

# CS450

## Structure of Higher Level Languages

Lecture 10: Currying, find-one, find-many

Tiago Cogumbreiro

# Today we will learn...

- (revisit) function evaluation
- (revisit) the abstract syntactic tree

# Exercise 1

What is the output of this program?

```
(define x 10)
(define (f x)
  (+ x 20))
(f 30)
```

# Exercise 1

What is the output of this program?

```
(define x 10)
(define (f x)
  (+ x 20))
(f 30)
```

**Output:** 50

Because, parameter x shadows the outermost definition.

# Exercise 2

What is the output of this program?

```
(define x 10)
(define f (lambda (x) (+ x 20)))
(f 30)
```

# Exercise 2

What is the output of this program?

```
(define x 10)
(define f (lambda (x) (+ x 20)))
(f 30)
```

**Output:** 50

The code above is **equivalent** to the code below:

```
(define (f x) (+ x 20))
```

# Exercise 3

What is the output of this program?

```
(define (factory k)
  (lambda () k))

(factory 10)
```

# Exercise 3

What is the output of this program?

```
(define (factory k)
  (lambda () k))

(factory 10)
```

**Output:** #<procedure>

Although if Racket displayed code, we would get: (lambda () 10)

```
((factory 10))
; Outputs: 10
```



# Exercise 3

## Step-by-step evaluation

```
(factory 10) =
( (lambda (k) (lambda () k)) 10) =
; ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the value *bound* to factory
(lambda () 10)
```

Why is factory replaced by a lambda?

User input

```
(define (factory k)
  (lambda () k))
```

Internal representation

```
(define factory
  (lambda (k)
    (lambda () k)))
```

# Exercise 3

Looking at function application more closely

```
(
  (lambda (k)      ; ← parameter k
    (lambda () k)) ; ← body of function
  10               ; ← argument
)
; Remove outer lambda and replace each parameter by argument
; (lambda () k)      ← body of function
; \--- replace parameter k by argument 10
(lambda () 10)      ; ← return value
```

# Exercise 4

Q1: What is the output of this program?

```
(define (f x y)
  (lambda (b)
    (cond [b x] [else y])))

(define g (f 1 2))

g
```

# Exercise 4

Q1: What is the output of this program?

```
(define (f x y)
  (lambda (b)
    (cond [b x] [else y])))

(define g (f 1 2))

g
```

**Output:** (lambda (b) (cond [b 1] [else 2]))

Q2: How do I call g to obtain 1?

# Exercise 4

Q1: What is the output of this program?

```
(define (f x y)
  (lambda (b)
    (cond [b x] [else y])))

(define g (f 1 2))

g
```

**Output:** (lambda (b) (cond [b 1] [else 2]))

Q2: How do I call g to obtain 1?

**Solution:** (g #t)

# The abstract syntactic tree (AST)

Representing code as data structures

# The AST of values

```
value = number | void | func-dec
func-dec = (lambda (variable* ) term+ )
```

## Implementation

```
(define (r:value? v)
  (or (r:number? v)
      (r:void? v)
      (r:lambda? v)))
(struct r:void () #:transparent)
(struct r:number (value) #:transparent)
(struct r:lambda (params body) #:transparent)
```

## How do we represent?

1. 10
2. (void)
3. (lambda () 10)

## AST

# The AST of values

```
value = number | void | func-dec
func-dec = (lambda (variable* ) term+ )
```

## Implementation

```
(define (r:value? v)
  (or (r:number? v)
      (r:void? v)
      (r:lambda? v)))
(struct r:void () #:transparent)
(struct r:number (value) #:transparent)
(struct r:lambda (params body) #:transparent)
```

## How do we represent?

1. 10
2. (void)
3. (lambda () 10)

## AST

```
(r:number 10)      ; ← 1
(r:void)           ; ← 2
(r:lambda (list) ; ← 3
  (list (r:number 10)))
```



# The AST of expressions

```
expression = value | variable | apply
apply = ( expression+ )
```

## Implementation

```
(define (r:expression? e)
  (or (r:value? e)
      (r:variable? e)
      (r:apply? e)))
(struct r:variable (name) #:transparent)
(struct r:apply (func args) #:transparent)
```

## How do we represent?

1. x
2. (f 10)

AST

# The AST of expressions

*expression* = *value* | *variable* | *apply*  
*apply* = ( *expression*<sup>+</sup> )

## Implementation

```
(define (r:expression? e)
  (or (r:value? e)
      (r:variable? e)
      (r:apply? e)))
(struct r:variable (name) #:transparent)
(struct r:apply (func args) #:transparent)
```

## How do we represent?

1. x
2. (f 10)

## AST

```
; 1:
(r:variable 'x)
; 2:
(r:apply
  (r:variable 'f)
  (list (r:number 10)))
```

# The AST of terms

```
term = define | expression
define = ( define identifier expression ) | ( define ( variable+ ) term+)
```

```
(define (r:term? t)
  (or (r:define? t)
      (r:expression? t)))
(struct r:define (var body) #:transparent)
```

Which Racket code is this?

```
(r:define (r:variable 'f)
  (r:lambda (list (r:variable 'y))
    (list
      (r:apply (r:variable '+)
                (list (r:variable 'y) (r:number 10)))))))
```

# The AST of terms

```
term = define | expression
define = ( define identifier expression ) | ( define ( variable+ ) term+)
```

```
(define (r:term? t)
  (or (r:define? t)
      (r:expression? t)))
(struct r:define (var body) #:transparent)
```

Which Racket code is this?

Answer 1

```
(r:define (r:variable 'f)
  (r:lambda (list (r:variable 'y))
    (list
      (r:apply (r:variable '+)
                (list (r:variable 'y) (r:number 10)))))))
```

```
(define (f y) (+ y 10))
```

Answer 2

```
(define f
  (lambda (y) (+ y 10)))
```