CS4 — Functional Programming
Vanier

Yubo Su

Contents

Chapter 1

05/01/15 — Introduction

This class will be taught in Scheme (Racket dialect), Typed Racket (type checking) and Ocaml. Website is on Moodle. Course is not practical but conceptual programming course. Three types of programming are function, imperative, and object-oriented. Scheme is very similar to Javascript, and was initially intended to be Javascript until parentheses became a rightfully-deserved turn off. Haskell is the archetypal functional language, CS115; both Ocaml and Racket support functional programming.

1.1 Programming fundamentals

Let's first examine some fundamentals of programming

1) Infix notation — the binary operator is in the middle, such as `3+4`; we use parentheses to specify order.

2) Assignment — Assigning a variable a value, such as `y=4` or `x=y+z`.

3) Function — Takes an argument and returns a value.

4) Prefix notation — Just like functions, the operator proceeds the input, like `f(x,y)`. An alternative prefix notation separates with commas, like `(f 3 2)`.

Scheme uses prefix notation such as `(+ 3 4)` or `(+ 3 (* 4 5))`. In fact you can write `(+ 2 3 4)`. Assignment goes `(define x 3)`.

Lambda means "this combination represents a function." Often we can write things like `(define a (lambda (r) (* pi (expt r 2))))`.

Comments are proceeded with a semicolon and go until the end of the line. Next time we will talk about the substitution model of evaluation.

Chapter 2

07/01/15 — The Substitution Model, booleans and conditionals

We examine today how expressions are evaluated: the *substitution model.* Recall `(operator operand1 operand2 ...)` is the Scheme way. Thus, the three steps to evaluating such an expression are

- Evaluate operator
- Evaluate operand(s)
- Apply operator procedure to operands

Notably, we specify that reading the `+` operator must call the *primitive procedure* corresponding to addition. Note also that *evaluating* the operands takes care of all the nested parentheses that we see all over the place.

There is some special forms which evaluates differently from the above. One is the `define` protocol, which goes like `(define x 3)`. We evaluate only the second operand and associate it to the first operand. The `define` protocol returns the value assigned.

Another is the `lambda` operator, which returns a procedure; *none* of the operands are evaluated. A procedure can then be called, and the way we apply a procedure to its arguments is to simply evaluate its arguments and then substitute them for the variables in the function *body.*

Note the following **Syntactic Sugar**:
`(define f (lambda (x) (+ x 1)))` is equivalent to `(define (f x) (+ x 1))`

This is the end of the *substitution model*, which is mechanical enough to be described in terms for a computer.

2.1 Booleans, Conditionals

We first introduce the *boolean*, which contains either a true or false flag (`#t` and `#f`) respectively. Some functions evaluate to booleans, such as `(= 2 3)`. Moreover, other functions expect boolean operands such as `(and #t #f)`.

Note that the `and`, `or` operators are smart but don't follow the standard evaluation protocol; they *will* short-circuit. They can also take more than two operands.

A typical convention is for functions returning booleans to be named ending with a question mark, called *predicates*.

The `if` conditional is the most powerful way to use booleans, comprising the form

```
(if <boolean-expression> <true-case> <false-case>)
```

Note that now that we have more than one data type, we must touch on types. Scheme does not type check before running code, so type errors show up at runtime.

Chapter 3

09/01/15 — Recursion and Iteration

Let's open with a small math problem; how do we turn a repeating decimal into a rational number? Alternatively, how do we sum an infinite series? Mathematically, this is very easy; the trick in both is to turn the problem into a recursive definition.

3.1 Recursion

Let's try to apply this sort of reasoning to a very simple problem, summing the first N integers. Note that the sum of the first N integers is simply $N+$ the sum of the first $N-1$ integers. Taking care to write in our base case, the full code to do this in Scheme is

```
(define (sum-integers n)
    (if (= n 0)
        0
        (+ n (sum-integers (- n 1)))))
```

In general, recursion requires a three step design pattern

1. Test whether base case

2. Base case

3. Recursive case

Proving that a recursive algorithm is correct usually invokes mathematical induction.

To discuss the efficiency of this recursive algorithm, we note that evaluating for N results in a chain of N deferred operations and N total function calls. This is called a *linear recursive* process. A linear recursive process requires an amount of time and space both of which are linear in the size of the input.

3.2 Iteratiion

Consider another algorithm, summing as we go:

```
(define (sum-int n)
    sum-iter 0 n 0)
(define (sum-iter current max sum)
    (if (> current max)
    sum
    (sum-iter (+ 1 current) max (+ current sum))))
```

Of course we still require N total function calls, but how about the number of deferred operations? Constant! Compare

```
    (sum-integers 3)                         (sum-int 3)
    (+ 3 (sum-integers 2))                   (sum-iter 0 3 0)
    (+ 3 (+ 2 (sum-integers 1)))             (sum-iter 1 3 0)
    ...                                      (sum-iter 2 3 1)
    (+ 3 (+ 2 (+ 1 0)))                      ...
                                             (sum-iter 4 3 6)
```

The second algorithm is called a *linear iterative process.* It still makes a linear number of calls but the state of computation is kept in a constant number of state variables, so it requires constant storage space. While both are recursive procedures, the algorithms differ in that recursive processes require pending operations while iterative ones don't.

Usually the iterative version of an algorithm is preferred, though the recursive is almost always easier to implement and prove correctness. Also, sometimes space isn't the constraining factor and so it proves okay to use the recursive implementation and be lazy.

Note above that we required a helper function `sum-iter` to keep track of the state of the program; we will see later how to put helper functions "inside" functions that use them.

3.3 Fibonacci Numbers

Consider the age-old Fibonacci numbers. We note that a recursive implementation requires two recursive calls per execution. This is called *tree recursive* instead of linear because calls fan out in a tree. The number of calls is exponential in the size of the argument (the precise form is nontrivial since the tree is not symmetric). The inefficiency of the naive recursive implementation lies in the fact that numbers previously computed are recomputed; we will discuss more about optimizing this out and general asymptotic efficiencies next lecture!

Chapter 4

12/01/15 —

4.1 Iterative Fibonacci

Let's finish the Fibbonacci problem we discussed earlier. Consider the code

```
(define (fib-iter fnext f cnt)
    (if (= cnt 0)
        f
        (fib-iter (+ fnext f)
            fnext
            (- cnt 1))))
```

Again, instead of an exponential growth we observe a linear growth. Recall that this is called a linear iterative process, so it requires $n+1$ calls and constant space to solve the problem.

Blurb: Note that we can use what is called *memoization* to improve the highly inefficient tree-recursive Fibonacci procedure we saw last lecture by memorizing values so we don't have to recalculate. We won't discuss this now but maybe later.

4.2 Efficiency of algorithms

The most important quantifier of the efficiency of an algorithm is the runtime's dependence. We introduce *Big O* notation, such that a function $g(n)$ is $O(f(n))$ if for some n_0, there exists $n > n_0, g(n) \leq Cf(n)$ for some constant C. In other words, $g(n)$ grows no faster than $f(n)$ for sufficiently large n.

Since big O is a bound from above, we also need a bound from below, which is satisfied by Θ, big theta notation. Usually we say "big oh" for Θ, and we notate it like O already, but technically they are different! Θ is the lower bound, O is the upper. For example, the recursive Fibonacci algorithm is $O(2^n)$ but $\Theta(\varphi^n), \varphi = 1.618\ldots$ the golden ratio.

We care about this because, like in the case of the Fibonacci implementations, the recursive goes like $O(2^n)$ while the iterative goes merely like $O(n)$!.

Note also that all logarithmic behaviors with different bases are equivalent up to a constant and so we don't care about the base in big O notation.

4.3 Internal Procedures

Suppose we want to write a helper function, say to evaluate $(a+b)*c$, but don't want it to pollute the global namespace. Thus, we can write an internal procedure

```
(define (timesPlus a b c)
    (define (plusPlus a b)
        (+ a b))
    (* (plusPlus a b) c))
```

The advantage is that we have easier code to understand and to write variable names for, but it is generally harder to debug.

4.4 `cond` Evaluation

The functional way to do else ifs is the `cond` operator. It uses syntax

```
(cond (<test1> <expr1>)
    (<test2> <expr2>)
    (<test3> <expr3>)
    (...)
    (else <expr>)
```

Chapter 5

14/01/15 — Higher order functions

In math, we have seen functions as arguments. For example, when we write $\sum_{i=0}^{n} f(n)$ the Σ is actually a function taking $f(n)$ as an argument! We call this Σ a *higher-order function*, and in Scheme we are allowed to define such functions.

Consider this exact problem. We will write the implementation

```
(define (sum f low high)
    (if (> low high)
        0
        (+ (f low)
            (sum f (+ low 1) high))))
```

which is clearly a linear recursive process. Note that if we want, say, a sum of squares, we can call with

```
(sum (lambda (x) (* x x)) 2 4)
```

We can also write up the iterative version, for which we will need extra state variables.

Consider the slight extension of the above `sum` code, which gives the ability to specify the next number in the summation

```
(define (gsum f low fnext high)
    (if (> low high)
        0
        (+ (f low)
            (gsum f (fnext low) fnext high))))
```

Now we can specify a whole class of summations simply by passing `lambda` functions into `f` and `fnext`. Lambda is all powerful!

We can next study how to solve problems, such as sum of prime factors. The general approach will be to successively find the smallest prime factor, sum it, divide it, and repeat.

5.1 Lambda functions

One might wonder why we call these functions "lambda functions." This is because back in 1930s, Alonzo Church saw lambda calculus, the theory of computation of the time, which was based on pure functions, and sought to eliminate as many kinds of data as possible. He kept only one type of data, *functions*. There are then only three types of expressions: variables, functions of one argument, and function applications. Since lambda expressions also give higher-order procedures, it's possible to solve everything with just lambda expressions! That's all.

Chapter 6

16/01/15 — COOKIES

Cookies were brought to class.

6.1 `Let` construct

Anyways, we can define something with very local scope with the `let` construct, via

```
(let ([x 10]
    [y (- 2 4)])
    (* x y))
```

`x, y` will not have any more scope outside of the `let` construct. Neato!

We note that this sort of substitution model is eerily reminiscent of functions; `let` is actually just another way of writing `lambda`! The following two pieces of code are equivalent

```
(let ( (<var1> <expr1>)
    (<var2> <expr2>))
    <body>)

( (lambda (<var1> <var2) <body)
    <expr1> <expr2>)
```

We also exhibit the syntax `let*`, which turns into a bunch of nested `let`s. This is helpful if the later `let`s are dependent on the earlier ones.

6.2 `begin` clause

Sometimes, if you want multiple lines under the same execution block (curly braces in C, Java etc.) one can do so by enclosing in `begin`

```
(if (> x 10)
    (begin
        (display x)
        (newline)
        (* x 2))
    (/ x 2))
```

begin returns the result of the last line only. Note also we used display, newline to print things and to print a new line.

6.3 Returning functions

Now let's get onto the really interesting stuff. Can we return a function? Consider a general function to generate functions that add n

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

Then if we want a function that adds 2, for example, we can just (define add2 (make-addn 2)).

There becomes a bit of a subtlety here; you cannot substitute into a lambda function's own arguments. So for example, (define (weird n) (lambda (n) (+ n n))), the n on the inner lambda is *not* substituted! lambda protects its own arguments.