Part I
Identifier Address

Suppose that

\[
\{\text{let } \{[x \ 88]\}\}
\{+ \ x \ y\}\}
\]

appears in a program; the body is eventually evaluated:

\[
\{+ \ x \ y\}
\]

where will \(x\) be in the environment?

**Answer:** always at the beginning:

\[
x = 88 \ \ldots
\]
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

```plaintext
{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

\[(\text{compile} \ 1 \ \ldots) \Rightarrow 1\]

\[(\text{compile} \ \{+ \ 1 \ 2\} \ \ldots) \Rightarrow \{+ \ 1 \ 2\}\]

\[(\text{compile} \ x \ \ldots) \Rightarrow \text{compile: free identifier}\]

\[(\text{compile} \ \{\text{lambda} \ {x} \ \{+ \ 1 \ x\}\} \ \ldots)\]

\[\Rightarrow \{\text{lambda} \ \{+ \ 1 \ \{\text{at} \ 0\}\}\}\]

\[(\text{compile} \ \{\text{lambda} \ {y} \ \{\text{lambda} \ {x} \ \{+ \ x \ y\}\}\} \ \ldots)\]

\[\Rightarrow \{\text{lambda} \ \{\text{lambda} \ \{+ \ \{\text{at} \ 0\} \ \{\text{at} \ 1\}\}\}\}\]
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

\[
\text{(define } c \ {\{\{\text{lambda } \{x\} \\
\quad \{\text{lambda } \{y\} \\
\quad \quad \{\text{lambda } \{z\} \ {+ \ {+ \ x \ x} \ {+ \ x \ x}\}\}\}\\
\quad 1\}\\
\quad 2\}\\
\quad 3\}\})
\]

\[
\text{(define } d \ \text{(compile } c \ \text{mt-env)})
\]

then

\[
\text{(interp } d \ \text{empty)}
\]

is significantly faster than

\[
\text{(interp } c \ \text{mt-env)}
\]

Using the built-in \textbf{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:

\[ \text{ExprC} \rightarrow \text{ExprD} \]

\[
\begin{align*}
\{ \text{lambda } \{x\} \} & \quad \{ \text{lambda } \{ + 1 \ \{x\} \} \} \\
\{ + 1 \ \{x\} \} & \quad \{ + 1 \ \{\text{at 0}\} \}
\end{align*}
\]

Eliminates all run-time names
From Racket to Machine Code

Step 2:

\[ \text{interp} \rightarrow \text{interp} + \text{continue} \]

Eliminates implicit continuations
From Racket to Machine Code

Step 3:

function calls $\rightarrow$ registers and \texttt{goto}
From Racket to Machine Code

Step 3:

function calls $\rightarrow$ registers and goto

\[
\begin{align*}
\text{(interp l} & \quad \text{(begin} \\
\text{env} & \quad \text{(set! expr-reg l)} \\
\text{(addSecondK r} & \quad \text{(set! k-reg (addSecondK r} \\
\text{env} & \quad \text{env-reg} \\
\text{k))} & \quad \text{k-reg))} \\
& \quad \text{(interp))}
\end{align*}
\]

Makes argument passing explicit
Part 5
From Racket to Machine Code

Step 4:

\[
\begin{align*}
\text{(multSecondK } r & \rightarrow \text{ (malloc3 3)} \\
& \quad \text{(ref expr-reg 2)} \\
& \quad \text{(env-reg env-reg 2)} \\
& \quad \text{(k-reg k-reg k-reg)}
\end{align*}
\]
From Racket to Machine Code

Step 4:

\[
\begin{align*}
doneK & \rightarrow 1 \\
addSecondK & \rightarrow 2 \\
\ldots & \\
numD & \rightarrow 8 \\
plusD & \rightarrow 9 \\
\ldots & \\
numV & \rightarrow 15 \\
closV & \rightarrow 16
\end{align*}
\]
From Racket to Machine Code

Step 4:

\[
\begin{align*}
(type-case \ Cont \ k-reg & \rightarrow \ (case \ (ref \ k-reg \ 0) \\
\ldots & \ldots \\
[\text{multSecondK} \ (r \ env \ k) & [(3) \\
\ldots \ r & \ldots (ref \ k-reg \ 1) \\
\ldots \ env & \ldots (ref \ k-reg \ 2) \\
\ldots \ k \ ..] & \ldots (ref \ k-reg \ 3) \ ....] \\
\ldots) & \ldots)
\end{align*}
\]
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

\[ o = \begin{array}{c}
\text{posn} \\
5 \\
7 
\end{array} \]

\[
\begin{array}{l}
\text{posn} \\
x \\
y \\
\text{mdist} \{+ \{\text{get this x}\} \{\text{get this y}\}\} \\
\text{addDist} \{+ \{\text{send this mdist 0}\} \{\text{send arg mdist 0}\}\}
\end{array}
\]

\{\text{send } o \text{ mdist 0}\}

\textbf{send} follows reference to class table, searches method list
Run-Time Dispatch by Name

\[ o = \begin{array}{c}
\text{posn} \\
5 \\
7 
\end{array} \]

\[
\begin{array}{c}
posn \\
x \\
y \\
mdist \{+ \{\text{get this } x\} \{\text{get this } y\}\} \\
addDist \{+ \{\text{send this } mdist 0\} \{\text{send arg } mdist 0\}\}
\end{array}
\]

\{send \ o \ mdist \ 0\}

\text{class posn extends object}
\{[x : \text{num}] \ [y : \text{num}]\}
\{mdist : \text{num} \rightarrow \text{num} \}
\{+ \{\text{get this } x\} \{\text{get this } y\}\}\}
\{addDist : \text{posn} \rightarrow \text{num} \}
\{+ \{\text{send this } mdist 0\} \{\text{send arg } mdist 0\}\}
\}

⇒ typechecking ensures search will succeed

If we preserve method order in flattening, method will always be first in list
Run-Time Dispatch by Name

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

\[ o = \begin{array}{c}
  \text{posn} \\
  5 \\
  7 \\
\end{array} \]

\[ \begin{array}{c}
  \text{posn} \\
  x \\
  y \\
  \text{mdist} \{+ \{get this x\} \{get this y\}\}
  \text{addDist} \{+ \{send this mdist 0\} \{send arg mdist 0\}\}
\end{array} \]

\[ \Rightarrow \text{typechecking ensures search will succeed} \]

Similarly, we can rely on field positions as long as we add subclass fields to the end
Run-Time Dispatch by Position

\[ o = \begin{array}{c} \text{posn} \\ 5 \\ 7 \end{array} \rightarrow \begin{array}{l} \text{posn} \{+ \{\text{get this 0}\} \{\text{get this 1}\}\} \\ \{+ \{\text{send this 0 0}\} \{\text{send arg 0 0}\}\} \end{array} \{\text{send o 0 0}\} \]
Run-Time Dispatch by Position

\[ o = \begin{array}{c}
\text{posn3D} \\
1 \\
2 \\
3 \\
\end{array} \]

\begin{tabular}{|c|}
\hline
\text{posn3D} \\
\hline
{+ {get this 2} {ssend \ldots}} \\
{+ {send this 0 0} {send arg 0 0}} \\
\hline
\end{tabular}

\{send o 0 0\}
Run-Time Dispatch by Position

\[
\begin{array}{c}
\text{o} = \begin{array}{c}
\text{posn4D} \\
0 \\
0 \\
0 \\
1
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{posn4D} \\
\{+ \{\text{get this 3}\} \{\text{ssend ...}\}\} \\
\{+ \{\text{send this 0 0}\} \{\text{send arg 0 0}\}\} \\
\{\text{get this 3}\} \\
\{\text{new posn} \{\text{get this 0}\} \{\text{get this 1}\}\}
\end{array}
\]

\[
\{\text{send o 0 0}\}
\]