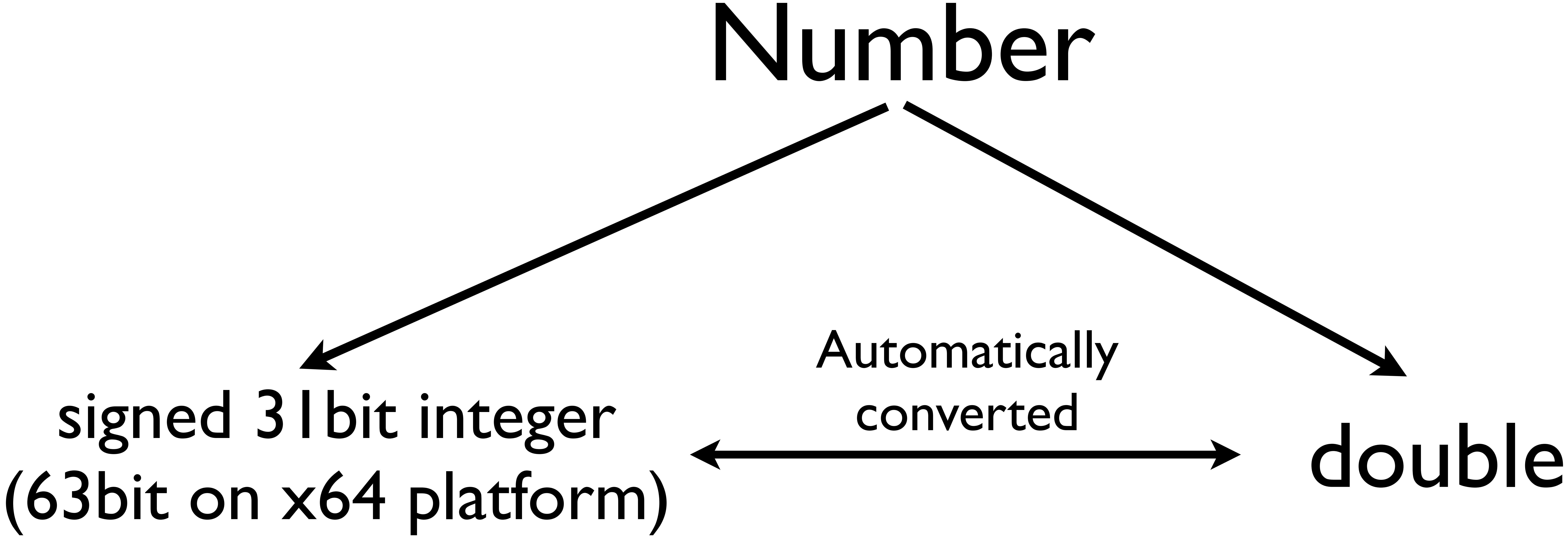
nil <=> undefined



Booleans

true

false



# Number

signed 31bit integer
(63bit on x64 platform)

Automatically
converted

double

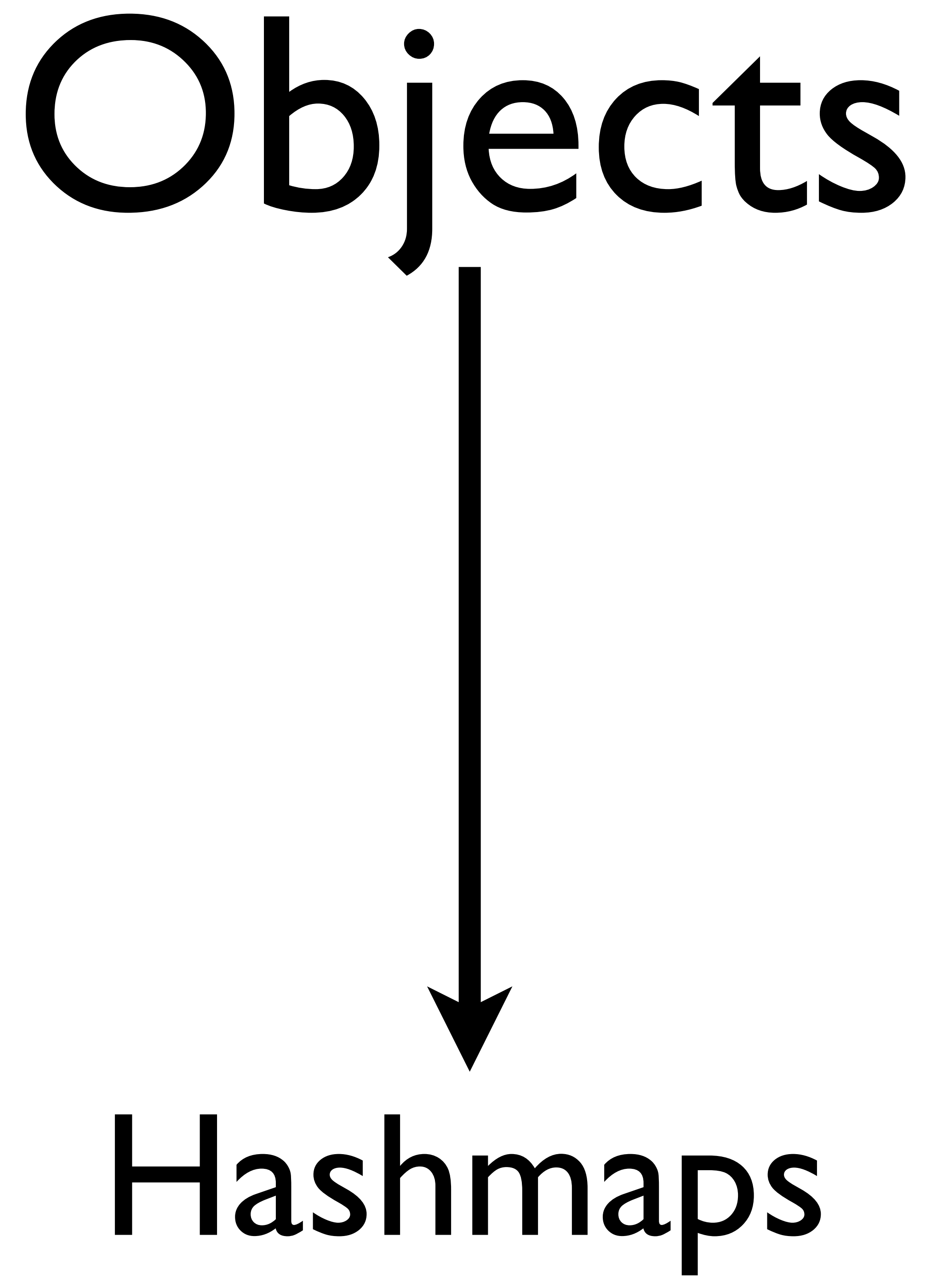
Examples: `123`, `123.456`

# Strings

↓

Just strings?

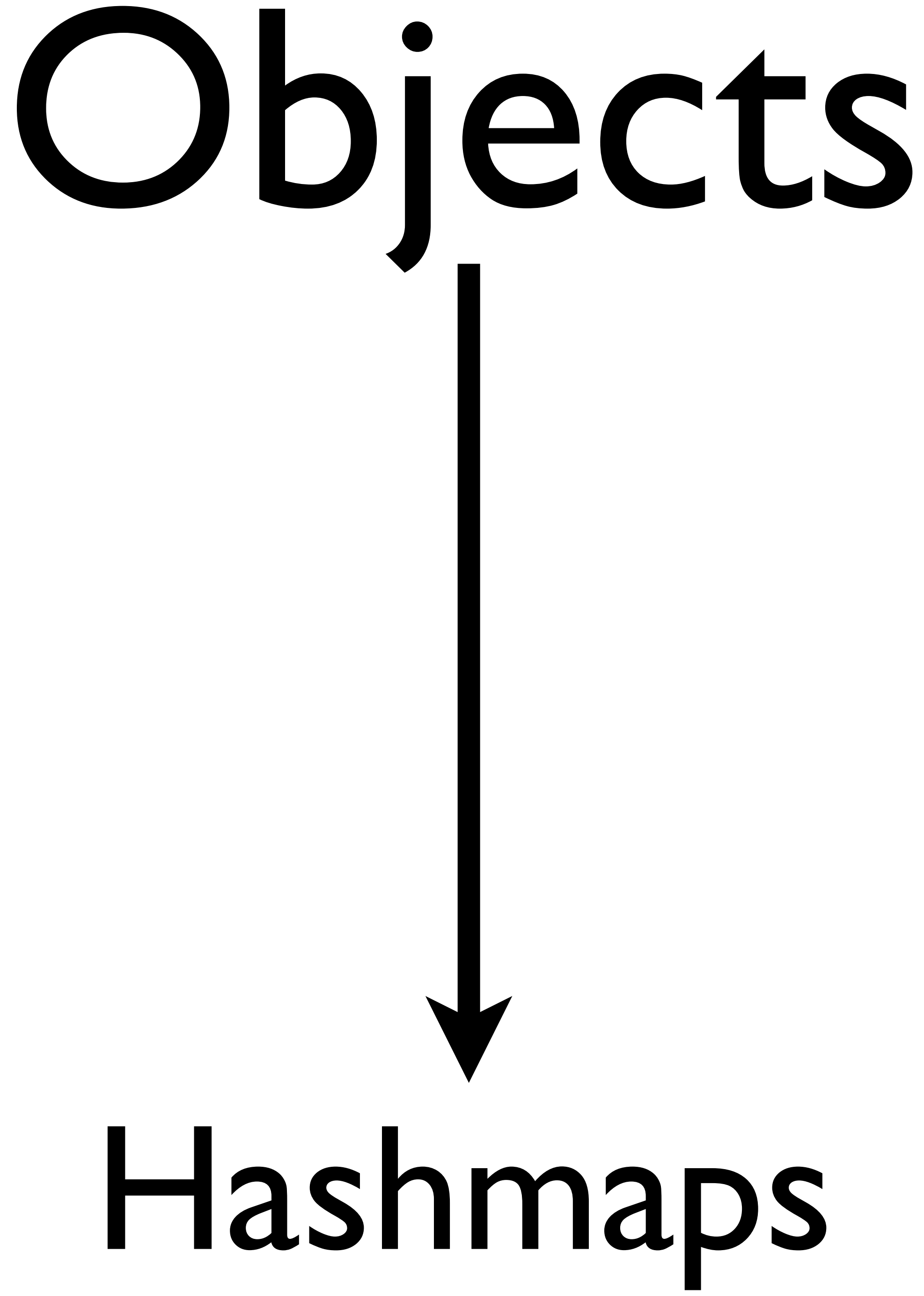
Example: “hello world”



# Objects

Hashmaps

Example: { a : 1, b: 2, “c”:  {} }



# Objects

Hashmaps

No language-specific properties with magic behaviour!

Example: { a : 1, b: 2, “c”:  {} }

# Arrays

Example: [1,2,3,”a”,”b”,{a:1,b:2,c:3}]

# Functions

Example: a(x,y,z) { return x + y + z }

# Functions with vararg

Example: a(x, y, z...) { return x + y + z[0] }

Call:  a(x, y, z...) or a(x, y, a, b, c)

# CData

Container for a C structure or pointer

# How I can interact with these types?

# How I can interact with these types?

Keywords

# How I can interact with these types?

Keywords

Binary operations

# How I can interact with these types?

Keywords

Binary operations

Unary operations

# How I can interact with these types?

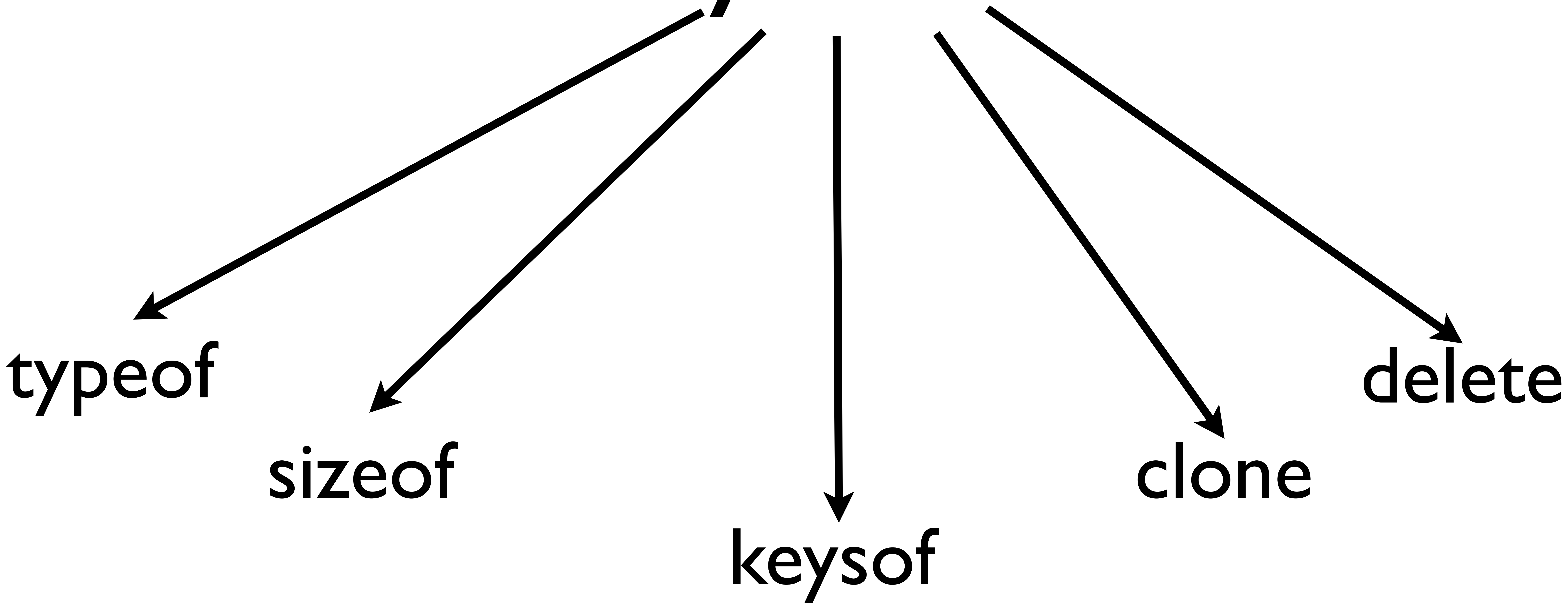
Keywords

Binary operations

Unary operations

Functions

# Keywords



typeof

sizeof

keysof

clone

delete

# typeof

```
typeof nil === “nil”
typeof 1 === “number”
typeof “value” === “string”
typeof {} === “object”
typeof [] === “array”
typeof () {} === “function”
typeof cdata === “cdata”
```

# sizeof

sizeof nil === 0, sizeof 1 === 0, ...

sizeof “value” === 5

sizeof [1,2,3] === 3

# keysof

keysof nil === [], keysof 1 === [], ...

keysof [1,2,3] === [0,1,2]

keysof {a:1,b:2} === [“a”, “b”]

# delete

delete obj.property

# clone

```
obj = { a: 1, b: 2 }
cobj = clone obj
print(cobj.a) // 1
```

===

!==

||

==

+

/

>>>

# Binary operations

<<

&&

Almost the same as in javascript

%

|

>>

^

&

*

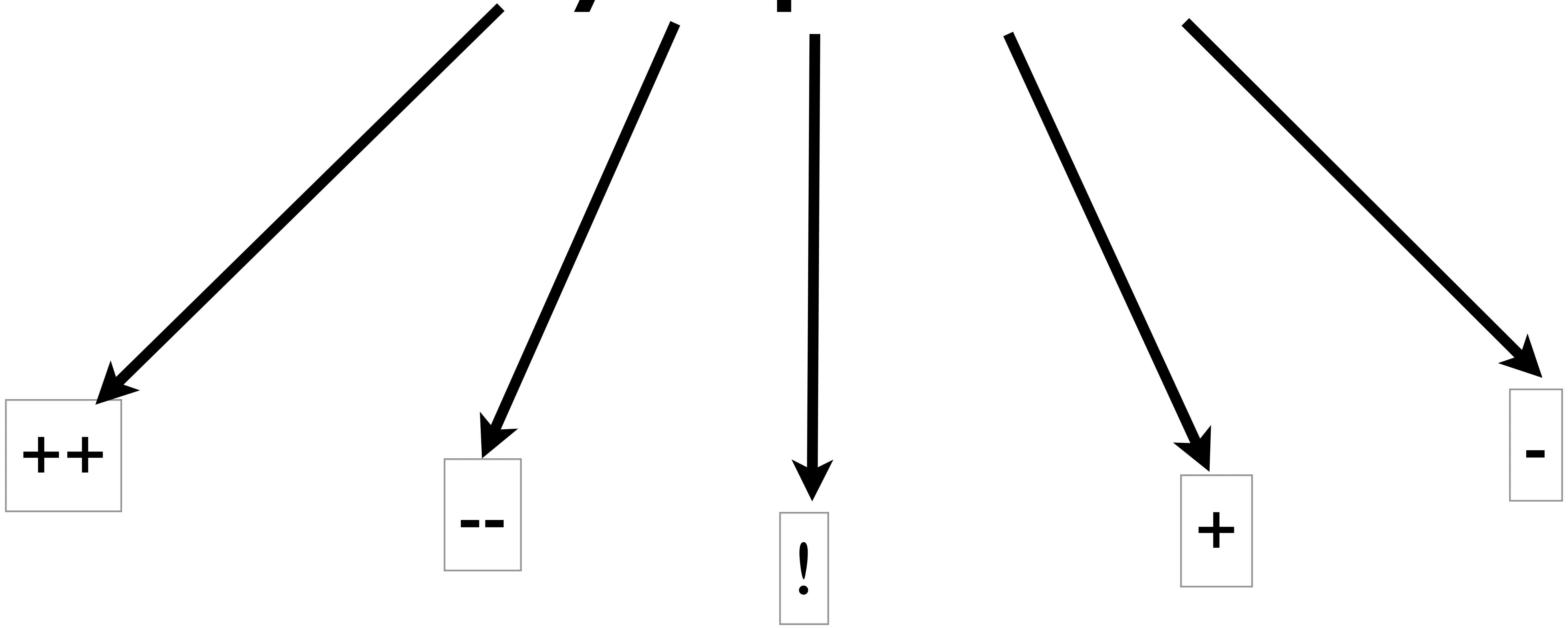
!=

-

# Type coercion

- coerce(nil, any) = (coerceTo(typeof any, nil), any)
- coerce(‘string’, other) = (‘string’, toString(other))
- coerce(boolean, other) = (boolean, toBoolean(other))
- coerce(number, other) = (number, toNumber(other))
- coerce(function|object|array|cdata:first, other) = depending on operation:
    - (toString(first), toString(other))
    - (toNumber(first), toNumber(other))

# Unary operations



++

--

!

+

-

# Control flow syntax

```
if (a) {
  // when a == true
} else {
  // when a == false
}
```

```
while (a) {
  // code
}
```

# Control flow syntax

```
if (a) {
  // when a == true
} else {
  // when a == false
}
```

```
while (a) {
  // code
}
```

No **for** loops!

# Array wrapper example

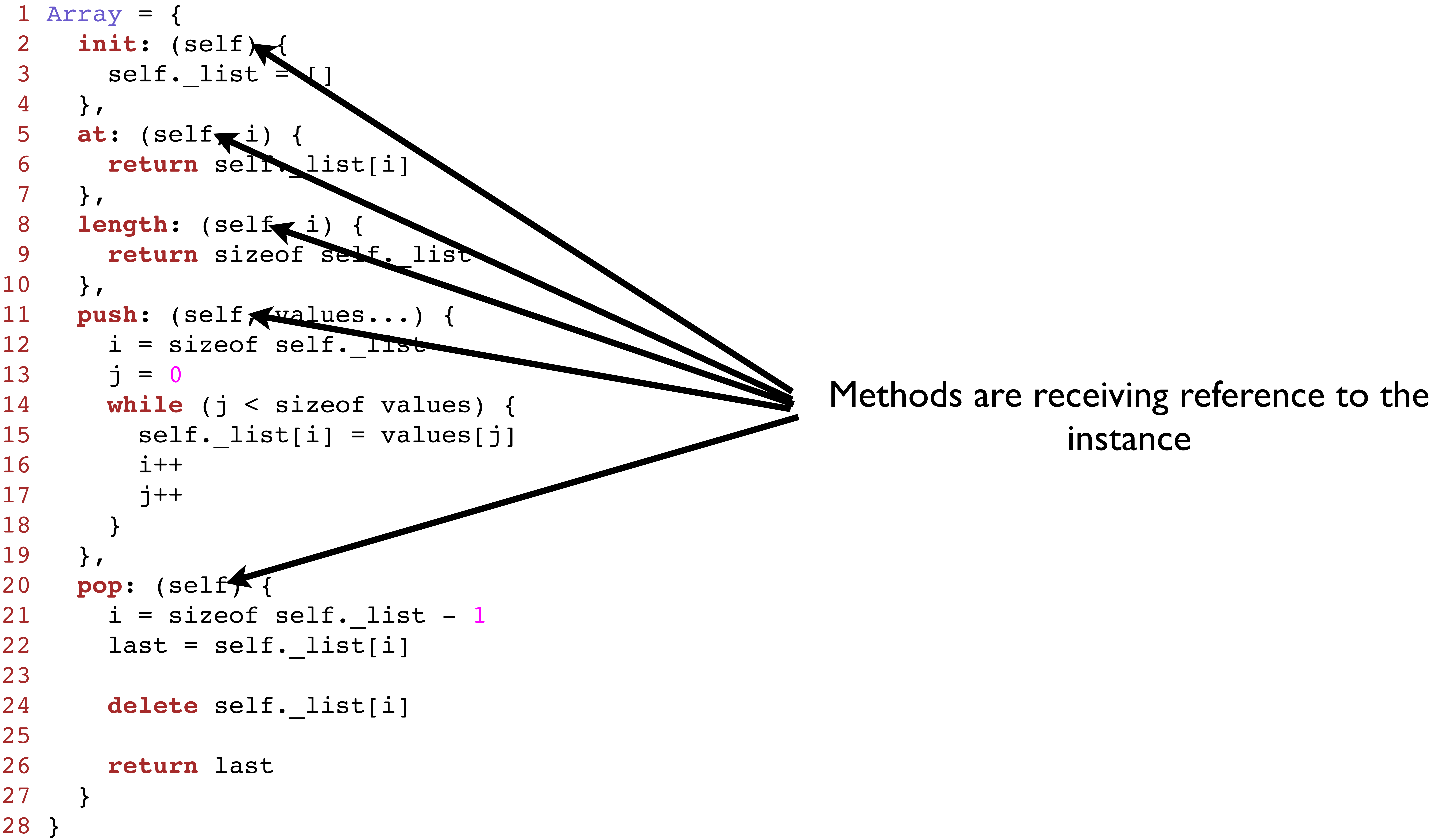
```
Array = {
  init: (self) {
    self._list = []
  },
  at: (self, i) {
    return self._list[i]
  },
  length: (self, i) {
    return sizeof self._list
  },
  push: (self, values...) {
    i = sizeof self._list
    j = 0
    while (j < sizeof values) {
      self._list[i] = values[j]
      i++
      j++
    }
  },
  pop: (self) {
    i = sizeof self._list - 1
    last = self._list[i]

    delete self._list[i]

    return last
  }
}
```

# Wrapper is an object

```
Array = {
  init: (self) {
    self._list = []
  },
  at: (self, i) {
    return self._list[i]
  },
  length: (self, i) {
    return sizeof self._list
  },
  push: (self, values...) {
    i = sizeof self._list
    j = 0
    while (j < sizeof values) {
      self._list[i] = values[j]
      i++
      j++
    }
  },
  pop: (self) {
    i = sizeof self._list - 1
    last = self._list[i]

    delete self._list[i]

    return last
  }
}
```

```
Array = {
  init: (self) {
    self._list = []
  },
  at: (self, i) {
    return self._list[i]
  },
  length: (self, i) {
    return sizeof self._list
  },
  push: (self, values...) {
    i = sizeof self._list
    j = 0
    while (j < sizeof values) {
      self._list[i] = values[j]
      i++
      j++
    }
  },
  pop: (self) {
    i = sizeof self._list - 1
    last = self._list[i]

    delete self._list[i]

    return last
  }
}
```

Constructor/Initializer



```
Array = {
  init: (self) {
    self._list = []
  },
  at: (self, i) {
    return self._list[i]
  },
  length: (self, i) {
    return sizeof self._list
  },
  push: (self, values...) {
    i = sizeof self._list
    j = 0
    while (j < sizeof values) {
      self._list[i] = values[j]
      i++
      j++
    }
  },
  pop: (self) {
    i = sizeof self._list - 1
    last = self._list[i]

    delete self._list[i]

    return last
  }
}
```

Methods are receiving reference to the instance

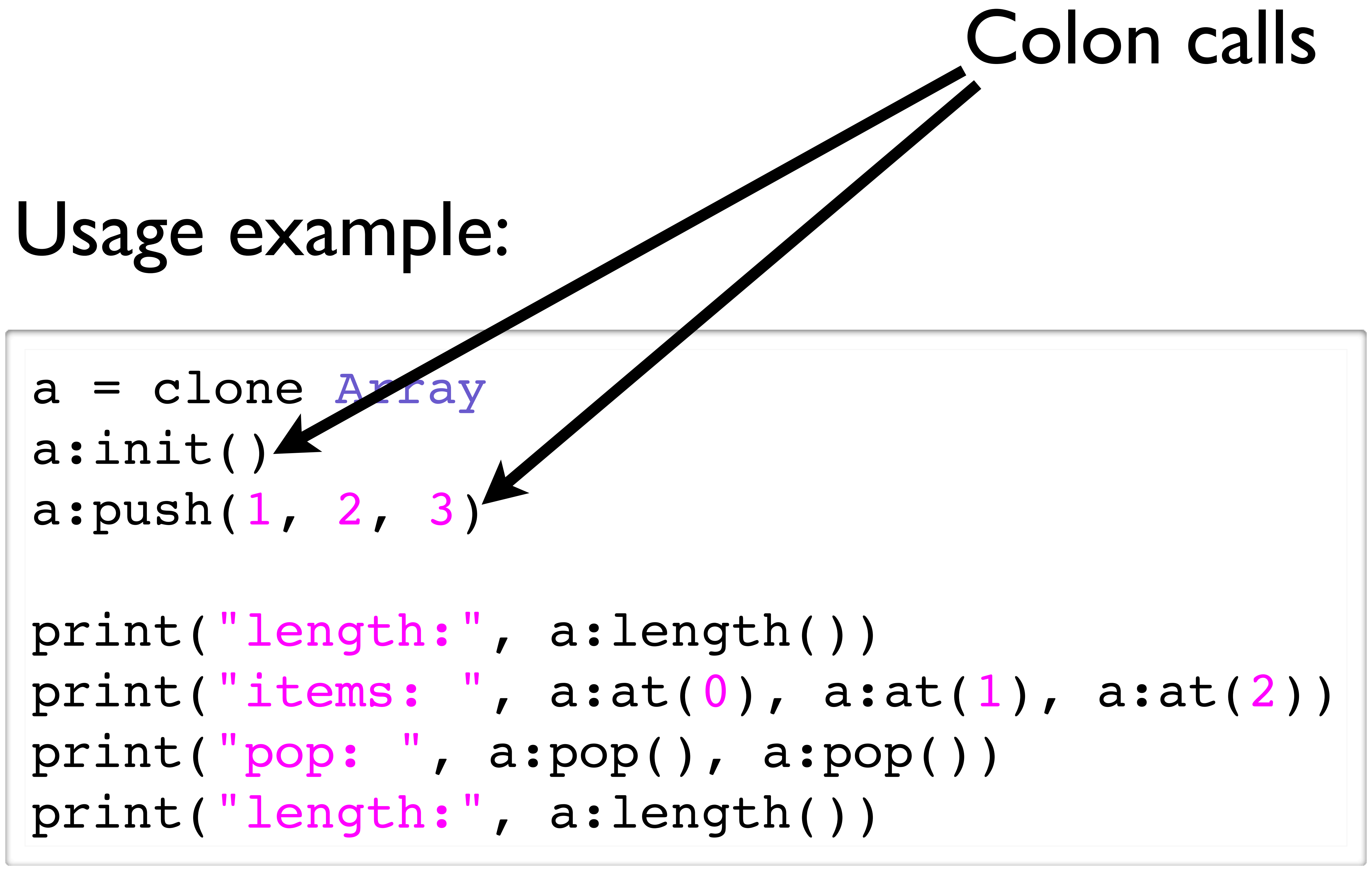
```
Array = {
  init: (self) {
    self._list = []
  },
  at: (self, i) {
    return self._list[i]
  },
  length: (self, i) {
    return sizeof self._list
  },
  push: (self, values...) {
    i = sizeof self._list
    j = 0
    while (j < sizeof values) {
      self._list[i] = values[j]
      i++
      j++
    }
  },
  pop: (self) {
    i = sizeof self._list - 1
    last = self._list[i]

    delete self._list[i]

    return last
  }
}
```

## Usage example:

```
a = clone Array
a:init()
a:push(1, 2, 3)

print("length:", a:length())
print("items: ", a:at(0), a:at(1), a:at(2))
print("pop: ", a:pop(), a:pop())
print("length:", a:length())
```

```
Array = {
  init: (self) {
    self._list = []
  },
  at: (self, i) {
    return self._list[i]
  },
  length: (self, i) {
    return sizeof self._list
  },
  push: (self, values...) {
    i = sizeof self._list
    j = 0
    while (j < sizeof values) {
      self._list[i] = values[j]
      i++
      j++
    }
  },
  pop: (self) {
    i = sizeof self._list - 1
    last = self._list[i]

    delete self._list[i]

    return last
  }
}
```

Usage example:



```
a = clone Array
a:init()
a:push(1, 2, 3)

print("length:", a:length())
print("items: ", a:at(0), a:at(1), a:at(2))
print("pop: ", a:pop(), a:pop())
print("length:", a:length())
```

```
Array = {
  init: (self) {
    self._list = []
  },
  at: (self, i) {
    return self._list[i]
  },
  length: (self, i) {
    return sizeof self._list
  },
  push: (self, values...) {
    i = sizeof self._list
    j = 0
    while (j < sizeof values) {
      self._list[i] = values[j]
      i++
      j++
    }
  },
  pop: (self) {
    i = sizeof self._list - 1
    last = self._list[i]

    delete self._list[i]

    return last
  }
}
```

a:init() <=> a.init(a)

## Usage example:

```
a = clone Array
a.init(a)
a.push(a, 1, 2, 3)

print("length:", a.length(a))
print("items: ", a.at(a, 0),
                 a.at(a, 1),
                 a.at(a, 2))
print("pop: ", a.pop(a), a.pop(a))
print("length:", a.length(a))
```

```
Array = {
  init: (self) {
    self._list = []
  },
  at: (self, i) {
    return self._list[i]
  },
  length: (self, i) {
    return sizeof self._list
  },
  push: (self, values...) {
    i = sizeof self._list
    j = 0
    while (j < sizeof values) {
      self._list[i] = values[j]
      i++
      j++
    }
  },
  pop: (self) {
    i = sizeof self._list - 1
    last = self._list[i]

    delete self._list[i]

    return last
  }
}
```

## Usage example:

```
a = clone Array
a.init(a)
a.push(a, 1, 2, 3)

print("length:", a.length(a))
print("items: ", a.at(a, 0),
                 a.at(a, 1),
                 a.at(a, 2))
print("pop: ", a.pop(a), a.pop(a))
print("length:", a.length(a))
```

Very Lua-like

# How can I run Candor code?

# How can I run Candor code?

- Using website ( http://candor-lang.org/ )
- Using Candor.js ( https://github.com/creationix/candor.js )
- Using development version of JIT VM ( https://github.com/indutny/candor )

# Wait, have you said JIT VM?

```
x00000001000d5688: rex.W test $0x1,%al
x00000001000d568c: jne    0x1000d56a6
x00000001000d5692: add    $0x2,%rax
x00000001000d5699: jno    0x1000d56cf
x00000001000d569f: sub    $0x2,%rax
x00000001000d56a6: mov    %rax,-0x10(%rbp)
x00000001000d56ad: movabs $0x2,%rax
x00000001000d56b7: mov    %rax,%rbx
x00000001000d56ba: mov    -0x10(%rbp),%rax
x00000001000d56c1: movabs $0x1000d5320,%r11
x00000001000d56cb: rex.WB callq *%r11
x00000001000d56ce: nop
x00000001000d56cf: mov    %rax,-0x10(%rbp)
x00000001000d56d6: mov    -0x10(%rbp),%r11
x00000001000d56dd: mov    %r11,-0x8(%rbp)
x00000001000d56e4: mov    %r11,%rax
```

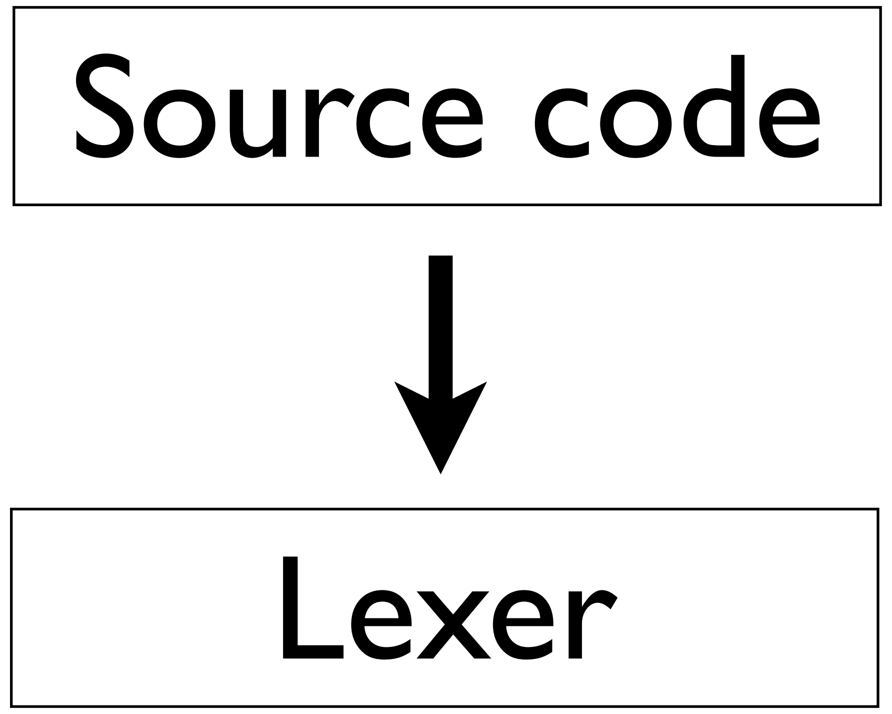
Yes!

Candor is not just another language that compiles to javascript

# VM

Compiler + Runtime + Heap + GC

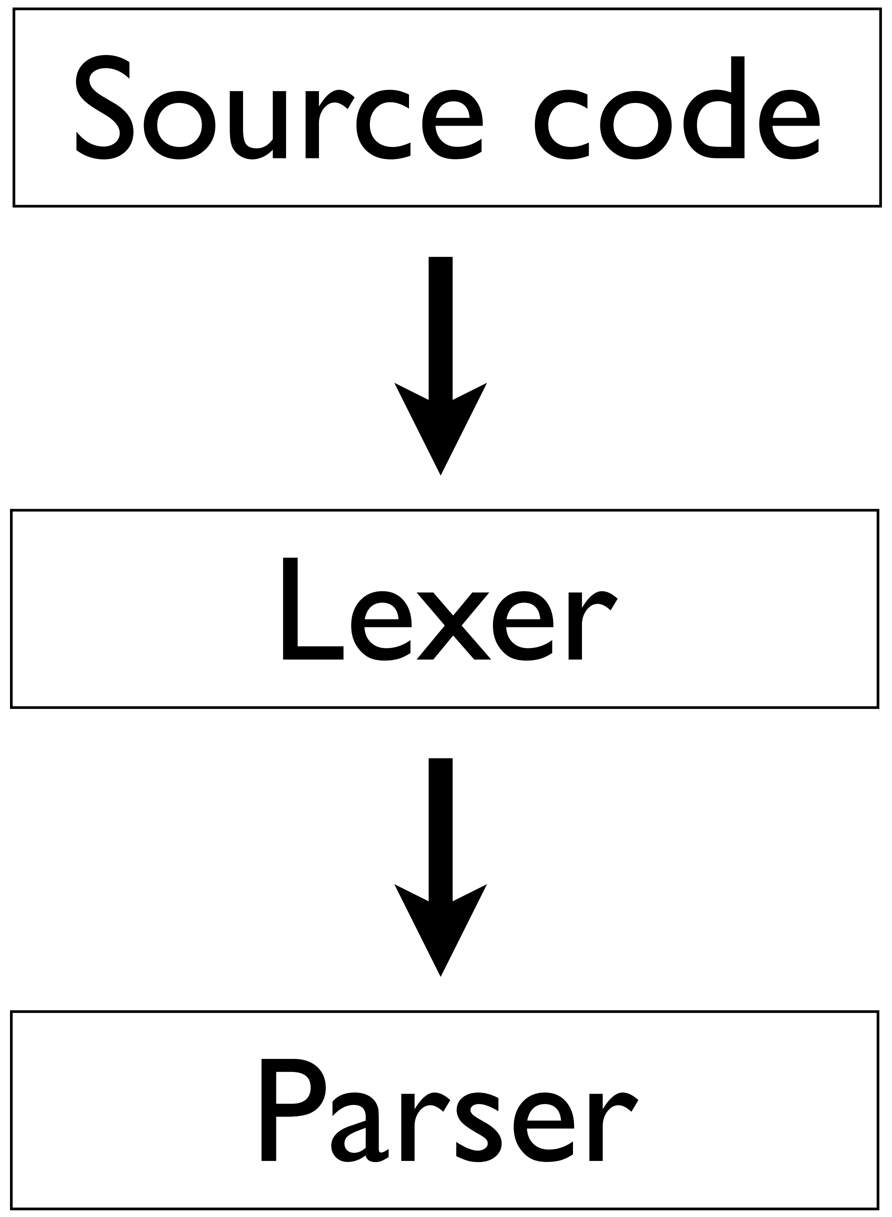
# Compiler’s structure



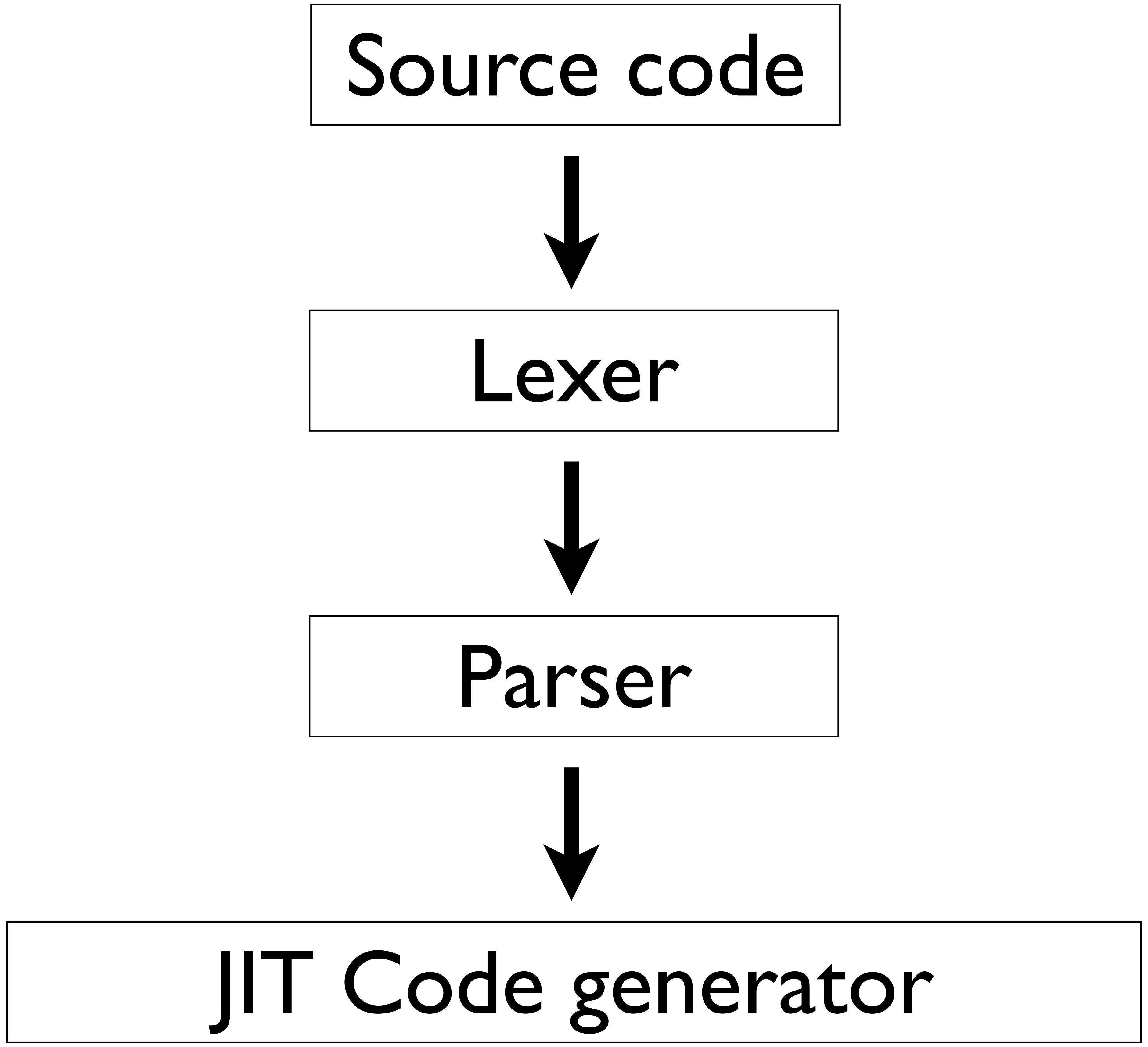
Source code

Lexer

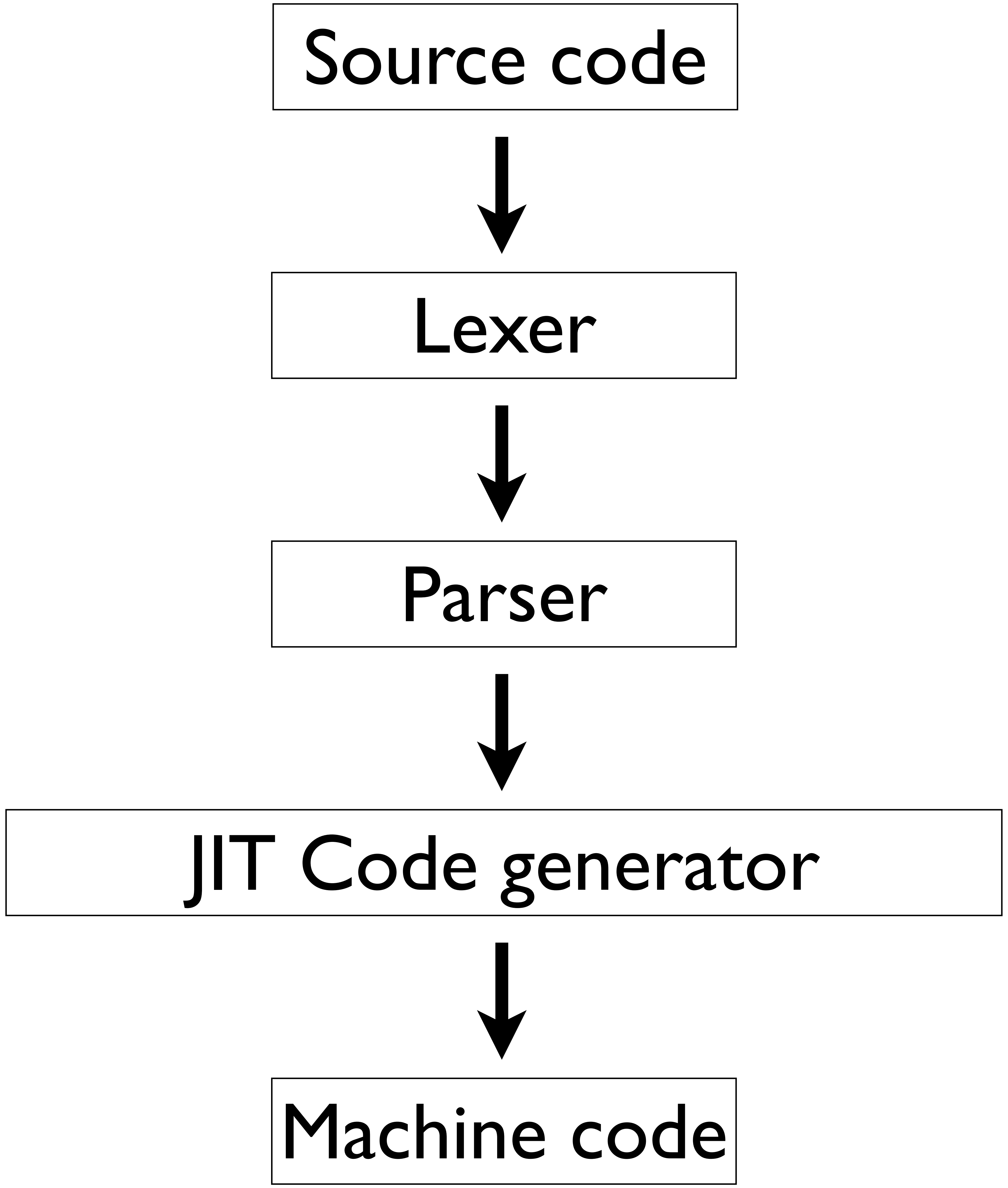
# Compiler’s structure



# Compiler's structure



# Compiler's structure



Source code

Lexer

Parser

JIT Code generator

Machine code

# It wasn’t really fast

```
#> time ./can test/benchmarks/while.can
real 0m7.079s
user 0m7.072s
sys 0m0.006s

#> time node while.js
real 0m1.492s
user 0m1.482s
sys 0m0.020s
```

# It wasn’t really fast

```
#> time ./can test/benchmarks/while.can
real 0m7.079s
user 0m7.072s
sys 0m0.006s

#> time node while.js
real 0m1.492s
user 0m1.482s
sys 0m0.020s
```

x5 times slower!

# WARNING!

Assembly code in next slide!

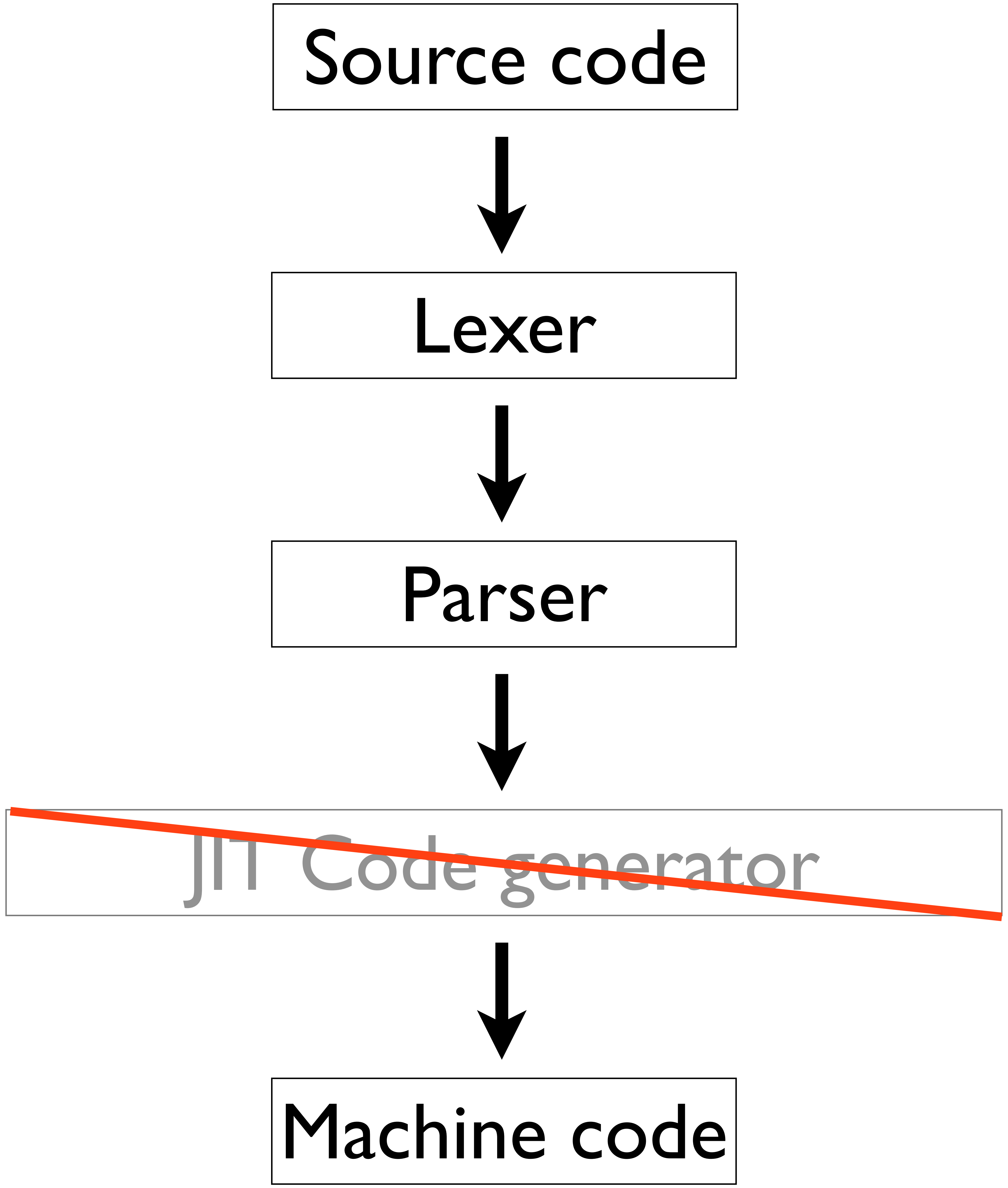
```
x00000001000d5688: rex.W test $0x1,%al
x00000001000d568c: jne    0x1000d56a6
x00000001000d5692: add    $0x2,%rax
x00000001000d5699: jno    0x1000d56cf
x00000001000d569f: sub    $0x2,%rax
x00000001000d56a6: mov    %rax,-0x10(%rbp)
x00000001000d56ad: movabs $0x2,%rax
x00000001000d56b7: mov    %rax,%rbx
x00000001000d56ba: mov    -0x10(%rbp),%rax
x00000001000d56c1: movabs $0x1000d5320,%r11
x00000001000d56cb: rex.WB callq *%r11
x00000001000d56ce: nop
x00000001000d56cf: mov    %rax,-0x10(%rbp)
x00000001000d56d6: mov    -0x10(%rbp),%r11
x00000001000d56dd: mov    %r11,-0x8(%rbp)
x00000001000d56e4: mov    %r11,%rax
```

```
x00000001000d5688: rex.W test $0x1,%al
x00000001000d568c: jne    0x1000d56a6
x00000001000d5692: add    $0x2,%rax
x00000001000d5699: jno    0x1000d56cf
x00000001000d569f: sub    $0x2,%rax
x00000001000d56a6: mov    %rax,-0x10(%rbp)
x00000001000d56ad: movabs $0x2,%rax
x00000001000d56b7: mov    %rax,%rbx
x00000001000d56ba: mov    -0x10(%rbp),%rax
x00000001000d56c1: movabs $0x1000d5320,%r11
x00000001000d56cb: rex.WB callq *%r11
x00000001000d56ce: nop
x00000001000d56cf: mov    %rax,-0x10(%rbp)
x00000001000d56d6: mov    -0x10(%rbp),%r11
x00000001000d56dd: mov    %r11,-0x8(%rbp)
x00000001000d56e4: mov    %r11,%rax
```

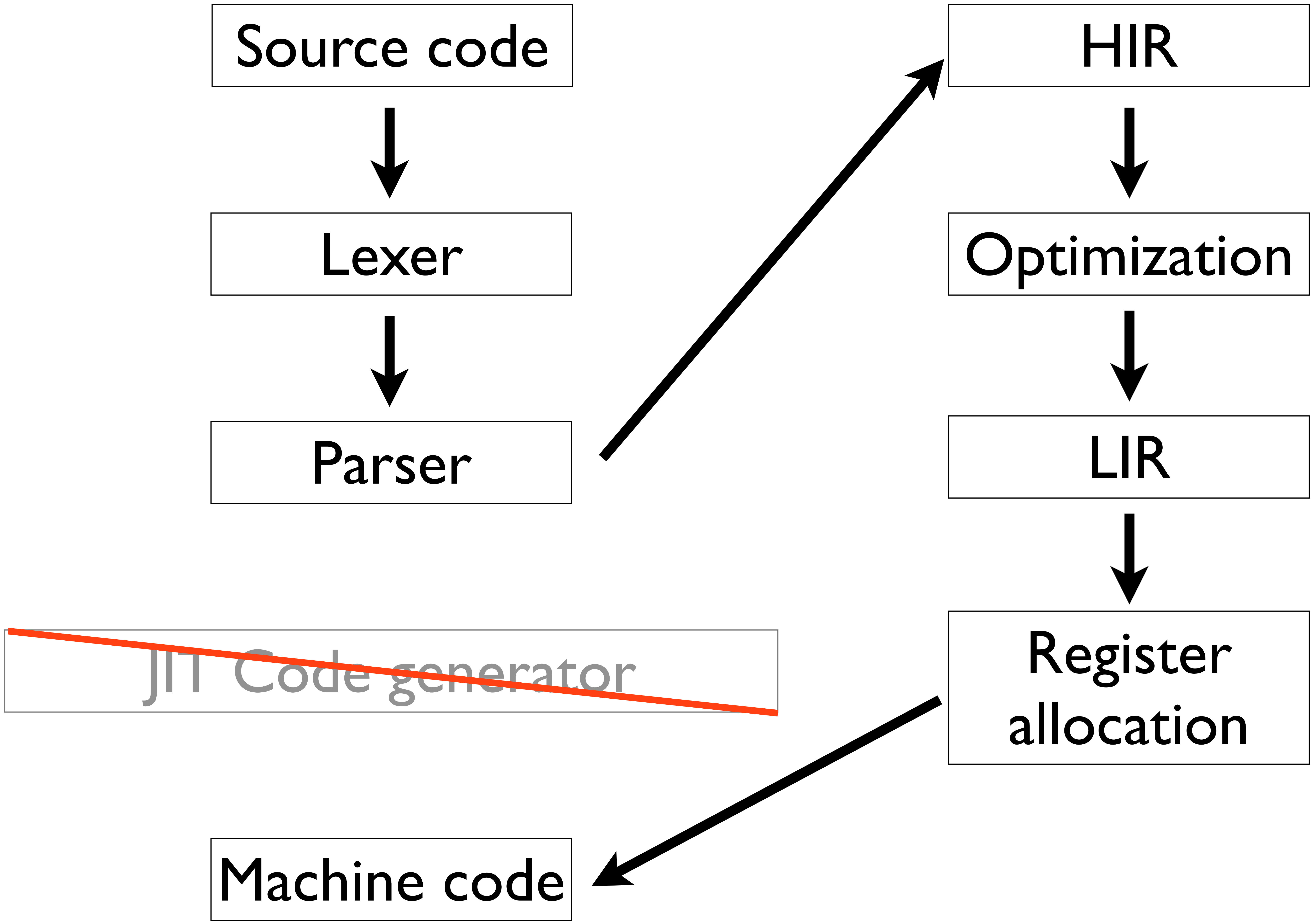
# And that’s just “i + 1”

Good news:
New compiler is in development!

# Compiler's structure



# Compiler's structure



Source code

Lexer

Parser

HIR

Optimization

LIR

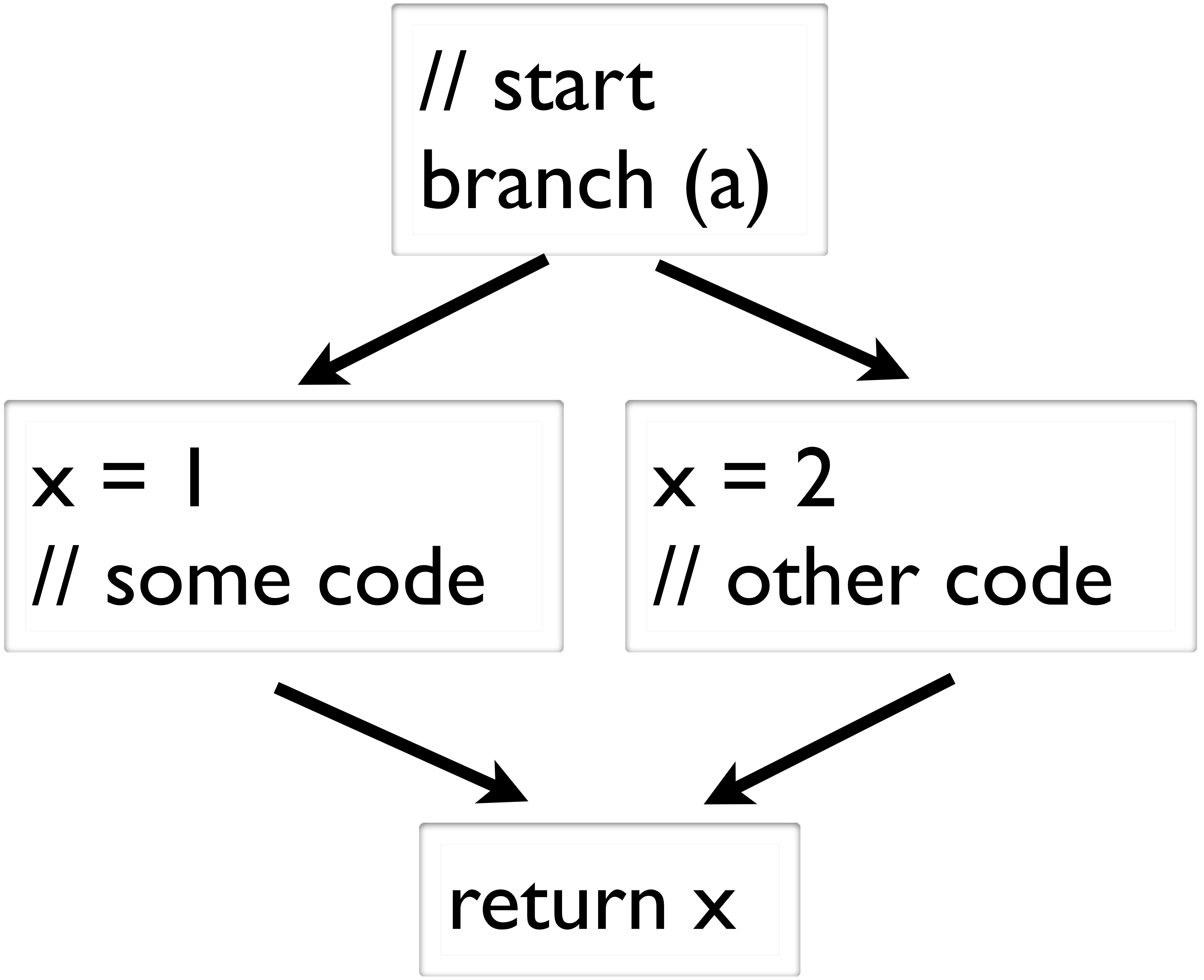
Register allocation

~~JIT Code generator~~

Machine code

# HIR (CFG)

```
// start
if (a) {
  x = 1
  // some code
} else {
  x = 2
  // other code
}
return x
```



// start
branch (a)

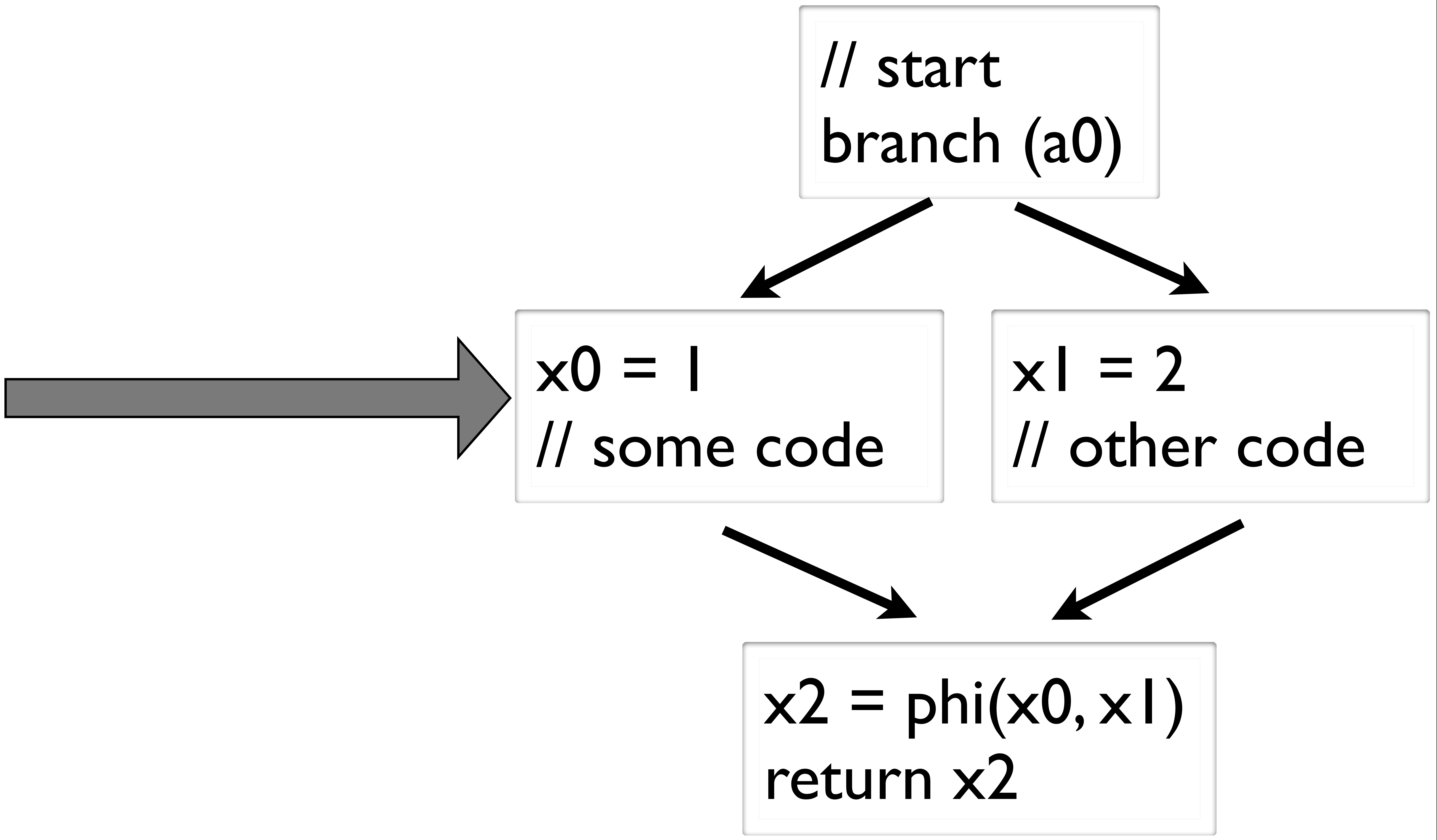
x = 1
// some code

x = 2
// other code

return x

# HIR (SSA)

```
// start
if (a) {
  x = 1
  // some code
} else {
  x = 2
  // other code
}
return x
```



```
// start
branch (a0)
```

```
x0 = 1
// some code
```

```
x1 = 2
// other code
```

```
x2 = phi(x0, x1)
return x2
```

# Optimizations

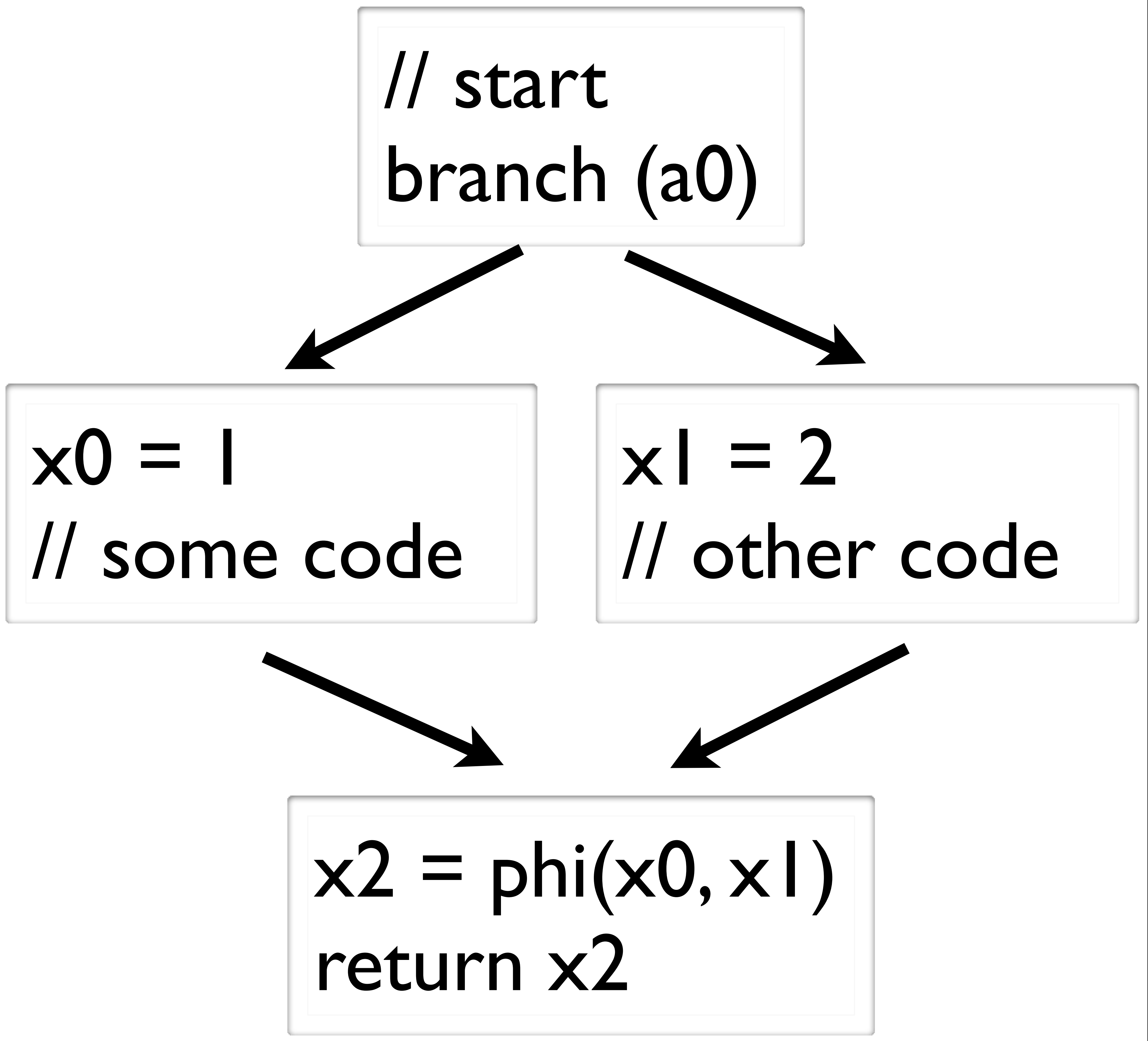
- Dead-code elimination
- Common subexpression elimination
- Hoisting code out of hot loop
- And some others

# LIR

Back from graph to the linear representation

# LIR

```
cmp r0
jz &else_branch

mov 1, r1
jmp &end_if

else_branch:

mov 2, r2

end_if:
// resolved phi
mov r3, %ax
ret
```



```
// start
branch (a0)
```

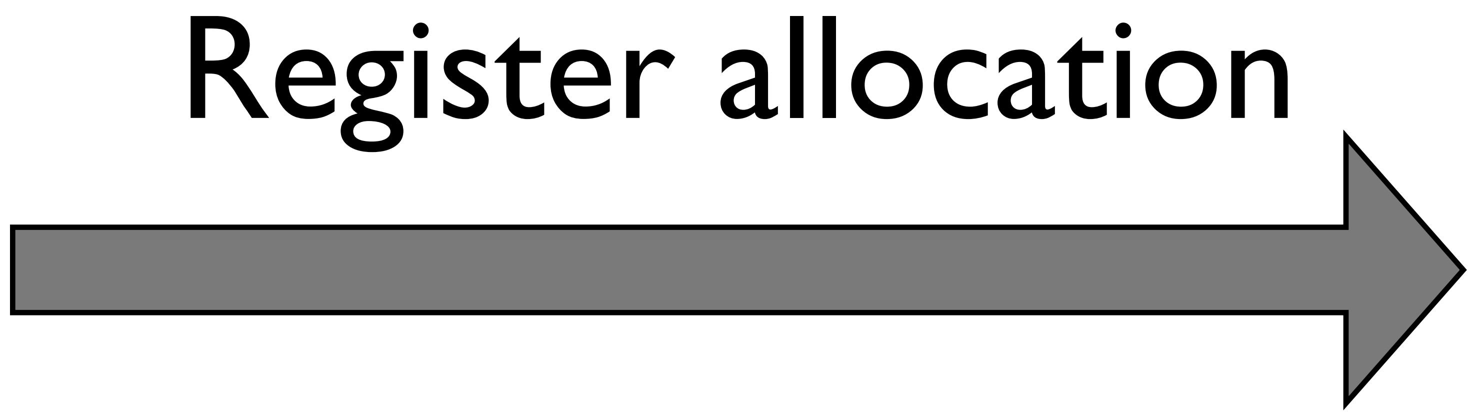
```
x0 = 1
// some code
```

```
x1 = 2
// other code
```

```
x2 = phi(x0, x1)
return x2
```

# LIR

```
cmp r0
jz &else_branch

mov 1, r1
jmp &end_if

else_branch:

mov 2, r2

end_if:
// resolved phi
mov r3, %ax
ret
```



Register allocation

```
cmp %bx
jz &else_branch

mov 1, %eax
jmp &end_if

else_branch:

mov 2, %eax

end_if:
ret
```

# Feel free to contribute!

To `feature-ssa` branch on github:

https://github.com/indutny/candor/tree/feature-ssa

# Future plans:

- Finishing feature-ssa branch of compiler
- Implementing SSA form optimizations
- Creating debugger for JIT VM
- And improving language!

http://candor-lang.org/

# Thank you!

Sorry, no Q&A