

Section Notes 9

CS51—Spring 2009

Week of April 12, 2009

1 Outline

1. Macros
2. Lazy Lists
3. Least squares fitting
4. Gradient descent
5. Mutation and aliasing

At the end of this section, you should be able to write simple macros, understand streams, and understand some of the concepts behind fitting 2D data to the line of best fit.

2 Macros

Recall John Scheme, and his attempt to rewrite `or` in the following way:

```
(define (my-or val1 val2) (if val1 #t val2))
```

Why didn't his version work? Because the Scheme interpreter will evaluate `val1` and `val2` as they get passed into `my-or`, as opposed to `or`, which properly delays evaluation.

`or` is a special-form in Scheme, as are many of the other built-in functions we've worked with, such as `and` and `cond` and `if`, which all have special evaluation rules that are different from the typical order in which Scheme evaluates functions.

If we want to delay evaluation of certain functions, we can define a macro in Scheme. Defining a macro is like defining a function, but instead of immediately evaluating the arguments, as with regular functions, it will replace the syntax around the arguments with whatever is specified in the definition, then evaluate it. Macros are defined as follows:

```
(define-syntax-rule (NAME [ARG...])  
  (DEFINITION incorporating the ARGS provided))
```

Use `...` in place of `ARGS` to specify an unknown number of arguments.

Macros do *syntactic* replacement of the parameters with the structure in the definition. A fixed definition of `my-or`:

```
(define-syntax-rule (my-or-fixed val1 val2) (if val1 #t val2))
```

Here's a mental model for thinking about how Scheme handles macros:

1. Find all the calls to macros in the program.

2. Replace each call with the body of the macro, replacing the arguments with the *expressions* that are being passed as parameters.¹
3. Now that there aren't any macros, run the program using the usual Scheme evaluation rules that we know and love.

Thus, a call to `(my-or-fixed (= a 0) (/ a (- 2 2)))` will take the arguments and replace the `val1` and `val2` arguments to the macro with the expressions passed: `(if (= a 0) #t (/ a (- 2 2)))`. Then it'll use the rules for `if`, and thus avoid evaluating the division by 0.

2.1 Macro examples

Define a macro to increment a variable by a certain value.

```
; Here is an example in use:
(define x 5)
  (+= x 2)
; x => 7

(define-syntax-rule (+= variable value)
  (
```

Define a macro to add one or more values to a variable.

```
; Here is an example in use:
(define x 2)
(+= x 7)
; x => 9
(+= x 3 4)
; x => 16

(define-syntax-rule (+= variable value ...)
  (
```

Define a macro for swapping the contents of two variables.

```
(define-syntax-rule (swap x y)
  (
```

Define a macro for the while-loop.

¹There is a bit more magic about replacing duplicated variable names, but this is pretty close. See the lecture notes for an example.

```
(define-syntax-rule (while e1 e2 ...)
  (letrec
```

3 Lazy Lists or Streams

A stream consists of 1) a current item, and 2) a “promise” to produce the rest of the stream on request. In Scheme, a stream is a cons cell whose cdr is a function that returns a stream. Note the similarity to the definition of a list. The critical difference is that the cdr is not itself a stream, but an unevaluated *lambda*. This is what allows a stream to be “infinite”: we don’t insist on keeping the whole thing around at once; the rest of the stream is always a function call away.

3.1 Lazy-List Macros

For convenience, we use `lcons` to create a stream and `lcar/lcdr` to retrieve elements of the stream, instead of using an explicit lambda function.

For your reference, we have pasted the lazy-list macros below:

```
; Wrapping e2 in a lambda causes its evaluation to be delayed.
(define-syntax-rule (lcons e1 e2)
  (mcons e1 (lambda () e2)))

; Lazy-car works the same as car.
(define (lcar lcell)
  (mcar lcell))

; Lazy-cdr needs to evaluate the lambda that lcons put in.
(define (lcdr lcell)
  (let* ([v ((mcdr lcell))])
    ([f (lambda () v)] ; memoize the result
     (begin (set-mcdr! lcell f) v))))
```

Note that we used mutable functions (`mcons`, etc) to define the lazy-list macros, so we can take advantage of *memoization*, in which we remember the result of a computation instead of re-computing it each time.

Finally, we also have lazy-list functions, such as `lmap` and `lfilter`, that use the lazy-list macros, which work like their corresponding non-lazy-list functions.

`zip` combines 2 infinite lists, and builds off of lazy-list macros as well:

```
(define (lzip f xs ys)
  (lcons (f (lcar xs) (lcar ys))
        (lzip f (lcdr xs) (lcdr ys))))
```

3.2 Examples

Write a lazy list of multiples of 3. You can assume the `nats` stream of the natural numbers is defined.

```
(define 3-mult
  (
```

Now let's think about how to actually define `nats`: Without looking at the answers, try to define the list of all natural numbers starting from 0.

```
(define nats
  (
```

Write a function `(stream-max s1 s2)` that takes two streams and returns a new stream of the maximum of the first elements of `s1` and `s2`, followed by the maximum of the second element of each, and so on. (For example, `s1` and `s2` might represent simultaneous rolls of two dice, and we want a stream representing the maximum in each pair of rolls.) You may assume that both streams are infinite.

```
(stream-max s1 s2)
  (
```

Write a function `(stream->list stream n)` that returns the first `n` elements of the stream as a list. `stream->list` must run in $O(n)$.

```
(define (stream->list stream n)
  (
```

The function e^x may be computed by the power series

$$\sum_{i=0}^{\infty} \frac{x^i}{i!}.$$

Write a function `(e-stream x)` that returns a stream of approximations for $e^{|x|}$ like $1, 1+x, 1+x+\frac{x^2}{2}, \dots$, in which each successive approximation contains one more term of the power series.

```
(define (e-stream x)
  (
```

4 Least squares

We often want to use computers to process data that came from the “real world” somehow—measured, guessed, typed in by users, etc. Such data invariably has imperfections of various sorts, and we’ll be talking about various ways to deal with them over the next few weeks. We’re starting with the simplest data model: fitting a line to a set of 2D points.

The model we’re using for least squares is straight lines. Note that in general, other models are possible. Remember plotting and estimating runtimes in PS4? We used higher degree polynomials and exponential functions there. However, it turns out that things are easier in the linear case, so we’ll stick to that for now.

So, the problem we’re trying to solve: given a set of data points (x_i, y_i) , we want to find a “good” line $y = mx + b$ that fits those points. Least squared defines an error function that specifies how “good” a particular line is. Not surprisingly given the name, the error function is the sum of squared residuals:

$$Err(m, b) = \sum_i (y_i - (mx_i + b))^2$$

To find the best line, we need to find the parameters that minimize the error. For the least squares error function, it turns out that there’s actually a closed form solution that can be derived. However, we’ll use gradient descent instead, because it’s a very useful general method that applies when there is no known closed form solution.

5 Gradient descent

One very convenient property of the least squares error function defined above is that it is *convex*—it has a single global minimum, and smoothly approaches it from all sides, with no other “dips”. This makes it possible to “walk downhill” to find the minimum, and be sure that we won’t get stuck. We can start at some initial values for the parameters, and then try adjusting them a bit in each direction to see which direction makes the function smaller. If neither direction makes it smaller, we’re done. (Why?) Otherwise, we move in the direction of decreasing function value and keep going.

Here is one way to implement gradient descent for a 1D function:

```
; Do gradient descent
(define (minimize f delta)
  (letrec ([start 1.0]
           [loop (lambda (cur)
                    (cond [(< (f (+ cur delta)) (f cur))
                          (loop (+ cur delta))]
                          [(< (f (- cur delta)) (f cur))
                          (loop (- cur delta))]
                          [else cur]))])
    (loop start)))
```

What if the function we want to minimize is not convex? How can we change the algorithm a bit so it works on a wider class of functions? (Hint: we may need to give up on having a guarantee that it will always find the global minimum.)

There are some functions that require exhaustive search to find the minimum: For example, imagine that you had a 2D function that was totally flat over the entire domain³, with a single low dip⁴ somewhere.

²Recall from lecture that this way of representing lines is actually pretty bad. We’ll use it anyway because it’s simple

³Say, Kansas

⁴Water well?

Unless you have some extra info about the structure of the function, there is no better way to find the lowest point except to search the entire space⁵.

Exercise: Given the `mimize` definition above, write `(maximize f delta)` that maximizes a convex function.

Exercise: Write a `my-sqrt` function that uses `minimize` to find the square root of a number to a given precision:

```
(define (my-sqrt n delta)
  (
```

6 Mutation and Aliasing

We're now switching gears to talk about mutation and aliasing, since this will come up in the project. As we've already mentioned, mutation often makes reasoning about programs harder. Here, we'd like to briefly talk about the problems caused by *aliasing*. Aliasing happens when the same mutable data structure is referenced from multiple locations. This means that changes in one part of your program can affect other parts in non-obvious ways.

A simple example:

```
; Make lst a mutable cons cell
(define lst (mcons 1 2))

; lst2 points to the _same_ cell
(define lst2 lst)

(mcar lst) ; returns 1
(mcar lst2) ; returns 1 (*)

(set-mcar! lst 42) ; change the car of the first list

(mcar lst) ; returns 42, which totally makes sense
(mcar lst2) ; also returns 42! A bit more surprising, since we
              ; haven't directly changed lst2 since the call above in (*)
```

Why would someone write such silly code, you may ask. Well, here's a slightly more realistic example from Project 2:

Inside the game engine, we have something more or less like this:

⁵An excuse for an exploratory bike trip?

```

; Called whenever a powerup reappears on the graph
(define (appear powerup node)
  ; Change the powerup state to reflect its new status and location
  (set-powerup-visible! powerup \#t)
  (set-powerup-location! powerup node)

  ; also change its size
  (set-powerup-size! powerup (+ alpha (powerup-size powerup)))

  ; Add it to the list we'll pass to collect-data later
  (set! new-powerups (cons (cons node powerup) new-powerups)))

...

(define (step)
  (if (not (empty? new-powerups))
      (begin
        ; tell the agent about the newly appeared powerups
        (send student-agent collect-data new-powerups)
        (set! new-powerups empty)) ; reset the list
      (void) ; otherwise just don't do anything
  ))

```

The powerup agents periodically appear and disappear, and change internal state such as location and size over time. The collect-data function that you have to write periodically gets called with a list of newly appeared powerups. Imagine that you tried writing it like this:

```

; ***** NOTE: BROKEN CODE *****

(define powerup-history empty)

(define (collect-data powerups)
  ; add each (node . powerup) pair to our history list
  (map (lambda (x)
        (set! powerup-history (cons x (powerup-history))))
       powerups)

; and then try to use the history later on:

; return the list of powerup sizes at a particular node.
(define (get-sizes node)
  (map powerup-size
       (filter (lambda (x) (equal? (car x) node)) powerup-history)))

```

This all seems perfectly reasonable, right? We store all the powerups we've seen, and then get a list of sizes at a node by extracting just the ones that appeared at that node and getting their size. What goes wrong? Do you see how this is similar to the simplified example above?

7 Answers

- (define-syntax-rule (= variable value)
 (set! variable (= variable value)))

- (define-syntax-rule (+= variable value ...)
 (set! variable (+ variable value ...)))
- (define-syntax-rule (swap x y)
 (let ([tmp x])
 (set! x y)
 (set! y tmp)))
- (define-syntax-rule (while e1 e2 ...)
 (letrec ([loop (lambda ()
 (if e1 (begin e2 ... (loop)) (void))))]
 (loop)))
- (define 3-mult
 (lfilter (lambda (x) (= (modulo 3 x) 0)) nats))
- (define nats
 (lcons 0 (lmap (lambda (z) (+ 1 z)) nats)))
- (define (stream-max s1 s2)
 (if (> (lcar s1) (lcar s2))
 (lcons (lcar s1)
 (stream-max (lcdr s1) (lcdr s2)))
 (lcons (lcar s2)
 (stream-max (lcdr s1) (lcdr s2)))))
- (define (stream->list stream n)
 (if (= n 0) '()
 (cons (lcar stream)
 (stream->list (lcdr stream) (- n 1)))))
- ;; e-stream: Number -> Stream of Number
 ;; To find successive approximations of e^x .
 ;;
 ;; e-stream wraps to e-stream-helper, providing the initial values of
 ;; 1 (the first exponent to be calculated) and 1 (the previous term,
 ;; with exponent 0).
 (define (e-stream x)
 (e-stream-helper x 1 1))

 ;; e-stream-helper: Number x Integer x Number -> Stream of Number
 ;;
 ;; e-stream-helper does the work of constructing the stream of
 ;; approximations for e^x . It takes the value of x as an argument,
 ;; and it keeps track of the value of i in the formula $(x^i)/(i!)$ and
 ;; the value of the previous term, both of which are needed for the
 ;; mathematical calculation of the term's value.
 (define (e-stream-helper x i prev-term)
 (lcons prev-term
 (e-stream-helper x (+ 1 i) (e-stream-calc x i prev-term))))

 ;; e-stream-calc: Number x Integer x Number -> Number
 ;;
 ;; e-stream-calc actually calculates the value of a given term based

```

;; on the formula for the power series of e^x.
(define (e-stream-calc x i prev-term)
  (+ prev-term (/ (expt x i) (factorial i))))

;; factorial: Number -> Number
;;
;; factorial returns (unsurprisingly) the factorial of its argument.
;; This is called by e-stream-calc to calculate the factorial of i.
(define (factorial n)
  (if (= 1 n) n
      (* n (factorial (- n 1)))))

```

- `(define (maximize f delta) (minimize (lambda (x) (- (f x))) delta))`
- `(define (my-sqrt n delta) (minimize (lambda (x) (square (- n (square x)))) delta))`

- **Gradient descent modifications:** If the function you're minimizing isn't convex, but is not too crazy, gradient descent may still do something reasonable. Now the answer will depend on where you start your search: if you have some info about where the minimum is likely to be, starting there can help. Another approach is to do random restarts: run the algorithm with multiple starting points and choosing the best one. Of course, this isn't guaranteed to find the global optimum, but it often does well in practice.
- **Aliasing:** the problem with the code is that the `powerup-history` will have lots of references to the same `powerup` objects that appear at different places over the course of the game. This means that when the engine code does something like `(set-powerup-size! powerup...)`, it will change the size of the `powerups` referenced in your history! So your `get-sizes` function will return a list that has lots of repeats of the same value. This is tricky to debug, because the changes aren't happening anywhere in your code.