National University
Of Computer and Emerging Sciences

Theory of Programming Languages

Functional Programming Languages

Sajid Anwer

Department of Computer Science,
FAST-NUCES, CFD Campus

National University
Of Computer and Emerging Sciences

Chapter Outline

- Introduction
- Mathematical Functions
- Fundamentals of Functional Programming Languages
  - » The First Functional Programming Language: LISP
  - » Introduction to Scheme
  - » Common LISP
  - » ML
  - » Haskell
  - » F#
- Support for Functional Programming in Primarily Imperative 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

National University Of Computer and Emerging Sciences

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

  `λ(x) x * x * x`

  for the function `cube(x) = x * x * x`

Mathematical Functions – Lambda expressions

- Lambda expressions describe *nameless functions.*

- Lambda expressions can be applied to parameter(s) by placing the parameter(s) after the expression
  e.g., `(λ(x) x * x * x)(2)`
  which evaluates to `8`

National University Of Computer and Emerging Sciences

Mathematical Functions – Functional Form

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

- 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, are called *functional composition.*

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 $(3 * x) + 2$

Mathematical Functions – 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) ≡ x * x`

  α`(h, (2, 3, 4))` yields `(4, 9, 16)`

Fundamentals of FPL

- 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

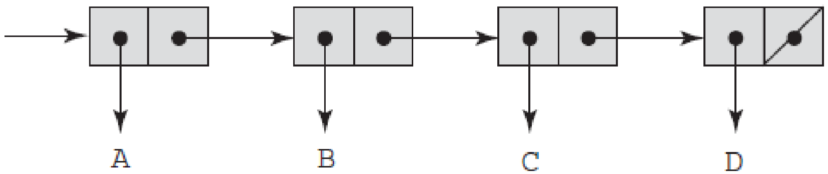
- *Referential Transparency* - In an FPL, the evaluation of a function always produces the same result given the same parameters

LISP Data Types and Structures

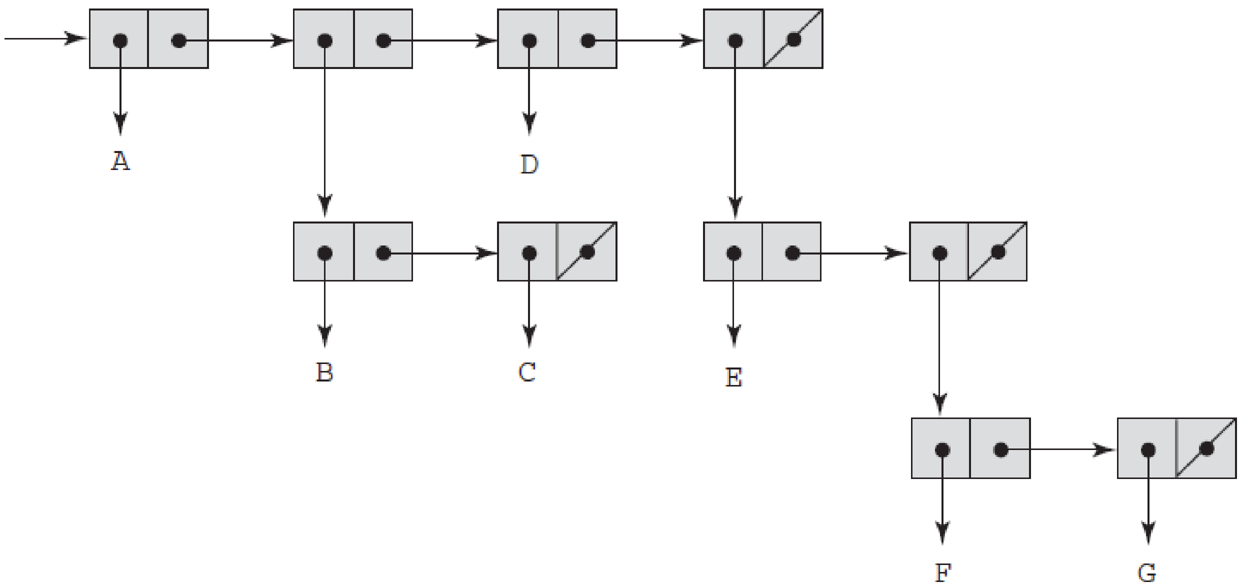
- *Data object types*: originally only atoms and lists
- *List form*: parenthesized collections of sublists and/or atoms

  e.g., `(A B (C D) E)`
- Originally, LISP was a typeless language
- LISP lists are stored internally as single-linked lists

LISP Interpretation



(A B C D)



(A (B C) D (E (F G)))

LISP Interpretation

- Lambda notation is used to specify functions and function definitions. Function applications and data 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`
  - » 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, passed as parameters, and returned from functions

The Scheme Interpreter

- In interactive mode, the Scheme interpreter is an infinite read-evaluate-print loop (REPL)
  - » This form of interpreter is also used by Python and Ruby
- Expressions are interpreted by the function `EVAL`
- Literals evaluate to themselves

Primitive Functions

- 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 Arithmetic Functions: `+`, `-`, `*`, `/`, `ABS`, `SQRT`, `REMAINDER`, `MIN`, `MAX`

Expression	*Value*
`42`	`42`
`(* 3 7)`	`21`
`(+ 5 7 8)`	`20`
`(- 5 6)`	`-1`
`(- 15 7 2)`	`6`
`(- 24 (* 4 3))`	`12`

`LAMBDA` Expressions

- Lambda Expressions
  - Form is based on λ notation

  e.g., `(LAMBDA (x) (* x x)`

  `x` is called a bound variable

- Lambda expressions can be applied to parameters

  e.g., `((LAMBDA (x) (* x x)) 7)`

- `LAMBDA` expressions can have any number of parameters

  `(LAMBDA (a b x) (+ (* a x x) (* b x)))`

Special Form Function: `DEFINE`

- `DEFINE` - Two forms:
  1. To bind a symbol to an expression

     e.g., `(DEFINE pi 3.141593)`

     Example use: `(DEFINE two_pi (* 2 pi))`

     These symbols are not variables – they are like the names bound by Java's **`final`** declarations

  2. To bind names to lambda expressions (`LAMBDA` is implicit)

     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.

Special Form Function: `DEFINE`

- Usually not needed, why?
  - » Because the interpreter always displays the result of a function evaluated by *EVAL*.

- Scheme has *`PRINTF`*, which is similar to the *`printf`* function of C

- Note: explicit input and output are not part of the pure functional programming model, why?
  - » Because input *operations change the state* of the program and *output operations are side effects.*

National University Of Computer and Emerging Sciences

Numeric Predicate Functions

- #T (or #t) is true and #F (or #f) is false (sometimes () is used for false)

- =, <>, >, <, >=, <=
- EVEN?, ODD?, ZERO?, NEGATIVE?

- The NOT function *inverts the logic* of a Boolean expression

- A *nonempty* list returns as true and *empty* as false.

Control Flow

- Selection- the special form, IF

  `(IF` predicate then_exp else_exp`)`

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

- Multiple selector

  » General form of a call to `COND`:

```
(COND
  (predicate1 expression1)
   …
  (predicaten expressionn)
  [(ELSE expressionn+1)]
)
```

Control Flow

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

List Functions

- 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))

List Functions (continued)

- Examples:

  `(CAR '((A B) C D))` returns `(A B)`

  `(CAR 'A)` is an error

  `(CDR '((A B) C D))` returns `(C D)`

  `(CDR 'A)` is an error

  `(CDR '(A))` returns `()`

  `CAADR?`

  `(CONS '() '(A B))` returns `(() A B)`

  `(CONS '(A B) '(C D))` returns `((A B) C D)`

List Functions (continued)

- LIST is a function for building a list from any number of parameters

`(LIST ′apple ′orange ′grape)` returns

`(apple orange grape)`

Predicate Function: `EQ?`

- EQ? takes *two expressions as parameters* (usually two atoms); it returns #T if both parameters have the same pointer value; otherwise #F

`(EQ? 'A 'A)` yields `#T`

`(EQ? 'A 'B)` yields `#F`

`(EQ? 'A '(A B))` yields `#F`

`(EQ? '(A B) '(A B))` yields `#T` or `#F`

`(EQ? 3.4 (+ 3 0.4)))` yields `#T` or `#F`

Predicate Function: `EQV?`

- EQV? is like EQ?, except that it works for both *symbolic and numeric atoms*; it is a *value* comparison, not a *pointer* comparison

`(EQV? 3 3)` yields `#T`

`(EQV? 'A 3)` yields `#F`

`(EQV 3.4 (+ 3 0.4))` yields `#T`

`(EQV? 3.0 3)` yields `#F` (floats and integers are different)

Predicate Functions: `LIST?` and `NULL?`

- LIST? takes one parameter; it returns #T if the parameter is a list; otherwise #F

  `(LIST? '())` yields `#T`

- NULL? takes one parameter; it returns #T if the parameter is the empty list; otherwise #F

  `(NULL? '(()))` yields `#F`

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

Example Scheme Function: `equalsimp`

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

```
(DEFINE (equalsimp list1 list2)
(COND
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((EQ? (CAR list1) (CAR list2))
          (equalsimp(CDR list1)(CDR list2)))
    (ELSE #F)
))
```

National University Of Computer and Emerging Sciences

Example Scheme Function: `equal`

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

```
(DEFINE (equal list1 list2)
  (COND
   ((NOT (LIST? list1))(EQ? list1 list2))
   ((NOT (LIST? lis2)) #F)
   ((NULL? list1) (NULL? list2))
   ((NULL? list2) #F)
   ((equal (CAR list1) (CAR list2))
        (equal (CDR list1) (CDR list2)))
   (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 list1 list2)
  (COND
    ((NULL? list1) list2)
    (ELSE (CONS (CAR list1)
                (append (CDR list1) list2)))
))
```

Example Scheme Function: `LET`

- `LET` is actually shorthand for a `LAMBDA` expression applied to a parameter

```
(LET ((alpha 7))(* 5 alpha))
```

is the same as:

```
((LAMBDA (alpha) (* 5 alpha)) 7)
```

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)))
  (LIST (+ minus_b_over_2a root_part_over_2a))
        (- minus_b_over_2a root_part_over_2a))
))
```

Functional Form - Composition

- Composition
  - » If `h` is the composition of `f` and `g`, `h(x) = f(g(x))`

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
(DEFINE h x) (+ 2 (* 3 x))) (The composition)
```

  - » In Scheme, the functional composition function `compose` can be written:

```
(DEFINE (compose f g) (LAMBDA (x) (f (g x))))
```

`((compose CAR CDR) '((a b) c d))` yields `c`

```
(DEFINE (third a_list)

((compose CAR (compose CDR CDR)) a_list))
```

is equivalent to `CADDR`

National University
Of Computer and Emerging Sciences

Functional Form – Apply-to-All

- Apply to All - one form in Scheme is `map`
  - » Applies the given function to all elements of the given list;

```
(DEFINE (map fun a_list)
  (COND
    ((NULL? a_list) '())
    (ELSE (CONS (fun (CAR a_list))
                (map fun (CDR a_list))))
))
```

`(map (LAMBDA (num) (* num num num)) '(3 4 2 6))` yields `(27 64 8 216)`

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 a_list)
  (COND
    ((NULL? a_list) 0)
    (ELSE (EVAL (CONS '+ a_list)))
))
```

- 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

Support for Functional Programming in Primarily Imperative Languages

- Support for functional programming is increasingly creeping into imperative languages
  - » Anonymous functions (lambda expressions)
    - JavaScript: leave the name out of a function definition
    - C#: `i => (i % 2) == 0` (returns true or false depending on whether the parameter is even or odd)
    - Python: `lambda a, b : 2 * a - b`
- Python supports the higher-order functions filter and map (often use lambda expressions as their first parameters)

```
map(lambda x : x ** 3, [2, 4, 6, 8])
```

Returns `[8, 64, 216, 512]`

Comparing Functional and Imperative Languages

- Imperative Languages:
  - » Efficient execution
  - » Complex semantics
  - » Complex syntax
  - » Concurrency is programmer designed
- Functional Languages:
  - » Simple semantics
  - » Simple syntax
  - » Less efficient execution
  - » Better readability
  - » Programs can automatically be made concurrent