Part I
Identifier Address

Suppose that

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

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

\[
\begin{align*}
\text{let } \{\text{y 1}\} \\
\text{+ x y}\end{align*}
\]

appears in a program; the body is eventually evaluated:

\[
\begin{align*}
\text{+ x y}\end{align*}
\]

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

**Answer:** always at the beginning:

\[
\begin{align*}
y = 1 \\
\ldots
\end{align*}
\]
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  ...
```
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

\[
\begin{align*}
\text{let } & \{ ([y \ 1]) \\
\text{let } & \{ ([w \ 10]) \\
\text{let } & \{ ([z \ 9]) \\
\text{let } & \{ ([x \ 0]) \\
\text{let } & \{ + \ x \ y \}) \} \}
\end{align*}
\]

appears in a program; the body is eventually evaluated:

\[
\begin{align*}
& \ldots \\
& \{ + \ x \ y \}
\end{align*}
\]

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

**Answer:** always first and fourth:

\[
\begin{align*}
x = 0 & \quad z = 9 & \quad w = 10 & \quad y = 1 & \quad \ldots
\end{align*}
\]
Identifier Address

Suppose that

\[
\begin{align*}
\{ \text{let } & \{ [y \{ \text{let } \{ [r \, 9] \} \{ \ast \, r \, 8 \}] \}] \} \\
\{ \text{let } & \{ [w \, 10] \} \\
\{ \text{let } & \{ [z \{ \text{let } \{ [q \, 9] \} \, q \}] \} \\
\{ \text{let } & \{ [x \, 0] \} \\
\{ + \, x \, y \} \}& \}
\end{align*}
\]

appears in a program; the body is eventually evaluated:

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

where will \textit{x} and \textit{y} be in the environment?

\textbf{Answer:} always first and fourth:

\[
x = 0 \quad z = 9 \quad w = 10 \quad y = 1 \quad \ldots
\]
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
Compilation of Variables

A compiler can transform an Exp expression to an expression without identifiers — only lexical addresses

; compile : Exp ... -> ExpD

(define-type Exp
  (numE [n : Number])
  (addE [l : Exp]
    [r : Exp])
  (multE [l : Exp]
    [r : Exp])
  (idE [n : Symbol])
  (lamE [n : Symbol]
    [body : Exp])
  (appE [fun : Exp]
    [arg : Exp]))

(define-type ExpD
  (numD [n Number])
  (addD [l : ExpD]
    [r : ExpD])
  (multD [l : ExpD]
    [r : ExpD])
  (atD [pos : Number])
  (lamD [body : ExpD])
  (appD [fun : ExpD]
    [arg : ExpD]))
Compile Examples

(\texttt{compile 1 \ldots}) \Rightarrow 1

(\texttt{compile \{+ 1 2\} \ldots}) \Rightarrow \{+ 1 2\}

(\texttt{compile x \ldots}) \Rightarrow \texttt{compile: free identifier}

(\texttt{compile \{\texttt{lambda} \{x\} {+ 1 x}\} \ldots})
\Rightarrow \{\texttt{lambda} {+ 1 \{at 0\}}\}

(\texttt{compile \{\texttt{lambda} \{y\} \{\texttt{lambda} \{x\} {+ x y}\}\} \ldots})
\Rightarrow \{\texttt{lambda} \{\texttt{lambda} {+ \{at 0\} \{at 1\}}\}\}
Implementing the Compiler

(define (compile [a : Exp] [env : EnvC])
  (type-case Exp a
    [(numE n) (numD n)]
    [(plusE l r) (plusD (compile l env)
                        (compile r env))]
    [(multE l r) (multD (compile l env)
                        (compile r env))]
    [(idE n) (atD (locate n env))]
    [(lamE n body-expr)
     (lamD (compile body-expr
              (extend-env (bindE n)
                          env)))]
    [(appE 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
  (bindE [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))))]))
```
Almost the same as \texttt{interp} for \texttt{Exp}:

\[
\text{(define (interp a env)}
\text{(type-case ExpD a)
\begin{align*}
\text{[(numD n) (numV n)]} \\
\text{[(plusD l r) (num+ (interp l env)}
\text{(interp r env))]} \\
\text{[(multD l r) (num* (interp l env)}
\text{(interp r env))]} \\
\text{[(atD pos) (list-ref env pos)]} \\
\text{[(lamD body-expr)
\text{(closV body-expr env)]}} \\
\text{[(appD fun-expr arg-expr)
\text{(let ([fun-val (interp fun-expr env)}
\text{[arg-val (interp arg-expr env)]})
\text{(interp (closV-body fun-val)}
\text{(cons arg-val
\text{(closV-env fun-val))})})])})}
\end{align*}
\)
Timing Effect of Compilation

Given

```scheme
(define c {{{lambda {x}
    {lambda {y}
        {lambda {z} [+ [+ x x] [+ x x]]}]
        1}
    2}
3}

(define d (compile c mt-env))
```

then

```scheme
(interp d empty)
```

is significantly faster than

```scheme
(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
From Racket to Machine Code

- Everything must be a number
From Racket to Machine Code

- Everything must be a number
- No `define-type` or `type-case`
From Racket to Machine Code

• Everything must be a number
• No `define-type` or `type-case`
• No implicit continuations
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:

\[
\text{Exp} \rightarrow \text{ExpD}
\]

\[
\lambda \{ x \} \rightarrow \lambda \{ \text{at 0} \}
\]

{\lambda \{ x \} \rightarrow \{ + 1 \ {\lambda \{ x \}} \}}

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 ➔ registers and goto
From Racket to Machine Code

Step 3:

function calls → registers and goto

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

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

Makes argument passing explicit
Part 5
From Racket to Machine Code

Step 4:

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

Step 4:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>doneK</td>
<td>1</td>
</tr>
<tr>
<td>plusSecondK</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>numD</td>
<td>8</td>
</tr>
<tr>
<td>plusD</td>
<td>9</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>numV</td>
<td>15</td>
</tr>
<tr>
<td>closV</td>
<td>16</td>
</tr>
</tbody>
</table>
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 memory (make-vector 1500 0))
(define ptr-reg 0)

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

Makes all allocation explicit

Makes everything a number