Functional Programming in Scheme and Lisp

Overview
- In a functional programming language, functions are first class objects
- You can create them, put them in data structures, compose them, specialize them, apply them to arguments, etc.
- We’ll look at how functional programming things are done in Lisp

eval
- Remember: Lisp code is just an s-expression
- You can call Lisp’s evaluation process with the eval function
  > (define s (list 'cadr ' ' (one two three)))
  > s
  (cadr ' (one two three))
  > (eval s)
  two
  > (eval (list 'cdr (car '((quote (a . b)) c))))
  b

apply
- apply takes a function & a list of arguments for it & returns the result of applying the function to them
  > (apply + ' (1 2 3))
  6
- It can be given any number of arguments, so long as the last is a list:
  > (apply + 1 2 ' (3 4 5))
  15
- A simple version of apply could be written as
  (define (apply f list) (eval (cons f list)))

lambda
- The define special form creates a function and gives it a name
- However, functions don’t have to have names, and we don’t need to use define to create them
- The primitive way to create functions is to use the lambda special form
- These are often called lambda expressions, e.g.
  (lambda (x) (+ x 1))
Lambda expression

- A lambda expression is a list of the symbol `lambda`, followed by a list of parameters, followed by a body of one or more expressions:

```lisp
> (define f (lambda (x) (+ x 2)))
> f
#<procedure:f>
> (f 100)
102
> ( (lambda (x) (+ x 2)) 100)
102
```

- Other languages like python and javascript have adopted the idea

Define vs. Define

```lisp
(define (add2 x)
  (+ x 2))
(define add2
  (lambda (x) (+ x 2)))
(define add2 #f)
(set! add2
  (lambda (x) (+ x 2)))
```

- The define special form comes in two varieties
- The three expressions to the right are entirely equivalent
- The first define form is just more familiar and convenient when defining a function

Functions as Objects

- While many PLs allow functions as arguments, nameless lambda functions add flexibility

```lisp
> (sort '((a 100)(b 10)(c 50))
  (lambda (x y) (< (second x) (second y))))
((b 10) (c 50) (a 100))
```

- There is no need to give the function a name

Lambda Expressions

- Lambda is a special form
- When evaluated, it creates a function and returns a reference to it
- The function does not have a name
- A lambda expression can be the first element of a function call:

```lisp
> ( (lambda (x) (+ x 100)) 1)
101
```

Lambdas in Other Languages

- Lambda expressions are found in many modern languages, e.g., Python:

```python
>>> f = lambda x,y: x*x + y
>>> f
<function <lambda> at 0x10048a230>
>>> f(2, 3)
7
>>> (lambda x,y: x*x+y)(2,3)
7
```

Mapping Functions

- Lisp & Scheme have several mapping functions
- `map` (mapcar in Lisp) is the most useful

```lisp
> (map abs '(3 -4 2 -5 -6))
(3 4 2 5 6)
> (map + '(1 2 3) (4 5 6))
(5 7 9)
> (map + '(1 2 3) '(4 5 6) '(7 8 9))
(12 15 18)
```
More map examples

> (map cons '(a b c) '(1 2 3))
((a . 1) (b . 2) (c . 3))
> (map (lambda (x) (+ x 10)) '(1 2 3))
(11 12 13)
> (map + '(1 2 3) '(4 5))
map: all lists must have same size; arguments were:
#<procedure:+> (1 2 3) (4 5)

Defining map

Defining a simple “one argument” version of map is easy

(define (map1 func list)
  (if (null? list)
      null
      (cons (func (first list))
            (map1 func (rest list)))))

Define Lisp’s every and some

• every and some take a predicate and one or more sequences
• When given just one sequence, they test whether the elements satisfy the predicate
  > (every odd? '(1 3 5))
  #t
  > (some even? '(1 2 3))
  #f
• If given >1 sequences, the predicate takes as many args as there are sequences and args are drawn one at a time from them:
  > (every > '(1 3 5) '(0 2 4))
  #f

Defining every is easy

(define (every1 f list)
  ;; note the use of the and function
  (if (null? list)
      #t
      (and (f (first list))
           (every1 f (rest list)))))

Define some similarly

(define (some1 f list)
  (if (null? list)
      #f
      (or (f (first list))
          (some1 f (rest list)))))

Will this work?

• You can prove that P is true for some list element by showing that it isn’t false for every one
• Will this work?
  > (define (some1 f list)
      (not (every1 (lambda (x) (not (f x))) list)))
  > (some1 odd? '(2 4 6 7 8))
  #t
  > (some1 (lambda (x) (> x 10)) '(4 8 10 12))
  #t
filter

(filter <f> <list>) returns a list of the elements of <list> which satisfy the predicate <f>

> (filter odd? '(0 1 2 3 4 5))
(1 3 5)
> (filter (lambda (x) (> x 98.6))
'(101.1 98.6 98.1 99.4 102.2))
(101.1 99.4 102.2)

Example: filter

Example: filter

• Define integers as a function that returns a list of integers between a min and max

(define (integers min max)
(if (> min max)
null
(cons min (integers (add1 min) max)))))

• Do prime? as a predicate that is true of prime numbers and false otherwise

> (filter prime? (integers 2 20))
(2 3 5 7 11 13 17 19)

Here’s another pattern

• We often want to do something like sum the elements of a sequence

(define (sum-list l)
(if (null? l)
0
(+ (first l) (sum-list (rest l)))))

• Other times we want their product

(define (multiply-list l)
(if (null? l)
1
(* (first l) (multiply-list (rest l)))))

Example: reduce

• Reduce takes (i) a function, (ii) a final value and (iii) a list of arguments
  Reduce of +, 0, (v1 v2 v3 ... vn) is just V1 + V2 + V3 + ... Vn + 0

• In Scheme/Lisp notation:
  > (reduce + 0 '(1 2 3 4 5))
  15
  (reduce * 1 '(1 2 3 4 5))
  120
Example: reduce

(define (reduce function final list)
  (if (null? list)
      final
      (function
        (first list)
        (reduce function final (rest list))))))

Using reduce

(define (sum-list list)
  ;; returns the sum of the list elements
  (reduce + 0 list))

(define (mul-list list)
  ;; returns the sum of the list elements
  (reduce * 1 list))

(define (copy-list list)
  ;; copies the top level of a list
  (reduce cons '() list))

(define (append-list list)
  ;; appends all of the sublists in a list
  (reduce append '() list))

The roots of mapReduce

- MapReduce is a software framework developed by Google for parallel computation on large datasets on computer clusters
- It’s become an important way to exploit parallel computing using conventional programming languages and techniques.
- See Apache’s Hadoop for an open source version
- The framework was inspired by functional programming’s map, reduce & side-effect free programs

Function composition

- Math notation: $g \cdot h$ is a composition of functions $g$ and $h$
- If $f = g \cdot h$ then $f(x) = g(h(x)$
- Composing functions is easy in Scheme

> compose
#<procedure:compose>
> (define (sq x) (* x x))
> (define (dub x) (* x 2))
> (sq (dub 10))
400
> (dub (sq 10))
200
> (define sd (compose sq dub))
> (sd 10)
400
> ((compose dub sq) 10)

Defining compose

- Here’s compose for two functions in Scheme
  (define (compose2 f g) (lambda (x) (f (g x)))))
- Note that compose calls lambda which returns a new function that applies $f$ to the result of applying $g$ to $x$
- We’ll look at how the variable environments work to support this in the next topic, closures
- But first, let’s see how to define a general version of compose taking any number of args

Functions with any number of args

- Defining functions that takes any number of arguments is easy in Scheme
  (define (foo . args) (printf "My args: ~a\n" args)))
- If the parameter list ends in a symbol as opposed to null (cf. dotted pair), then it’s value is the list of the remaining arguments’ values
  (define (f x y . more-args) ...)
  (define (map f . lists) ...)

Compose in Scheme

(define (compose . FS)
  ;; Returns the identity function if no args given
  (if (null? FS)
      (lambda (x) x)
      (lambda (x) ((first FS) ((apply compose (rest FS)) x))))
  ;; examples
  (define (add-a-bang str) (string-append str "!"))
  (define givebang
    (compose string->symbol add-a-bang symbol->string))
  (givebang 'set) ; ===> set!
  ; anonymous composition
  ((compose sqrt negate square) 5) ; ===> 0+5i

A general every

• We can easily re-define other functions to take more than one argument
  (define (every fn . args)
    (cond ((null? args) #f)
          ((null? (first args)) #t)
          ((apply fn (map first args))
           (apply every fn (map rest args)))
          (else #f)))
  (every > '(1 2 3) '(0 2 3)) => #t
  (every > '(1 2 3) '(0 20 3)) => #f

Functional Programming Summary

• Lisp is the archetypal functional programming language
• It treated functions as first-class objects and uses the same representation for data & code
• The FP paradigm is a good match for many problems, esp. ones involving reasoning about or optimizing code or parallel execution
• While no pure FP languages are considered mainstream, many PLs support a FP style