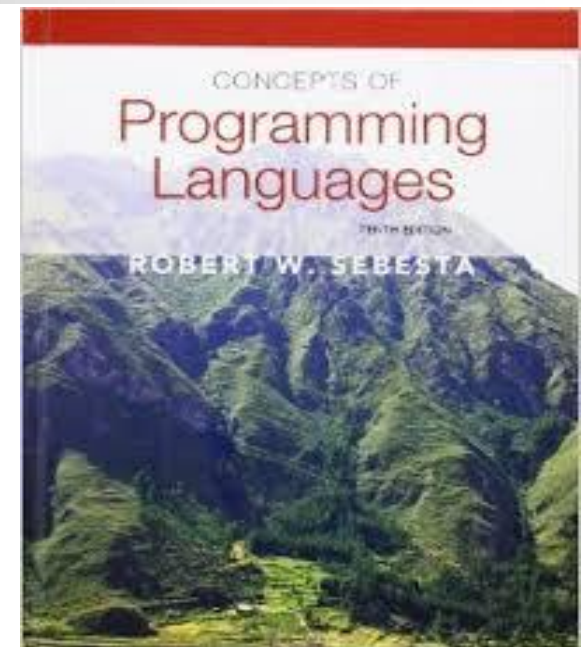


Chapter 15 :Functional Programming Languages

Saad Alshahrani

Email: salshahrani@ut.edu.sa

Department of Computer Science
Duba University College





Covered Topics

- Introduction
- Mathematical Functions
- Fundamentals of Functional Programming Languages
- The First Functional Programming Language: LISP
- Introduction to Scheme
- COMMON LISP
- ML
- Haskell
- Applications of Functional Languages
- Comparison of Functional and Imperative Languages



Introduction

- 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



Mathematical Functions

- A mathematical function is a *mapping* of members of one set, called the *domain set*, to another set, called the *range set*
- A *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

$\lambda (x) \quad x * x * x$

for the function $\text{cube} (x) = x * x * x$



Lambda Expressions

- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

e.g., $(\lambda (x) \ x \ * \ x \ * \ x) (2)$

which evaluates to 8



Functional Forms

- A higher-order function, or *functional form*, is one that either takes functions as parameters or yields a function as its result, or both



Function Composition

- A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

Form: $h \equiv f \circ g$

which means $h(x) \equiv f(g(x))$

For $f(x) \equiv x + 2$ and $g(x) \equiv 3 * x$,

$h \equiv f \circ g$ yields $(3 * x) + 2$



Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

Form: α

For $h(x) \equiv x * x$

$\alpha(h, (2, 3, 4))$ yields $(4, 9, 16)$



Fundamentals of Functional Programming Languages

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible
- The basic process of computation is fundamentally different in a FPL than in an imperative language
 - In an imperative language, operations are done and the results are stored in variables for later use
 - Management of variables is a constant concern and source of complexity for imperative programming
- In an FPL, variables are not necessary, as is the case in mathematics



Fundamentals of Functional Programming Languages - continued

- *Referential Transparency* - In an FPL, the evaluation of a function always produces the same result given the same parameters
- *Tail Recursion* – Writing recursive functions that can be automatically converted to iteration



LISP Data Types and Structures

- *Data object types*: originally only atoms and lists
- *List form*: parenthesized collections of sublists and/or atoms
e.g., (A B (C D) E)
- Originally, LISP was a typeless language
- LISP lists are stored internally as single-linked lists



LISP Interpretation

- Lambda notation is used to specify functions and function definitions. Function applications and data have the same form.

e.g., If the list `(A B C)` is interpreted as data it is a simple list of three atoms, `A`, `B`, and `C`

If it is interpreted as a function application, it means that the function named `A` is applied to the two parameters, `B` and `C`

- The first LISP interpreter appeared only as a demonstration of the universality of the computational capabilities of the notation



Origins of Scheme

- A mid-1970s dialect of LISP, designed to be a cleaner, more modern, and simpler version than the contemporary dialects of LISP
- Uses only static scoping
- Functions are first-class entities
 - They can be the values of expressions and elements of lists
 - They can be assigned to variables and passed as parameters



Evaluation

- Parameters are evaluated, in no particular order
- The values of the parameters are substituted into the function body
- The function body is evaluated
- The value of the last expression in the body is the value of the function



Primitive Functions

- Arithmetic: `+`, `-`, `*`, `/`, `ABS`, `SQRT`, `REMAINDER`, `MIN`, `MAX`
e.g., `(+ 5 2)` yields 7
- `QUOTE` - takes one parameter; returns the parameter without evaluation
 - `QUOTE` is required because the Scheme interpreter, named `EVAL`, always evaluates parameters to function applications before applying the function. `QUOTE` is used to avoid parameter evaluation when it is not appropriate
 - `QUOTE` can be abbreviated with the apostrophe prefix operator
`' (A B)` is equivalent to `(QUOTE (A B))`



Function Definition: LAMBDA

- Lambda Expressions

- Form is based on λ notation

e.g., $(\text{LAMBDA } (x) \ (* \ x \ x))$

x is called a bound variable

- Lambda expressions can be applied

e.g., $((\text{LAMBDA } (x) \ (* \ x \ x)) \ 7)$



Special Form Function: `DEFINE`

- A Function for Constructing Functions `DEFINE` - Two forms:
 1. To bind a symbol to an expression
e.g., `(DEFINE pi 3.141593)`
Example use: `(DEFINE two_pi (* 2 pi))`
 2. To bind names to lambda expressions
e.g., `(DEFINE (square x) (* x x))`
Example use: `(square 5)`
- The evaluation process for `DEFINE` is different! The first parameter is never evaluated. The second parameter is evaluated and bound to the first parameter.



Output Functions

- `(DISPLAY expression)`
- `(NEWLINE)`



Numeric Predicate Functions

- $\#T$ is true and $\#F$ is false (sometimes $()$ is used for false)
- $=$, $<>$, $>$, $<$, $>=$, $<=$
- `EVEN?`, `ODD?`, `ZERO?`, `NEGATIVE?`



Control Flow: IF

- Selection- the special form, IF

`(IF predicate then_exp
else_exp)`

e.g.,

```
(IF (<> count 0)
    (/ sum count)
    0)
```



Control Flow: COND

- Multiple Selection - the special form, COND

General form:

(COND

 (*predicate_1* *expr* { *expr* })

 (*predicate_1* *expr* { *expr* })

 ...

 (*predicate_1* *expr* { *expr* })

 (ELSE *expr* { *expr* }))

- Returns the value of the last expression in the first clause whose predicate is true.



Example of COND

```
(DEFINE (compare x y)
  (COND
    ((> x y) "x is greater than y")
    ((< x y) "y is greater than x")
    (ELSE "x and y are equal")
  )
)
```



List Functions: CONS and LIST

- CONS takes two parameters, the first of which can be either an atom or a list and the second of which is a list; returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result
e.g., (CONS 'A ' (B C)) returns (A B C)
- LIST takes any number of parameters; returns a list with the parameters as elements



List Functions: CAR and CDR

- CAR takes a list parameter; returns the first element of that list

e.g., (CAR ' (A B C)) yields A

(CAR ' ((A B) C D)) yields (A B)

- CDR takes a list parameter; returns the list after removing its first element

e.g., (CDR ' (A B C)) yields (B C)

(CDR ' ((A B) C D)) yields (C D)



Predicate Function: EQ?

- EQ? takes two symbolic parameters; it returns #T if both parameters are atoms and the two are the same; otherwise #F
e.g., (EQ? 'A 'A) yields #T
(EQ? 'A 'B) yields #F
- Note that if EQ? is called with list parameters, the result is not reliable
- Also EQ? does not work for numeric atoms



Predicate Functions: LIST? and NULL?

- LIST? takes one parameter; it returns #T if the parameter is a list; otherwise #F
- NULL? takes one parameter; it returns #T if the parameter is the empty list; otherwise #F
 - Note that NULL? returns #T if the parameter is ()



Example Scheme Function: `member`

- `member` takes an atom and a simple list; returns `#T` if the atom is in the list; `#F` otherwise

```
DEFINE (member atm lis)
  (COND
    ((NULL? lis) #F)
    ((EQ? atm (CAR lis)) #T)
    ((ELSE (member atm (CDR lis)))
     ) )
```



Example Scheme Function: `equalsimp`

- `equalsimp` takes two simple lists as parameters; returns `#T` if the two simple lists are equal; `#F` otherwise

```
(DEFINE (equalsimp lis1 lis2)
  (COND
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((EQ? (CAR lis1) (CAR lis2))
      (equalsimp (CDR lis1) (CDR lis2)))
    (ELSE #F)
  ))
```



Example Scheme Function: `equal`

- `equal` takes two general lists as parameters; returns `#T` if the two lists are equal; `#F` otherwise

```
(DEFINE (equal lis1 lis2)
  (COND
    ((NOT (LIST? lis1)) (EQ? lis1 lis2))
    ((NOT (LIST? lis2)) #F)
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((equal (CAR lis1) (CAR lis2))
     (equal (CDR lis1) (CDR lis2)))
    (ELSE #F)
  ))
```



Example Scheme Function: `append`

- `append` takes two lists as parameters; returns the first parameter list with the elements of the second parameter list appended at the end

```
(DEFINE (append lis1 lis2)
  (COND
    ((NULL? lis1) lis2)
    (ELSE (CONS (CAR lis1)
                  (append (CDR lis1) lis2))))
  ) )
```



Example Scheme Function: LET

- General form:

```
(LET (
  (name_1 expression_1)
  (name_2 expression_2)
  ...
  (name_n expression_n) )
  body
)
```

- Evaluate all expressions, then bind the values to the names; evaluate the body



LET Example

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
      (/ (SQRT (- (* b b) (* 4 a c))) (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
    (DISPLAY (+ minus_b_over_2a root_part_over_2a))
    (NEWLINE)
    (DISPLAY (- minus_b_over_2a root_part_over_2a))
  ))
```




Tail Recursion in Scheme

- Definition: A function is *tail recursive* if its recursive call is the last operation in the function
- A tail recursive function can be automatically converted by a compiler to use iteration, making it faster
- Scheme language definition requires that Scheme language systems convert all tail recursive functions to use iteration



Tail Recursion in Scheme - continued

- Example of rewriting a function to make it tail recursive, using helper a function

Original:

```
(DEFINE (factorial n)
  (IF (= n 0)
      1
      (* n (factorial (- n 1)))
  ))
```

Tail recursive:

```
(DEFINE (facthelper n factpartial)
  (IF (= n 0)
      factpartial
      facthelper((- n 1) (* n factpartial)))
  ))

(DEFINE (factorial n)
  (facthelper n 1))
```



Scheme Functional Forms

■ Composition

- The previous examples have used it
- `(CDR (CDR ' (A B C)))` returns `(C)`

■ Apply to All - one form in Scheme is `mapcar`

- Applies the given function to all elements of the given list;

```
(DEFINE (mapcar fun lis)
  (COND
    ((NULL? lis) ())
    (ELSE (CONS (fun (CAR lis))
                 (mapcar fun (CDR lis)))))
  ))
```



Functions That Build Code

- It is possible in Scheme to define a function that builds Scheme code and requests its interpretation
- This is possible because the interpreter is a user-available function, `EVAL`



Adding a List of Numbers

```
((DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (EVAL (CONS '+ lis))))
))
```

- The parameter is a list of numbers to be added; `adder` inserts a `+` operator and evaluates the resulting list
 - Use `CONS` to insert the atom `+` into the list of numbers.
 - Be sure that `+` is quoted to prevent evaluation
 - Submit the new list to `EVAL` for evaluation



COMMON LISP

- A combination of many of the features of the popular dialects of LISP around in the early 1980s
- A large and complex language--the opposite of Scheme
- Features include:
 - records
 - arrays
 - complex numbers
 - character strings
 - powerful I/O capabilities
 - packages with access control
 - iterative control statements

- A static-scoped functional language with syntax that is closer to Pascal than to LISP
- Uses type declarations, but also does *type inferencing* to determine the types of undeclared variables
- It is strongly typed (whereas Scheme is essentially typeless) and has no type coercions
- Includes exception handling and a module facility for implementing abstract data types
- Includes lists and list operations



ML Specifics

- Function declaration form:

`fun name (parameters) = body;`

e.g., `fun cube (x : int) = x * x * x;`

- The type could be attached to return value, as in

`fun cube (x) : int = x * x * x;`

- With no type specified, it would default to

int (the default for numeric values)

- User-defined overloaded functions are not allowed, so if we wanted a `cube` function for real parameters, it would need to have a different name

- There are no type coercions in ML



ML Specifics (continued)

- ML selection

```
if expression then then_expression  
  else else_expression
```

where the first expression must evaluate to a Boolean value

- Pattern matching is used to allow a function to operate on different parameter forms

```
fun fact(0) = 1  
|      fact(n : int) : int =  
                                n * fact(n - 1)
```



ML Specifics (continued)

■ Lists

Literal lists are specified in brackets

`[3, 5, 7]`

`[]` is the empty list

`CONS` is the binary infix operator, `::`

`4 :: [3, 5, 7]`, which evaluates to `[4, 3, 5, 7]`

`CAR` is the unary operator `hd`

`CDR` is the unary operator `tl`

```
fun length([]) = 0
```

```
| length(h :: t) = 1 + length(t);
```

```
fun append([], lis2) = lis2
```

```
| append(h :: t, lis2) = h :: append(t, lis2);
```



ML Specifics (continued)

- The `val` statement binds a name to a value (similar to `DEFINE` in Scheme)

```
val distance = time * speed;
```

- As is the case with `DEFINE`, `val` is nothing like an assignment statement in an imperative language



Haskell

- Similar to ML (syntax, static scoped, strongly typed, type inferencing, pattern matching)
- Different from ML (and most other functional languages) in that it is *purely* functional (e.g., no variables, no assignment statements, and no side effects of any kind)

Syntax differences from ML

```
fact 0 = 1
```

```
fact n = n * fact (n - 1)
```

```
fib 0 = 1
```

```
fib 1 = 1
```

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



Function Definitions with Different Parameter Ranges

```
fact n
|  n == 0 = 1
|  n > 0 = n * fact (n - 1)
```

```
sub n
|  n < 10      = 0
|  n > 100     = 2
|  otherwise   = 1
```

```
square x = x * x
```

- Works for any numeric type of x



Lists

- List notation: Put elements in brackets
e.g., `directions = ["north",
"south", "east", "west"]`
- Length: `#`
e.g., `#directions` is 4
- Arithmetic series with the `..` operator
e.g., `[2, 4..10]` is `[2, 4, 6, 8, 10]`
- Catenation is with `++`
e.g., `[1, 3] ++ [5, 7]` results in `[1, 3, 5, 7]`
- `CONS`, `CAR`, `CDR` via the colon operator (as in Prolog)
e.g., `1 : [3, 5, 7]` results in `[1, 3, 5, 7]`



Factorial Revisited

`product [] = 1`

`product (a:x) = a * product x`

`fact n = product [1..n]`



List Comprehension

- Set notation
- List of the squares of the first 20 positive integers: $[n * n \mid n \leftarrow [1..20]]$
- All of the factors of its given parameter:
$$\text{factors } n = [i \mid i \leftarrow [1..n \text{ div } 2],$$
$$n \text{ mod } i == 0]$$



Quicksort

```
sort [] = []
```

```
sort (a:x) =
```

```
    sort [b | b ← x; b ≤ a] ++
```

```
    [a] ++
```

```
    sort [b | b ← x; b > a]
```



Lazy Evaluation

- A language is *strict* if it requires all actual parameters to be fully evaluated
- A language is *nonstrict* if it does not have the strict requirement
- Nonstrict languages are more efficient and allow some interesting capabilities – *infinite lists*
- Lazy evaluation - Only compute those values that are necessary
- Positive numbers

```
positives = [0..]
```

- Determining if 16 is a square number

```
member [] b = False
```

```
member (a:x) b = (a == b) || member x b
```

```
squares = [n * n | n <- [0..]]
```

```
member squares 16
```



Member Revisited

- The member function could be written as:

```
member [] b = False
```

```
member (a:x) b = (a == b) || member x b
```

- However, this would only work if the parameter to squares was a perfect square; if not, it will keep generating them forever. The following version will always work:

```
member2 (m:x) n
```

```
    | m < n = member2 x n
```

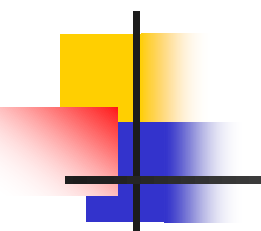
```
    | m == n = True
```

```
    | otherwise = False
```



Applications of Functional Languages

- APL is used for throw-away programs
- LISP is used for artificial intelligence
 - Knowledge representation
 - Machine learning
 - Natural language processing
 - Modeling of speech and vision
- Scheme is used to teach introductory programming at some universities



Comparing Functional and Imperative Languages

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



Summary

- Functional programming languages use function application, conditional expressions, recursion, and functional forms to control program execution instead of imperative features such as variables and assignments
- LISP began as a purely functional language and later included imperative features
- Scheme is a relatively simple dialect of LISP that uses static scoping exclusively
- COMMON LISP is a large LISP-based language
- ML is a static-scoped and strongly typed functional language which includes type inference, exception handling, and a variety of data structures and abstract data types
- Haskell is a lazy functional language supporting infinite lists and set comprehension.
- Purely functional languages have advantages over imperative alternatives, but their lower efficiency on existing machine architectures has prevented them from enjoying widespread use