CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout

THE UNIVERSITY OF ARIZONA

Version: 1

February 12, 2020

Today

- **Finish Let constructs**
- **Lambda functions**

Let construct (FINISHED?)

- **To introduce local names into environment**

```
(let ([x1 e1]
      ...
      [xn en])
   e)
```

- **Evaluate e1 through en, bind answers to x1, …, xn**

$$\frac{\begin{array}{c} x_1, \ldots, x_n \text{ all distinct} \\ \ell_1, \ldots, \ell_n \notin \operatorname{dom} \sigma_n \text{ (and all distinct)} \\ \sigma_0 = \sigma \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e, \rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{LET}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\text{LET})$$

Let* construct

- **How is let* different than let?**

```
(let* ([x1 e1]
       ...
       [xn en])
   e)
```

$$\frac{\begin{array}{c} \rho_0 = \rho \qquad \sigma_0 = \sigma \\ \langle e_1, \rho_0, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_0' \rangle \qquad \ell_1 \notin \operatorname{dom} \sigma_0' \qquad \rho_1 = \rho_0\{x_1 \mapsto \ell_1\} \qquad \sigma_1 = \sigma_0'\{\ell_1 \mapsto v_1\} \\ \vdots \\ \langle e_n, \rho_{n-1}, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_{n-1}' \rangle \\ \ell_n \notin \operatorname{dom} \sigma_{n-1}' \qquad \rho_n = \rho_{n-1}\{x_n \mapsto \ell_n\} \qquad \sigma_n = \sigma_{n-1}'\{\ell_n \mapsto v_n\} \\ \langle e, \rho_n, \sigma_n \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \mathrm{LETSTAR}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\textsc{Letstar})$$

Letrec construct

- **How is letrec different than let and let*?**

```
(letrec ([x1 e1]
         ...
         [xn en])
    e)
```

$$\frac{\begin{array}{c}\ell_1,\ldots,\ell_n \notin \operatorname{dom}\sigma \text{ (and all distinct)} \\ x_1,\ldots,x_n \text{ all distinct} \\ e_i \text{ has the form } \text{LAMBDA}(\cdots),\ 1 \le i \le n \\ \rho' = \rho\{x_1 \mapsto \ell_1,\ldots,x_n \mapsto \ell_n\} \qquad \sigma_0 = \sigma\{\ell_1 \mapsto \text{unspecified},\ldots,\ell_n \mapsto \text{unspecified}\} \\ \langle e_1,\rho',\sigma_0\rangle \Downarrow \langle v_1,\sigma_1\rangle \\ \vdots \\ \langle e_n,\rho',\sigma_{n-1}\rangle \Downarrow \langle v_n,\sigma_n\rangle \\ \langle e,\rho',\sigma_n\{\ell_1 \mapsto v_1,\ldots,\ell_n \mapsto v_n\}\rangle \Downarrow \langle v,\sigma'\rangle\end{array}}{\langle \text{LETREC}(\langle x_1,e_1,\ldots,x_n,e_n\rangle,e),\rho,\sigma\rangle \Downarrow \langle v,\sigma'\rangle} \quad (\text{LETREC})$$

Letrec construct

- **Why not just use multiple global defines?**

$$\frac{x \notin \mathrm{dom}\,\rho \qquad \ell \notin \mathrm{dom}\,\sigma \qquad \langle e, \rho\{x \mapsto \ell\}, \sigma\{\ell \mapsto \text{unspecified}\}\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \mathrm{VAL}(x, e), \rho, \sigma\rangle \to \langle \rho\{x \mapsto \ell\}, \sigma'\{\ell \mapsto v\}\rangle} \quad (\textsc{DefineNewGlobal})$$

$$\frac{\begin{array}{c} \ell_1, \ldots, \ell_n \notin \mathrm{dom}\,\sigma \text{ (and all distinct)} \\ x_1, \ldots, x_n \text{ all distinct} \\ e_i \text{ has the form } \mathrm{LAMBDA}(\cdots),\ 1 \le i \le n \\ \rho' = \rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\} \qquad \sigma_0 = \sigma\{\ell_1 \mapsto \text{unspecified}, \ldots, \ell_n \mapsto \text{unspecified}\} \\ \langle e_1, \rho', \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \\ \vdots \\ \langle e_n, \rho', \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ \langle e, \rho', \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\}\rangle \Downarrow \langle v, \sigma'\rangle \end{array}}{\langle \mathrm{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad (\textsc{Letrec})$$

Letrec construct

- **How is letrec different than ApplyClosure?**

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \notin \operatorname{dom} \sigma \text{ (and all distinct)} \\ x_1, \ldots, x_n \text{ all distinct} \\ e_i \text{ has the form } \mathrm{LAMBDA}(\cdots),\ 1 \le i \le n \\ \rho' = \rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\} \qquad \sigma_0 = \sigma\{\ell_1 \mapsto \text{unspecified}, \ldots, \ell_n \mapsto \text{unspecified}\} \\ \langle e_1, \rho', \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho', \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e, \rho', \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle\end{array}}{\langle \mathrm{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\textsc{Letrec})$$

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \notin \operatorname{dom} \sigma_n \text{ (and all distinct)} \\ \langle e, \rho, \sigma \rangle \Downarrow \langle (\!|\mathrm{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e_c), \rho_c |\!), \sigma_0 \rangle \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e_c, \rho_c\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle\end{array}}{\langle \mathrm{APPLY}(e, e_1, \ldots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\textsc{ApplyClosure})$$

Impcore to uScheme

- **Things that should bother you about Impcore**
  - Looking up a function and looking up a variable require different interfaces
  - To get a variable you must check 2 or 3 environments
  - Can't create a function without giving it a name
    - High cognitive overhead
    - A sign of a second-class citizenship
- **All these problems have one solution: Lambda!**

Anonymous, first-class functions

- From Church's lambda-calculus

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

- "The function that maps x to x plus x"
- At top level, like define. Define is a synonym for a lambda that also gives the lambda a name.
- In general, `\x.E` or `(lambda (x) E)`
  - x is bound in E
  - Other variables are free in E (enables capturing these variables)
- Functions become just like any other value

First-class, nested functions

- What does this mean?

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

- Lambda expressions can be passed as parameters

```
(sort (lambda (v w) (< v w)) xs)
```

- Lambda expressions can be nested (i.e. returned)

```
(val add (lambda (x) (lambda (y) (+ x y))
```

- => Can we do this in C? Java? Python?

Lambda Expressions in Python

- **Example**

```
>>> f = lambda x, y : x + y
>>> f(1,1)
2
```

- **=> How does the below work?**

```
>>> fib = [0,1,1,2,3,5,8,13,21,34,55]
>>> result = filter(lambda x: x%2, fib)
>>> print result
[1, 1, 3, 5, 13, 21, 55]
>>> result = filter(lambda x : x%2 == 0, fib)
>>> print result
[0, 2, 8, 34]
```

Lambda Expressions for Java Event Handling

- **Only need to specify the handle method**
- **Lambda expressions enable doing that with less syntax**

```java
    public void start(Stage stage) {
        BorderPane p = new BorderPane();

        // Handling a button press
        Button b = new Button("PressMe");
        b.setOnAction((e) -> {
                System.out.println("Main5: Button was pushed");
        });
```

Lambda Expression Syntax in Java

- **Syntax**

```
() -> expr
(p1, p2, ..., pn) -> expr
(p1, p2, ..., pn) -> {...; return expr;}
```

- **=> Activity (1)**

Using Lambda Expressions in Java

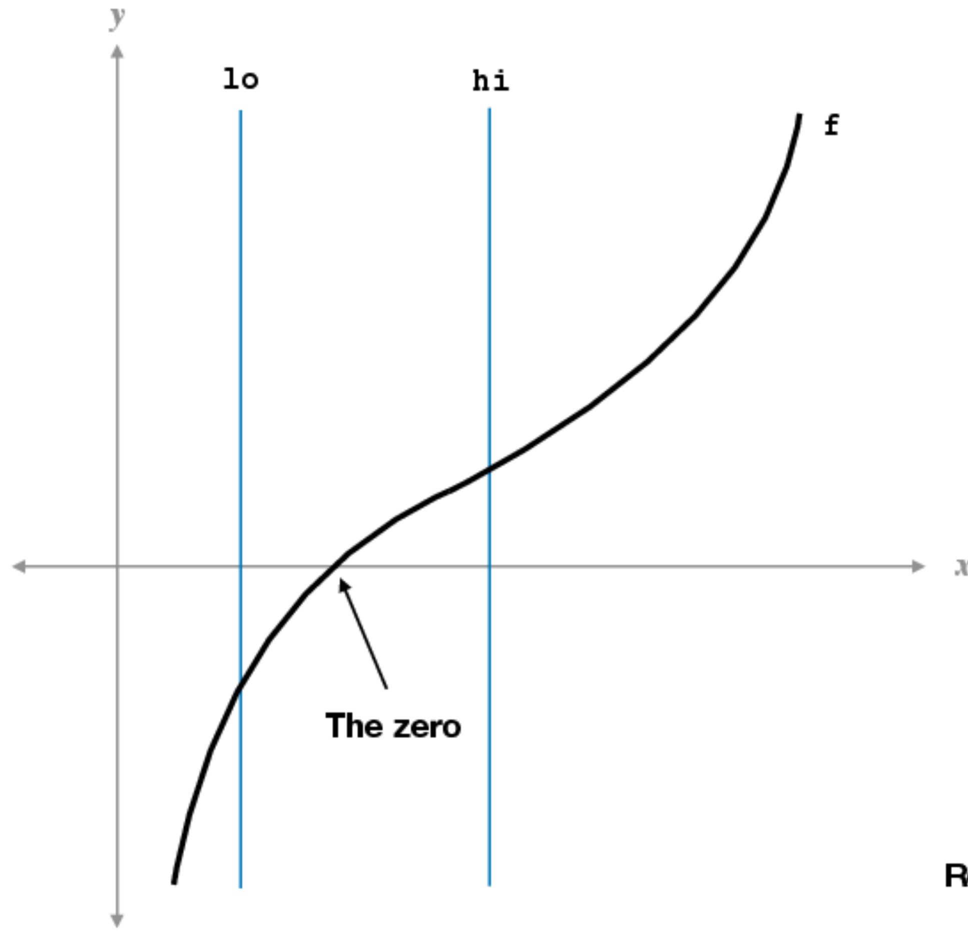
- Passing as parameters
- Assigning to a variable
- Capturing variables
  - Locals and parameters, just copies, need to be essentially final
  - Static and member variables, capture references to these

- => Activity (2)
- => Activity (3)

Using Nested Lambda Functions in uScheme

- **Finding roots**
  - Given n and k, find an x such that x^n = k
  - Step 1: write a function that computes x^n = k
  - Step 2: write a function that finds a zero between lo and hi bounds
  - Algorithm uses binary search over integer interval between lo and hi.
  - Finds x point in interval where function x^n - k is closest to zero.

Using binary search to find the zero of a function



Repeat…

A function that computes x^n = k

- Assigning lambda to local var and returning that

```
-> (define to-the-n-minus-k (n k)
       (let ((x-to-the-n-minus-k
              (lambda (x) (- (exp x n) k))))
       x-to-the-n-minus-k))
-> (val x-cubed-minus-27 (to-the-n-minus-k 3 27))
-> (x-cubed-minus-27 2)
-19
```

- to-the-n-minus-k is a higher-order function, why?

- No need to name the escaping function

```
-> (define to-the-n-minus-k (n k)
     (lambda (x) (- (exp x n) k)))
-> (val x-cubed-minus-27 (to-the-n-minus-k 3 27))
-> (x-cubed-minus-27 2)
-19
```

The zero-finder

- **Finding roots**
  - Given n and k, find an x such that x^n = k
  - Step 1: write a function that computes x^n = k

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

  - Step 2: write a function that finds a zero between lo and hi bounds

```
(define findzero-between (f low hi)
    ;; binary search
    (if (>= (+ lo 1) hi)
        hi
        (let ((mid (/ (+ lo hi) 2)))
            (if (< (f mid) 0)
                (findzero-between f mid hi)
                (findzero-between f mid hi)))))
(define findzero (f) (findzero-between f 0 100))
```

Your turn! Lambda questions

- **What are the results?**

```
(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) == ?
```