**Chapter 10 Functional Programming Languages**

Key: **express your solution as functions**

10.1 Background - Based on and inspired by mathematical functions.

Mathematically

mapping

Function: value (from Domain) 🡪 value (belong to Range)

This mapping can be expressed:

1. Formula or existing (or known) functions

e.g., f(x) = x\*2+3

y = (sin(x))2

2. Table,

e.g.,

|  |  |
| --- | --- |
| X | Y |
| 1 | 2 |
| 2 | 4 |
| 3 | 10 |
| 4 | 99 |
| 5 | 66 |

Characteristics of functional languages:

1. Programs are sets of mathematical relationships between objects, express as functions
   * explicit memory management is not important, but also not illegal.
2. Evaluation order is controlled by recursion and conditional expression.

Ex., sum function.

f(n) = n + f(n-1) if n>0

0 if n is 0

Express f(n) in terms of f(n-1)+n

1. minimum side effects – natural extension of property 1.  
   Side effects: (*unexpected?)* change of variable values [that affect the outside world]

Ex., change of heap (pointer) var with aliases – created with shallow access of a binary tree

1. *Pure* functional languages have no mutable variable, no assignment statements, only return values, thus, guarantee of no side effect.
   * Variables in lisp and scheme are mutable.
   * Variables in prolog are immutable, although prolog is not an FL
   * In contrast to traditional, procedural languages which specify data movement and computation based on von Neumann architecture
2. Functions are [first-class objects](https://en.wikipedia.org/wiki/First-class_function)

Some typical functional features (may or may not be in a traditional imperative languages)

* First class (or higher order) functions
* powerful and flexible list facilities – representation and methods (ex., build-in list function)
* structured function returns
* garbage collection

Why FL?

* A different way of thinking and problem solving.
* True recursive thinking by using short, self-contained function
* Implicit polymorphism

Related concepts:

1. lambda (or anonymous, nameless) functions – the inspiration for functional programming (first proposed by [Alonzo Church](http://en.wikipedia.org/wiki/Alonzo_Church))

* + Functions need not have names   
    Ex.,

λ (y) y\*y

is the same as

square(y) = y\*y

so, instead of invoking f explicitly, as in square(2.5), we could call the function using it form directly, without a name, as in

( λ (y) y\*y ) (2.5) (🡪 6.25)

* Nameless (or anonymous) functions, thus, are also called Lambda functions, or lambda abstraction.

Benefit: efficient description, more general form, good for one-time use, efficient code segment,

1. Higher-Order Functions (HOF) – functions are first class object

(a) returns a function

(b) accepts functions as its argument (the actual function, not a call)

(c) Apply-to-all ( map function)

Example: Apply a function (accepted as the 1st parameter) to the rest of arguments

(map procedure list-arg1 list-arg2 …) 🡪 (procedure list-arg1 list-arg2 ..)

Ex.,

(map – ‘(1 2 3 4 5)) ; negate every number in class

(map (lambda (x) (\* x 1.5)) ‘(1 2 3 4 5)) ; change very number using a lambda function

Why should I learn FL?

<https://www.47deg.com/blog/five-reason-to-learn-functional-programming/>

<https://thealmarty.com/2021/03/10/why-you-should-learn-functional-programming/>

10.2 Functional Programming Concepts

Objective: mimic mathematical functions as much as possible.

* Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are attempts to realize Church's lambda calculus in practical form as a programming language
* The key idea: do everything by composing functions.
  + algorithm is a sequence of, possibly nested and/or recursive, functional calls.
  + variables, if any, are local.
  + no mutable state 🡪 no side effects.
  + Limited use of assignment statements.
  + Repetition is done by recursion.
  + Loop is optional.

Four basic components of most functional languages:

* 1. Primitive functions/operators - carry out the basic operations.
  2. Function forms - used to construct complex function from more primitive ones.
  3. Function application operation (i.e., function call)
  4. Structure for representing data - most commonly used is (linked) list, which often contain many levels of sublists: (a b (c d) (e))

How do we get anything done in a functional language?

Function calls

How do operations get repeated?

Nested and Recursion function calls

* In general, we can get the effect of a series of assignments  
   x := 0 ... // f1  
   y := expr1 ... // expr1 contains x, replace with f2

z := expr2 ... // expr2 contains y and/or x, represented by f3

With

f3(f2(f1(0)))

where each f expects the value of x as an argument, f1 returns expr1, and f2 returns expr2, etc

* Recursion even does a nifty job of replacing looping  
  int x = 0, i = 1, j = 100;  
  while (i < j)

{  
 x += i\*j;

i++;   
 j--;

} end while  
return x

(define SIP

(lambda (x i j)

(if (< i j) (SIP (+ x (\* i j)) (+ 1 i) (- j 1)) x)))

// (if cond true-result false-result)

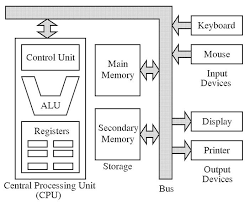
// exp has the same form as function: both use prefix notation

* Recursion is NOT direct, mechanical replacement/substitution of iteration.

🡪 It is best when you think recursively.



Can programming be liberated from von Neumann Style? (John Backus, 1977 Turing speech)



Lisp – 1st functional language:

* Designed and implemented by [John McCarthy](http://www-formal.stanford.edu/jmc/index.html) in 1958-1959, as a language for general symbolic processing to study *computability, among other things*. (very good intro [here](https://www.youtube.com/watch?v=92WHN-pAFCs))
* Lisp stands for LISt Processing (or Lots of Insipid and Stupid Parenthesis)
* As a whole, Lisp is not a pure functional language (i.e., most dialects have lots of imperative language features)
* Use two basic types of data objects:
  + Atom, ex., A
  + List, ex., (A B C D), *(define double (lambda (n) 2n))*
* Just about everything in lisp is represented in list form, why? (see picture on board)
  + Study of computability, ex halting problem – does a program halt on a particular input?
    - machine (or model) itself can be studied just like any other input, i.e., A lisp function is represented the same way as its data. Both are referred to as s-expression (symbolic expression)

Ex., dfa.rtk

* + A universal function, called eval, can evaluate any other function,
* Many variations:
  + Pure (original) Lisp
  + Interlisp, MacLisp, Emacs Lisp
  + Common Lisp
  + Scheme – a simpler, elegant dialect

Other functional languages:

* ML
* Miranda
* Haskell – leading language in research of functional languages
* FP

10.3 Overview of scheme

* Scheme was introduced in 1975. It was a dialect of lisp that emphasizes conceptual elegance and simplicity.
* Designed to better match lambda calculus
* It’s multi-paradigm language, best known for its support of functional programming.
* ANSI/IEEE standard was established in 1991.
* [More here](http://www.engin.umd.umich.edu/CIS/course.des/cis400/scheme/scheme.html)

Important characteristics:

* A small language (minimalist approach)
* Can be compiled or interpreted
* A versatile language, can be used for many applications
* Highly portable

See SchemeBasic.docx for basic intro