Lecture 3: Higher Order Procs

CS 61A
Summer 2006
Instructor: Kevin Lin

1

Administrative stuff

- Lab 1B and Homework 1B are online
- Activate your cardkeys! 387 Soda
- My OH today: 3:30-4:30 PM, 751 Soda
- Permanent OH: will be posted online this weekend
- TAs should also set up their office hours by early next week

2

How to do well in this class

- Start homework and projects EARLY!
- The time that you spend playing around with the STk interpreter and programming should be AT LEAST as much as the time you spend on the reading.
- The best way to learn the material is to try it out yourself by programming.

3

Applicative order / Normal order

This is Exercise 1.5 from the textbook:

```
(define (p) (p))
(define (test x y)
  (if (= x 0)
      0
      y))
(test 0 (p))
```

What happens here with normal order evaluation? What happens here with applicative order evaluation?

4

cond

The syntax for cond is as follows:

```
(cond (<if 1> <then 1>)
      (<if 2> <then 2>)
      ...
      (<if n> <then n>)
      (else <expr>))
```

cond is a special form! (What are the other special forms that we've seen so far?)

5

Internal defines

The Pig Latin procedure could have been defined as follows:

```
(define (pigl wd)
  (define (pl-done? wd)
    (vowel? (first wd)))
  (define (vowel? letter)
    (member? letter '(a e i o u)))
  (if (pl-done? wd)
      (word wd 'ay)
      (pigl (word (bf wd) (first wd)))))
```

6

Programming practice!!!

Write a procedure called reverse that takes a word and returns a new word with the characters reversed.

```
> (reverse 'paranoid)
dionarap
```

Write a procedure called interleave that takes two sentences of the same length and returns a new sentence with the words interleaved.

```
> (interleave '(a b c) '(1 2 3))
(a 1 b 2 c 3)
```

Write a procedure called dupls-removed that, given a sentence as input, returns the result of removing duplicate words from the sentence. (This is from lab.)

7

pigl-sent and argue

```
(define (pigl-sent sent)
  (if (empty? sent)
      '()
      (se (pigl (first sent))
          (pigl-sent (bf sent)))))
(define (argue sent)
  (if (empty? sent)
      '()
      (se (opposite (first sent))
          (argue (bf sent)))))
```

8

every

```
(define (<proc-name> sent)
  (if (empty? sent)
      '()
      (se (<word-proc> (first sent))
          (<proc-name (bf sent)))))
```

How can we generalize this?

9

every

```
(define (every proc sent)
  (if (empty? sent)
      '()
      (se (proc (first sent))
          (every proc (bf sent)))))
```

We can do this because in Scheme, procedures are first-class. They can be passed in as arguments to other procedures!

10

sum

```
(define (sum-squares a b)
  (if (> a b)
      0
      (+ (square a)
         (sum-squares (+ a 1) b))))
(define (square x) (* x x))
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a)
         (sum-cubes (+ a 1) b))))
(define (cube x) (* x x x))
```

11

sum

Generalizing the pattern ...

```
(define (sum fn a b)
  (if (> a b)
      0
      (+ (fn a) (sum fn (+ a 1) b))))
```

Now we can do things like:

```
(define (sum-squares a b)
  (sum square a b))
(sum first a b)
(sum cos a b)
```

12

every

```
(define (pigl-sent sent)
  (every pigl sent))
(define (argue sent)
  (every opposite sent))
(define (square-sent sent)
  (every square sent))
(define (plural-sent sent)
  (every plural sent))
```

This makes it easier and faster to write a lot of different and useful procedures!

every

How do you use every properly?

```
> (every first '(abba zabba))
(a z)
> (every (first) '(abba zabba))
Error: too few arguments to (first)
> (every (* x x) '(1 2 3 4))
Error: unbound variable x
```

Is every a special form?
No.

Higher order procedures

A data type is considered *first-class* in a language if it can be:
-the value of a variable (i.e. named)
-an argument to a function
-the return value from a function
-a member of an aggregate

So far we've seen that numbers and words are first-class.
We'll see that procedures are also first class.

Higher order procedures

A data type is considered *first-class* in a language if it can be:
-the value of a variable (i.e. named)
-an argument to a function
-the return value from a function
-a member of an aggregate

We've seen named procedures
We've seen procedures as argument to other procedures (as in the case of every)
Now what about procedures as return values?

Higher order procedures

A higher order procedure is simply a procedure that does at least one of the following:

- takes another procedure as argument
- returns a procedure

Back to every

Suppose we wanted to add 3 to every number in a sentence.

```
(define (plus-3 x) (+ 3 x))
(every plus-3 '(50 29 38 30 3))
```

If we only need to do this once, then it seems kind of wasteful to define a procedure we're only going to use once, isn't it?

Back to every

So instead, we can do this:

```
(every (lambda (x) (+ 3 x)) '(1 2 3 4))
```

This works as expected, and returns the sentence (4 5 6 7).

19

Lambda

Lambda is a special form that returns a nameless function!

```
(lambda (<parameters>) <body>)

> (every (lambda (x) (word (first x)
(last x)) '(hey you over there))
(hy yu or te)
```

20

Lambda

Now, using lambda, we can write procedures that return other procedures!

```
(define (make-adder n)
  (lambda (x) (+ x n)))

> ((make-adder 3) 5)
8
> (every (make-adder 5) '(1 2 3))
(6 7 8)
> ((make-adder 9) 11)
20
```

21

Lambda

We can also do funky things like procedures that take procedures as argument and also return procedures.

```
(define (compose-with-1+ proc)
  (lambda (x) (+ 1 (proc x))))
> (compose-with-1+ square)
#[closure...]
> ((compose-with-1+ (lambda (x) (* 2 x))) 2)
5
> ((compose-with-1+ square) 3)
10
```

22

Lambda

Now, how about a procedure that returns a procedure that returns a procedure?

```
(define (f x)
  (lambda (y) (lambda (z) (+ x y z))))
> (((f 1) 2) 3)
6

(define (g)
  (lambda (x) (lambda () (+ x x))))
> (((g) 2))
4
```

23