CS 61A Lecture Notes Second Half of Week 1

Topic: Higher-order procedures

Reading: Abelson & Sussman, Section 1.3

Note that we are skipping 1.2; we'll get to it later. Because of this, never mind for now the stuff about iterative versus recursive processes in 1.3 and in the exercises from that section.

We're all done teaching you the syntax of Scheme; from now on it's all big ideas!

This lecture's big idea is *function as object* (that is, being able to manipulate functions as data) as opposed to the more familiar view of function as process, in which there is a sharp distinction between program and data.

The usual metaphor for function as process is a recipe. In that metaphor, the recipe tells you what to do, but you can't eat the recipe; the food ingredients are the "real things" on which the recipe operates. But this week we take the position that a function is just as much a "real thing" as a number or text string is.

Compare the *derivative* in calculus: It's a function whose domain and range are functions, not numbers. The derivative function treats ordinary functions as things, not as processes. If an ordinary function is a meat grinder (put numbers in the top and turn the handle) then the derivative is a "metal grinder" (put meat-grinders in the top...).

• Using functions as arguments.

Arguments are used to generalize a pattern. For example, here is a pattern:

```
;;;;;                          In file cs61a/lectures/1.3/general.scm
(define pi 3.141592654)

(define (square-area r) (* r r))

(define (circle-area r) (* pi r r))

(define (sphere-area r) (* 4 pi r r))

(define (hexagon-area r) (* (sqrt 3) 1.5 r r))
```

In each of these procedures, we are taking the area of some geometric figure by multiplying some constant times the square of a linear dimension (radius or side). Each is a function of one argument, the linear dimension. We can generalize these four functions into a single function by adding an argument for the shape:

```
;;;;;                          In file cs61a/lectures/1.3/general.scm
(define (area shape r) (* shape r r))

(define square 1)
(define circle pi)
(define sphere (* 4 pi))
(define hexagon (* (sqrt 3) 1.5))
```

We define names for shapes; each name represents a constant number that is multiplied by the square of the radius.

In the example about areas, we are generalizing a pattern by using a variable *number* instead of a constant number. But we can also generalize a pattern in which it's a *function* that we want to be able to vary:

```
;;;;;                          In file cs61a/lectures/1.3/general.scm
(define (sumsquare a b)
  (if (> a b)
      0
      (+ (* a a) (sumsquare (+ a 1) b)) ))

(define (sumcube a b)
  (if (> a b)
      0
      (+ (* a a a) (sumcube (+ a 1) b)) ))
```

Each of these functions computes the sum of a series. For example, `(sumsquare 5 8)` computes $5^2 + 6^2 + 7^2 + 8^2$. The process of computing each individual term, and of adding the terms together, and of knowing where to stop, are the same whether we are adding squares of numbers or cubes of numbers. The only difference is in deciding which function of `a` to compute for each term. We can generalize this pattern by making *the function* be an additional argument, just as the shape number was an additional argument to the area function:

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

Here is one more example of generalizing a pattern involving functions:

```
;;;;;                          In file cs61a/lectures/1.3/filter.scm
(define (evens nums)
  (cond ((empty? nums) '())
        ((= (remainder (first nums) 2) 0)
         (se (first nums) (evens (bf nums))) )
        (else (evens (bf nums))) ))

(define (ewords sent)
  (cond ((empty? sent) '())
        ((member? 'e (first sent))
         (se (first sent) (ewords (bf sent))) )
        (else (ewords (bf sent))) ))

(define (pronouns sent)
  (cond ((empty? sent) '())
        ((member? (first sent) '(I me you he she it him her we us they them))
         (se (first sent) (pronouns (bf sent))) )
        (else (pronouns (bf sent))) ))
```

Each of these functions takes a sentence as its argument and *filters* the sentence to return a smaller sentence containing only some of the words in the original, according to a certain criterion: even numbers, words that contain the letter `e`, or pronouns. We can generalize by writing a `filter` function that takes a predicate function as an additional argument.

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

• Unnamed functions.

Suppose we want to compute

$$\sin^2 5 + \sin^2 6 + \sin^2 7 + \sin^2 8$$

We can use the generalized `sum` function this way:

```
> (define (sinsq x) (* (sin x) (sin x)))
> (sum sinsq 5 8)
2.408069916229755
```

But it seems a shame to have to define a named function `sinsq` that (let's say) we're only going to use this once. We'd like to be able to represent the function *itself* as the argument to `sum`, rather than the function's name. We can do this using `lambda`:

```
> (sum (lambda (x) (* (sin x) (sin x))) 5 8)
2.408069916229755
```

`Lambda` is a special form; the formal parameter list obviously isn't evaluated, but the body isn't evaluated *when we see the* `lambda`, either—only when we invoke the function can we evaluate its body.

• First-class data types.

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

In most languages, numbers are first-class; perhaps text strings (or individual text characters) are first-class; but usually functions are not first-class. In Scheme they are. So far we've seen the first two properties; we're about to look at the third. (We haven't really talked about aggregates yet, except for the special case of sentences, but we'll see in chapter 2 that functions can be elements of aggregates.) It's one of the design principles of Scheme that everything in the language should be first-class. Later, when we write a Scheme interpreter in Scheme, we'll see how convenient it is to be able to treat Scheme programs as data.

• Functions as return values.

```
(define (compose f g) (lambda (x) (f (g x))))
(define (twice f) (compose f f))
(define (make-adder n) (lambda (x) (+ x n)))
```

The derivative is a function whose domain and range are functions.

People who've programmed in Pascal might note that Pascal allows functions as arguments, but *not* functions as return values. That's because it makes the language harder to implement; you'll learn more about this in 164.

• Let.

We write a function that returns a sentence containing the two roots of the quadratic equation $ax^2+bx+c=0$ using the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

(We assume, to simplify this presentation, that the equation has two real roots; a more serious program would check this.)

```
;;;;;                          In file cs61a/lectures/1.3/roots.scm
(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 inefficient that we have to compute the square root twice. We'd like to avoid that by computing it once, giving it a name, and using that name twice in figuring out the two solutions. We know how to give something a name by using it as an argument to a function:

```
;;;;;                          In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  (define (roots1 d)
    (se (/ (+ (- b) d) (* 2 a))
        (/ (- (- b) d) (* 2 a)) ))
  (roots1 (sqrt (- (* b b) (* 4 a c)))) )
```

`Roots1` is an internal helper function that takes the value of the square root in the formula as its argument `d`. `Roots` calls `roots1`, which constructs the sentence of two numbers.

This does the job, but it's awkward having to make up a name `roots1` for this function that we'll only use once. As in the `sum` example earlier, we can use `lambda` to make an unnamed function:

```
;;;;;                          In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  ((lambda (d)
     (se (/ (+ (- b) d) (* 2 a))
         (/ (- (- b) d) (* 2 a)) ))
   (sqrt (- (* b b) (* 4 a c))) ))
```

This does exactly what we want. The trouble is, although it works fine for the computer, it's a little hard for human beings to read. The connection between the name `d` and the `sqrt` expression that provides its value isn't obvious from their positions here, and the order in which things are computed isn't the top-to-bottom order of the expression. Since this is something we often want to do, Scheme provides a more convenient notation for it:

```
;;;;;                          In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  (let ((d (sqrt (- (* b b) (* 4 a c)))))
    (se (/ (+ (- b) d) (* 2 a))
        (/ (- (- b) d) (* 2 a)) )))
```

Now we have the name next to the value, and we have the value of `d` being computed above the place where it's used. But you should remember that `let` does not provide any new capabilities; it's merely an abbreviation for a `lambda` and an invocation of the unnamed function.

The unnamed function implied by the `let` can have more than one argument:

```
;;;;;                          In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  (let ((d (sqrt (- (* b b) (* 4 a c))))
        (-b (- b))
        (2a (* 2 a)))
    (se (/ (+ -b d) 2a)
        (/ (- -b d) 2a) )))
```

Two cautions: (1) These are not long-term "assignment statements" such as you may remember from other languages. The association between names and values only holds while we compute the body of the `let`. (2) If you have more than one name-value pair, as in this last example, they are not computed in sequence! Later ones can't depend on earlier ones. They are all arguments to the same function; if you translate back to the underlying `lambda`-and-application form you'll understand this.

Another point of interest: Please note how, by using a language with first-class functions, we can construct local variables. We say, in this case, that the **expressive power** of first-class functions includes the ability to construct local variables. Indeed, the notion that first-class unnamed procedures give us other types of functionality "for free" will be a recurring theme of this course.