CS3L:
Introduction to Symbolic Programming

Lecture 16:
Let and Lambda

Summer 2008 Colleen Lewis
colleenL@berkeley.edu

Today

- Homework
  - **Mon:** Mini-project 2 due Thursday at 11:59 pm
- **`let`**
- **`lambda`**

Treating functions as things

- "define" associates a name with a value
  - The usual form associates a name with a object that is a function

```
(define (square x) (* x x))
(define (pi) 3.1415926535)
```

  - You can define other objects, though:

```
(define *pi* 3.1415926535)
(define *month-names*
   '(january february march april may
     june july august september
     october november december))
```

"Global variables"

- Functions are "global", in that they can be used anywhere:

```
(define (pi) 3.1415926535)
(define (circle-area radius)
     (* (pi) radius radius))
```

- A "global" variable, similarly, can be used anywhere:

```
(define *pi* 3.1415926535)
(define (circle-area radius)
     (*  *pi*  radius radius))
```

Are these the same?

Consider two forms of "month-name":

```
(define (month-name1 date)
    (first date))

(define month-name2 first)
```

Let

```
( let
  ( (variable1 value1) ;;definition 1
   (variable2 value2) ;;definition 2
   )
   statement1 ;;body
   statement2 ... )
```

Using `let` to define temporary variables

- `let` lets you define variables within a procedure:

```
(define (scramble-523 wd)
   (let ((second (first (bf wd)))
         (third  (first (bf (bf wd))))
         (fifth  (item 5 wd))
         )
     (word fifth second third) ) )
```

```
(scramble-523 'meaty) ➔ yea
```

Let

```
(let
  ((variable1 value1) ;;definition 1
   (variable2 value2) ;;definition 2
   )
   statement1 ;;body
   statement2 ...  )
```

Three ways to define a variable

1. In a procedure call (e.g., the variable `proc`):

```
(define (doit proc value)
    ;; proc is a procedure here…
    (proc value))
```

2. As a global variable

```
(define *alphabet* '(a b c d e … ))
(define *month-name* '(january … ))
```

3. With `let`

Which pi?

```
(define (square x)
  (let ((pi 3.1))
     (* pi pi)))          (square 1) ➔

(define (square pi)
  (let ((pi 3.1))
     (* pi pi)))          (square 1) ➔

(define pi 3.1415)
(define (square pi)
  (let ((pi 3.14))
     (* pi pi)))          (square 1) ➔
```

Which pi?

```
(define pi 3.1415)
(define (square x)
     (* pi pi))          (square 1) ➔

(define pi 3.1415)
(define (square pi)
     (* pi pi))          (square 1) ➔
```

Anonymous functions:

using `lambda`

the lambda form

- "lambda" is a special form that returns a function.

```
(lambda (arg1 arg2 …)
   statements
      )
```

```
(lambda        (x)        (*    x    x))
   ↑            ↑          ↑    ↑    ↑
a procedure  that takes one argument  and multiplies  it  by itself
```

Using lambda with define

- These are the same:

```
(define (square x)
   (* x x))


(define square
    (lambda (x) (* x x))
    )
```

Using lambda with define

- These are VERY DIFFERENT:

```
(define (adder-1 y)
   (lambda (x) (+ x 1)))


(define adder-2
   (lambda (x) (+ x 1)))
```

Accumulate

```
(define (my-accum1 accum-proc num-sent)
  (if (= (count num-sent) 1)
    (first num-sent)
    (accum-proc
      (first num-sent)
      (my-accum1 accum-proc (bf num-sent)) ) )


          > (my-accum1 - '245)
```

```
(define (my-accum1 accum-proc num-sent)
  (if (= (count num-sent) 1)
    (first num-sent)
    (accum-proc
      (first num-sent)
      (my-accum1 accum-proc (bf num-sent)) ) )
```

```
> (my-accum1 - '245)
```

(my-accum1 - '(2 4 5))

(my-accum1 - '(4 5))

(my-accum1 - '(5))

5

(- 4

(- 2

```
➔ (- 2 (- 4 5))
➔ 3
```