Lecture 4: More on higher order procedures

Summer 2006

CS 61A

Instructor: Kevin Lin

1

Administrative stuff

- No class next Tuesday (Independence Day).
- Reading for next week: 1.2.1-4;
  1.2.6 optional but in my opinion it's very cool and fun.
- Homework 1A and 1B due next Wednesday; Project 1 due next Thursday.
- I encourage you to use the newsgroup for questions related to homework/projects.
- Web access to newsgroup:
  http://inst.eecs.berkeley.edu/webnews
  (Use your cs61a-xx login)

2

Find the error

```
(define f lambda(x) (+ x x))

(lambda (x) (+ x x))(1)

(cond ((even? 3) (+ 1 2))
      ((odd? 4) 'cs)
      (else ('61a)))

(every (lambda (x) *) '(1 2 3))
```

Sentences can only hold words! That means: no procedures, no booleans, ...

3

"Non-linear" recursion

We've seen lots of examples of "linear" recursion, where we march down a sentence, doing one thing or another to each word in the sentence.

Don't think that this is the only way we'll ever do things!

Write a procedure `palindrome?` that takes a word and checks whether it is a palindrome.
Note: it may be helpful (but is not necessary) to use the procedure `count`, which returns the length of a word or a sentence.

4

Scoping rules of Scheme

Scheme uses what's called static scope (also known as lexical scope in the textbook.)
What's the return value of the final expression?

```
(define (f x)
  (define (g x)
    (+ x x))
  (g 5))
(f 10)
```

5

Scoping rules of Scheme

What's the return value of the final expression?

```
(define y 2)
(define (f x y)
  (g x))
(define (g z)
  (+ z y))
(f 5 6)
```

6

Scoping rules of Scheme

Scope can be tricky. We'll learn the detailed rules later. It can get pretty complicated. However, if every variable you use has a different name, it's usually very easy to figure out what's what. Unfortunately that's not always very convenient.

You don't have to worry about this.
Yet.

7

Scheme idiosyncrasies

```
; We didn't have time to go over this in
; lecture. We'll talk about it next time.

; What gets evaluated and what doesn't?
(define pi 3.1415926)
(define temp (+ 4 5))
(define f (lambda (x) (+ x x)))

; What gets evaluated and what doesn't?
(define (foo) (+ 4 5))
(define (f x) (+ x x))
```

8

Back to higher order procedures

Write a procedure `keep` that takes a predicate procedure (that is, a true/false procedure) and a sentence and returns a new sentence of words which satisfy the predicate, e.g.

```
> (keep (lambda (x) (= x 1)) '(1 2 1 2))
(1 1)

(define (keep pred sent)
  (cond ((empty? sent) '())
        ((pred (first sent))
                  (se (first sent)
                      (keep pred (bf sent))))
        (else
              (keep pred (bf sent)))))
```

9

Higher order procedures

The keep and every higher order procedures are built-in functions in the version of scheme that we use.

The built-in versions of keep and every also work on words, except the behavior of every on words is kind of weird, so keep it in mind if you ever decide to use it.

10

Higher order procedures

```
> (keep (lambda (x) (equal? 't (first x)))
'(to ma toes)
(to toes)
> (keep (lambda (x) (member? x '(e i)))
'aeiieai)
eiiei
> (every square '(1 2 3 4))
(1 4 9 16)
> (every (lambda (x) (word x x)) 'hello)
(hh ee ll ll oo)
; somewhat surprising behavior!
```

11

Higher order procedures

```
(define (plus1 x) (+ 1 x))
(define (plus2 x) (+ 2 x))
(define (plus3 x) (+ 3 x))
... This is getting tedious

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

> ((make-adder 1) 5)
6
> ((make-adder 7) 3)
10
```

12

Higher order procedures

```
(define (times1 x) (* 1 x))
(define (times2 x) (* 2 x))
(define (times3 x) (* 3 x))
... This is getting tedious

(define (make-mult n)
  (lambda (x) (* x n)))

> ((make-mult 1) 5)
5
> ((make-mult 7) 3)
21
```


Higher order procedures

```
(define (make-adder n)
  (lambda (x) (+ x n)))
(define (make-mult n)
  (lambda (x) (* x n)))
(define (make-div n)
  (lambda (x) (/ x n)))
(define (make-minus n)
  (lambda (x) (- x n)))
Okay ... this is getting tedious again.
```


Higher order procedures

```
(define (specialize f)
  (lambda (n)
    (lambda (x) (f x n))))

(define make-adder (specialize +))
(define make-mult (specialize *))
etc.
> ((specialize +) 2)
#[closure...]
> ( ((specialize +) 2) 3)
5
> (every ((specialize *) 2) '(1 2 3))
(2 4 6)
```


Higher order procedures

Remember the quadratic equation?

```
; for a polynomial ax^2+bx+c

(define (roots a b c)
  (se (/ (+ (- b) (sqrt (- (* b b) (* 4 a c))))
         (* 2 a))
      (/ (- (- b) (sqrt (- (* b b) (* 4 a c))))
         (* 2 a)) ))
```

This works fine, but it's a little bit inefficient. Why?

Higher order procedures

Okay then, how about this?

```
(define (roots a b c)
  (define (roots1 d)
    (se (/ (+ (- b) d) (* 2 a))
        (/ (- (- b) d) (* 2 a)) ))
  (roots1 (sqrt (- (* b b) (* 4 a c)))) )
```

This does the job, but …
It's awkward having to make up a name roots1 for this function that we'll only use once.
So how about making a temporary *unnamed* function?

Higher order procedures

Alright. Now we know what to do.

```
(define (roots a b c)
  ((lambda (d)
     (se (/ (+ (- b) d) (* 2 a))
         (/ (- (- b) d) (* 2 a)) ))
   (sqrt (- (* b b) (* 4 a c))) ))
```

This is exactly what we want. However …
Although the computer can parse it and understand it just fine, it's a little hard for humans to read.
To compensate …

Higher order procedures

Scheme provides a more convenient notation.

```
(define (roots a b c)
  (let ((d (sqrt (- (* b b) (* 4 a c)))))
    (se (/ (+ (- b) d) (* 2 a))
        (/ (- (- b) d) (* 2 a)) )))
```

Awesome!

Let

```
(let ((<var> <val>) (<var> <val>) ...)
   <body>)

Same as:
( (lambda (<var> <var> ...) <body>)
  <val> <val> ...)
```


Let

```
(define (fact n)
  (let ((temp (- n 1)))
    (if (= n 1)
        1
        (* n (fact temp)))))

(let ((a 1) (b 2) (c 3))
  (+ a b c))

(let ((a 1) (b (+ a 1)))
  (+ a b))
; ERROR! Use let* instead.
```