ICOM 4036: PROGRAMMING LANGUAGES

Lecture 6

Functional Programming

The Case of Scheme

Required Readings

- **Texbook (Scott PLP)**
  - Chapter 11 Section 2: Functional Programming
- **Scheme Language Description**
  - Revised Report on the Algorithmic Language Scheme
    (available at the course website in Postscript format)

At least one exam question will cover these readings

Functional Programming Impacts

Functional programming as a minority discipline in the field of programming languages nears a certain resemblance to socialism in its relation to conventional, capitalist economic doctrine. Their proponents are often brilliant intellectuals perceived to be radical and rather unrealistic by the mainstream, but little-by-little changes are made in conventional languages and economics to incorporate features of the radical proposals.

- Morris [1982] “Real programming in functional languages

Functional Programming Highlights

- **Conventional Imperative Languages Motivated by von Neumann Architecture**
- **Functional programming= New machanism for abstraction**
- **Functional Composition = Interfacing**
- **Solutions as a series of function application**

  f(a), g(f(a)), h(g(f(a))), ...
- **Program is an notation or encoding for <u>a value</u>**
- **Computation proceeds by rewriting the program into that value**
- **Sequencing of events not as important**
- **In pure functional languages there is no notion of state**

Functional Programming Phylosophy

- Symbolic computation / Experimental programming
- Easy syntax / Easy to parse / Easy to modify.
- Programs as data
- High-Order functions
- Reusability
- No side effects (Pure!)
- Dynamic & implicit type systems
- Garbage Collection (Implicit Automatic Storage management)

Garbage Collection

- **At a given point in the execution of a program, a memory location is garbage if no continued execution of the program from this point can access the memory location.**
- **Garbage Collection: Detects unreachable objects during program execution & it is invoked when more memory is needed**
- **Decision made by run-time system, not by the program (Memory Management).**

What's wrong with this picture?

- **Theoretically, every imperative program can be written as a functional program.**

- **However, can we use functional programming for practical applications?**

  (Compilers, Graphical Users Interfaces, Network Routers, ...)

Eternal Debate: But, most complex software today is written in imperative languages

LISP

- Lisp= List Processing
- Implemented for processing symbolic information
- McCarthy: “Recursive functions of symbolic expressions and their computation by machine” Communications of the ACM, 1960.
- 1970’s: Scheme, Portable Standard Lisp
- 1984: Common Lisp
- 1986: use of Lisp ad internal scripting languages for GNU Emacs and AutoCAD.

History (1)

Fortran

$\downarrow$

FLPL (Fortran List Processing Language)
No recursion and conditionals within expressions.

$\downarrow$

Lisp (List processor)

History (2)

- Lisp (List Processor, McCarthy 1960)
  - * Higher order functions
  - * conditional expressions
  - * data/program duality
  - * scheme (dialect of Lisp, Steele & Sussman 1975)
- APL (Inverson 1962)
  - * Array basic data type
  - * Many array operators

History (3)

- IFWIM (If You Know What I Mean, Landin 1966)
  * Infix notation
  * equational declarative

- ML (Meta Language – Gordon, Milner, Appel, McQueen 1970)
  * static, strong typed language
  * machine assisted system for formal proofs
  * data abstraction
  * Standard ML (1983)

History (4)

- FP (Backus 1978)
  * Lambda calculus
  * implicit data flow specification

- SASL/KRC/Miranda (Turner 1979,1982,1985)
  * math-like sintax

Scheme: A dialect of LISP

- **READ-EVAL-PRINT Loop (interpreter)**
- **Prefix Notation**
- **Fully Parenthesized**
- **(* (* (+ 3 5) (- 3 (/ 4 3))) (- (* (+ 4 5) (+ 7 6)) 4))**
-
```
(* (*  (+ 3 5)
       (- 3 (/ 4 3)))
   (- (* (+ 4 5)
         (+ 7 6))
    4))
```

A scheme expression results from a pre-order traversal of an expression syntax tree

Scheme Definitions and Expressions

- (define pi 3.14159)   ; bind a variable to a value

  pi
- pi

  3.14159
- (* 5 7)

  35
- (+ 3 (* 7 4))

  31   ; parenthesized prefix notation

Scheme Functions

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

  square

- (square 5)

  25

- ((lambda (x) (*x x)) 5)   ; unamed function

  25

The benefit of lambda notation is that a function value can appear within expressions, either as an operator or as an argument.

Scheme programs can construct functions dynamically

Functions that Call other Functions

- (define square (x) (* x x))
- (define square (lambda (x) (* x x)))
- (define sum-of-squares (lambda (x y)

```
                    (+ (square x) (square y))))
```

Named procedures are so powerful because they allow us to hide details and solve the problem at a higher level of abstraction.

Scheme Conditional Expressions

- (If P E1 E2)   ; if P then E1 else E2
- (cond (P1 E1)   ; if P1 then E1

```
          .....
         (Pk Ek)            ; else if Pk  then Ek
        (else Ek+1))     ;  else Ek+1
```

- (define (fact n)

```
    (if (equal? n 0)
       1
      (*n (fact (- n 1)))  ) )
```

Blackboard Exercises

- Fibonacci
- GCD

Scheme: List Processing (1)

- (null? ( ))

  #t
- (define x ‘((It is great) to (see) you))

  x
- (car x)

  (It is great)
- (cdr x)

  (to (see) you)
- (car (car x))

  It
- (cdr (car x))

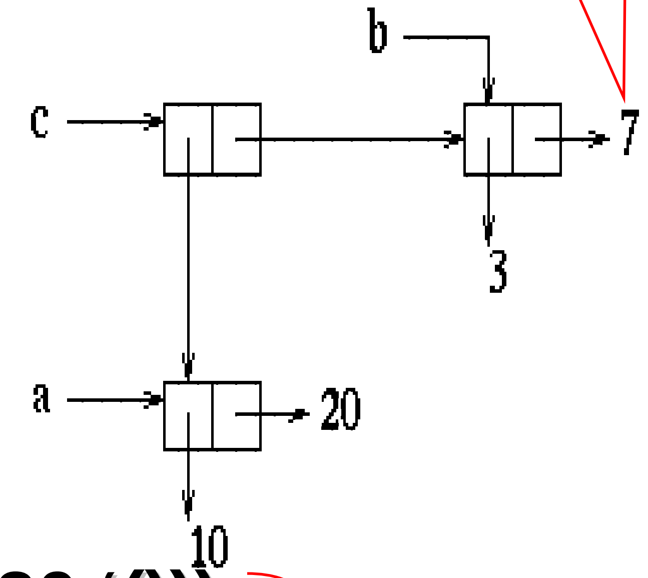
  (is great)

Quote delays evaluation of expression

Scheme: List Processing (2)

- (define a (cons 10 20))
- (define b (cons 3 7))
- (define c (cons a b))

Not a list!!
(not null terminated)



- (define a (cons 10 (cons 20 ‘())))
- (define a (list 10 20))

Equivalent

Scheme List Processing (3)

- (define (lenght x)

  (cond ((null? x) 0)

  (else (+ 1 (length (cdr x))))))

- (define (append x z)

  (cond ((null? x) z)

  (else (cons (car x) (append (cdr x) z)))))

- (append \`(a b c) \`(d))

  (a b c d)

Backboard Exercises

- Map(List,Funtion)
- Fold(List,Op,Init)
- Fold-map(List,Op,Init,Function)

More Backboard Exercises

- **Using fold and map as abstractions**
  - Compute the length of a list
  - Determine if list has a list inside
  - Determine if a list of numbers includes a negative
  - Determine is all elements in the list satisfy a predicate p
  - Determine is all elements in the list satisfy a predicate p

Scheme: Implemeting Stacks as Lists

- Devise a representation for staks and implementations for the functions:

  push (h, st)   returns stack with h on top

  top (st)   returns top element of stack

  pop(st)   returns stack with top element removed

-

  Solution:

  represent stack by a list

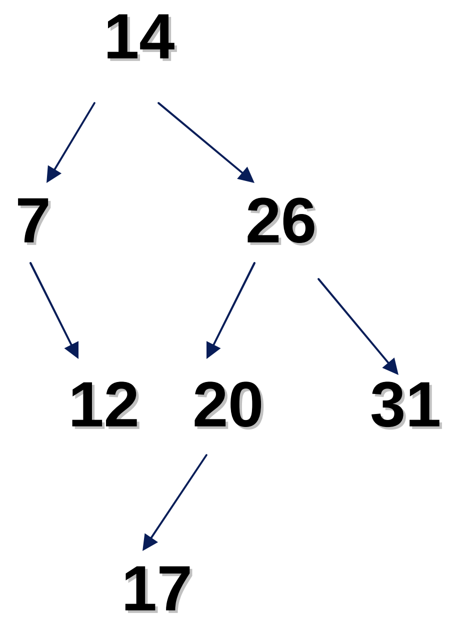
  push=cons

  top=car

  pop=cdr

List Representation for Binary Search Trees

```
'(14 (7  ()
         (12()()))
     (26 (20
           (17()())
           ())
         (31()())))
```


$$\begin{tikzcd}
 & & 14 \arrow[dll] \arrow[dr] & & \\
7 \arrow[dr] & & & 26 \arrow[dl] \arrow[dr] & \\
 & 12 & 20 \arrow[d] & & 31 \\
 & & 17 & &
\end{tikzcd}$$

Binary Search Tree Data Type

- (define make-tree (lambda (n l r) (list n l r)))
- (define empty-tree? (lambda (bst) (null? bst)))
- (define label (lambda (bst) (car bst)))
- (define left-subtree (lambda (bst) (car (cdr bst))))
- (define right-subtree (lambda (bst) (car (cdr (cdr bst)))))

Searching a Binary Search Tree

```
(define find
  (lambda (n bst)
    (cond
        ((empty-tree? bst) #f)
        ((= n (label bst)) #t)
        ((< n (label bst)) (find n (left-subtree bst)))
        ((> n (label bst)) (find n (right-subtree bst))))))
```

Recovering a Binary Search Tree Path

```
(define path
  (lambda (n bst)
    (if (empty-tree? bst)
        ‘()                    ;; didn't find it
        (if (< n (label bst))
            (cons 'L (path n (left-subtree bst)))      ;; in the left subtree
            (if (> n (label bst))
                (cons 'R (path n (right-subtree bst)))  ;; in the right subtree
                '()                                     ;; n is here, quit
                )
            )
        )
   ))
```

Blackboard Exercise

- Write a Scheme interpreter in Scheme

List Representation of Sets

$$\text{Math} \longrightarrow \{\, 1, 2, 3, 4 \,\}$$

Scheme $\longrightarrow$ **(list 1 2 3 4)**

List Representation of Sets

- (define (member? e set)
```
    (cond
        ((null? set) #f)
        ((equal? e (car set)) #t)
        (else (member? e (cdr set)))
    )
)
```

- (member? 4 (list 1 2 3 4))
```
  > #t
```

Set Difference

```
(define (setdiff lis1 lis2)
        (cond
             ((null? lis1) '())
             ((null? lis2) lis1)
             ((member? (car lis1) lis2)
                 (setdiff (cdr lis1) lis2))
             (else (cons (car lis1) (setdiff (cdr lis1) lis2)))
        )
)
```

Set Intersection

```
(define (intersection lis1 lis2)
        (cond
             ((null? lis1) '())
             ((null? lis2) '())
             ((member? (car lis1) lis2)
                 (cons (car lis1)
                       (intersection (cdr lis1) lis2)))
             (else (intersection (cdr lis1) lis2))
        )
)
```

Set Union

```
(define (union lis1 lis2)
      (cond
          ((null? lis1) lis2)
          ((null? lis2) lis1)
          ((member? (car lis1) lis2)
             (cons (car lis1)
                  (union (cdr lis1)
                        (setdiff lis2 (cons (car lis1) '()))))))
          (else (cons (car lis1) (union (cdr lis1) lis2)))
      )
)
```

Functional Languages: Remark 1

- In Functional Languages, you can concern yourself with the higher level details of what you want accomplished, and not with the lower details of how it is accomplished. In turn, this reduces both development and maintenance cost

Functional Languages: Remark 2

- **Digital circuits are made up of a number of functional units connected by wires. Thus, functional composition is a direct model of this application. This connection has caught the interest of fabricants and functional languages are now being used to design and model chips**
    - Example: Products form *Cadence Design Systems,* a leading vendor of electronic design automation tools for IC design, are scripted with SKILL (a proprietary dialect of LISP)

Functional Languages: Remark 3

- **Common Language Runtime (CLR) offers the possibility for multi-language solutions to problems within which various parts of the problem are best solved with different languages, at the same time offering some layer of transparent inter-language communication among solution components.**
  - Example: Mondrian (http://www.mondrian-script.org) is a purely functional language specifically designed to leverage the possibilities of the .NET framework. Mondrian is designed to interoperate with object-oriented languages (C++, C#)

Functional Languages: Remark 4

- **Functional languages, in particular Scheme, have a significant impact on applications areas such as**
  - Artificial Intelligence (Expert systems, planning, etc)
  - Simulation and modeling
  - Applications programming (CAD, Mathematica)
  - Rapid prototyping
  - Extended languages (webservers, image processing)
  - Apps with Embedded Interpreters (EMACS lisp)

Functional Languages: Remark 5

- If all you have is a hammer, then everything looks like a nail.

END OF LECTURE 5