Functional Programming Concepts

Features

- Defines output of a program as a function of input
- Miranda,Haskell,PH,Sisal,Single Assignment C
- **Features**
- First class function values & higher order functions
  - Can be passed as parameter,returned fromfunction or assigned into a variable
- Extensive polymorphism
- Supports list types and operators
- Structured function returns
- Constructors for structured objects
- Garbage collection

Scheme

- Dialect of Lisp
- Emerged from MIT in mid 1990
- Small size
- Use of static scoping
- Function is a first class entity

Scheme Interpreter

- Read-eval-print loop
- EVAL function

  (+3 4)=>7

  7=>7
- Each parameter expression is evaluated
- Primitive function '+' is applied
- Result is displayed
- Read i/p from file
- (load"my_scheme_pgm")

- +,-,*,/
- *and + takes 0 or more parameters
- If * is given no parameter
-   It returns 1
- If + is given no parameter
-   -It returns 0
- / - takes 2 or more parameters
- Uses Cambridge polish notation for expressions

- (+ 3 4) $\Longrightarrow$ 7
- ((+ 3 4)) $\Longrightarrow$ error

- When inner paranthesis found
  - Calls function '+' passing 3 and 4 as arguments
- When outer paranthesis found
  - Calls 7 as a zero argument function which is a runtime error

- **Sqrt**
- Returns square root of non-negative numeric parameters

Defining Functions

- Scheme program is a collection of functions
- Nameless functions are defined using lambda expressions
- Lambda expressions are unnamed functions of the form:
- (lambda(id..) exp)
- (id...) -> list of formal parameters
- exp --> body of lambda expression

```
((lambda (x) (* x x))
((lambda (x) (* x x)) 3) ==> 9
```

- x-> bound variable
- Keyword DEFINE serves 2 purposes
- To bind a name to value
- To bind a name to a lambda expression

To Bind a Name to Value

- (define symbol expression)
- (define pi 3.14)
- (define two_pi(* 2 pi))

  pi=> 3.14

  two_pi=> 6.28

- Rules for creating names:
  - Can have letters,digits and special symbols
  - Paranthesis must be used
  - Must not begin with digit
  - They are case sensitive

- (define(function-name parameters)(expression))

- (define (square number)(* number number))
- (square 5) =>25

- (define min (lambda (a b) (if (< a b) a b)))

- (min 123 456) =>123

Output Functions

- (display expression)
- (newline)

Numeric Predicate Functions

- Predicate functions return a boolean value

Function	Meaning
=	Equal
< >	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
Even?	Is it an even number?
Odd?	Is it an odd number?
Zero?	Is it zero?
(boolean? X)	Is x a boolean?

- (char? x) ; is x a character?
- (string? x) ; is x a string?
- (symbol? x) ; is x a symbol?
- (number? x) ; is x a number?
- (list? x) ; is x a (proper) list?

Example

- (negative?-6) returns t
- (zero?44) return f
- (>6,2) returns t

Symbols

- procedure: symbol?Object
- Returns true if the object is a symbol,otherwise returns false

- (symbol? ’x$_%:&=*!) =⇒ #t

- Type is not determined at runtime
- (if(> a 0)(+2 3)(+ 2 "foo")
- Will evaluate to 5 if 'a' is positive
- Will produce a type clash error if 'a' is negetive or 0

Quote

- Prevent interpreter from evaluating an expression
- (quote A) <=> '(A) => A
- (quote (+ 3 4)) =⇒ '(+ 3 4) =⇒(+ 3 4)
- '3; =⇒ 3
- '"hi"; =⇒ "hi"
- 'a; =⇒ a
- '(+3 4); =⇒ (list'+ '3'4)
- '(abc); =⇒(list 'a 'b 'c)

Control Flow

- If and cond constructs are available
- If has 3 parameters
- (if predicate then_expression else_expression)

- if(<2 3)4 5) => 4
- (if #f 2 3) => 3

```
(define (compare x y)
(cond
((> x y)" x is greater than y")
((< x y)"y is greater thanx")
(else "x and y are equal")
)
```

Bindings- Nested Scope

- Names can be bound to values by introducing a nested scope:
- 'let' careates a new local static scope
- Short hand for lambda expressions

```
(let ((alpha 7))
(* 5 alpha)
)
```

```
(let ((a 3)
(b 4)
(square (lambda (x) (* x x)))
(plus +))
(sqrt (plus (square a) (square b)))) ==> 5.0
```

- let evaluates all bindings with respect to the level above it
- let* does it sequentially

(let ((a 3))

(let ((a 4)

(b a)) -Since b is assigned a before the defenition for 2nd a is completed, b takes the value of 1st a

(+ a b))) =⇒ 7

(let ((a 3))

(let* ((a 4)

(b a)) -Since b is assigned a, it takesthe value of a just above it

(+ a b))) =⇒ 8

- letrec allows to bind recursive values
- (letrec ((fact (lambda (n)
- (if (=n 1)
- 1
- (* n(fact(-n 1)))))))

- (fact 5)=> 120

List and Numbers

- 3 main functions to manipulate lists
- car - which returns the head of a list
- cdr - everything after the head
- cons-joins a head to the rest of a list

(car '(2 3 4)) $\Longrightarrow$ 2

(cdr '(2 3 4)) $\Longrightarrow$ (3 4)

(cons 2 '(3 4)) $\Longrightarrow$ (2 3 4)

EqualityTesting and Searching

- General Purpose - eq? eqv? equal? can be used
- Eq? check if 2 parameters represent the same object
- Eqv? – if they have same value
- Equal? – checks for recursive values
- (eq? x y)

- eqv? returns true for same primitive values

(eqv? 2 2) => t

(eqv? "a" "a") => t

- equal? - test lists, vectors etc.,

```
(define(x '(2 3))
(define(y '(2 3))
(eq?(x y)) =>  #f
(equal? x y) => t
(define y x)
(eq? x y) =>t
```

Search elements in List

- 3 functions are there- memq, memv and member
- Each one takes an element and a list as argument and returns longest suffix of list
- memq uses eq, memv uses eqv and member uses equal
- (memq 'z '(x y z w)) $\Longrightarrow$ (z w)
- (memv '(z) '(x y (z) w)) $\Longrightarrow$ #f (recursive)
- (member '(z) '(x y (z) w)) $\Longrightarrow$ ((z) w)

Association (A-list)

- The functions assq, assv, and assoc search for values in *association lists*
- *List is a dictionary*
- *1st element – key*
- *2nd element- information*
- Assq uses eq , assv uses eqv, and assoc uses equal to search in alist

- (define e '((a 1)(b 2) (c 3)))
- (assq 'a e) => (a 1)
- (assq 'b e) => (b 2)
- (assq 'd e) => f
- (assoc ‘(a) ‘(((a)) ((b)) ((c))) => ((a))
- (assv 5 ‘((2 3) (5 7) (11 13))) => (5 7)

- Scheme do not support side effect
- It is supported by

1. Assignment
2. Iteration
3. Sequencing

Assignment

(let ((x 2) ; initialize x to 2

(l ’(a b))) ; initialize l to (a b)

(set! x 3) ; assign x the value 3

(set-car! l ’(c d)) ; assign head of l the value (c d)

(set-cdr! l ’(e)) ; assign rest of l the value (e)

... x $\Longrightarrow$ 3

... l $\Longrightarrow$ ((c d) (e))

Sequencing

Controls the flow of control

```
(begin
(display "hi ")
(display "mom"))
```

Iteration - do

```
(do ((<var1> <init> <step>))
(<test> <exp>)
<command>)
(define i 0)
(do()
((=i 5))
(display i)
(set! i(+ i 1)))
01234
```

for-each

- (for-each (lambda (a b) (display (* a b)) (newline))
- ’(2 4 6)
- ’(3 5 7))
- // a and b are 2list
- 6 20 42 ()

Programs as Lists

```
(define compose
(lambda (f g)
(lambda (x) (f (g x)))))
((compose car cdr) '(1 2 3)) =⇒ 2
```

- Compose takes a pair of function as arguments
- It returns a function that takes x as parameter, as result

```
(define compose2
(lambda (f g)
(eval (list ‘lambda ’(x) (list f (list g ’x)))
(scheme-report-environment 5))))
((compose2 car cdr) ’(1 2 3)) =⇒ 2
```

- eval function evaluates a list
- compose2 performs same function
- Function list returns a list that is evaluated
- 2nd argument to eval is the RE in which expression is to be evaluated

Eval *and* Apply

Takes scheme object and evaluates it

(define fn '*)

(define x 3)

(define y(list '+ x 5))

(define z(list fn 10 y))

x => 3

y =>(+3 5)

(eval y) => 8

(eval z) => 80

Example 2

- (define a 'b)
- (define b 'c)
- (define c 50)
- a=>b
- (eval a) = c
- (eval(eval a)) =>50

Apply

- Apply takes two arguments: a function and a list
- Apply a function to an expression

- (apply factorial ‘(3)) => 6
- (apply + ‘(1 2 3 4)) =>10
- (define (sum s) (apply + s))
- (sum ‘(1 2 3)) => 6

Evaluation Order

- Applicative order: Evaluate arguments before
- Normal Order: Pass arguments unevaluated
- Scheme -> Applicative order

- (define double(lambda(x)(+x x)))

Applicative Order Evaluation

```
(define double(lambda(x)(+ x x)))
(double(* 3 4))
(double(12))
=> (+ 12 12)
=>24
```

Normal Order Evaluation

```
(define double(lambda(x)(+ x x)))
(double (* 3 4))
==> (+ (* 3 4) (* 3 4))
==> (+ 12 (* 3 4))
==> (+ 12 12)
==> 24
```

Strictness and Lazy Evaluation

- Evaluation order effects execution speed and program correctness
- Program that finds dynamic semantic error or infinite loop in applicative order may terminate in normal order
- Applicative order - faster

- A strict function denoted an expression which does not return a normal value
  - either because it loops endlessly
  - because it aborts due to an error such as division by zero
- A function which is not strict is called **non strict**

- Strict language - functions are always strict
- NonStrict language - allow non strict functions
- Strict language follows applicative order evaluation
- ML and Scheme - strict
- Miranda and Haskell - nonstrict

Lazy Evaluation

- Lazy evaluation uses normal order evaluation
  - - Arguments are not evaluated unnecessarily
- Tag every argument with a memo indicating that its value is known
- Attempt to evaluate argument sets value in memo or returns value if already set
- Useful for infinite data structures

Example

- (define double(lambda(x)(+ x x)))

```
(define f
     (lambda ()
          (let ((done #f) ; memo initially unset
               (memo '())
               (code (lambda () (* 3 4))))
          (if done memo ; if memo is set, return it
             (begin
             (set! memo (code)) ; remember value (set! done #t)
                              memo))))) ; and return it
...
(double (f))
=⇒ (+ (f) (f))
=⇒ (+ 12 (f)) ; first call computes value
=⇒ (+ 12 12) ; second call returns remembered value
=⇒ 24
```

Delay and force

- Lazy evaluation is available in Scheme
- The delay primitive returns a promise
- redeemed by the force primitive

- (delay expr) => promise
- (force promise) => value

- **Expression** **Value**
- (delay(+ 5 4)) #Promise
- (let((delayed(delay (+ 5 6)))
- (force delayed)) 11