SI 413: The most intelligent way to misuse a computer

Professor Keith Sullivan

August 27, 2023

Pure Functional Programming

- Referential Transparency
- Functions are first class

First class Functions

What does it mean for a function to be first class?

- Can be given names
- Can be arguments to procedures
- Can be returned by procedures
  - Higher order functions
- Can be stored in data structures

Procedures Returning Procedures

Get the predicate for the type of input

```
(define (test-my-type something)
        (cond ((number? something) number?)
              ((symbol? something) symbol?)
              ((list? something)   list? )))
```

Useful when combined with higher-order procedures:

```
(define (like-the-first L)
        (filter (test-my-type (car L)) L))
```

Storing Procedures in a List

Maybe we want to apply different functions to the same data

```
(define (apply-all alof alon)
        (if (null? alof)
            '()
            (cons ((car alof) alon)
                  (apply-all (cdr alof) alon))))
```

Then we can get statistics on a list of numbers:

```
(apply-all (list length mean stddev)
                (list 2.4 5 3.2 8))
```

History Time!

- The lambda calculus is a way of expressing computation (similar to Turning machines)
- Alonzo Church in the 1930's
- Believed to cover everything that is computable
- *Everything* is a function: numbers, points, booleans, ...
- Functions are just a kind of data!
- Consists of constructing lambda terms and performing reductions

Anonymous Functions

lambda is a special form in Scheme that creates a “nameless” (or anonymous) function:

```
(lambda (arg1 arg2 . . . )
        expr-using-args)
```

lambda is a function that returns a function

`(lambda (x) (+ x 5))` → `#<procedure>`

`((lambda (x) (+ x 5)) 8)` → 13

Exercises

- Write the `make-adder` function:
  `((make-adder 5) 10)` ⇒ 15
- Write the `double` function:
  `((double sqrt) 81)` ⇒ 3

Behind the Curtain

You have already been using **lambda**!

```
(define (f x1 x2 ... xn)
        (cool-stuff-with-xs))
```

```
(let ((x1 e1) (x2 e2) ... (xn en))
        expression-using-xs)
```

How to rewrite these using **lambda**?

More Lambda!

```
(define fold
  (lambda (f i l)
    (if (null? l)
        i
        (f (car l) (fold f i (cdr l))))))

(define total
  (lambda (L)
    (fold + 0 L)))

(define total-all (lambda (L) (map total L)))
```

Yet more λ

```
(define make-double (lambda (f)
                      (lambda (x)
                       (f x x))))
(define twice (make-double +))
(define square (make-double *))


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

(define incr (make-adder 1))

(define add123 (make-adder 123))
```

Exercise

- In the language APL, most arithmetic functions can be applied either to a number or to a vector (similar to **apply** in Scheme). For example, the function sqrt applied to 16 returns 4 as in Scheme, but sqrt can also be applied to a list such as (16 49) and it returns (4 7).
  Write a procedure `aplize` that takes as its argument a one-argument arithmetic procedure. It should return an APLized procedure that also accepts lists:
  \> (define apl-sqrt (aplize sqrt))
  \> (apl-sqrt 36)
  6
  \> (apl-sqrt '(1 100 25 16))
  (1 10 5 4)

Stack Space

Recursive calls use a lot of memory

```
(define (ssq n)
        (if (= n 0)
                0
                (+ (* n n) (ssq (- n 1)))))
```

Why does `(ssq 4000000)` run out of memory?

Stack Space

This code does the same thing, but takes an extra argument that acts as an accumulator

```
(define (ssq-better n accum)
        (if (= n 0)
                accum
                (ssq-better (- n 1)
                            (+ (* n n) accum))))
```

Now it works!

Tail Recursion

- The second version worked since there is no stack of recursive calls.
- A function is tail recursive if its output expression in every recursive call is only the recursive call.
- In Scheme, this means the recursive call is outermost in the returned expression.

Tail Recursion

```scheme
(define (mult L)
  (if (null? L)
      1
      (* (car L) (mult (cdr L)))))

(define (sqrt-prod n)
  (mult (map sqrt (range 1 n))))


(define (sqrt-prod-apply n)
  (apply * (map sqrt (range 1 n))))
```

Tail recursion for Fibonacci

To implement tail recursion we usually make a helper function:

```scheme
(define (fib-helper n i fib-of-i fib-of-i+1)
  (if (= i n)
      fib-of-i
      (fib-helper
       n
       (+ i 1)
       fib-of-i+1
       (+ fib-of-i
          fib-of-i+1))))
```

The main function then becomes:

```scheme
(define (fib-tail n) (fib-helper n 0 0 1))
```

Exercises

- Write a tail-recursive version of `mult`. Recall the non tail recursive version

  ```scheme
  (define (mult L)
    (if (null? L)
        1
        (* (car L) (mult (cdr L)))))
  ```

- Write a tail-recursive procedure `all-positive` that returns true if all elements of a list are positive, false otherwise.
- Write a tail recursive procedure called `myfilter` that operates the same way as the built-in function **filter** .

Side Effects

Remember the intro to the Scheme standard:

Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including functional, imperative, and message passing styles, find convenient expression in Scheme.

What do we have to give up to get side effects?

Controlling Output

Displaying text to the screen is a kind of side effect.

Here are some useful functions for screen output:

- `(display X)`
- `(newline)`
- `(printf format args...)`
  The catch-all format flag is `~a`.

(Note: Strings in Scheme are made using double quotes, like `"This is a string"`.)

Structuring Code with Side Effects

How to print every element of a list?

```
(define (print-list L)
   (if (null? L)
      ;; BASE CASE
      (display (car L))
      (newline)
      (print-list (cdr L))
)
```

Structuring Code with Side Effects

How to print every element of a list?

```
(define (print-list L)
   (if (not (null? L))
      (begin (display (car L))
             (newline)
             (print-list (cdr L))
      )
)
```

Mutation

The built-in special form (set! x val) changes the value of x to be val.

Say we want a function that will print out how many times it's been called. The following factory produces one of those:

```
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ 1 count))
      (display count)
      (newline))))
```

Closures

- Notice that make-counter makes a different **count** variable each time it is called.
- This is because each **lambda** call produces a closure — the function along with its referencing environment.
- **Save yourself a lot of trouble:**
  The changing "state" variable (i.e., the **let**) must be inside the function (i.e., the **define**), **but** outside the lambda.

Objects in Scheme

We can use closures and mutation to do OOP in Scheme!

```
(define (make-counter-obj)
  (let ((count 0))
    (lambda (command)
      (cond ((symbol=? command 'get) count)
            ((symbol=? command 'inc)
                  (set! count (+ 1 count)))
            ((symbol=? command 'reset)
                  (set! count 0))))))
```

The object has three methods: **get**, `inc`, and `reset`.

Built-in Data Structures

Scheme has some useful built-in data structures:

- Arrays (called "vectors").

```
(define A (make-vector 5))
(vector-set! A 3 'something)
(vector-ref A 3) ; produces 'something
(vector-ref A 5) ; error:  out of bounds
```

- Hash tables

```
(define H (make-eqv-hashtable))
(hashtable-set! H 2 'something)
(hashtable-set! H 'another-key 'crazy!)
(hashtable-contains? H 2)                ;true
(hashtable-ref H 'another-key 'default) ;'crazy!
(hashtable-ref H 1234 'default)          ;'default
```

Inefficiency in Scheme

Recall the problem of computing Fibonacci numbers.

```
(define (fib n)
  (if (<= n 1)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

Why is this function so slow?

Memoization in Scheme

Recall: Memoization is remembering the results of previous function calls, and never repeating the same computation.

Why is functional programming *perfect* for memoization?

Scheme's built-in hashes can be used to memoize.

Memoizing Fibonacci

```scheme
(define fib-memo
  (let ((memo (make-eqv-hashtable)))
    (define (fib-internal n)
      (cond ((<= n 1) n]
            ((hashtable-contains? memo n)
                 (hashtable-ref memo n '()))
            (else
                 (let ((val (+ (fib-internal (- n 1))
                               (fib-internal (- n 2)))))
                      (hashtable-set! memo n val)
                  val))))
    fib-internal))
```