CMSC 430 - 24 June 2026

Lambdas and first-class functions

- Loot - Lambda
  - wrapping up closures
- Knock - Pattern matching
  - interpreting patterns

CMSC 430 - 24 June 2026

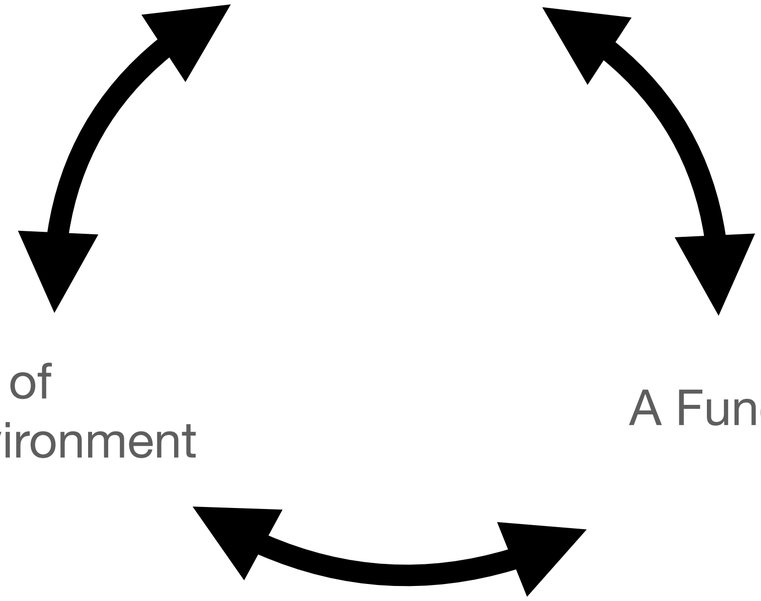
Announcements

- Assignment 7 due Thursday
- Assignment 6 grades released
- Exam 2 Friday
- Practice exam 2 out later today
- Quiz released after class, due by start of class tomorrow

Loot

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

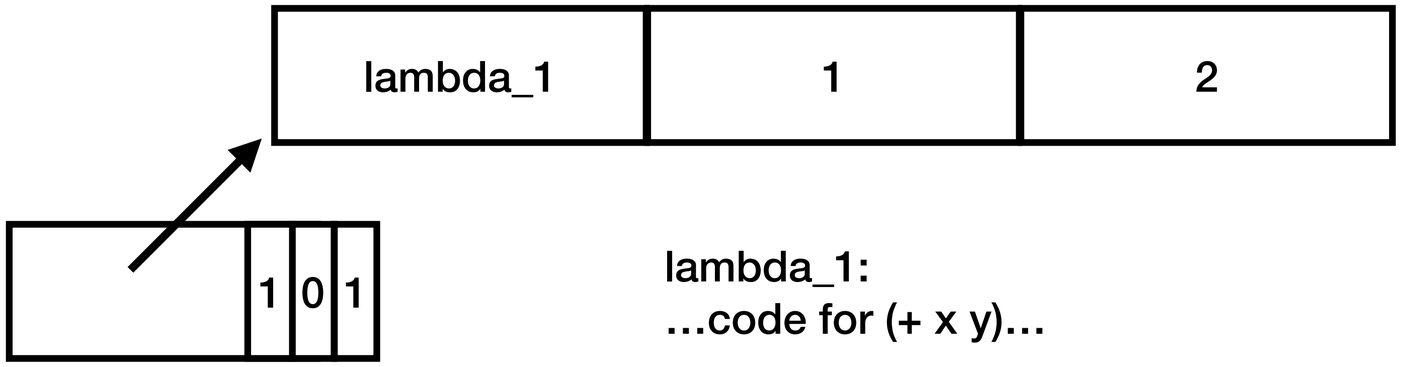
```
;; type Value =
;; ...
;; | (Closure [Listof Id] Expr Env)
(struct Closure (xs e r) #:prefab)
```

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)))]))
```

Knock

Example Knock program

Concrete example

```
#lang racket
(define (add1-lon xs)
  (match xs
    ['() '()]
    [(cons x xs)
     (cons (add1 x)
           (add1-lon xs))]))

(add1-lon (cons 1 (cons 2 (cons 3 '()))))
```

Syntax

Match and patterns

```
;; type Expr = ...
;;           | (Match Expr (Listof Pat) (Listof Expr))

;; type Pat  = (Var Id)
;;           | (Lit Datum)
;;           | (Box Pat)
;;           | (Cons Pat Pat)
;;           | (Conj Pat Pat)
```

Interpretation

Match expressions

```
;; Expr Env Defns -> Value { raises 'err }
(define (interp-e e r ds)
  (match e
    ;...
    [(Match e ps es)
     (let ((v (interp-e e r ds)))
       (interp-match v ps es r ds))]))
```

Interpretation

Match expressions

```
;; Value [Listof Pat] [Listof Expr] Env Defns
;;  -> Value { raises 'err }
(define (interp-match v ps es r ds)
  (match* (ps es)
    [('() '()) (raise 'err)]
    [((cons p ps) (cons e es))
     (match (interp-match-pat p v r)
       [#f (interp-match v ps es r ds)]
       [r  (interp-e e r ds)])]))
```

Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  ...)
```

The heart of pattern matching in the interpreter.

Interpretation

Patterns: Examples

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  ...)
```

```
> (interp-match-pat (Var '_) 1 '())
'()
> (interp-match-pat (Var 'x) 1 '())
'((x 1))
> (interp-match-pat (Lit 1) 1 '())
'()
> (interp-match-pat (Lit 2) 1 '())
#f
```

Interpretation

Patterns: Examples

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  ...)
```

```
> (interp-match-pat (Box (Var '_))
                    (box 1)
                    '())
'()
> (interp-match-pat (Box (Var 'x))
                    (box 1)
                    '())
'((x 1))
> (interp-match-pat (Box (Var 'x))
                    1
                    '())
#f
```

Interpretation

Patterns: Examples

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  ...)
```

```
> (interp-match-pat (Cons (Var '_) (Var '_))
                    (cons 1 2)
                    '())
'()
> (interp-match-pat (Cons (Var 'x) (Var 'y))
                    (cons 1 2)
                    '())
'((y 2) (x 1))
> (interp-match-pat (Cons (Var 'x) (Var 'y))
                    1
                    '())
#f
```

Interpretation

Patterns: Examples

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  ...)
```

```
> (interp-match-pat (Conj (Lit 1) (Lit 1))
                    1
                    '())
'()
> (interp-match-pat (Conj (Lit 1) (Lit 2))
                    1
                    '())
#f
> (interp-match-pat (Conj (Lit 1) (Var 'x))
                    1
                    '())
'((x 1))
```

Interpretation

Patterns: Examples

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  ...)
```

```
> (interp-match-pat (Conj (Var 'y) (Var 'x))
                    1
                    '())
'((x 1) (y 1))
> (interp-match-pat (Conj (Var 'p) (Cons (Var 'x) (Var 'y)))
                    (cons 1 2)
                    '())
'((y 2) (x 1) (p (1 . 2)))
```

Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    ; ...
    [(Var '_) r]))
```

Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    ; ...
    [(Var x) (ext r x v)]))
```

Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    ; ...
    [(Lit l) (and (eqv? l v) r)]))
```

Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    ; ...
    [(Box p)
     (match v
       [(box v)
        (interp-match-pat p v r)]
       [_ #f])]))
```

Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    ; ...
    [(Box p)
     (match v
       [(box v)
        (interp-match-pat p v r)]
       [_ #f])]))
```

Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    ; ...
    [(Cons p1 p2)
     (match v
       [(cons v1 v2)
        (match (interp-match-pat p1 v1 r)
          [#f #f]
          [r1 (interp-match-pat p2 v2 r1)])]
       [_ #f])]))
```

Interpretation

Patterns

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    ; ...
    [(Conj p1 p2)
     (match (interp-match-pat p1 v r)
       [#f #f]
       [r1 (interp-match-pat p2 v r1)])]))
```

Interpretation

Patterns

Putting it all together…

```
;; Pat Value Env -> [Maybe Env]
(define (interp-match-pat p v r)
  (match p
    [(Var '_) r]
    [(Var x) (ext r x v)]
    [(Lit l) (and (eqv? l v) r)]
    [(Box p)
     (match v
       [(box v)
        (interp-match-pat p v r)]
       [_ #f])]
    [(Cons p1 p2)
     (match v
       [(cons v1 v2)
        (match (interp-match-pat p1 v1 r)
          [#f #f]
          [r1 (interp-match-pat p2 v2 r1)])]
       [_ #f])]
    [(Conj p1 p2)
     (match (interp-match-pat p1 v r)
       [#f #f]
       [r1 (interp-match-pat p2 v r1)])]))
```

Compiling a match expression

```
;; Expr [Listof Pat] [Listof Expr] CEnv Bool -> Asm
(define (compile-match e ps es c t?)
  (let ((done (gensym)))
    (seq (compile-e e c #f)
         (Push rax)     ;; save away to be restored by each clause
         (compile-match-clauses ps es (cons #f c) done t?)
         (Jmp 'err)
         (Label done)   ;; where to jump if you've finished a RHS
         (Add rsp 8)))) ;; pop the saved value being matched
```

Compiling a list of match clauses

```
;; [Listof Pat] [Listof Expr] CEnv Symbol Bool -> Asm
(define (compile-match-clauses ps es c done t?)
  (match* (ps es)
    [('() '()) (seq)]
    [((cons p ps) (cons e es))
     (seq (compile-match-clause p e c done t?)
          (compile-match-clauses ps es c done t?))]))
```

Just the sequence of compiling each clause

Compiling a match clause

```
;; Pat Expr CEnv Symbol Bool -> Asm
(define (compile-match-clause p e c done t?)
  (let ((next (gensym)))
    ;; Compiling a pattern needs to know two things:
    ;; - a compile-time environment that describes bindings made so far
    ;; - where to jump in case the pattern doesn't match
    (match (compile-pattern p '() next)
      ;; Compiling a pattern produces two things:
      ;; - instructions that will either:
      ;;   * match value currently in rax,
      ;;     binding things by pushing on the stack
      ;;   * not match, clear out the bindings, and jump to next
      ;; - a compile-time environment that describe the bindings
      ;;   of the pattern, should it succeed
      [(list i cm)
       (seq (Mov rax (Offset rsp 0))       ;; restore value being matched into rax
            i                              ;; do matching, binding, maybe jumping to next
            ;; If you made it here, the pattern matched and the bindings described
            ;; by cm have been pushed onto the stack
            (compile-e e (append cm c) t?) ;; Do the RHS
            (Add rsp (* 8 (length cm)))    ;; Clear out the bindings
            (Jmp done)                     ;; Jump to bottom of clauses code
            (Label next))])))              ;; Where to jump if p doesn't match
```

Compiling a wildcard or variable pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    ;; - Always matches
    ;; - No new bindings
    [(Var '_)
     (list (seq) cm)]
    ;; - Always matches
    ;; - Binds x to value in rax
    [(Var x)
     (list (seq (Push rax)) (cons x cm))]
    #;...))
```

Compiling a literal pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    ;; - Matches if l is equal to value in rax
    ;; - No new bindings
    [(Lit l)
     (let ((ok (gensym)))
       (list (seq (Mov r8 rax)
                  (compile-value l) ;; puts l in rax
                  (Cmp rax r8)
                  (Je ok)
                  (Add rsp (* 8 (length cm)))
                  (Jmp next)
                  (Label ok))
             cm))]
    #;...))
```

Compiling a box pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    ;; - matches if rax is a box and value in box matches p
    ;; - Adds any bindings that p adds
    [(Box p)
     (match (compile-pattern p cm next)
       [(list i1 cm1)
        (let ((ok (gensym)))
          (list
           (seq (Mov r8 rax)
                (And r8 ptr-mask)
                (Cmp r8 type-box)
                (Je ok)
                ; haven't pushed anything yet
                (Add rsp (* 8 (length cm)))
                (Jmp next)
                (Label ok)
                (Xor rax type-box)
                (Mov rax (Offset rax 0))
                i1)
           cm1))])]
    #;...))
```

Compiling an and pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    [(Conj p1 p2)
     (match (compile-pattern p1 (cons #f cm) next)
       [(list i1 cm1)
        (match (compile-pattern p2 cm1 next)
          [(list i2 cm2)
           (list
            (seq (Push rax) ;; save away value so it can be restored for i2
                 i1
                 ;; Reach up into the stack to grab the stowed away value
                 (Mov rax (Offset rsp (* 8 (- (sub1 (length cm1)) (length cm)))))
                 i2)
            cm2)])])]
    #;...))
```

Compiling a cons pattern

```
;; Pat CEnv Symbol -> (list Asm CEnv)
(define (compile-pattern p cm next)
  (match p
    [(Cons p1 p2)
     (match (compile-pattern p1 (cons #f cm) next)
       [(list i1 cm1)
        (match (compile-pattern p2 cm1 next)
          [(list i2 cm2)
           (let ((ok (gensym)))
             (list
              (seq (Mov r8 rax)
                   (And r8 ptr-mask)
                   (Cmp r8 type-cons)
                   (Je ok)
                   (Add rsp (* 8 (length cm))) ; haven't pushed anything yet
                   (Jmp next)
                   (Label ok)
                   (Xor rax type-cons)
                   (Mov r8 (Offset rax 0))
                   (Push r8)                   ; push cdr
                   (Mov rax (Offset rax 8)) ; mov rax car
                   i1
                   (Mov rax (Offset rsp (* 8 (- (sub1 (length cm1)) (length cm)))))
                   i2)
              cm2))])])]
    #;...))
```

For next time

- Read Knock notes
- Take ELMS quiz