CS164 Lecture
Compiling Closures

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

Kevin Läufer

ADEPT

- 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

CHISEL

Review: Function Calls

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

Stack frame layout for **id**

Review: Function Calls

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

x
<return address>

Stack frame layout for **id**

Review: Function Calls

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

x
<return address>

Stack frame layout for **id**

<main return address>

Calling **id** from our main function

Review: Function Calls

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

x
<return address>

Stack frame layout for **id**

//////////
<main return address>

Calling **id** from our main function

Review: Function Calls

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

x
<return address>

Stack frame layout for **id**

4
<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
```

x
<return address>

Stack frame layout for **id**

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
```

x
<return address>

Stack frame layout for **id**

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
```

x
<return address>

Stack frame layout for **id**

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
```

x
<return address>

Stack frame layout for **id**

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
```

x
<return address>

Stack frame layout for **id**

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
```

x
<return address>

Stack frame layout for **id**

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
```

x
<return address>

Stack frame layout for **id**

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
```

x
<return address>

Stack frame layout for **id**

4 << 2
<id return address>
//////////
<main return address>

Calling **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

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

Review: First-Class Functions

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

Review: First-Class Functions

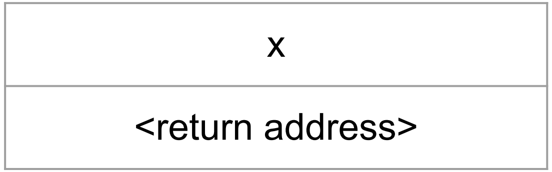
```
(define (id x) x)
(print (let ((f id)) (f 4)))
```

x
<return address>

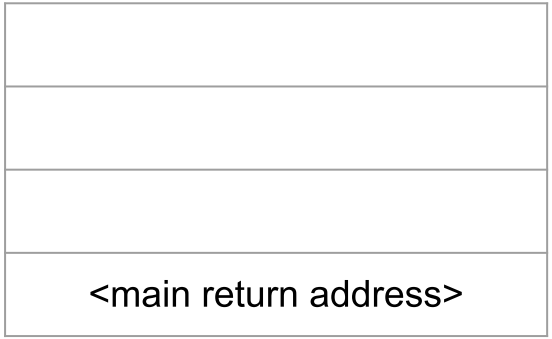
Stack frame layout for **id**

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)))
```

x
<return address>

Stack frame layout for **id**

<address of function_id_…>
<main return address>

Calling **id** from our main function

Review: First-Class Functions

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

x
<return address>

Stack frame layout for **id**

4 << 2
<id return address>
<address of function_id_…>
<main return address>

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
```

x
<return address>

Stack frame layout for **id**

4 << 2
<id return address>
<address of function_id_…>
<main return address>

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
```

x
<return address>

Stack frame layout for **id**

4 << 2
<id return address>
<address of function_id_...>
<main return address>

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
```

x
<return address>

Stack frame layout for **id**

4 << 2
<id return address>
<address of function_id_…>
<main return address>

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
```

x
<return address>

Stack frame layout for **id**

4 << 2
<id return address>
<address of function_id_...>
<main return address>

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
```

x
<return address>

Stack frame layout for **id**

4 << 2
<id return address>
<address of function_id_…>
<main return address>

Calling **id** from our main function

Review: First-Class Functions Bookkeeping

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

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

```
tab : int symtab

defns : defn list
```

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

```
| 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

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

```
tab : int symtab

defns : defn list
```

```
| Var var when is_defn defns var ->
    [
      LeaLabel (Reg Rax, defn_label var)
      ; Or (Reg Rax, Imm fn_tag)
    ]
```

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) …)
(define (map f l) …)
(define (g x) (+ x 1))
(print (map g (range 0 2)))
```

```
type expr
```

```
(define (range lo hi) …)
(define (map f l) …)
(print (map (lambda (x) (+ x 1)) (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)))
```

`type expr`

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

`type expr_lam`

Review: Anonymous Functions

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

type expr

expr_of_expr_lam

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

type expr_lam

Review: Anonymous Functions

```
(define (range lo hi) …)
(define (map f l) …)
(define (_lambda__16 x) (+ x 1))
(print (map _lambda__16 (range 0 2)))
```

type expr

expr_of_expr_la
m

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

type expr_lam

Review: Closures

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

Review: Closures

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

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

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

Review: Closures

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

x is a
free variable.

Review: Closures

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

x is a
<u>free variable</u>.

We need to find all free variables in a lambda.
→ *Discuss the <u>fv</u> 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)
)))
```

Create closure.

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)
)))
```

Create closure.

Call closure.

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

```
(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)
))))
```

Create closure.

Call closure.

Q: What does the program print?

Review: Closures in the Interpreter

Environment	Code	
	`(print`	
{}	`(let ((x 2))`	
{x:2}	`(let ((f (lambda (y) (+ y x))))`	Create closure.
{x:2, f:...}	`(let ((x -2))`	
{x:-2, f:...}	`(f 3)`	Call closure.
	`))))`	

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)
))))
```

{}

{x:2}

{x:2, f:...}

{x:-2, f:...}

Create closure.

Call closure.

Q: What does the program print?

Review: Closures in the Interpreter

Environment	Code	
	`(print`	
{}	`(let ((x 2))`	
{x:2}	`(let ((f (lambda (y) (+ y x))))`	Create closure.
{x:2, f:...}	`(let ((x -2))`	
{x:-2, f:...}	`(f 3)`	Call closure.
	`))))`	

Contains a copy of the environment. In our case: {x: 2}

Closures in the Compiler

```
(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))))
  (let ((x -2))
  (f 3)
))))
```

{}

{x:-8}

Create closure.

{x:-8, f: -16}

{x:-24, f:-16}

Closures in the Compiler

Heap structure for our lambda

<address of function label>
value of x

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

{}

{x:-8}

Create closure.

{x:-8, f: -16}

{x:-24, f:-16}

Closures in the Compiler

Heap structure for our lambda

<address of function label>
value of x

Copy all free variables from stack!

Create closure.

```
(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

Heap structure for our lambda

<address of function label>
value of x

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

{}

{x:-8}

Copy all free variables from stack!

Create closure.

{x:-8, f: -16}

{x:-24, f:-16}

→ ***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

```
(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)
))))
```

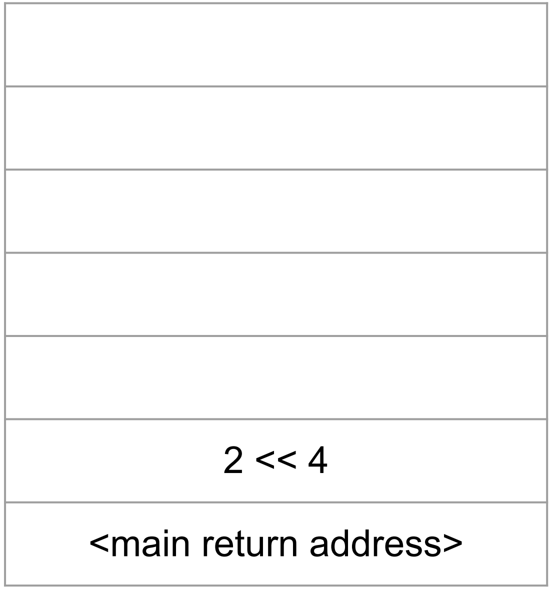
Call closure.

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.



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>
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.

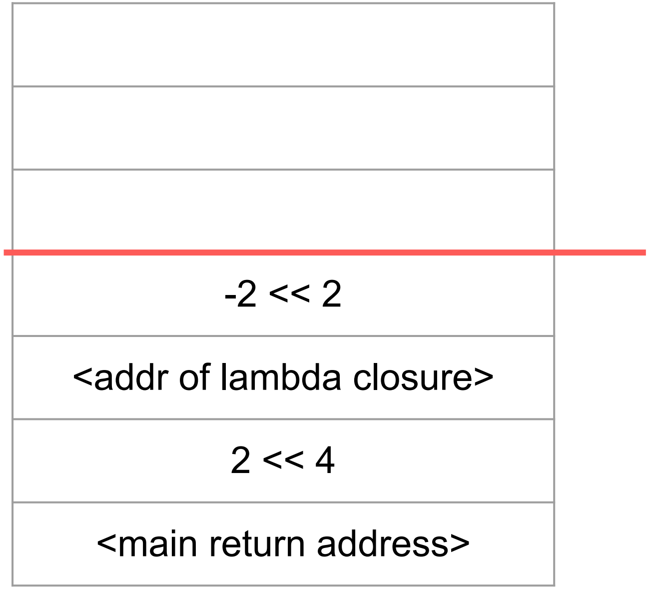
-2 << 2
<addr of lambda closure>
2 << 4
<main return address>

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.

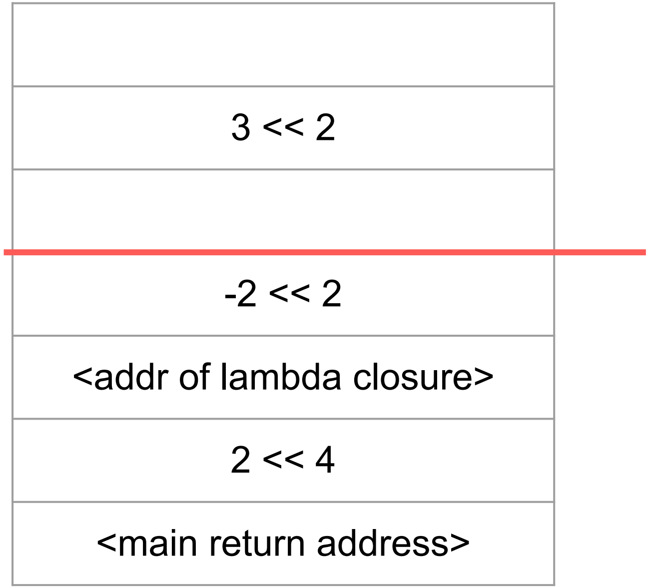


Calling the lambda from main

Closures in the Compiler

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

Call closure.



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)
))))
```

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

→ ***Show*** *how calling closures is implemented!*

Closures in the Compiler

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

<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

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

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

Stack frame layout

Closures in the Compiler

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

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

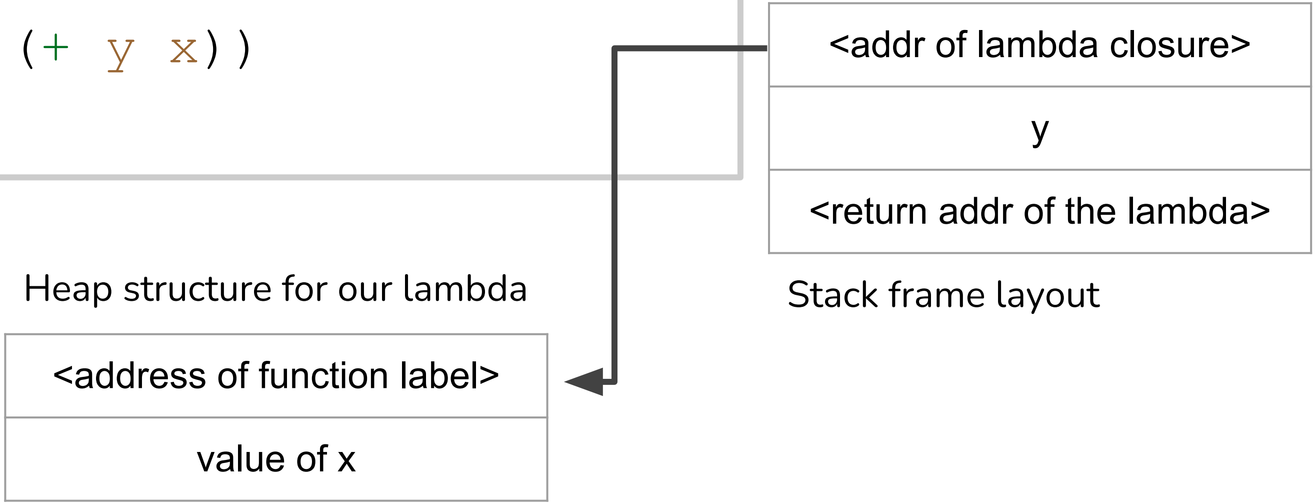
Stack frame layout

Heap structure for our lambda

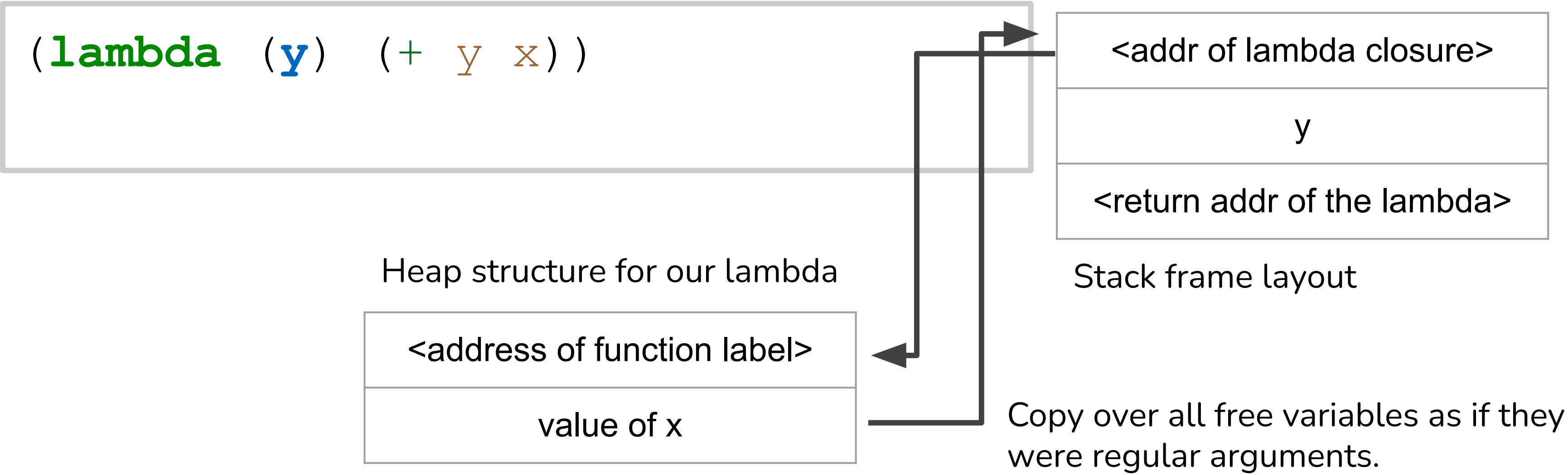
<address of function label>
value of x

Closures in the Compiler

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

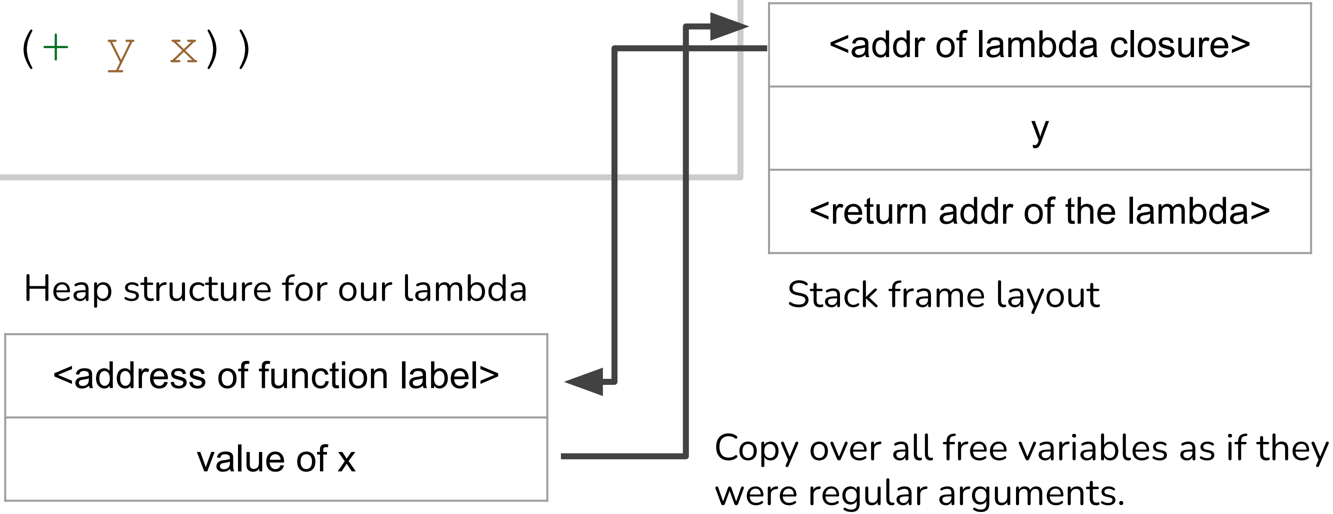


Closures in the Compiler



Closures in the Compiler

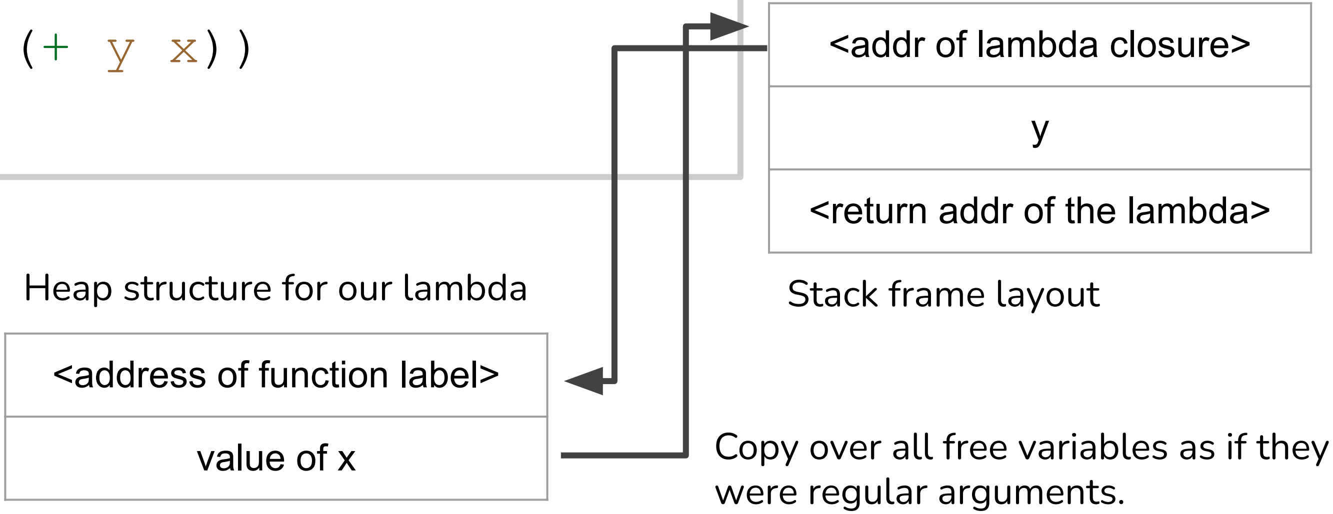
```
(lambda (y) (+ y x))
```



Q: What does our symbol table look like?

Closures in the Compiler

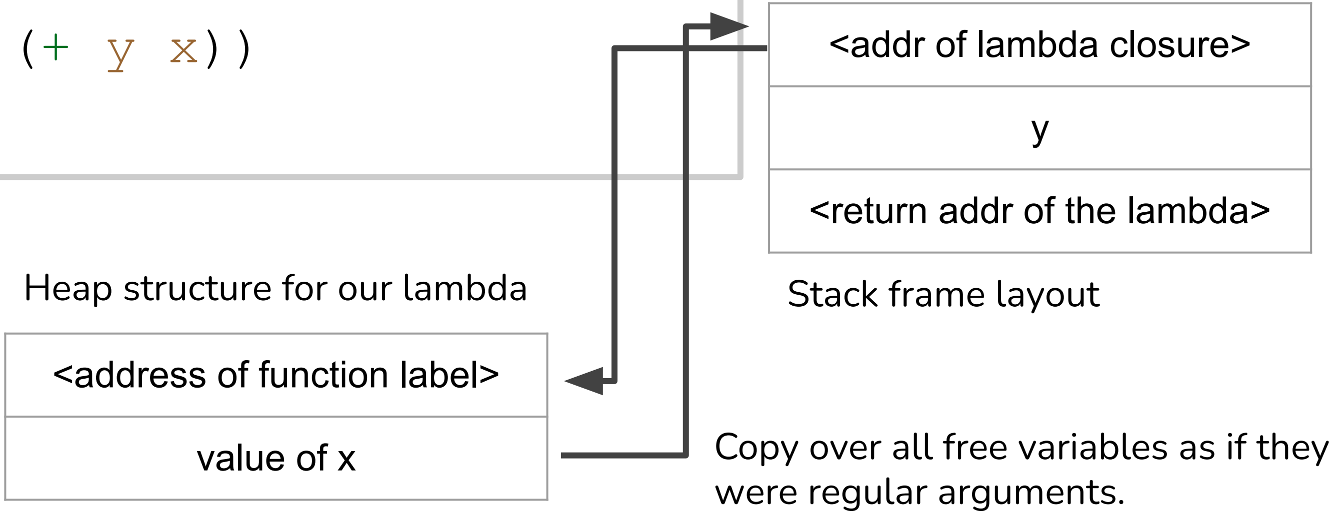
```
(lambda (y) (+ y x))
```



Q: What does our symbol table look like? **A:** { y : -8, x : }

Closures in the Compiler

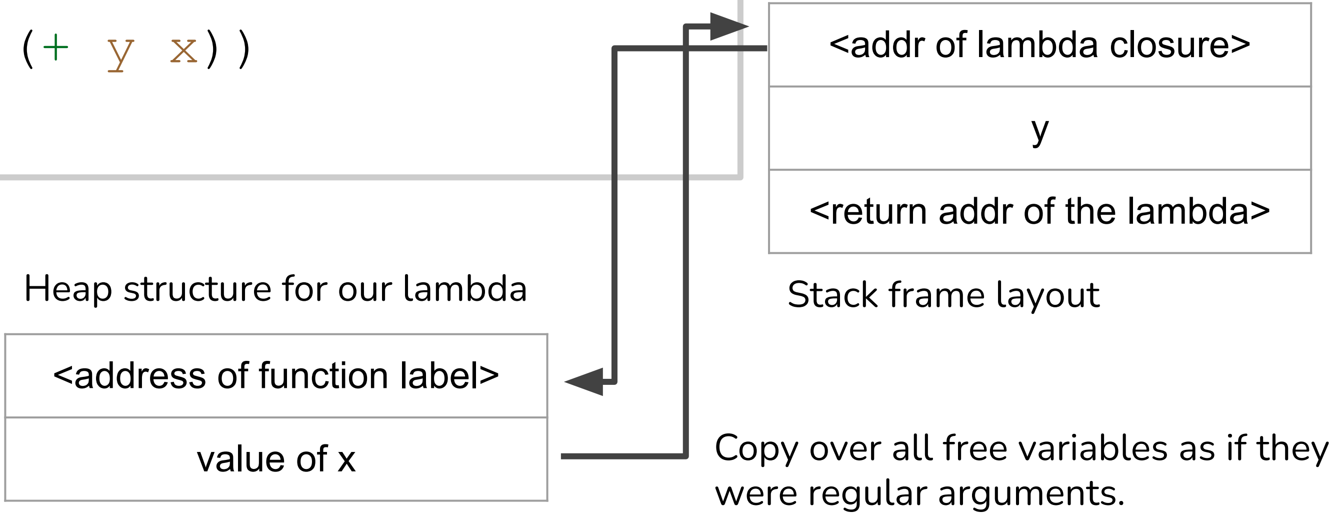
```
(lambda (y) (+ y x))
```



Q: What does our symbol table look like? **A:** { y : -8, x : -16 }

Closures in the Compiler

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



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