FP Foundations, Scheme

In Text: Chapter 15
Functional Programming -- Prelude

- We have been discussing imperative languages
  - C/C++, Java, Fortran, Pascal etc. are imperative languages
  - Imperative languages follow the von Neumann architecture

- There is a different way of looking at things
  - Functional/Applicative programming
  - Based on Mathematics; Math functions
Functional Programming -- Mathematical Foundations

- A mathematical function is a mapping of members of one set to another set
  - The “input” set is called the domain
  - The “output” set is called the range

- Function definition -- usual form:
  \[ \text{cube}(x) = x \times x \times x, \text{ where } x \text{ is a real number} \]
  then, e.g., \( \text{cube}(2) = 8 \)

- Lambda notation (Alonzo Church, 1941) provides for nameless functions:
  \( (x) x \times x \times x \)

- Can apply just like a named function:
  \( (x) x \times x \times x)(2) = 8 \)

- Functions do not have state; they are side-effect free
Function composition: has two function parameters, yields a function whose value is the first function applied to the result of the second

\[ h = f \circ g \text{ -- means apply } g \text{ first, then apply } f \text{ to the result} \]

example: if \( f(x) = x + 2 \)

\( g(y) = 3 \times y \)

then \( h(z) = f(g(z)) = (3 \times z) + 2 \)

construction: takes a list of functions and applies each in turn to the argument, creating a list of results

written by enclosing function names in brackets, e.g. \([g,h,i]\)

example: if \( g(x) = x \times x \)

\( h(x) = 2 \times x \)

\( i(x) = x / 2 \)

then \([g, h, i](4)\) yields \((16, 8, 2)\)
Functional Programming -- Mathematical Foundations

- Functional Forms
  - apply-to-all: takes a single function and applies it to a list of arguments, creating a list of values
denoted by \( \alpha \)
example: if \( h(x) = x \times x \)
then \( \alpha(h, (2, 3, 4)) \) yields \( (4, 9, 16) \)
Functional Programming

- LISP: John McCarthy 1958 MIT
  - List Processing => Symbolic Manipulation

- First functional programming language
  - Every version after the first has imperative features, but we will discuss a subset that is purely functional

- Functional subset
  - No assignment statement
  - No looping statement
  - Recursion recursion recursion
LISP Data Types

- “Data Types:” everything in LISP are *S-expressions*
  - **Atoms**: identifiers, numbers, symbols
    - a
    - 100
    - +
  - **Lists**
    - (a b c d)
    - (a (b c) d e)

- All data structures in Lisp are single-linked lists
  - each node has 2 pointers, one to element, the other to next node in the list
Data Structures

- Single atom:

- List of atoms: (a b c)

- List containing list: (a (b c) d)
LISP Primitives

- Primitive numeric functions
  - +, -, *, /
    - 42 => 42
    - (* 3 6) => 18
    - (+ 1 2 3) => 6
    - (sqrt 16) => 4

- Numeric Predicate Functions
  - Predicate Functions return true (#t), or false (nil ()
    - =, <, >, >=, <=, =>, even? odd? zero?
      - ( = 16 16 ) => #t
      - ( even? 29 ) => ()
      - ( > 10 ( * 2 4 )) => #t
      - ( zero? ( - 10 ( * 2 5 ))) => #t
Defining functions

( define (function_name arg1 arg2 ... argn)
    Sexp
)

( define (square x)
    (* x x)
)

( define (hypotenuse side1 side2)
    (sqrt (+ (square side1) (square side2)))
)

You can also define named constants:

( define pie 3.14159)

( define (circumference r)
    (* 2 pie r)
)
Control Flow

- Control flow is modeled after math functions

\[ f(x) = \begin{cases} 
1 & \text{if } x = 0 \\
x * f(x - 1) & \text{if } x > 0 
\end{cases} \]

( if predicate then_expr else_expr )

( define ( factorial x )
  ( if (= x 0 )
    1
    (* x (factorial (- x 1)))
  )
)
List Functions

- **quote**  =>  

  (quote a)  =>  'a = a

  (quote (a b c))  =>  '(a b c) = (a b c)

- **car**  :  List  =>  Sexp

  - One input arg  =>  List

  - Returns first element of that list

    (car '(a b))  =>  a

    (car '((a b) c))  =>  (a b)

    (car '(a (b c)))  =>  a

    (car 'a)  =>  undefined

    (car '())  =>  undefined
List Functions

- **cdr**: List => List
  - One input arg => List
  - Returns list of all elements but the first element
    - \( (\text{cdr} \ 'a \ b \ c) \) => (b c)
    - \( (\text{cdr} \ '((a \ b) \ (c))))) \) => ((c))
    - \( (\text{cdr} \ 'a) \) => ()
    - \( (\text{cdr} \ 'a) \) => undefined
    - \( (\text{cdr} \ '()) \) => undefined
    - \( (\text{cdr} \ '55) \) => undefined
List Functions

- \texttt{cons : Sexp \times X \rightarrow List}
  - 2 args as input: (cons \ a1 \ a2)
    \begin{itemize}
    \item \ a1 : Sexp
    \item \ a2 : List
    \end{itemize}
  - Returns \ a2 with \ a1 inserted as its first element
    \begin{itemize}
    \item \ (\texttt{cons 'a '(b c)}) \Rightarrow (a \ b \ c)
    \item \ (\texttt{cons 'a '()}) \Rightarrow (a)
    \item \ (\texttt{cons '(a b) '((c d) e)}) \Rightarrow ((a \ b) (c \ d) e)
    \end{itemize}
  - Be careful, \texttt{cons} will take non list (atom) arguments and form "dotted" pairs, e.g.
    \begin{itemize}
    \item \ (\texttt{cons 'a 'b}) \Rightarrow (a . b)
    \end{itemize}
  - \textbf{Note:} \ \texttt{(cons? X)} is a predicate form in Dr. Scheme and returns true if \(X\) is a list
More Predicates

- the following return #t if the arguments are of the indicated type, and nil () otherwise

<table>
<thead>
<tr>
<th>(symbol? 'a)</th>
<th>=&gt;</th>
<th>#t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(symbol? '())</td>
<td>=&gt;</td>
<td>()</td>
</tr>
<tr>
<td>(symbol? 55)</td>
<td>=&gt;</td>
<td>()</td>
</tr>
<tr>
<td>(number? '55)</td>
<td>=&gt;</td>
<td>#t</td>
</tr>
<tr>
<td>(number? 55)</td>
<td>=&gt;</td>
<td>#t</td>
</tr>
<tr>
<td>(number? '(a))</td>
<td>=&gt;</td>
<td>()</td>
</tr>
</tbody>
</table>

**LIST?:** Dr. Scheme

<table>
<thead>
<tr>
<th>(cons? '(a))</th>
<th>=&gt;</th>
<th>#t</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cons? 'a)</td>
<td>=&gt;</td>
<td>()</td>
</tr>
<tr>
<td>(cons? '())</td>
<td>=&gt;</td>
<td>#t</td>
</tr>
<tr>
<td>(null? '(a c))</td>
<td>=&gt;</td>
<td>()</td>
</tr>
<tr>
<td>(null? '()</td>
<td>=&gt;</td>
<td>#t</td>
</tr>
</tbody>
</table>
More Predicates

- **eq?** \( \text{Sexp X Sexp} \Rightarrow \{ \#t , () \} \)
  Returns true if objects are equal through pointer comparison. Guaranteed to work on symbols

- **equal?** \( \text{Sexp X Sexp} \Rightarrow \{\#t , () \} \)
  Recursively compares two objects to determine if they are equal (works on symbols, atoms, numbers, and lists)
Examples

How do we implement equal?

(define (atom? atm)
  (cond
    ((cons? atm) (null? atm))
    ((symbol? atm) #t)
    ((number? atm) #t)
    (else ()
      )))

(define (equal? lis1 lis2)
  (cond
    ((atom? lis1) (eq? lis1 lis2))
    ((atom? lis2) ()
      )
    ((equal? (car lis1) (car lis2))
      (equal? (cdr lis1) (cdr lis2)))
    (else ()
      ))
  )
(define (member? atm lis)
  (cond
    ((null? lis) ())
    ((eq? atm (car lis)) #t)
    (else (member? atm (cdr lis))))
)

(define (append lis1 lis2)
  (cond
    ((null? lis1) lis2)
    (else (cons (car lis1) (append (cdr lis1) lis2))))
)
Additional Functions

- **Additional control flow primitives that are available:**
  
  \[
  \text{(if } Sexp_1 \ Sexp_2 \ [Sexp_3 ]) \\]
  
  if \((Sexp_1 )\) then \(Sexp_2 \) [else \(Sexp_3 \)]

  \[
  \text{(while } Sexp_1 \ Sexp_2 \ ) \\]
  
  while \((Sexp_1 )\) do \((Sexp_2 )\) od

- **Blocking primitive:**
  
  \[
  \text{(begin } Sexp_1 \ Sexp_2 \ ... \ Sexp_n \ ) \\]

- **Variable initialization primitive:**
  
  \[
  \text{(set! } x \ Sexp) \\]

- **USE THESE FOR TESTING PURPOSES ONLY!**
Lambda Expressions

- Intuitively, *lambda expressions* allow one to define and use nameless functions and to pass them to be used in other functions

\[(\text{lambda (lis) (car (cdr lis))})\]

- Given to a lisp interpreter, the above function definition returns the second element in a list, e.g.

\[
((\text{lambda (lis) (car (cdr lis))}) '(a b c))
\]

returns "b"
Lambda Expressions

- We CAN integrate the lambda expression into a function definition:

  (define second
   (lambda (lis) (car (cdr lis))))

- Once "evaled" by the interpreter, the function definition is "bound" to the name "second" such that
  (second '(a b c)) => b

- But, our "standard" way of defining functions will work too ...

  (define (second lis)
   (car (cdr lis)))

  (second '(a b c)) => b

- **SO, WHAT DOES THE LAMBDA EXPRESSION BUY US?**
WE NOW HAVE THE CAPABILITY TO PASS FUNCTION DEFINITIONS AS PARAMETERS!

- Suppose that we want to write an "apply-to-all" function that takes a function definition and list as its arguments and applies its function argument to all elements in the list.

```scheme
(define (mapcar fctn lis)
  (cond
    ((null? lis) ())
    (else (cons (fctn (car lis))
                 (mapcar fctn (cdr lis))))))
```

```scheme
(mapcar (lambda (num) (* num num)) '(2 4 6))
```
returns a list containing the square of all elements in the original list, i.e.,

(4 16 36)
Lambda Expressions

- Why not simply define a function “f” that performs a specified operation on one element and then pass it to mapcar, e.g.

  (define (square x)
    (* x x)
  )

  (define (mapcar fctn lis)
    (cond
      ((null? lis) ())
      (else (cons (fctn (car lis))
        (mapcar fctn (cdr lis)))))
  )

  (mapcar square '(2 4 6))
Scoping in LISP

- Lisp allows programs to reference "unbound" variables, e.g.

    (define (f atm)
        (cons atm y) ; y is an unbound variable
    )

- What are the implications of this capability with respect to scoping?
Scoping in LISP

- In reality, Lisp does allow one to define global and local variables, e.g.
  
  (define x 5) ; Global x
  (set! x (car '(a b c)) ; Gbl/Lcl x

  In reality, Lisp does allow programs to reference "unbound" variables, e.g.
  
  (define (f atm)
      (cons x y) ; y is an unbound variable
  ) ; x assumes the prev (set! x.... )

  What are the implications of these capabilities with respect to scoping?
Scoping in LISP

- Consider the following example:

```scheme
(define (A ...) 
    ... (car X) ... ; unbound ref
)

(define (B Fctn X) 
    ... Fctn ... ; invoke Fctn *
)

(define (C X Z) 
    ... A ... ; invoke fctn A ** 
    ... (B A Z) ... ; invoke fctn B ***
)
Scoping in LISP

- Assuming that "our" Lisp is *statically* scoped (and most current Lisps are statically scoped), let's consider the impact of the following invocation of C:

  \[(C \,(i\ j)\ ,(k\ l\ m))\]

  - What is the binding of X in A after being invoked at ** ?
  - What is the binding of X in A after being sent to B through the call *** and being invoked at * ?
  - Is it what you expected?

- What is needed is the ability to bind an *execution environment* at same time a function is passed as a parameter! (Funarg Problem)

  - Solution: ... \((B\ \text{(function} \ A)\ Z)\) ...
Consider the same scenario again:

```
(define (A ...)  
  ... (car X) ...  
)
```

```
(define (B Fctn X)  
  ... Fctn ...  
)
```

```
(define (C X Z)  
  ... A ...  
  ... (B (function A) Z) ...  
)
```

In Pascal, static scoping and lexical scoping are effectively synonymous.

Although we are able to achieve "static" scoping through the use of the *function* primitive, is this also a "lexical" scoping?
SCHEME

- SCHEME is a dialect of LISP
- SCHEME extends LISP
  - Statically scoped
  - Minor syntax changes
    - will not affect us
  - Has extensions
    - additional functions we will not use
- We will use only the “pure” LISP parts of SCHEME
A Little Help

- Access Help Desk
  - Start => Programs => PTL Scheme => Help Desk

Software
  Tour
  Manuals