Part 1

Identifier Address

Suppose that

```
{let {[x 88]}
  {+ x y}}
```

appears in a program; the body is eventually evaluated:

```
...
```

```
{+ x y}
```

where will **`x`** be in the environment?

Answer: always at the beginning:

```
x = 88    ...
```

Identifier Address

Suppose that

```
{let {[y 1]}
  {+ x y}}
```

appears in a program; the body is eventually evaluated:

```
...
{+ x y}
```

where will `y` be in the environment?

Answer: always at the beginning:

```
y = 1    ...
```

Identifier Address

Suppose that

```
{let {[y 1]}
  {let {[x 2]}
    {+ x y}}}
```

appears in a program; the body is eventually evaluated:

```
...
{+ x y}
```

where will `y` be in the environment?

Answer: always second:

```
x = 2    y = 1    ...
```

Identifier Address

Suppose that

```
{let {[y 1]}
  {let {[x 88]}
    {* {+ x y} 17}}}
```

appears in a program; the body is eventually evaluated:

```
...
{+ x y}
```

where will **`x`** and **`y`** be in the environment?

Answer: always first and second:

```
x = 88    y = 1    ...
```

Identifier Address

Suppose that

```
{let {[y 1]}
  {let {[w 10]}
    {let {[z 9]}
      {let {[x 0]}
        {+ x y}}}}}
```

appears in a program; the body is eventually evaluated:

```
...
{+ x y}
```

where will **x** and **y** be in the environment?

Answer: always first and fourth:

```
x = 0   z = 9   w = 10   y = 1   ...
```

Identifier Address

Suppose that

```
{let {[y {let {[r 9]} {* r 8}}]}
  {let {[w 10]}
    {let {[z {let {[q 9]} q}]}
      {let {[x 0]}
        {+ x y}}}}}
```

appears in a program; the body is eventually evaluated:

```
...
{+ x y}
```

where will **x** and **y** be in the environment?

Answer: always first and fourth:

```
x = 0    z = 9    w = 10    y = 1    ...
```

Lexical Scope

- For any expression, we can tell which identifiers will be in the environment at run time
- The order of the environment is predictable

Part 2

Compilation of Variables

A compiler can transform an **`ExprC`** expression to an expression without identifiers — only lexical addresses

```
; compile : ExprC ... -> ExprD
```

```
(define-type ExprC
  [numC (n : number)]
  [addC (l : ExprC)
        (r : ExprC)]
  [multC (l : ExprC)
         (r : ExprC)]
  [idC (n : symbol)]
  [lamC (n : symbol)
        (body : ExprC)]
  [appC (fun : ExprC)
        (arg : ExprC)])
```

```
(define-type ExprD
  [numD (n number)]
  [addD (l : ExprD)
        (r : ExprD)]
  [multD (l : ExprD)
         (r : ExprD)]
  [atD (pos : number)]
  [lamD (body : ExprD)]
  [appD (fun : ExprD)
        (arg : ExprD)])
```

Compile Examples

```
(compile [1] ...) ⇒ [1]

(compile [{+ 1 2}] ...) ⇒ [{+ 1 2}]

(compile [x] ...) ⇒ compile: free identifier

(compile [{lambda {x} {+ 1 x}}] ...)
⇒ [{lambda {+ 1 {at 0}}}]

(compile [{lambda {y} {lambda {x} {+ x y}}}] ...)
  ⇒ [{lambda {lambda {+ {at 0} {at 1}}}}]
```

Implementing the Compiler

```
(define (compile [a : ExprC] [env : EnvC])
  (type-case ExprC a
    [numC (n) (numD n)]
    [plusC (l r) (plusD (compile l env)
                        (compile r env))]
    [multC (l r) (multD (compile l env)
                        (compile r env))]
    [idC (n) (atD (locate n env))]
    [lamC (n body-expr)
          (lamD
           (compile body-expr
                    (extend-env (bindC n)
                                env)))]
    [appC (fun-expr arg-expr)
          (appD (compile fun-expr env)
                (compile arg-expr env))]))
```

Compile-Time Environment

Mimics the run-time environment, but without values:

```
(define-type BindingC
  [bindC (name : symbol)])

(define-type-alias EnvC (listof BindingC))

(define (locate name env)
  (cond
   [(empty? env) (error 'locate "free variable")]
   [else (if (symbol=? name (bindC-name (first env)))
             0
             (+ 1 (locate name (rest env))))]))
```

`interp` for Compiled

Almost the same as `interp` for `ExprC`:

```
(define (interp a env)
  (type-case ExprD a
    [numD (n) (numV n)]
    [plusD (l r) (num+ (interp l env)
                       (interp r env))]
    [multD (l r) (num* (interp l env)
                       (interp r env))]
    [atD (pos) (list-ref env pos)]
    [lamD (body-expr)
          (closV body-expr env)]
    [appD (fun-expr arg-expr)
          (let ([fun-val (interp fun-expr env)]
                [arg-val (interp arg-expr env)])
            (interp (closV-body fun-val)
                    (cons arg-val
                          (closV-env fun-val))))]))
```

Timing Effect of Compilation

Given

```
(define c {{{{lambda {x}
                {lambda {y}
                  {lambda {z} {+ {+ x x} {+ x x}}}}}
              1}
             2}
            3})
(define d (compile c mt-env))
```

then

```
(interp d empty)
```

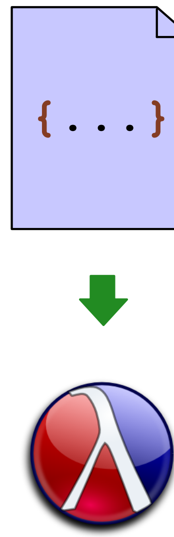
is significantly faster than

```
(interp c mt-env)
```

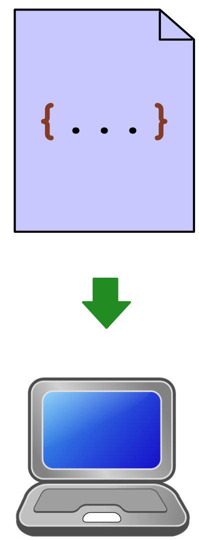
Using the built-in `list-ref` simulates machine array indexing, but don't take timings too seriously

Part 3

From Racket to Machine Code



From Racket to Machine Code



- Everything must be a number
- No **`define-type`** or **`type-case`**
- No implicit continuations
- No implicit allocation

Part 4

From Racket to Machine Code

Step 1:

ExprC → ExprD

```
{lambda {x}        {lambda
  {+ 1 x}}           {+ 1 {at 0}}}
```

Eliminates all run-time names

From Racket to Machine Code

Step 2:

`interp` → `interp` + `continue`

Eliminates implicit continuations

From Racket to Machine Code

Step 3:

function calls → regsiters and **`goto`**

From Racket to Machine Code

Step 3:

function calls → regsiters and **goto**

```
(interp l
        env
        (addSecondK r
                    env
                    k))
```

```
(begin
  (set! expr-reg l)
  (set! k-reg (addSecondK r
                          env-reg
                          k-reg))
  (interp))
```

Makes argument passing explicit

Part 5

From Racket to Machine Code

Step 4:

```
(multSecondK r           →  (malloc3 3
             env-reg                (ref expr-reg 2)
             k-reg)                 env-reg
                                    k-reg)
```

From Racket to Machine Code

Step 4:

```
doneK       → 1
addSecondK  → 2
…
numD        → 8
plusD       → 9
…
numV        → 15
closV       → 16
```

From Racket to Machine Code

Step 4:

```
(type-case Cont k-reg            →   (case (ref k-reg 0)
  ...                                  ...
  [multSecondK (r env k)               [(3)
               ... r                    ... (ref k-reg 1)
               ... env                  ... (ref k-reg 2)
               ... k ..]                ... (ref k-reg 3) ...]
  ...)                                 ...)
```

From Racket to Machine Code

Step 4:

```
(define (malloc3 tag a b c)
  (begin
    (vector-set! memory ptr tag)
    (vector-set! memory (+ ptr 1) a)
    (vector-set! memory (+ ptr 2) a)
    (vector-set! memory (+ ptr 3) a)
    (set! ptr (+ ptr 4))
    (- ptr 4)))
```

Makes all allocation explicit

Makes everything a number

Part 6

Compiling Class-Based Programs

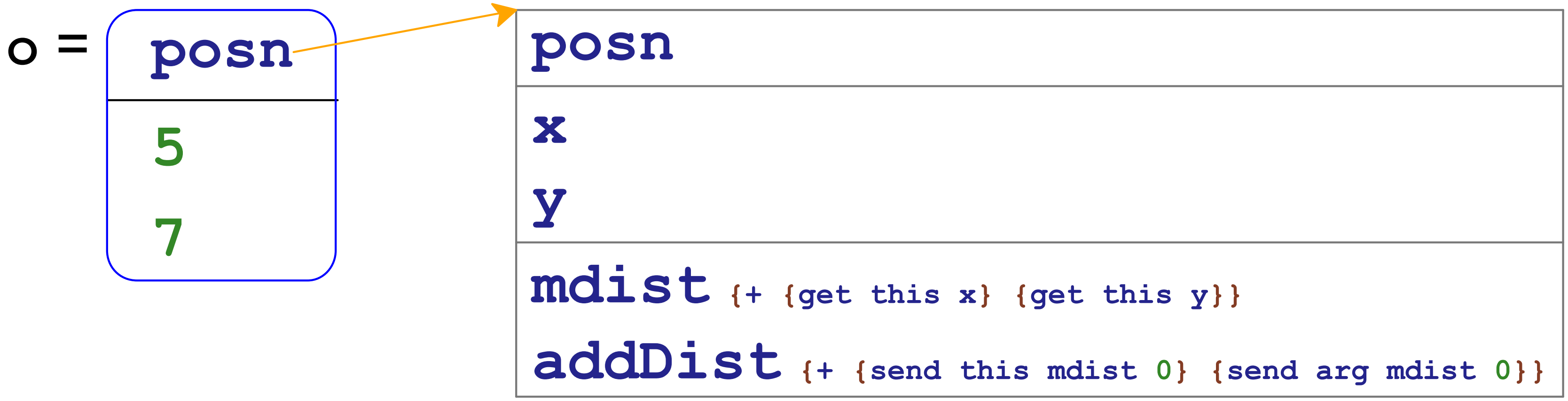
```
{class posn extends object
  {[x : num]  [y : num]}
  {mdist : num -> num
       {+ {get this x} {get this y}}}
  {addDist : posn -> num
         {+ {send this mdist 0} {send arg mdist 0}}}}

{send {new posn 1 2} mdist 0}
```

`this` and `arg` — no search

field and method names — search

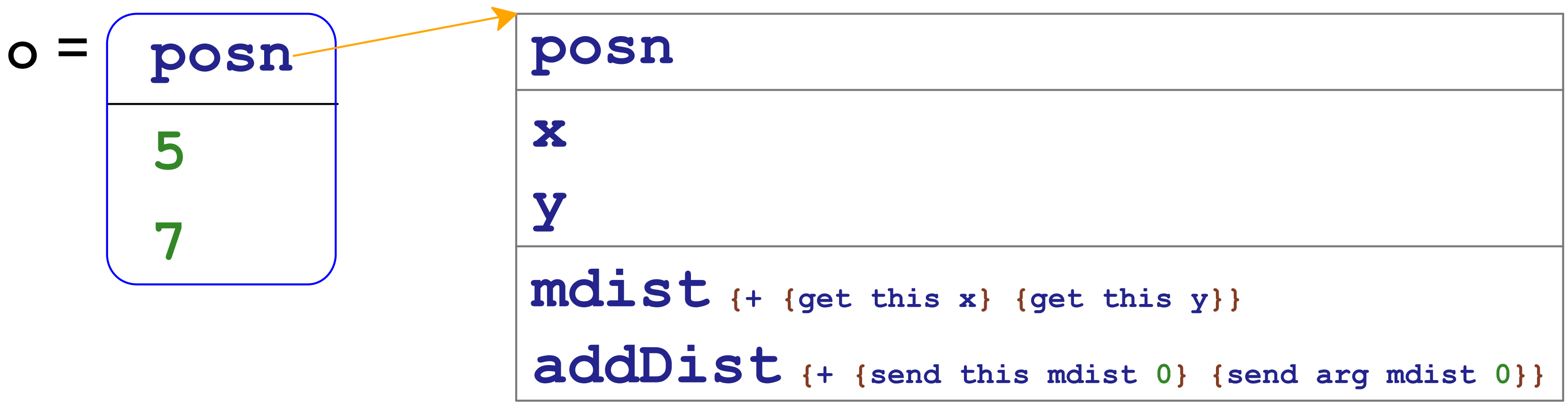
Run-Time Dispatch by Name



`{send o mdist 0}`

`send` follows reference to class table, searches method list

Run-Time Dispatch by Name




```
{send o mdist 0}
```

```
{class posn extends object
  {[x : num]  [y : num]}
  {mdist : num -> num
         {+ {get this x} {get this y}}}
  {addDist : posn -> num
           {+ {send this mdist 0} {send arg mdist 0}}}}
```

$\Rightarrow$ typechecking ensures search will succeed

If we preserve method order in flattening, method will always be first in list

Run-Time Dispatch by Name

o =

posn
5
7

posn
x y
mdist {+ {get this x} {get this y}} addDist {+ {send this mdist 0} {send arg mdist 0}}

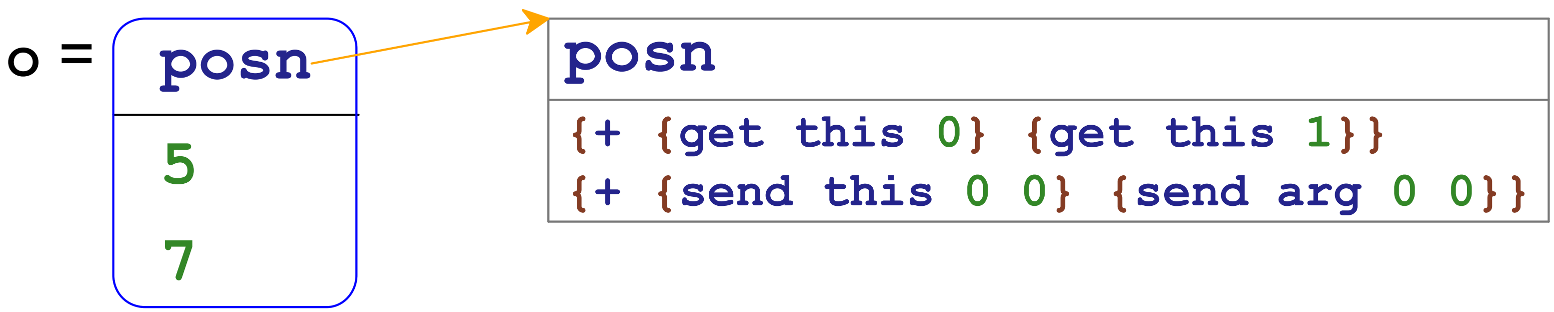
```
{send o mdist 0}
```

```
{class posn extends object
  {[x : num]  [y : num]}
  {mdist : num -> num
         {+ {get this x} {get this y}}}
  {addDist : posn -> num
           {+ {send this mdist 0} {send arg mdist 0}}}}
```

$\Rightarrow$ typechecking ensures search will succeed

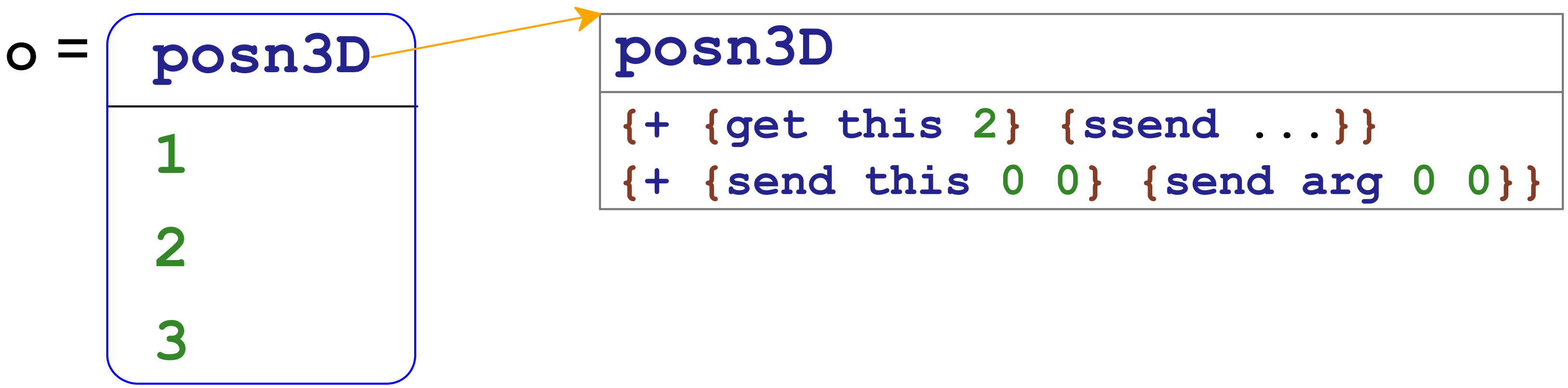
Similarly, we can rely on field positions as long as we add subclass fields to the end

Run-Time Dispatch by Position




```
{send o 0 0}
```

Run-Time Dispatch by Position




```
{send o 0 0}
```

Run-Time Dispatch by Position

o =

```
posn4D
------
0
0
0
1
```

```
posn4D
------
{+ {get this 3} {ssend ...}}
{+ {send this 0 0} {send arg 0 0}}
{get this 3}
{new posn {get this 0} {get this 1}}
```

```
{send o 0 0}
```