CMSC 430 - 10 Nov 2026

Lambdas and first-class functions

- Loot - Lambda
  - interpreter lambda expressions w/ and w/o functions
  - compiling lambda expressions

CMSC 430 - 10 Nov 2026

Announcements

- Assignment 8 due Tuesday
- No quiz today
- No lecture on Thursday!
- Exam 2 is not graded (there are still make-ups to be taken)

Loot

Example Loot program

Concrete example

```
#lang racket
(define (map f xs)
  (match xs
    ['() '()]
    [(cons x xs)
     (cons (f x) (map f xs))]))

(define (addn n)
  (λ (m) (+ m n)))

(map (addn 5) (cons 1 (cons 2 (cons 3 '()))))
```

Syntactic matters

```
;; type Expr = ...
;;           | (App Expr (Listof Expr))
;;           | (Lam Id (Listof Id) Expr)
```

- Apps now have an arbitrary expression in function position
- Lambdas have an id (inserted by the parser) to “name” it

Syntactic matters

```
;; type Expr = ...
;;           | (App Expr (Listof Expr))
;;           | (Lam Id (Listof Id) Expr)

> (parse-e '(λ (x) x))
'#s(Lam lambda6795915 (x) #s(Var x))
> (parse-e '(λ (x) x))
'#s(Lam lambda6795947 (x) #s(Var x))
> (parse-e '(λ (x y z) x))
'#s(Lam lambda6795991 (x y z) #s(Var x))
>
```

Syntactic matters

```
;; type Expr = ...
;;           | (App Expr (Listof Expr))
;;           | (Lam Id (Listof Id) Expr)
```

```
> (parse-e '(f x))
'#s(App #s(Var f) (#s(Var x)))
> (parse-e '((if y f g) x))
'#s(App #s(If #s(Var y) #s(Var f) #s(Var g)) (#s(Var x)))
```

Semantic matters

New kind of value: function

```
;; type Value = ...
;; | Function (???)
```

Semantic matters

New kind of value: function

```
;; type Value =
;; | ((Listof Value) -> Value { raises 'err })
```

Idea: use Racket functions to represent Loot functions

Lambda expressions

Remember the environment they were created in

```
(let ((f (let ((x 5))
           (λ (y) (+ x y)))))
  (f 1))
```

Lambda expressions

Remember the environment they were created in

```
(let ((f (let ((x 5))
           (λ (y) (+ x y)))))
  (let ((x 6))
    (f 1)))
```

Semantic matters

New kind of value: function

```
;; Expr Env Defns -> Value { raises 'err }
(define (interp-e e r ds) ;; where r closes e
  (match e
    ; ...
    [(Lam _ xs e)
     (λ (vs)
       ; check arity matches
       (if (= (length xs) (length vs))
           (interp-e e (append (zip xs vs) r) ds)
           (raise 'err)))]
    [(App e es)
     (let ((f (interp-e e r ds))
           (vs (interp-e* es r ds)))
       (if (procedure? f)
           (f vs)
           (raise 'err)))]))
```

Lambda without Lambda

The essence of a closure

"Save the environment! Create a closure today!" — Cormac Flanagan

A closure pairs together

- a computation that can be run later
- the environment in which it was created

Semantic matters

New kind of value: function (represented as a closure)

```
;; type Value = ...
;; | (Closure [Listof Id] Expr Env)
(struct Closure (xs e r) #:prefab)
```

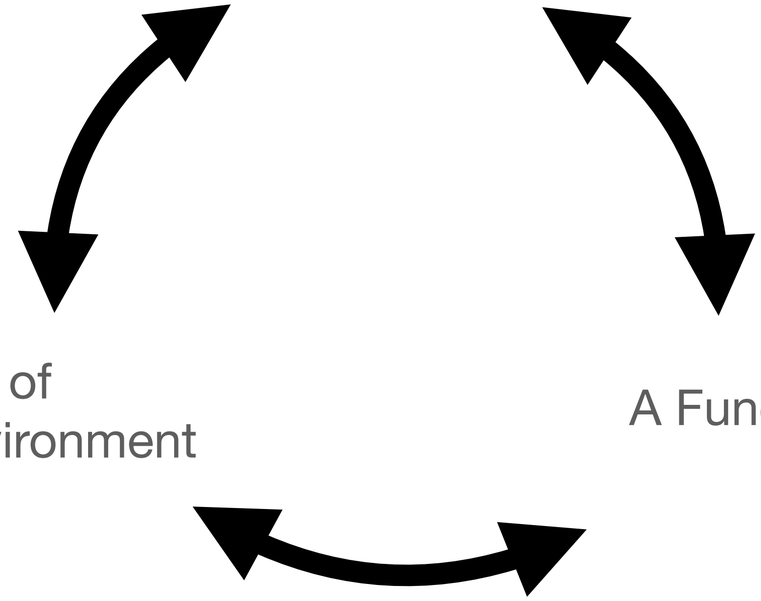
Semantic matters

New kind of value: function (represented as a closure)

```
;; Expr Env Defns -> Value { raises 'err }
(define (interp-e e r ds) ;; where r closes e
  (match e
    ; ...
    [(Lam f xs e)
     (Closure xs e (restrict r (fv (Lam f xs e))))]
    [(App e es)
     (let ((f (interp-e e r ds))
           (vs (interp-e* es r ds)))
       (match f
         [(Closure xs e r)
          ; check arity matches
          (if (= (length xs) (length vs))
              (interp-e e (append (zip xs vs) r) ds)
              (raise 'err))]
         [_ (raise 'err)]))]))
```

The three facets of first-class functions

Calling of a function: Unpack the data structure and
install the values on the stack

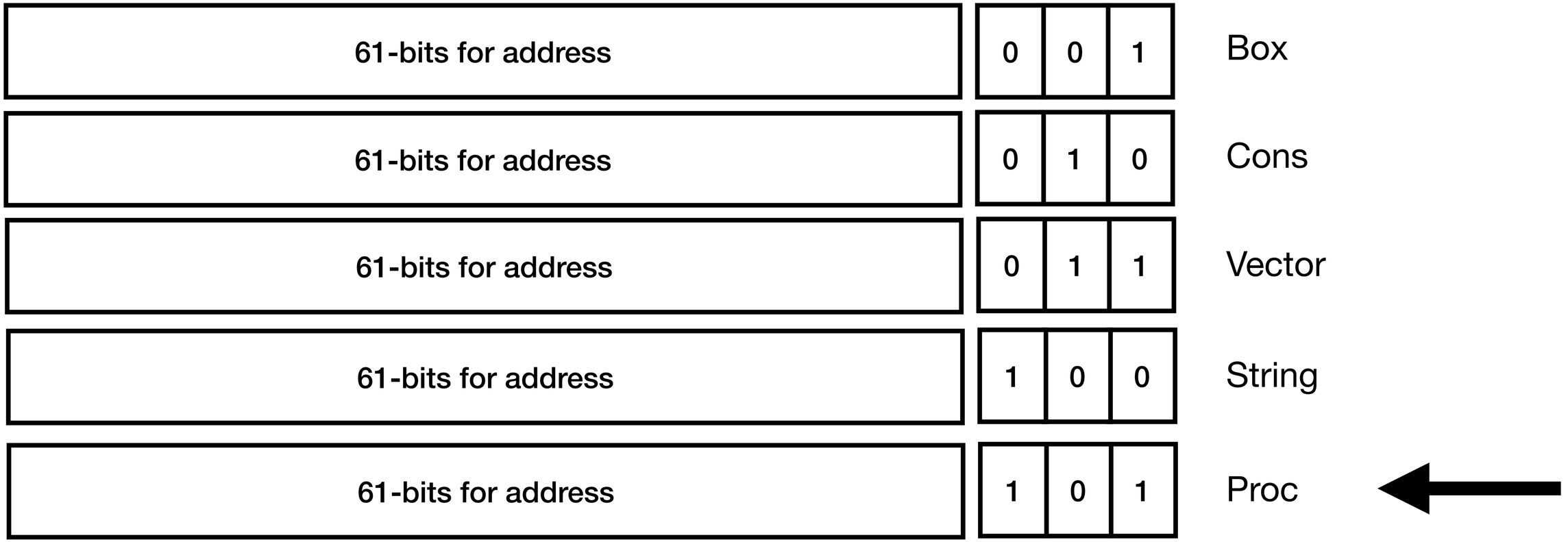


An Expression: Allocation of
data structure packaging up environment

A Function Definition: Code for executing
the body of the function

Encoding pointer values (Loot)

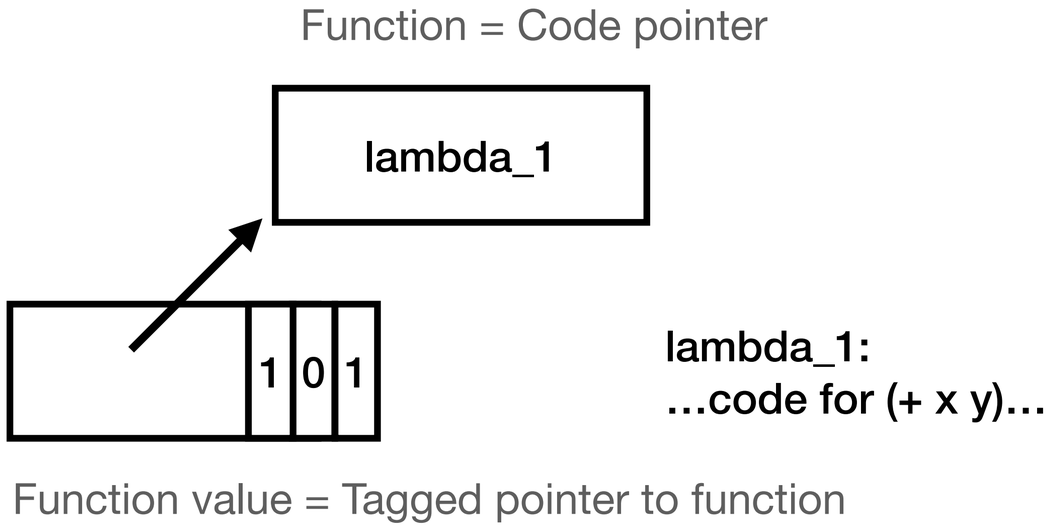
Type tag in least significant bits



Representing (closed) function values

Tagged pointer to memory that holds a code pointer

```
(λ (x y)
   (+ x y))
```



Lambda expressions define functions

First take: compile a function definition for each lambda

```
;; Lam -> Asm
(define (compile-lambda-define l)
  (match l
    [(Lam f xs e)
     (let ((env (reverse (cons #f xs))))
       (seq (Label (symbol->label f))
            (Cmp r8 (length xs))
            (Jne 'err)
            (compile-e e env #t)
            (Add rsp (* 8 (length env)))
            (Ret)))]))
```

Assumes closed!!

Lambda expressions evaluate to function values

First take: lambda expressions evaluate to tagged pointers to code label

Assumes closed!!

```
;; Id [Listof Id] Expr CEnv -> Asm
(define (compile-lam f xs e c)
  (seq (Lea rax (symbol->label f))
       (Mov (Mem rbx) rax)
       (Mov rax rbx) ; return value
       (Xor rax type-proc)
       (Add rbx 8)))
```

Function calls made indirectly

First take: eval function, untag, dereference, and jump

```
;; Id [Listof Expr] CEnv -> Asm
(define (compile-app-nontail f es c)
  (let ((r (gensym 'ret)))
    (seq (Lea rax r)
         (Push rax)
         (compile-es es (cons #f c))
         (Mov r8 (length es))
         (Jmp f)
         (Label r))))
```

```
;; Expr [Listof Expr] CEnv -> Asm
(define (compile-app-nontail e es c)
  (let ((r (gensym 'ret)))
    (seq (Lea rax r)
         (Push rax)
         (compile-es (cons e es) (cons #f c))
         (Mov r8 (length es))
```



Compiler for Jig

Whoops — broke function call protocol with an extra argument

Aside: What if we did the function last?

Could solve our problem and be more efficient!

```
;; Expr [Listof Expr] CEnv -> Asm
(define (compile-app-nontail e es c)
  (let ((r (gensym 'ret)))
    (seq (Lea rax r)
         (Push rax)
         (compile-es es (cons #f c))
         (compile-e e (append (make-list (add1 (length es)) #f) c))
         (Mov r8 (length es))
         (assert-proc rax)
         (Jmp (Mem rax (- type-proc))))
         (Label r)))
```

Pros:
- saves one word word on the stack
- No need to fetch the function from the stack — it's in rax already!
- No longer sends one too many arguments to the function

Cons:
- Broken (w.r.t. our spec)

```
#lang racket
((begin (write-byte 97) (λ (x y) x))
 (begin (write-byte 98) #t)
 (begin (write-byte 99) #f))
```

Aside: What if we did function last?

Could solve our problem and be more efficient!

We could change the spec to go right to left….

```
OCaml version 4.14.0
Enter #help;; for help.

# ((Printf.printf "a"; (fun x y -> x))
   (Printf.printf "b"; true)
   (Printf.printf "c"; false));;
cba- : bool = true
```

OCaml: designed by compiler hackers, for compiler hackers!

OCaml's philosophy: don't rely on order of evaluation in function application; for a particular order, be explicit, e.g.:

```
# let f1 = Printf.printf "a"; (fun x y -> x) in
  let v1 = Printf.printf "b"; true in
  let v2 = Printf.printf "c"; false in
  f1 v1 v2;;
abc- : bool = true
```

Lambda expressions define functions

Second take: every function takes implicit extra argument

Assumes closed!!

```
;; Lam -> Asm
(define (compile-lambda-define l)
  (match l
    [(Lam f xs e)
     (let ((env (reverse (cons #f xs))))
       (seq (Label (symbol->label f))
            (Cmp r8 (length xs))
            (Jne 'err)
            (compile-e e env #t)
            (Add rsp (* 8 (length env)))
            (Ret)))]))
```

Representing functions

Functions are a new kind of value

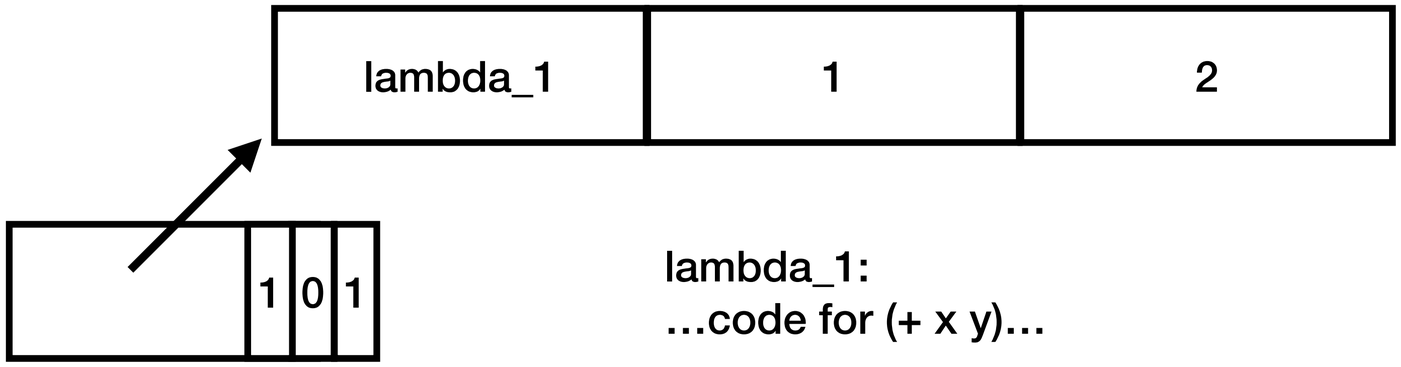
```
;; type Value =
;; ...
;; | (Closure [Listof Id] Expr Env)
(struct Closure (xs e r) #:prefab)
```

Can be used to define a function in assembly

Can be represented as an array of values

```
(let ((x 1))
  (let ((y 2))
    (λ (z) (+ x y))))
```

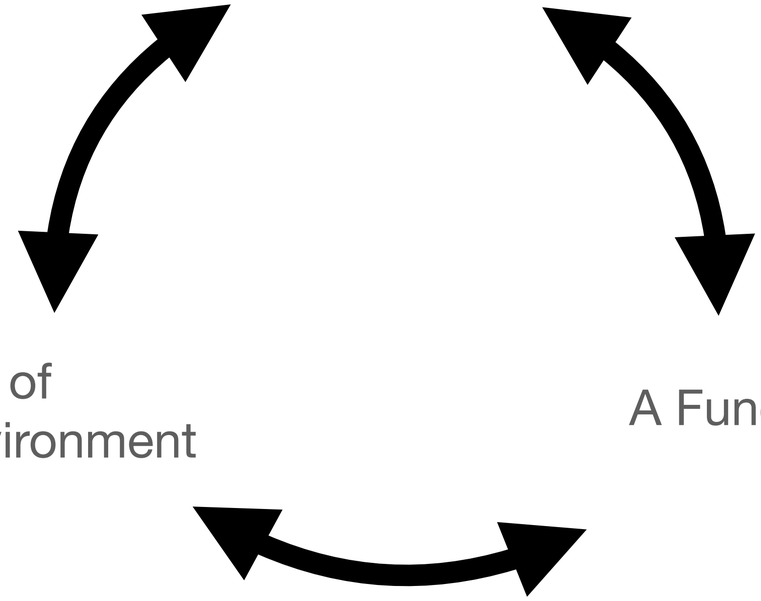
Closure = Code pointer + run-time environment



Function value = Tagged pointer to closure

The three facets of first-class functions

Calling of a function: Unpack the data structure and
install the values on the stack



An Expression: Allocation of
data structure packaging up environment

A Function Definition: Code for executing
the body of the function

Representing functions

Functions are a new kind of value

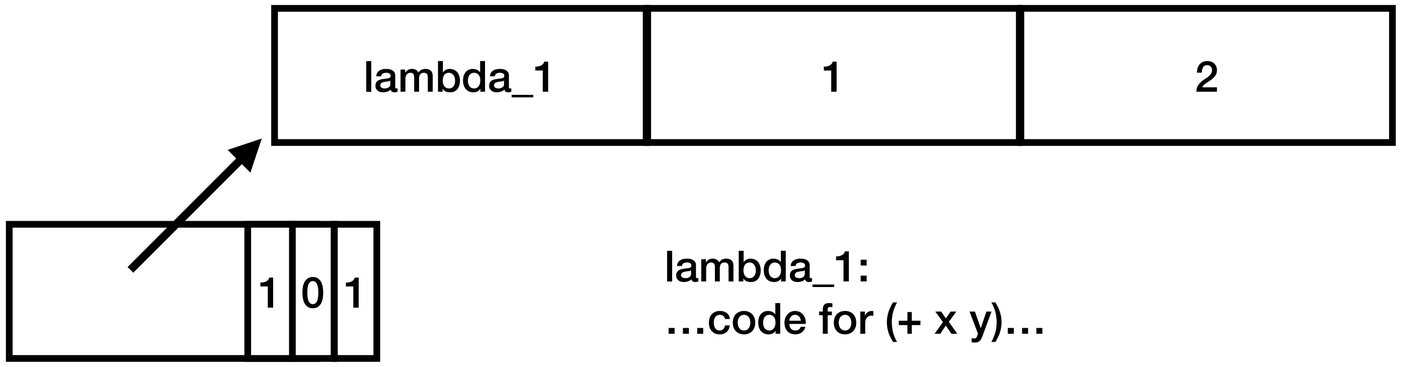
```
;; type Value =
;; ...
;; | (Closure [Listof Id] Expr Env)
(struct Closure (xs e r) #:prefab)
```

Can be used to define a function in assembly

Can be represented as an array of values

```
(let ((x 1))
  (let ((y 2))
    (λ (z) (+ x y))))
```

Closure = Code pointer + run-time environment



Function value = Tagged pointer to closure

Lambda expressions evaluate to function values

Final take: lambda expressions evaluate to tagged pointers to code label AND ENVIRONMENT

```
;; Id [Listof Id] Expr CEnv -> Asm
(define (compile-lam f xs e c)
  (seq (Lea rax (symbol->label f))
       (Mov (Mem rbx) rax)
       (Mov rax rbx) ; return value
       (Xor rax type-proc)
       (Add rbx 8)))
```

Assumes closed!!

```
;; Id [Listof Id] Expr CEnv -> Asm
(define (compile-lam f xs e c)
  (let ((fvs (fv (Lam f xs e))))
    (seq (Lea rax (symbol->label f))
         (Mov (Mem rbx) rax)
         (free-vars-to-heap fvs c 8)
         (Mov rax rbx) ; return value
         (Xor rax type-proc)
         (Add rbx (* 8 (add1 (length fvs)))))))
```

Lambda expressions evaluate to function values

Final take: lambda expressions evaluate to tagged pointers to code label AND ENVIRONMENT

```
;; [Listof Id] CEnv Int -> Asm
(define (free-vars-to-heap fvs c off)
  (match fvs
    ['() (seq)]
    [(cons x fvs)
     (seq (Mov r8 (Mem rsp (lookup x c)))
          (Mov (Mem rbx off) r8)
          (free-vars-to-heap fvs c (+ off 8)))]))
```

Lambda expressions define functions

Final take: functions install the environment of from the closure

```
;; Lam -> Asm
(define (compile-lambda-define l)
  (let ((fvs (fv l)))
    (match l
      [(Lam f xs e)
       (let ((env (append (reverse fvs) (reverse xs) (list #f))))
         (seq (Label (symbol->label f))
              (Cmp r8 (length xs))
              (Jne 'err)

              🤦

              (compile-e e env #t)
              (Add rsp (* 8 (length env))) ; pop env
              (Ret)))])))
```

```
(let ((x 1))
  (let ((y 2))
    (λ (z)
      (+ x (+ y z)))))
```

Lambda expressions define functions

Final take: functions install the environment of from the closure

```
;; [Listof Id] Int -> Asm
;; Copy the closure environment at given offset to stack
(define (copy-env-to-stack fvs off)
  (match fvs
    ['() (seq)]
    [(cons _ fvs)
     (seq (Mov r9 (Mem rax (- off type-proc)))
          (Push r9)
          (copy-env-to-stack fvs (+ 8 off)))]))
```