CS164 Lecture
Compiling Closures

Substitute Lecturer: Kevin Laeufer
<laeufer@berkeley.edu>
Kevin Läufer

- B.Sc. in Electrical Engineering from RWTH Aachen University
- Advised by Jonathan Bachrach (Chisel) and Koushik Sen (Concolic Testing)
- Associated with the Adept lab
- building compilers and automated testing tools for circuits
- happy to use Scala and SMT solvers
Review: Function Calls

(define (id x) x)
(print (id 4))

Stack frame layout for id
Review: Function Calls

\[
\begin{align*}
\text{(define (id x) x)} \\
\text{(print (id 4))}
\end{align*}
\]

Stack frame layout for id

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;return address&gt;</td>
</tr>
</tbody>
</table>

\(x\)
Review: Function Calls

(define (id x) x)
(print (id 4))

Stack frame layout for id

Calling id from our main function
Review: Function Calls

```
(define (id x) x)
(print (id 4))
```

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;return address&gt;</td>
</tr>
</tbody>
</table>

Stack frame layout for `id`

```
/---------/

<main return address>
```

Calling `id` from our main function
Review: Function Calls

\[
\text{(define (id } x \text{) } x) \\
\text{(print (id 4))}
\]

Stack frame layout for \text{id}

<table>
<thead>
<tr>
<th>4</th>
<th>&lt;id return address&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>/// /// /// /</td>
<td></td>
</tr>
<tr>
<td>&lt;main return address&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Calling \text{id} from our main function
Review: Function Calls

```
(define (id x) x)
(print (id 4))
```

entry:
```assembly
mov rax, 16
mov [rsp + -24], rax
add rsp, -8
call function_id...
sub rsp, -8
... ; call print
function_id...
mov rax, [rsp + -8]
ret
```

Stack frame layout for id

|        |  
|--------|--------|
| x      | <return address> |
| 4 << 2 | <id return address> |
| ////////////// |
| <main return address> |

Calling id from our main function
Review: Function Calls

(define (id x) x)
(print (id 4))

entry:
  mov rax, 16
  mov [rsp + -24], rax
  add rsp, -8
  call function_id...
  sub rsp, -8
  ...; call print
function_id...:
  mov rax, [rsp + -8]
  ret

Stack frame layout for id

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>&lt;return address&gt;</td>
</tr>
</tbody>
</table>

Calling id from our main function

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4 &lt;&lt; 2</td>
<td>&lt;id return address&gt;</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>///////////////</td>
<td>&lt;main return address&gt;</td>
</tr>
</tbody>
</table>
Review: Function Calls

\[
(\text{define } (\text{id } x) \ x) \\
(\text{print } (\text{id } 4))
\]

```
(entry: 
  mov rax, 16 
  mov [rsp + -24], rax 
  add rsp, -8 
  call function_id...
  sub rsp, -8 
  ...; call print

function_id...: 
  mov rax, [rsp + -8] 
  ret)
```

Stack frame layout for id

```
<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;return address&gt;</td>
</tr>
</tbody>
</table>

4 \(\ll 2\)

\(<\text{id return address}>\)

<~~~~~~~~~~~>

<\text{main return address}>

Calling \text{id} from our main function
Review: Function Calls

```
(define (id x) x)
(print (id 4))
```

entry:
- `mov rax, 16`
- `mov [rsp + -24], rax`
- `add rsp, -8`
- `call function_id...`
- `sub rsp, -8`
- `... ; call print`

```
function_id...:
  mov rax, [rsp + -8]
  ret
```

Stack frame layout for `id`

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
</tr>
<tr>
<td>&lt;return address&gt;</td>
</tr>
</tbody>
</table>

Calling `id` from our main function

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>4 &lt;&lt; 2</code></td>
</tr>
<tr>
<td>&lt;id return address&gt;</td>
</tr>
<tr>
<td>//////////////</td>
</tr>
<tr>
<td>&lt;main return address&gt;</td>
</tr>
</tbody>
</table>
Review: Function Calls

\[(\text{define } (\text{id } x) \ x)\] \[(\text{print } (\text{id } 4))\]

entry:
\[
\begin{align*}
\text{mov rax, 16} \\
\text{mov [rsp + -24], rax} \\
\text{add rsp, -8} \\
\text{call function_id...} \\
\text{sub rsp, -8} \\
\text{... ; call print} \\
\text{function_id...:} \\
\text{mov rax, [rsp + -8]} \\
\text{ret}
\end{align*}
\]

Stack frame layout for \text{id}

\[
\begin{align*}
\text{x} \\
<\text{return address}> \\
\text{4 }\ll\text{ 2} \\
<\text{id return address}> \\
/\\/\\/\\/\\
<\text{main return address}>
\end{align*}
\]

Calling \text{id} from our main function
Review: Function Calls

```
(define (id x) x)
(print (id 4))
```

entry:
```
mov rax, 16
mov [rsp + -24], rax
add rsp, -8
call function_id_...
sub rsp, -8
... ; call print
```

```
function_id_...:
mov rax, [rsp + -8]
ret
```

Stack frame layout for id

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;return address&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4 &lt;&lt; 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;id return address&gt;</td>
</tr>
</tbody>
</table>

Calling id from our main function
Review: Function Calls

(define (id x) x)
(print (id 4))

entry:
  mov rax, 16
  mov [rsp + -24], rax
  add rsp, -8
  call function_id...
  sub rsp, -8
  ... ; call print
function_id...:
  mov rax, [rsp + -8]
  ret

Stack frame layout for id

<table>
<thead>
<tr>
<th>4 &lt;&lt; 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;id return address&gt;</td>
</tr>
</tbody>
</table>

Calling id from our main function
Review: Function Calls

(\texttt{define (id \ x) \ x})
(\texttt{print (id 4)})

entry:
\begin{verbatim}
mov rax, 16
mov [rsp + -24], rax
add rsp, -8
call function_id...
sub rsp, -8
... ; call print
function_id...:
mov rax, [rsp + -8]
ret
\end{verbatim}

Stack frame layout for \texttt{id}

<table>
<thead>
<tr>
<th>\texttt{x}</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;return address&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\texttt{4 \textless\textgreater\ 2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;id return address&gt;</td>
</tr>
</tbody>
</table>

Calling \texttt{id} from our main function
Review: Function Calls Bookkeeping

```
(define (id x) x)
(print (id 4))
```

tab : int symtab

defns : defn list
Review: Function Calls Bookkeeping

```
(define (id x) x)
(print (id 4))
```

```
tab : int symtab
defns : defn list
```

| Call (f, args) when is_defn defns f && not is_tail ->
  let defn = get_defn defns f in
  if List.length args = List.length defn.args then
```
Review: First-Class Functions

\[
\text{(define (id } x) \ x) \\
\text{(print (id 4))}
\]
Review: First-Class Functions

(define (id x) x)
(print (let ((f id)) (f 4)))
Review: First-Class Functions

(\(\text{define (id x) x}\))
(\(\text{print (let ((f id)) (f 4)))}\))

Stack frame layout for id

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;return address&gt;</td>
</tr>
</tbody>
</table>
**Review: First-Class Functions**

```
(define (id x) x)
(print (let ((f id)) (f 4)))
```

Stack frame layout for `id`

Calling `id` from our main function
Review: First-Class Functions

```
(define (id x) x)
(print (let ((f id)) (f 4)))
```

Stack frame layout for `id`

Calling `id` from our main function
Review: First-Class Functions

\begin{align*}
\text{(define (id x) x)} \\
\text{(print (let ((f id)) (f 4)))}
\end{align*}

Stack frame layout for \text{id}

\begin{tabular}{|c|}
\hline
4 \text{<<} 2 \\
\hline
\text{id return address} \\
\hline
\text{<address of function_id_…>} \\
\hline
\text{<main return address>} \\
\hline
\end{tabular}

Calling \text{id} from our main function
Review: First-Class Functions

(define (id x) x)
(print (let ((f id)) (f 4)))

entry:
lea rax, [function_id_...]
or rax, 6
mov [rsp + -8], rax
mov rax, 16
mov [rsp + -24], rax
mov rax, [rsp + -8]
;; ensure_fn
sub rax, 6
add rsp, -8
call rax
sub rsp, -8

Stack frame layout for id

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;return address&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4 &lt;&lt; 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;id return address&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;address of function_id_...&gt;</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>&lt;main return address&gt;</th>
</tr>
</thead>
</table>

Calling id from our main function
Review: First-Class Functions

\[
\text{(define (id x) x)}
\]
\[
\text{(print (let ((f id)) (f 4)))}
\]

entry:

```
lea rax, [function_id_...]
or rax, 6
mov [rsp + -8], rax
mov rax, 16
mov [rsp + -24], rax
mov rax, [rsp + -8]
;; ensure_fn
sub rax, 6
add rsp, -8
call rax
sub rsp, -8
```

Stack frame layout for id

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>&lt;return address&gt;</td>
</tr>
<tr>
<td>4 &lt;&lt; 2</td>
<td>&lt;id return address&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;address of function_id_...&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;main return address&gt;</td>
</tr>
</tbody>
</table>

Calling id from our main function
Review: First-Class Functions

(define (id x) x)
(print (let ((f id)) (f 4)))

entry:
  lea rax, [function_id_...]
or rax, 6
mov [rsp + -8], rax
mov rax, 16
mov [rsp + -24], rax
mov rax, [rsp + -8]
;; ensure_fn
sub rax, 6
add rsp, -8
call rax
sub rsp, -8

Stack frame layout for id

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;return address&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4 &lt;&lt; 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;id return address&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;address of function_id_...&gt;</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>&lt;main return address&gt;</th>
</tr>
</thead>
</table>

Calling id from our main function
Review: First-Class Functions

(define (id x) x)
(print (let ((f id)) (f 4)))

entry:

- lea rax, [function_id_...]
- or rax, 6
- mov [rsp + -8], rax
- mov rax, 16
- mov [rsp + -24], rax
- mov rax, [rsp + -8]

;; ensure_fn
- sub rax, 6
- add rsp, -8
- call rax
- sub rsp, -8

Stack frame layout for id

<table>
<thead>
<tr>
<th>x</th>
<th>&lt;return address&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 &lt;&lt; 2</td>
<td>&lt;id return address&gt;</td>
</tr>
<tr>
<td>&lt;address of function_id_...&gt;</td>
<td>&lt;main return address&gt;</td>
</tr>
</tbody>
</table>

Calling id from our main function
Review: First-Class Functions

\[(\text{define } (\text{id } x) \ x)\]  
\[(\text{print } (\text{let } ((f \ \text{id})) (f \ 4)))\]

entry:

\[
\begin{align*}
\text{lea } rax, & \text{ [function_id...]} \\
\text{or } rax, & \ 6 \\
\text{mov } [rsp + -8], & \ rax \\
\text{mov } rax, & \ 16 \\
\text{mov } [rsp + -24], & \ rax \\
\text{mov } rax, & \ [rsp + -8] \\
\text{;; ensure_fn} \\
\text{sub } rax, & \ 6 \\
\text{add } rsp, & \ -8 \\
\text{call } rax \\
\text{sub } rsp, & \ -8
\end{align*}
\]

Stack frame layout for \text{id}:

\[
\begin{array}{|c|}
\hline
x \\
\hline
<\text{return address}> \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
4 \ll 2 \\
\hline
<\text{id return address}> \\
\hline
<\text{address of function_id...}> \\
\hline
<\text{main return address}> \\
\hline
\end{array}
\]

Calling \text{id} from our main function
Review: First-Class Functions Bookkeeping

(define (id x) x)
(print (let (((f id)) (f 4))))

| Call (f, args) when is_defn defns f && not is_tail ->
| let defn = get_defn defns f in
| if List.length args = List.length defn.args then
Review: First-Class Functions Bookkeeping

\[(\text{define} \ (\text{id} \ x) \ x)\]
\[(\text{print} \ (\text{let} \ ((f \ \text{id})) \ (f \ 4)))\]

\[\text{tab} : \text{int} \ \text{symtab}\]
\[\text{defns} : \text{defn list}\]

Not a string anymore! We do not know the function name at compile-time.

| Call \((f, \text{args})\) when is_defn defns f && not is_tail ->
  let defn = get_defn defns f in
  if \List.length\ args = \List.length\ defn.args then
Review: First-Class Functions Bookkeeping

(define (id x) x)
(print (let (((f id)) (f 4))))
Review: Anonymous Functions

(define (range lo hi) ...) 
(define (map f l) ...) 
(define (g x) (+ x 1)) 
(print (map g (range 0 2))))
Review: Anonymous Functions

(define (range lo hi) ...)  
(define (map f l) ...)  
(define (g x) (+ x 1))  
(print (map g (range 0 2)))
Review: Anonymous Functions

```
(define (range lo hi) ...)
(define (map f l) ...)
(define (g x) (+ x 1))
(print (map g (range 0 2)))
```

```
(define (range lo hi) ...)
(define (map f l) ...)
(print (map (lambda (x) (+ x 1)) (range 0 2)))
```
Review: Anonymous Functions

```
(define (range lo hi) ...)  type expr
(define (map f l) ...)       
(define (g x) (+ x 1))       
(print (map g (range 0 2)))
```

```
(define (range lo hi) ...)  type expr
(define (map f l) ...)       
(define (g x) (+ x 1))       
(print (map (lambda (x) (+ x 1)) (range 0 2)))
```
Review: Anonymous Functions

(define (range lo hi) ...)  \text{type expr}
(define (map f l) ...)
\textcolor{red}{(define (g x) (+ x 1))}
(print (map g (range 0 2)))

(define (range lo hi) ...)  \text{type expr}
(define (map f l) ...)
\textcolor{blue}{(print (map (lambda (x) (+ x 1)) (range 0 2)))}
Review: Anonymous Functions

(\textbf{define} (\textbf{range} \textit{lo} \textit{hi}) \ldots)
(\textbf{define} (\textbf{map} \textit{f} \textit{l}) \ldots)
(\textbf{define} (\textit{g} \textit{x}) (+ \textit{x} 1))
(\textbf{print} (\textbf{map} \textit{g} (\textbf{range} 0 2)))
Review: Anonymous Functions

\[
\begin{align*}
&\text{(define (range lo hi) \ldots)} \\
&\text{(define (map f l) \ldots)} \\
&\text{(define (_lambda_16 x) (+ x 1))} \\
&\text{(print (map _lambda_16 (range 0 2)))}
\end{align*}
\]
Review: Closures

```plaintext
(print
  (let ((x 2))
    ((lambda (y) (+ y x)) 3)
  )
)
```
Review: Closures

\[
\begin{align*}
\text{print} \\
\quad \text{let} \ ((x \ 2)) \\
\quad \quad ((\text{lambda} \ (y) \ (+ \ y \ x)) \ 3)
\end{align*}
\]
Review: Closures

(print
  (let ((x 2))
    ((lambda (y) (+ y x)) 3)
  )
)

(define (_lambda_1 y) (+ y x))
Review: Closures

```
(print
  (let ((x 2))
    (_lambda_1 3))
)
```

```
(define (_lambda_1 y) (+ y x))
```
Review: Closures

(print
  (let ((x 2))
    (_lambda_1 3)
  )
)

(define (_lambda_1 y) (+ y x))

How do we “pass” x?
Review: Closures

```
(let ((x 2))
  ((lambda (y) (+ y x)) 3))
```
Review: Closures

```lisp
(print
  (let ((x 2))
    ((lambda (y) (+ y x)) 3))
)
```

x is a free variable.
Review: Closures

We need to find all free variables in a lambda.

→ Discuss the \texttt{fv} function.
Review: Closures

(print
  (let ((x 2))
    ((lambda (y) (+ y x)) 3)
  )
)
Review: Closures

```
(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (f 3)))
)
```
Review: Closures

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (f 3)))))
Review: Closures

(print
 (let ((x 2))
 (let ((f (lambda (y) (+ y x))))
 (f 3)))
)))

Create closure.

Call closure.
Review: Closures

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (f 3))))

Q: What does the program print?
Review: Closures

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
      )))
)
Review: Closures

```scheme
(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
      )))
)
```

Q: What does the program print?
Review: Closures in the Interpreter

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
        ))))

Q: What does the program print?
Review: Closures in the Interpreter

```scheme
(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3))))
))
```

Q: What does the program print?
Q: What does the program print?
Review: Closures in the Interpreter

Contains a copy of the environment. In our case: \{x: 2\}
(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)))))
)
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)))
  )
)

{}   
{x:-8}   
{x:-8, f: -16}   
{x:-24, f:-16}
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x)))))
      {}
      {x:-8}
      Create closure.
    (let ((x -2))
      {x:-8, f: -16}
    (f 3)
      {x:-24, f:-16}
)))))
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x)))))
    (let ((x -2))
      (f 3)))
))

Heap structure for our lambda

<address of function label>

<table>
<thead>
<tr>
<th>value of x</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
</tr>
<tr>
<td>{x:-8}</td>
</tr>
<tr>
<td>{x:-8, f: -16}</td>
</tr>
<tr>
<td>{x:-24, f:-16}</td>
</tr>
</tbody>
</table>
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x)))
      (let ((x -2))
        (f 3)))
  )))

Heap structure for our lambda

<address of function label>

<table>
<thead>
<tr>
<th>value of x</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
</tr>
<tr>
<td>{x:-8}</td>
</tr>
<tr>
<td>{}</td>
</tr>
<tr>
<td>{x:-8, f: -16}</td>
</tr>
<tr>
<td>{x:-24, f:-16}</td>
</tr>
</tbody>
</table>

Copy all free variables from stack! Create closure.
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x)))))
    (let ((x -2))
      (f 3)))))

→ Show how creating closures is implemented!
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
        Call closure.
      )))
))
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
          Call closure.
        )))
  )
)
Closures in the Compiler

```plaintext
(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
        Call closure.
      )))
))
```
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
        __________________________ Call closure.
      )))
)

<main return address>

Calling the lambda from main
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
        Call closure.
    )))
)

2 << 4
<main return address>

Calling the lambda from main
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
          __________________________ Call closure.
      )))
)
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
        Call closure.
    )))
)

-2 << 2
.addr of lambda closure>
2 << 4
.<main return address>

Calling the lambda from main
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
        ___________________________ Call closure.
    )))

-2 << 2
<addr of lambda closure>
2 << 4
<main return address>

Calling the lambda from main
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x)))))
    (let ((x -2))
      (f 3)
      ---------- Call closure.
)))

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &lt;&lt; 2</td>
<td></td>
</tr>
<tr>
<td>-2 &lt;&lt; 2</td>
<td></td>
</tr>
<tr>
<td>&lt;addr of lambda closure&gt;</td>
<td></td>
</tr>
<tr>
<td>2 &lt;&lt; 4</td>
<td></td>
</tr>
<tr>
<td>&lt;main return address&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Calling the lambda from main
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
        Call closure.
    )))

<addr of lambda closure>

| 3 << 2 |
| -2 << 2 |

<addr of lambda closure>

| 2 << 4 |

<main return address>

Calling the lambda from main
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)
          Call closure.
      )))
  )

<addr of lambda closure>
3 << 2

<return addr of the lambda>
-2 << 2

<addr of lambda closure>
2 << 4

<main return address>

Calling the lambda from main
Closures in the Compiler

(print
  (let ((x 2))
    (let ((f (lambda (y) (+ y x))))
      (let ((x -2))
        (f 3)))))

→ Show how calling closures is implemented!
### Closures in the Compiler

**(lambda (y) (+ y x))**

<table>
<thead>
<tr>
<th>&lt;addr of lambda closure&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &lt;&lt; 2</td>
</tr>
<tr>
<td>&lt;return addr of the lambda&gt;</td>
</tr>
<tr>
<td>-2 &lt;&lt; 2</td>
</tr>
<tr>
<td>&lt;addr of lambda closure&gt;</td>
</tr>
<tr>
<td>2 &lt;&lt; 4</td>
</tr>
<tr>
<td>&lt;main return address&gt;</td>
</tr>
</tbody>
</table>

Calling the lambda from main
Closures in the Compiler

\((\texttt{lambda } (y) \ ( + \ y \ x))\)

<table>
<thead>
<tr>
<th>&lt;addr of lambda closure&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
</tr>
<tr>
<td>&lt;return addr of the lambda&gt;</td>
</tr>
</tbody>
</table>

Stack frame layout
Closures in the Compiler

\\((\texttt{\textbf{lambda}} \ (y) \ (\ + \ y \ x))\\)

Heap structure for our lambda

<table>
<thead>
<tr>
<th>&lt;address of function label&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of x</td>
</tr>
</tbody>
</table>

Stack frame layout

<table>
<thead>
<tr>
<th>&lt;addr of lambda closure&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
</tr>
<tr>
<td>&lt;return addr of the lambda&gt;</td>
</tr>
</tbody>
</table>
Closures in the Compiler

```
(lambda (y) (+ y x))
```

Heap structure for our lambda:
- `<address of function label>`
- `value of x`

Stack frame layout:
- `<addr of lambda closure>`
- `y`
- `<return addr of the lambda>`
Closures in the Compiler

```
(lambda (y) (+ y x))
```

Heap structure for our lambda:

- <address of function label>
- value of x

Stack frame layout:

- <addr of lambda closure>
- y
- <return addr of the lambda>

Copy over all free variables as if they were regular arguments.
Closures in the Compiler

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

Heap structure for our lambda:
- `<address of function label>`
- `value of x`

Stack frame layout:
- `<addr of lambda closure>`
- `y`
- `<return addr of the lambda>`

Q: What does our symbol table look like?
Closures in the Compiler

\( \text{lambda (y) (+ y x)} \)

Heap structure for our lambda
- <address of function label>
- value of x

Stack frame layout
- <return addr of the lambda>
- y
- <addr of lambda closure>

Q: What does our symbol table look like? A: \{ y: -8, x: \}
Closures in the Compiler

\((\text{lambda } (y) \ ( + \ y \ x))\)

Heap structure for our lambda

- <address of function label>
- value of x

Stack frame layout

- <return addr of the lambda>
- y

Copy over all free variables as if they were regular arguments.

Q: What does our symbol table look like? A: \{ y : -8, x : -16 \}
Closures in the Compiler

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

Heap structure for our lambda:
- <address of function label>
- value of x

Stack frame layout:
- <addr of lambda closure>
- y
- <return addr of the lambda>

Copy over all free variables as if they were regular arguments.

→ **Show** how the prologue for closures is implemented!