Principles of Programming Languages

Module M03: Functional Programming

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

Presented jointly with Srijoni Majumder, January 17 & 19, 2022

Table of Contents

Functional Programming

Functions in Mathematics and Programming

- A function, in the *mathematical sense*, is a set of operations that perform some computation on the parameter(s) passed, and return a value
- A function, in a *programming language*, is a set of code whose purpose is to compute based on the parameter(s) and return a value
- Functions are often used to promote **modularity**
  - break down program tasks into small, roughly independent pieces
  - promotes structured programming in terms of design
  - aids debugging, coding, and maintenance
- **However, the function, as we see them in programming languages, does not necessarily reflect a mathematical function**

Functions in Mathematics and Programming

- **Functions** may act more like **procedures**
  - a procedure is another unit of modularity
  - the idea behind a procedure is to accomplish one or more related activities
  - the activities should make up some logical goal of the program but may not necessarily be based on producing a single result
  - procedures may return 0 items or multiple items unlike functions
- In languages like C, there are no procedures, so functions must take on multiple roles
  - mathematical types of functions
  - functions that act as procedures
- Such functions can produce **side effects**
  - mathematical functions do not produce side effects

Functional Design

- **Functions** should
  - be *concise*
    - accomplish only a single task or goal (pop in stack should not return the top element)
  - *return one item*
    - in C, we can wrap return multiple values in a structure
    - in C++, we can use reference parameters (side effect)
  - have no *side effect*
    - because the assignment operations are done by function calls, the function calls must produce side effects in such circumstances, but in general, the code should not produce side effects
    - most C/C++ function do (like printf, sort)
  - use *parameter passing* for communication
    - rather than global variables
  - exploit *recursion* when possible
    - this simplifies the body of a function and requires fewer or no local variables

What is Functional Programming?

- A *program* in a functional programming language is basically a *function call* which likely makes use of other functions
  - The basic *building block* of such programs is the *function*
  - Functions produce *results (based on arguments)*, but *do not change any memory state*
  - In other words, pure functions do not have any *side effects*
- Everything *is an expression, not an assignment*
  - In an imperative language, a program is a sequence of assignments
  - Assignments produce results by changing the memory state

Functional Programming Paradigm

- The *Functional Programming Paradigm* is one of the major programming paradigms
  - FP is a type of *declarative programming paradigm*
    - C is procedural
    - Java is object oriented
    - SQL is declarative, but it is not FP
  - Also known as *applicative programming* and *value-oriented programming*
- Idea: everything is a *function*
- Based on sound *theoretical frameworks* (for example, the *lambda calculus*)
- Examples of FP languages
  - Early FP language: **Lisp**
  - Important FPs: **ML**, **Haskell**, **Miranda**, **Scheme**, **Logo**
  - FPs in other paradigm languages: C++, Python, Java

Functional Programming Languages

- The design of the *imperative languages* is based directly on the *von Neumann architecture*
  - Efficiency is the primary concern, rather than the suitability of the language for software development
- The design of the *functional languages* is based on *mathematical functions*
  - A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

Programming Language Concepts in Functional Programming

- Anonymous Functions (Module 02)
- Curried functions (Module 02)
- Functional abstraction (Module 02)
- Recursion (Module 02)
- Higher-order functions (Module 02)
- Lazy evaluation (Module 04)
- Type inferencing (Module 04, 05, 07)
- Functions as First Class Objects (Module 06)
- Polymorphism (Module 04, 05)
- Formal (Denotational) Semantics (Module 08)

Characteristics of Pure FPLs

- Pure FP languages tend to
  - Have *no side-effects*
  - Have *no assignment statements*
  - Often have *no variables*!
  - Be built on a *small, concise framework*
  - Have a *simple, uniform syntax*
  - Be implemented via *interpreters* rather than *compilers*
  - Be mathematically *easier to handle*
  - Pure FPLs have *no side effects*
    - **Haskell** and **Miranda** are the two most popular examples
- Some FPLs try to be more practical and do allow some side effects
  - **Lisp** and its dialects (like **Scheme**)
  - **ML** (**Meta Language**) and **SML** (**Standard ML**)

Importance of Functional Programming

- In their pure form FPLs dispense with the *notion of assignment*
  - it is easier to program in them
  - easier to reason about programs written in them
- FPLs encourage thinking at *higher levels of abstraction*
  - support modifying and combining existing programs
  - encourage programmers to work in units larger than statements of conventional languages: *programming in the large*
- FPLs provide a paradigm for *parallel computing*
  - absence of assignment (or single assignment)
  - independence of evaluation order
  - ability to operate on entire data structures

Importance of Functional Programming

- FPLs are valuable in developing *executable specifications* and *prototype implementations*
  - Simple underlying semantics
    - rigorous mathematical foundations
    - ability to operate on entire data structures
    - ideal vehicle for capturing specifications
- FPLs are *very useful for AI* and other applications which require extensive symbol manipulation
- Functional Programming is tied to CS theory
  - provides framework for viewing *decidability questions*
  - Good introduction to *Denotational Semantics* (Module 08)
    - functional in form

Lisp

- Defined by John McCarthy in 1958 as a language for AI
- Originally, **LISP** was a *typeless* language with only two data types: *atom* and *list*
- **Lisp**'s lists are stored internally as single-linked lists
- Lambda notation was used to specify functions
- Function definitions, function applications, and data all have the same form
  - If the list (A B C) is interpreted as data it is a simple list of three atoms, A, B, and C but if interpreted as a function application, it means that the function named A is applied to the two parameters, B and C
- Example (early **Lisp**):

```
(defun fact (n) (cond ((lessp n 2) 1)(T (times n (fact (sub1 n))))))
```

- **Common Lisp** is the *ANSI standard Lisp specification*

Scheme

- In mid 70's Sussman and Steele (MIT) defined **Scheme** as a new **LISP**-like Language
- Goal was to move **Lisp** back toward it's simpler roots and incorporate ideas which had been developed in the PL community since 1960
  - Uses only *static scoping*
  - More uniform in treating *functions as first-class objects* which can be the values of expressions and elements of lists, assigned to variables and passed as parameters
  - Includes the ability to create and manipulate *closures* and *continuations*
    - A *closure* is a data structure that holds an expression and an environment of variable bindings in which it is to be evaluated. Closures are used to represent unevaluated expressions for FPLs with lazy evaluation (Module 04)
    - A *continuation* is a data structure which represents *the rest of a computation*
- Example:

```
(define (fact n) (if (< n 2) 1 (* n (fact (- n 1)))))
```

- **Scheme** has mostly been used as a language for teaching Computer programming concepts where as **Common Lisp** is widely used as a practical language

ML/SML & Haskell

- **ML** (**Meta Language**) is a *strict*, *static-scoped* functional language with a Pascal-like syntax created by Robin Milner et. al. in 1973. Common dialect: **Standard ML**
- First language to include *statically checked polymorphic typing*
  - Uses *type declarations*, but also does *type inferencing* to determine the types of undeclared variables (Modules 05, 07)
  - *Strongly typed* (whereas **Scheme** is essentially *typeless*) and has no type coercions
- Has exception handling, modules for implementing ADTs, GC and a formal semantics
- Example: `fun cube (x : int) = x * x * x;`
- **Haskell** is similar to **ML** (*syntax*, *static scoped*, *strongly typed*, *type inferencing*)
- *Purely functional* – *no variable*, *no assignment statement*, and *no side effect*
- Some key features:
  - Uses *lazy evaluation* (evaluate a subexpression only when needed) (Module 04)
  - Has *list comprehensions*, which allow it to deal with infinite lists
- Example:

```
fib 0 = 1
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
```

Python

- Python supports *λ functions*:
  - one-line mini-functions
  - can be used anywhere a function is required
- *Syntax of λ functions* in Pyhton
  - there is *no parenthesis* around the argument list
  - *no return* keyword
  - function has *no name*
  - can be called through the variable it is assigned to
  - can be use without even assigning it to a variable – just an *in-line function*
- To generalize, a λ function is a function that:
  - takes any number of arguments
  - returns the value of a single expression
  - λ functions cannot contain commands
  - λ functions cannot contain more than one expression
  - λ functions should be kept simple

Example:

```
>>> def f(x):
... return x*2
...
>>> f(3)
6
>>> g = lambda x: x*2
>>> g(3)
6
>>> (lambda x: x*2)(3)
6
>>> def f(n):
...return lambda x: x+n
>>> v = f(3)
>>> v(10)
13
```

- Functional Programming HOWTO
- Functional Programming In Python
- Functional Programming in Python
- Python and functional programming

C++

- C++ supports functional programming through λ's from **C++11** (Module 06)

```
#include <iostream>
#include <functional> // Provides template <class Ret, class... Args> class function<Ret(Args...)>;
using namespace std;

// lambda expressions
auto f =     [](int i) { return i + 3; };
auto sqr =   [](int i) { return i * i; };
auto twice = [](const function<int(int)>& g, int v) { return g(g(v)); };
auto comp =  [](const function<int(int)>& g, const function<int(int)>& h, int v) { return g(h(v)); };

int main() { auto a = 7, b = 5, c = 3; // Type inferred as int
    cout << f(a) << endl;                                // 10
    cout << twice(f, a) << " " << comp(f, f, a) << endl;       // 13 13
    cout << twice(sqr, b) << " " << comp(sqr, sqr, b) << endl; // 625 625
    cout << comp(sqr, f, c) << " " << comp(f, sqr, c) << endl; // 36 12
}
```

- The lambda's in C++ above correspond to the simply typed λ-expressions below:

$$
\begin{aligned}
f &\equiv \lambda(i : Int).\ i + 3 : Int \\
twice &\equiv \lambda(f : Int \rightarrow Int).\ \lambda(v : Int).\ f\ (f\ v) :\ Int \\
sqr &\equiv \lambda(i : Int).\ i * i : Int \\
comp &\equiv \lambda(f : Int \rightarrow Int).\ \lambda(g : Int \rightarrow Int).\ \lambda(v : Int).\ f\ (g\ v) : Int
\end{aligned}
$$

Applications of Functional Programming

- **Lisp** is used for artificial intelligence applications
  - Knowledge representation
  - Machine learning
  - Natural language processing
  - Modeling of speech and vision
- **Embedded Lisp** interpreters add programmability to some systems, such as Emacs
- **Scheme** is used to teach introductory programming at many universities
- FPLs are often used where rapid prototyping is desired
- Pure FPLs like **Haskell** are useful in contexts requiring some degree of program verification

Imperative vis-a-vis Functional Programming Languages

- **Imperative Languages**
  - Efficient execution
  - Complex semantics
  - Complex syntax
  - Concurrency is programmer designed
- **Functional Languages**
  - Inefficient execution
  - Simple semantics
  - Simple syntax
  - Programs can automatically be made concurrent

Haskell

Haskell

Haskell

Installations:

```
sudo add-apt-repository ppa:hvr/ghc
sudo apt-get update
sudo apt-get install ghc-8.0.2
or
sudo apt-get install ghc

Type ghci to start the interactive prompt

user@ubuntu:~$ ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude>
To run a haskell program

ghc -o fac fac.hs
```

Haskell - Basics

Simple Arithmetics

parenthesis rule obeyed

```
75+90
50 * (100 - 4999)
Negative of numbers
5 * (-3)  not 5 * -3
```

Boolean Algebra

```
True && False
False || True
not (True && True)
```

Haskell - Lists

In Haskell, lists are a homogenous data structure
We can use the let keyword to define a name in ghci

```
ghci> let lostNumbers = [4,8,15,16,23,42]
ghci> lostNumbers
[4,8,15,16,23,42]

ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]

ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

Haskell - Lists

Lists comparison

```
ghci> [3,2,1] > [2,1,0]
True
ghci> [3,2,1] > [2,10,100]
True
ghci> [3,4,2] > [3,4]
True
ghci> [3,4,2] == [3,4,2]
True
```

Nested Lists

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b   [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
```

Haskell - Lists


```
ghci> head [5,4,3,2,1]
5
ghci> tail [5,4,3,2,1]
[4,3,2,1]
ghci> last [5,4,3,2,1]
1
ghci> init [5,4,3,2,1]
[5,4,3,2]
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
-----------------------------------------
{Texas ranges}
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

Haskell - Lists


```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
-------------------------------------



Tuples:
[[1,2],[8,11],[4,5]].  and [[1,2],[8,11,5],[4,5]] both allowed
but not [(1,2),(8,11,5),(4,5)]
("Christopher", "Walken", 55)
ghci> let mhc = (('a', 50),('g', 40))
```

Haskell - Maps and Filters

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
-------------------------------
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
-------------------------------
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
```

Haskell - Functions

In Haskell, functions are called by writing the function name, a space and then the parameters, separated by spaces.

```
ghci> max 100 101
ghci> succ 8
ghci> (succ 9) + (max 5 4) + 1
```

```
ghci> bar (bar 3)?
```

Haskell - Functions

```
doubleMe x = x + x
```

```
Prelude> :l func2
[1 of 1] Compiling Main ( func2.hs, interpreted )
Ok, one module loaded.
*Main> doubleUs x y = doubleMe x + doubleMe y
*Main> doubleUs 4 5
18
```

Haskell - Functions

```
doubleMe x = x + x
doubleUs x y = x*2 + y*2
doubleSmallNumber x = if x > 100
                        then x
                        else x*2
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z

describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."
```

Haskell - Functions


```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)


main = print (describeList "pp")
```

Haskell - Functions

Different approaches to define a function

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse xs ++ [x]
```

```
reverse2 xs  = app ([], xs)
       where
       app (ys, [])      = ys
       app (ys, (x:xs)) = app ((x:ys), xs)
```

```
 main = print (reverse2 [1,2,3,4])
 main = print (reverse'[1,2,3,4])
[4,3,2,1]
```

Haskell - lambdas

Lambdas are basically anonymous functions that are used because we need some functions only once.

```
numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15)
(map chain [1..100]))
```

```
------------------------------------------------
map (+3) [1,6,3,2] and map (\x -> x + 3) [1,6,3,2] are equivalent
since both (+3) and (\x -> x + 3) are functions that take a number and
add 3 to it.
------------------------------------------------
Like normal functions, lambdas can take any number of parameters:
ghci> zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]
```

Python - Lambdas

```
def muliplyBy (n):
  return lambda x: x*n
double = multiplyBy(2)
---------------------------------------------------
sequences = [10,2,8,7,5,4,3,11,0, 1]
filtered_result = filter (lambda x: x > 4, sequences)
print(list(filtered_result))

[10, 8, 7, 5, 11]
-----------------------------------------------------
sequences = [10,2,8,7,5,4,3,11,0, 1]
filtered_result = map (lambda x: x*x, sequences)
print(list(filtered_result))

 [100, 4, 64, 49, 25, 16, 9, 121, 0, 1]
```

Haskell - Recursion

```
maximum' :: (Ord a) => [a] -> a
    maximum' [] = error "maximum of empty list"
    maximum' [x] = x
    maximum' (x:xs) = max x (maximum' xs)

    replicate' :: (Num i, Ord i) => i -> a -> [a]
    replicate' n x
        | n <= 0     = []
        | otherwise = x:replicate' (n-1) x
```

Haskell - Function Examples


```
Quicksort
----------------------------------------
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (pivot:xs) =
     quicksort [x | x <- xs, x < pivot] ++
     [pivot] ++
     quicksort [x | x <- xs, x >= pivot]

Finding a maximum element in a binary tree
-----------------------------------------
data Tree a = Leaf | Node a (Tree a) (Tree a)
maxElement :: (Ord a) => Tree a -> Maybe a
maxElement Leaf = Nothing
maxElement (Node v l r) = maximum [Just v, maxElement l, maxElement r]
```

Haskell - Types and Classes

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool

Types of functions
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
factorial :: Integer -> Integer
factorial n = product [1..n]
```

Haskell - Statically Typed

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."
error if string is 2
--------------------------------------------
haskell_programs/func.hs:13:54:
    No instance for (Num [Char]) arising from the literal ‘2’
    In the expression: 2
    In a case alternative: xs -> 2
    In the second argument of ‘(++)’, namely
      ‘case xs of {
         [] -> "empty."
         [x] -> "a singleton list."
         xs -> 2 }’
Failed, modules loaded: none.
```

Python – Strong typing, dynamic typing


```
p = 3
if p>2:
    p = "two"
    print("two")
else:
    p = 1
    print (p)

---------------------------

p = 3
if p>2:
    print("two" + p)
else:
    print (p)
```

Haskell - Evaluation Strategies

```
{Haskell
add x y = x + x

#Python
def add( x , y ) :
return x + x
```

Haskell vs Python

Lazy Evaluation in Haskell, Eager Evaluation in Python

```
{Haskell
add 4 5
Result : 8
add 10 (89/0)
Result : 20

#Python
print( add(4 , 5) )
Result : 8
print( add(10 , (89/0) ) )
Result : Traceback (most recent call last):
File \test.py", line 7, in <module>
print( add(10 , (89/0) ) )
ZeroDivisionError: division by zero
```

Haskell vs ML

Eager Evaluation in ML vs Lazy Evaluation in Haskell

> (square (square 2)) * (square (square 2))
>((square 2) * (square 2)) * ((square 2) * (square 2))
> ((2 * 2) * (2 * 2)) * ((2 * 2) * (2 * 2))
> (4 * 4) * (4 * 4)
> 16 * 16
> 256

> (square (square 2)) * (square (square 2))
> ((square 2) * (square 2)) * (square (square 2))
> ((2 * 2) * (square 2)) * (square (square 2))
> (4 * (square 2)) * (square (square 2))
> (4 * (2 * 2)) * (square (square 2))
> (4 * 4) * (square (square 2))
> 16 * (square (square 2))
> ... > 256

Haskell - Infinite Lists

`take 5 [ 4 .. ]` which gives us the first 5 elements of `[ 4 .. ]` which are `[4,5,6,7,8]`.

```
addDigits :: Int -> Int
addDigits n = foldl (\acc char -> acc + digitToInt char^len ) 0 $ show n
where len = length (show n)

isArmstrong :: Int -> Bool
isArmstrong n = addDigits n == n


armstrongNumbers :: Int -> [Int]
armstrongNumbers n = take n $ filter isArmstrong [100..]
```

Haskell - Input / Output


```
main = putStrLn "hello, world"
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
------------------------------------------------
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you can login!")
```

Lisp

Lisp

Lisp - Basics

```
sudo apt-get install sbcl
type sbcl
```

```
* (+ 3 2)
(setq x 10)
(print (type-of x))
(INTEGER 0 4611686018427387903)
```

Lisp - Arrays


```
(write (setf my-array (make-array '(2))))
(terpri)
(setf (aref my-array 0) 25)
(setf (aref my-array 1) 23)

(write my-array)
----------------------------------
(setf x (make-array '(3 3)
:initial-contents '((0 1 2 ) (3 4 5) (6 7 8))))
(write x)
----------------------------------
(setq a (make-array '(4 3)))
(dotimes (i 4)
(dotimes (j 3)
(setf (aref a i j) (list i 'x j '= (* i j))))
-----------------------------------
```

Lisp - Lists


```
(write (car '(a b c d e f)))
(terpri)
(write (cdr '(a b c d e f)))
(terpri)
(write (cons 'a '(b c)))
(terpri)
(write (list 'a '(b c) '(e f)))
(terpri)
(write (append '(b c) '(e f) '(p q) '() '(g)))
(terpri)
(write (last '(a b c d (e f))))
(terpri)
(write (reverse '(a b c d (e f))))
```

Lisp - Functions and Lambdas


```
(defun name (parameter-list)
"Optional documentation string."
body)
--------------------------------
(defun averagenum (n1 n2 n3 n4)
(/ ( + n1 n2 n3 n4) 4))
(write(averagenum 10 20 30 40))
---------------------------------
(defun area-circle(rad)
"Calculates area of a circle with given radius"
(terpri)
(format t "Radius: ~5f" rad)
(format t "~%Area: ~10f" (* 3.141592 rad rad)))
(area-circle 10)
```

Lisp - Lambdas and Functions

```
Finding if a list is Palindrome (2 ways)
------------------------------
(defun P (list)
  (loop
     :for left  :on list
     :for right :in (reversed-spine list)
     :until (or (eq left right) (eq (cdr left) right))
     :unless (eql (car left) (car right)) :do (return nil)
     :finally (return t)))
(defun P (list)
  (loop
     :with data = (coerce list 'vector)
     :for i :from 0
     :for j :from (1- (length data)) :by -1
     :while (< i j)
     :always  (eql (aref data i) (aref data j))))
```

Lisp - Lambdas and Functions


```
Finding duplicates in a list
------------------------------
(defun dupli (list)
  (mapcan (lambda (item) (list item item)) list))

(defun dupli (list)
  (loop
     :for item :in list
     :collect item
     :collect item))
```

Lisp - Lambdas and Functions

```
Retrieve a given number of randomly selected elements from a list.
--------------------------------
(defun remove-at (list index)
  (cond
    ((< index 1) (error "Invalid index"))
    ((= index 1) (rest list))
    (t      (cons (first list) (remove-at (rest list) (1- index))))))

(defun rnd-select (list count)
  (if (zerop count)
    '()
    (let ((i (random (length list))))
     (cons (elt list i) (rnd-select (remove-at list (1+ i)) (1- count))))))
```

Lisp - Input / Output

```
; the function AreaOfCircle
; calculates area of a circle
; when the radius is input from keyboard
(defun AreaOfCircle()
(terpri)
(princ "Enter Radius: ")
(setq radius (read))
(setq area (* 3.1416 radius radius))
(princ "Area: ")
(write area))
(AreaOfCircle)
```

Scheme

Scheme - Basics

```
sudo apt-get install mit-scheme
type mit-scheme
----------------
(+ 3 5)
(fac 6)
(append '(a b c) '(1 2 3 4))
(- 10 3) → 7
(* 2 3) → 6
(/ 29 3) → 29/3
(/ 9 6) → 3/2

(quotient 7 3) → 2
(modulo 7 3) → 1
(sqrt 8) → 2.8284271247461903
```

Scheme - Lists and Operations

```
(cons '1 '(2 3 4))
car -- returns the first member of a list or dotted pair.

         (car '(123 245 564 898))               is   123
         (car '(this (is no) more difficult))    is  this

cdr -- returns the list without its first item
         (cdr '(it rains every day))  is  (rains every day)
         (cdr (cdr '(a b c d e f)))   is   (c d e f)
         (car (cdr '(a b c d e f)))   is   b

(length '(1 3 5 9 11))                   is  5

(reverse '(1 3 5 9 11)) is  (11 9 5 3 1)

(append '(1 3 5)  '(9 11))  is  (1 3 5 9 11)
```

Scheme - Lists and Operations


```
(let ((list1 '(a b c)) (list2 '(d e f)))
  (cons (cons (car list1)
              (car list2))
        (cons (car (cdr list1))
              (car (cdr list2)))))

(let ([a 4] [b -3])
  (let ([a-squared (* a a)]
        [b-squared (* b b)])
    (+ a-squared b-squared)))

(let ([x 1])
  (let ([x (+ x 1)])
    (+ x x)))
```

Scheme - Dynamically typed


```
different return types allowed, due to dynamically typed

(if (> 3 2) 'yes '3)
```

Scheme - Lambdas and Functions

```
(define pi 3.14)

((lambda (x) (+ x x)) (* 3 4))  - Anonymous


(define square (lambda (x)  (* x x)))  - bindings

Factorial function

(define fac
           (lambda (n)
                   (if (= n 0)
                          1
                          (* n (fac (- n 1))))))
```

Scheme - Lambdas and Functions

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))              =>  45

  ---------------------------------
(define reverse-subtract
  (lambda (x y)
    (- y x)))
(reverse-subtract 7 10)
```

Scheme - Lambdas and Functions


```
Fibonacci Series
--------------------------------
(define fib
    (lambda (n)
         (if (= n 0)
                0
                (if (= n 1)
                          1
                          (+ (fib (- n 1)) (fib (- n 2)))))))
```

Scheme - Lambdas and Functions

```
GCD
------------------------------
(define gcd
    (lambda (a b)
        (if (= a b)
                a
            (if (> a b)
                 (gcd (- a b) b)
                 (gcd a (- b a))))))
```

Scheme - Input / Output

```
(+ 3 (read))

(display (+ 3 (read)))


(define prompt-read (lambda (Prompt)
   (display Prompt)
   (read)))
```