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

Mathematical Functions

Def: 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

l(x) x * x * x

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

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

- Lambda expressions describe nameless functions

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

e.g. (l(x) x * x * x)(3)

which evaluates to 27

Functional Forms

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

1. Function Composition

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

Form: h ∫ f ° g

which means h (x) ∫ f (g (x))

www.lecturetutorials.blogspot.com

2. Construction

A functional form that takes a list of functions as parameters and yields a list of the results of applying each of its parameter functions to a given parameter

Form: **[f, g]**

For f (x) ∫ x * x * x and g (x) ∫ x + 3,
[f, g] (4) yields (64, 7)

3. 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: **a**

For h (x) ∫ x * x * x
a (h, (3, 2, 4)) yields (27, 8, 64)

LISP - the first functional programming language

Data object types: **originally only atoms and lists**

List form: **parenthesized collections of sublists and/or atoms**
e.g., (A B (C D) E)

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

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

- In an FPL, the evaluation of a function always produces the same result given the same parameters
  - This is called *referential transparency*

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

A Bit of LISP

- Originally, LISP was a typeless language

- There were only two data types, atom and list

- LISP lists are stored internally as single-linked lists

- Lambda notation is used to specify functions and function definitions, function applications, and data all 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 parmeters, B and C

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

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

Scheme

- **A mid-1970s dialect of LISP, designed to be 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**

- ***Primitive Functions***

  1. Arithmetic: +, -, *, /, ABS, SQRT e.g., (+ 5 2) yields 7

  2. QUOTE -takes one parameter; returns the parameter without evaluation

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

- 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
  e.g., '(A B) is equivalent to (QUOTE (A B))

3. 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)

4. 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)

5. 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

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

e.g., (CONS 'A '(B C)) returns (A B C)

6. LIST - takes any number of parameters; returns a list with the parameters as elements

- *Predicate Functions:* (#T and () are true and false)

1. EQ? takes two symbolic parameters; it returns #T if both parameters are atoms and the two are the same

   e.g., (EQ? 'A 'A) yields #T
   (EQ? 'A '(A B)) yields ()

Note that if EQ? is called with list parameters, the result is not reliable
Also, EQ? does not work for numeric atoms

2. LIST? takes one parameter; it returns #T if the parameter is an list; otherwise ()

3. NULL? takes one parameter; it returns #T if the parameter is the empty list; otherwise ()

Note that NULL? returns #T if the parameter is ()

4. Numeric Predicate Functions
   =, <>, >, <, >=, <=, EVEN?, ODD?, ZERO?

www.lecturetutorials.blogspot.com

5. Output Utility Functions*:*

(DISPLAY expression) (NEWLINE)

- ***Lambda Expressions***

- **Form is based on l notation**

e.g.,
(LAMBDA (L) (CAR (CAR L)))

L is called a *bound variable*

- Lambda expressions can be applied

e.g.,
((LAMBDA (L) (CAR (CAR L))) '((A B) C D))

- A Function for Constructing Functions

DEFINE - Two forms:

1. To bind a symbol to an expression e.g.,
(DEFINE pi 3.141593)
(DEFINE two_pi (* 2 pi))

www.lecturetutorials.blogspot.com

Copyright © 1998 by Addison Wesley Longman, Inc.

WWW.KVRSOFTWARES.BLOGSPOT.COM

2. To bind names to lambda expressions e.g.,

(DEFINE (cube x) (* x x x))

- Example use:

(cube 4)

- Evaluation process (for normal functions):

1. **Parameters are evaluated, in no particular order**
2. **The values of the parameters are substituted into the function body**
3. **The function body is evaluated**
4. **The value of the last expression in the body is the value of the function**

(Special forms use a different evaluation process)

- Control Flow

- 1. Selection- the special form, **IF**

(IF predicate then_exp else_exp) e.g.,

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


www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

- 2. *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 expr in the first pair whose predicate evaluates to true

Example Scheme Functions

- 1. member - takes an atom and a list; returns #T if the atom is in the list; () otherwise

```
(DEFINE (member atm lis)
  (COND
    ((NULL? lis) '())
    ((EQ? atm (CAR lis)) #T) ((ELSE
    (member atm (CDR lis)))
  ))
```

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

- 2. equalsimp - takes two simple lists as parameters; returns #T if the two simple lists are equal; () otherwise

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

- 3. equal - takes two lists as parameters; returns #T if the two general lists are equal; () otherwise

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

www.kvrtutorials.blogspot.com

- 4. 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)))
 ))
```

Functional Forms

- 1. Composition
  - The previous examples have used it

- 2. Apply to All - one form in Scheme is mapcar
  - Applies the given function to all elements of the given list; result is a list of the results

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

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

- 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

    e.g., suppose we have a list of numbers that must be added together

```
((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 interprets the resulting list

Scheme includes some imperative features:

1. SET! binds or rebinds a value to a name
2. SET-CAR! replaces the car of a list
3. SET-CDR! replaces the cdr part of a list

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

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**

- **Includes:**
  - **records**
  - **arrays**
  - **complex numbers**
  - **character strings**
  - **powerful i/o capabilities**
  - **packages with access control**
  - **imperative features like those of Scheme**
  - **iterative control statements**

- ***Example* (iterative set membership, member)**

```
(DEFUN iterative_member (atm lst)
  (PROG ()
    loop_1
     (COND
       ((NULL lst) (RETURN NIL))
       ((EQUAL atm (CAR lst)) (RETURN T))
      )
    (SETQ lst (CDR
    lst)) (GO loop_1)
))
```

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

ML

- 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 (See Chapter 4)

- 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

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

- Function declaration form:

```
fun function_name (formal_parameters) =
        function_body_expression;
```

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

- Functions that use arithmetic or relational operators cannot be polymorphic--those with only list operations can be polymorphic

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

Lazy evaluation

- Infinite lists
  e.g.,

```
positives = [0..]
squares = [n * n | n ¨ [0..]]
```

(only compute those that are necessary)

e.g.,

```
member squares 16
```

would return True

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
```

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM

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 a significant number of 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**

www.lecturetutorials.blogspot.com

WWW.KVRSOFTWARES.BLOGSPOT.COM