CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout

THE UNIVERSITY OF ARIZONA

Version: 1

February 19, 2020

Your turn! Lambda questions

- **What are the results? divvy is? combine is?**

```
(define combine (p? q?)
    (lambda (x) (if (p? x) (q? x) #f)))

(define divvy (p? q?)
    (lambda (x) (if (p? x) #t (q? x))))

(val c-p-e (combine prime? even?))
(val d-p-o (divvy   prime? odd?))

(c-p-e 9) == ?
(c-p-e 8) == ?
(c-p-e 7) == ?

(d-p-o 9) == ?
(d-p-o 8) == ?
(d-p-o 7) == ?
```

Your turn! Lambda calculus questions

- **Answers for lambda calculus?**
  1. What is the abstract syntax? Syntax categories?
  2. What are the values?
  3. What environments are there? What are names mapped to?
  4. How are terms evaluated?
  5. What's in the initial basis? Primitives and otherwise, what is built in?
- **Why do we care about lambda calculus?**

Alpha Conversion and Beta Reduction

Answer to how to evaluate terms (Dr. Sethi's slide)

- Beta Reduction Rule

$$(\lambda x.\ M)\ N \quad \Rightarrow_{\beta} \quad [N\ /\ x]\ M$$

  - In words, $(\lambda x.M)N$ beta-reduces to $[N/x]M$
  - $[N/x]M$ is considered simpler than $(\lambda x.M)N$
  - $(\lambda x.M)N$ is called a *redex*, from "reducible expression"
  - Notation: $\Rightarrow_{\beta}^{*}$ represents a sequence of zero or more beta reductions

- Alpha Conversion Rule

$$\lambda x.\ M \quad \Rightarrow_{\alpha} \quad \lambda y.\ [y\ /\ x]\ M \qquad\qquad \text{if } y \text{ is not free in } M$$

Languages use lambda calculus concepts

- **Correspondence between lambda calculus and lambdas in scheme**
  - Lambda calculus

$M \rightarrow$	x	*variables*
$\vert$	$(M\,M)$	*function application*
$\vert$	$(\lambda\, x.\, M)$	*abstraction*

  - Scheme expressions

$$
\begin{array}{lcl}
\mathit{exp} & ::= & \mathit{literal} \\
 & \mid & \mathit{variable\text{-}name} \\
 & \mid & (\texttt{set}\ \ \mathit{variable\text{-}name}\ \mathit{exp}) \\
 & \mid & (\texttt{if}\ \ \mathit{exp}\ \mathit{exp}\ \mathit{exp}) \\
 & \mid & (\texttt{while}\ \ \mathit{exp}\ \mathit{exp}) \\
 & \mid & (\texttt{begin}\ \ \{\mathit{exp}\}) \\
 & \mid & (\mathit{exp}\ \{\mathit{exp}\}) \\
 & \mid & (\mathit{let\text{-}keyword}\ (\{[\mathit{variable\text{-}name}\ \mathit{exp}]\})\ \ \mathit{exp}) \\
 & \mid & (\texttt{lambda}\ \ (\mathit{formals})\ \ \mathit{exp})
\end{array}
$$

Midterm: Be able to identify free variables

THE UNIVERSITY OF ARIZONA
Computer Science

	Free vars
$(x\ y)$	________________
$\lambda x.x\ y$	________________
$\lambda x.(x\ y\ (\lambda\ y.zyx))$	________________

Midterm: Be able to do substitution

rules

$$
\begin{aligned}
[N/x]x &= N \\
[N/x]y &= y, \text{ if } x \neq y \\
[N/x]\,(M_1\ M_2) &= ([N/x]M_1)\ ([N/x]M_2) \\
[N/x]\,(\lambda x.M) &= \lambda x.M \\
[N/x]\,(\lambda y.M) &= \lambda y.\ [N/x](M), \text{ if } x \neq y \text{ and } y \notin free(N)
\end{aligned}
$$

Which cases of rules apply?

$[u / x]\ (x\ y)$ = ____________

$[\lambda y.y / y]\ x$ = ____________

$[u / x]\ (\lambda y.\ y\ x)$ = ____________

$[u / x]\ (\lambda u.\ x)$ = ____________

$[u / x]\ (\lambda u.\ u\ x)$ = ____________

Escaping functions (and why free vars are important)

- **"Escape" means "outlive the function in which lambda was evaluated"**
  - Typically returned
  - More rarely, stored in a global variable or a heap-allocated structure
- **We have already seen an example**

```
(define to-the-n-minus-k (n k)
    (lambda (x) (- (exp x n) k)))
```

- **Values that escape have to be allocated on the heap**
  - C programmers use malloc
  - In a language with first-class, nested functions, storage of escaping values is part of the semantics of the lambda

Closures represent escaping functions

Function value representation: $(\!|\lambda x.e, \rho|\!)$

(ρ binds the free variables of e)

A closure is a heap-allocated record containing

- **a pointer to the code**
- **an environment storing free variables**

$$(\!|\bullet, \{n \mapsto 3, k \mapsto 27\}|\!)$$

code for `x-to-the-n-minus-k`

What’s the closure for conjunction?

```
(define conjunction(p? q?)
    (lamba (x) (if (p? x) (q? x) #f)))
```

Closure for conjunction

$$\begin{tikzcd} (\!|\bullet, \{p? \mapsto \bullet, q? \mapsto \bullet\}|\!) \arrow[d, bend right] \\ \text{code for } \texttt{(lambda (x) (if ...))} \end{tikzcd}$$

Higher-order functions

THE UNIVERSITY OF ARIZONA
Computer Science

- **Preview: in math, what is the following?**

```
(f o g)(x) == ???
```

- **Another algebraic law, another function:**

```
(f o g)(x) == f(g(x))
(f o g) == \x. (f(g(x)))
```

- **Functions create new functions**

```
-> (define o (f g) (lambda (x) (f (g x))))
-> (define even? (n) (= 0 (mod n 2)))
-> (val odd? (o not even?))
-> (odd? 3)
?
-> (odd? 4)
?
```

Currying

- **Currying converts a binary function f(x,y) to**
  - A function f' that takes x and returns …
  - a function f'' that takes y and returns f(x,y)
- **Handy: now all functions just take one parameter!**

```
-> (val positive? (lambda (y) (< 0 y)))
-> (positive? 3)
?
-> (val <-c (lambda (x) (lambda (y) (< x y))))
-> (val positive? (<-c 0)) ;; partial application
-> (positive? 0)
??
```

What’s the algebraic law for curry?

- **Keep in mind that you can apply a function**

```
... (curry f) ... = ... f ...
```

```
(((curry f) x) y) = (f x y)
```

No need to curry by hand!

```
;; curry : binary function -> value -> function
-> (val curry
     (lambda (f)
       (lambda (x)
         (lambda (y) (f x y)))))
-> (val positive? ((curry <) 0))
-> (positive? -3)
#f
-> (positive? 11)
#t
```

Exercises

- **What is the result of the following expressions?**

```
-> (map ((curry +) 3) '(1 2 3 4 5))
?
-> (exists? ((curry =) 3) '(1 2 3 4 5))
?
-> (filter ((curry >) 3) '(1 2 3 4 5))
```

Exercises, vulnerable variables?

- **What is the problem with this?**

```
-> (val seed 1)
-> (val rand (lambda ()
          (set seed (mod (+ (* seed 9) 5) 1024))))
-> (rand)
14
-> (rand)
131
-> (set seed 1)
1
-> (rand)
14
```

Exercises, Lambda as abstraction barrier

THE UNIVERSITY OF ARIZONA
Computer Science

- **Instead use lambda to give a variable "private" access**

```
-> (val mk-rand (lambda (seed)
        (lambda () (set seed (
                              mod (+ (* seed 9) 5) 1024))))))
-> (val rand (mk-rand 1))
-> (rand)
14
-> (rand)
131
-> (set seed 1)
error: set unbound variable seed
-> (rand) 160
```