MODULE 5
Functional Programming Languages

Functional Programming Languages

- Introduction to LISP
- Introduction to Scheme
- Comparison of Functional and Imperative Languages

Functional Programming Languages

- Functional programming languages are specially designed to handle symbolic computation and list processing applications.
- Functional programming is based on mathematical functions.
- Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.
- Functional programming languages are categorized into two groups,
  - **Pure Functional Languages** – These types of functional languages support only the functional paradigms.
    For example – Haskell.
  - **Impure Functional Languages** – These types of functional languages support the functional paradigms and imperative style programming.
    For example – LISP.

Introduction to LISP

- The First Functional Programming Language.
- Many functional programming languages have been developed. The oldest and most widely used is LISP (or one of its descendants), which was developed by John McCarthy at MIT in 1959.

➢ **Data Types and Structures**

- There were only two categories of data objects in the original LISP: atoms and lists.
- List elements are pairs, where the first part is the data of the element, which is a pointer to either an atom or a nested list. The second part of a pair can be a pointer to an atom, a pointer to another element, or the empty list.
- Elements are linked together in lists with the second parts.
- Atoms and lists are not types in the sense that imperative languages have types.
- In fact, the original LISP was a typeless language.
- Atoms are either symbols, in the form of identifiers, or numeric literals.

- LISP originally used lists as its data structure
- Lists are specified in LISP by delimiting their elements with parentheses.
- The elements of **simple lists** are restricted to atoms, as in

  (A B C D)
- **Nested list** structures are also specified by parentheses. For example, the list

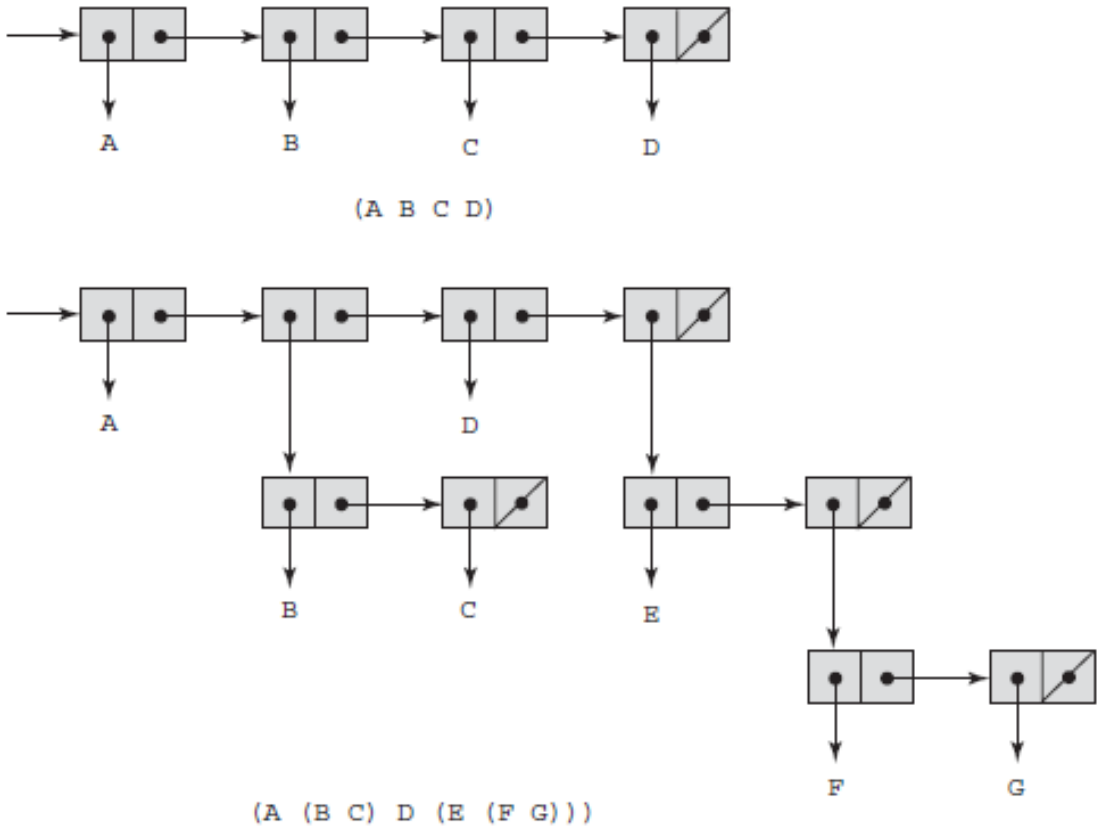
  (A (B C) D (E (F G)))

  is a list of four elements. The first is the atom A; the second is the sublist (B C); the third is the atom D; the fourth is the sublist (E (F G)), which has as its second element the sublist (F G).

- Internally, a list is usually stored as linked list structure in which each node has two pointers, one to reference the data of the node and the other to form the linked list.
- A list is referenced by a pointer to its first element.
- The internal representations of our two example lists are shown in Figure

Figure 15.1

Internal representation of two LISP lists



The First LISP Interpreter

- The original intent of LISP's design was to have a notation for programs. This notation was called M-notation, for meta-notation.
- There was to be a compiler that would translate programs written in M-notation into semantically equivalent machine code programs for the IBM 704.
- McCarthy thought that the functional processing of symbolic lists was a more natural model of computation than Turing machines, which operated on symbols written on tapes, which represented state.
- One of the common requirements of the study of computation is that one must be able to prove certain computability characteristics of the whole class of whatever model of computation is being used.
- From this concept came the idea of constructing a universal LISP function that could evaluate any other function in LISP.

- The first requirement for the universal LISP function was a notation that allowed functions to be expressed in the same way data was expressed.
- The **parenthesized list notation** had already been adopted for LISP data, so it was decided to invent conventions for function definitions and function calls that could also be expressed in list notation.
- Function calls were specified in a prefix list form originally called **Cambridge Polish**, as in the following:

  (function_name argument1 c argumentn)
- For example, if + is a function that takes two or more numeric parameters, the following two expressions evaluate to 12 and 20, respectively:

  (+ 5 7)

  (+ 3 4 7 6)

- The **lambda notation** was chosen to specify function definitions.
- It had to be modified, however, to allow the binding of functions to names so that functions could be referenced by other functions and by themselves.
- This name binding was specified by a list consisting of the function name and a list containing the lambda expression, as in

  (function_name (LAMBDA (arg1 … argn) expression))
- LISP functions specified in this new notation were called S-expressions, for *symbolic expressions*. Eventually, all LISP structures, both data and code, were called S-expressions. An S-expression can be either a list or an atom.
- McCarthy successfully developed a universal function that could evaluate any other function. This function was named **EVAL** and was itself in the form of an expression.
- An implementation of EVAL could serve as a LISP interpreter

- An interpreter for LISP can be written in LISP.
- Such an interpreter, which is not a large program, describes the operational semantics of LISP, in LISP.
- This is vivid evidence of the semantic simplicity of the language.

Introduction to Scheme

- We have chosen Scheme because it is relatively simple, it is popular in colleges and universities, and Scheme interpreters are readily available (and free) for a wide variety of computers.
- The version of Scheme described in this section is Scheme 4.

➢ Origins of Scheme

- The Scheme language, which is a dialect of LISP, was developed at MIT in the mid-1970s (Sussman and Steele, 1975).
- As an essentially typeless small language with simple syntax and semantics, Scheme is well suited to educational applications, such as courses in functional programming, and also to general introductions to programming.
- Most of the Scheme code in the following sections would require only minor modifications to be converted to LISP code.

The Scheme Interpreter

- A Scheme interpreter in interactive mode is an infinite read-evaluate-print loop (often abbreviated as REPL).
- It repeatedly reads an expression typed by the user (in the form of a list), interprets the expression, and displays the resulting value.
- Expressions are interpreted by the function EVAL.
- Literals evaluate to themselves. So, if you type a number to the interpreter, it simply displays the number.
- Expressions that are calls to primitive functions are evaluated in the following way: First, each of the parameter expressions is evaluated, in no particular order. Then, the primitive function is applied to the parameter values, and the resulting value is displayed.

Primitive Numeric Functions

- Scheme includes primitive functions for the basic arithmetic operations.
- These are +, −, *, and /, for add, subtract, multiply, and divide.
- * and + can have zero or more parameters.
- If * is given no parameters, it returns 1; if + is given no parameters, it returns 0.
- + adds all of its parameters together.
- * multiplies all its parameters together.
- / and − can have two or more parameters.
- In the case of subtraction, all but the first parameter are subtracted from the first.
- Division is similar to subtraction.

- Some examples are:

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

- There are a large number of other numeric functions in Scheme, among them MODULO, ROUND, MAX, MIN, LOG, SIN, and SQRT.
- SQRT returns the square root of its numeric parameter, if the parameter's value is not negative.
- If the parameter is negative, SQRT yields a complex number.
- In Scheme, note that we use uppercase letters for all reserved words and predefined functions.

Defining Functions

- A Scheme program is a collection of function definitions.
- Consequently, knowing how to define these functions is a prerequisite to writing the simplest program.
- In Scheme, a nameless function actually includes the word LAMBDA, and is called a **lambda expression**.
- For example,

  (LAMBDA (x) (* x x))

  is a nameless function that returns the square of its given numeric parameter.
- The following expression yields 49:

  ((LAMBDA (x) (* x x)) 7)
  - In this expression, x is called a **bound variable** within the lambda expression.
  - During the evaluation of this expression, x is bound to 7.

- Lambda expressions can have any number of parameters.
  - For example, we could have the following:

  (LAMBDA (a b c x) (+ (* a x x) (* b x) c))
- The Scheme special form function DEFINE serves two fundamental needs of Scheme programming: to bind a name to a value and to bind a name to a lambda expression.
- The simplest form of DEFINE is one used to bind a name to the value of an expression.
- This form is (DEFINE symbol expression)
- For example,

  (DEFINE pi 3.14159)

  (DEFINE two_pi (* 2 pi))
- If these two expressions have been typed to the Scheme interpreter and then pi is typed, the number 3.14159 will be displayed; when two_pi is typed, 6.28318 will be displayed.

- The second use of the DEFINE function is to bind a lambda expression to a name.
- In this case, the lambda expression is abbreviated by removing the word LAMBDA.
- To bind a name to a lambda expression, DEFINE takes two lists as parameters.
- The first parameter is the prototype of a function call, with the function name followed by the formal parameters, together in a list.
- The second list contains an expression to which the name is to be bound.
- The general form of such a DEFINE is

```
(DEFINE (function_name parameters)
        (expression)
)
```

- The following example call to DEFINE binds the name square to a functional expression that takes one parameter:

  (DEFINE (square number) (* number number))

- After the interpreter evaluates this function, it can be used, as in

  (square 5)

  which displays 25.

Output Functions

- Scheme includes a few simple output functions
- Scheme includes a formatted output function, PRINTF, which is similar to the printf function of C.

Numeric Predicate Functions

- A predicate function is one that returns a Boolean value (some representation of either true or false).
- Scheme includes a collection of predicate functions for numeric data.
- Among them are the following:

Function	*Meaning*
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
EVEN?	Is it an even number?
ODD?	Is it an odd number?
ZERO?	Is it zero?

- In Scheme, the two Boolean values are #T and #F (or #t and #f), although some implementations use the empty list for false.
- The Scheme predefined predicate functions return the empty list, (), for false.
- The NOT function is used to invert the logic of a Boolean expression.

Control Flow

- Scheme uses three different constructs for control flow: one similar to the selection construct of the imperative languages and two based on the evaluation control used in mathematical functions.
- The Scheme two-way selector function, named IF, has three parameters:

  a predicate expression, a then expression, and an else expression.
- A call to IF has the form

  (IF predicate then_expression else_expression)

- For example,

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

- the multiple selection of Scheme, COND
- Following is an example of a simple function that uses COND:

```
(DEFINE (leap? year)
  (COND
    ((ZERO? (MODULO year 400)) #T)
    ((ZERO? (MODULO year 100)) #F)
    (ELSE (ZERO? (MODULO year 4)))
))
```

- The third Scheme control mechanism is recursion, which is used, as in mathematics, to specify repetition.

List Functions

- This subsection introduces the Scheme functions for dealing with lists.
- Scheme programs are interpreted by the function application function, EVAL. When applied to a primitive function, EVAL first evaluates the parameters of the given function. This action is necessary when the actual parameters in a function call are themselves function calls.
- In some calls, however, the parameters are data elements rather than function references. When a parameter is not a function reference, it obviously should not be evaluated.

- Suppose we have a function that has two parameters, an atom and a list, and the purpose of the function is to determine whether the given atom is in the given list.
- To avoid evaluating a parameter, it is first given as a parameter to the primitive function QUOTE, which simply returns it without change.
- The following examples illustrate QUOTE:

  (QUOTE A) returns A

  (QUOTE (A B C)) returns (A B C)

- Two machine instructions, named CAR (*contents of the address part of a register)* and CDR (*contents of the decrement part of a register)* also used.

```
(CAR '(A B C)) returns A
(CAR '((A B) C D)) returns (A B)
(CAR 'A) is an error because A is not a list
(CAR '(A)) returns A
(CAR '()) is an error
(CDR '(A B C)) returns (B C)
(CDR '((A B) C D)) returns (C D)
(CDR 'A) is an error
(CDR '(A)) returns ()
(CDR '()) is an error
```

- second '(A B C) which returns B.
- Another example of a simple function, consider

  (DEFINE (second a_list) (CAR (CDR a_list)))

- Following are example calls to CONS:

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

- The two parameters to CONS become the CAR and CDR of the new list. Thus, if a_list is a list, then

  (CONS (CAR a_list) (CDR a_list))

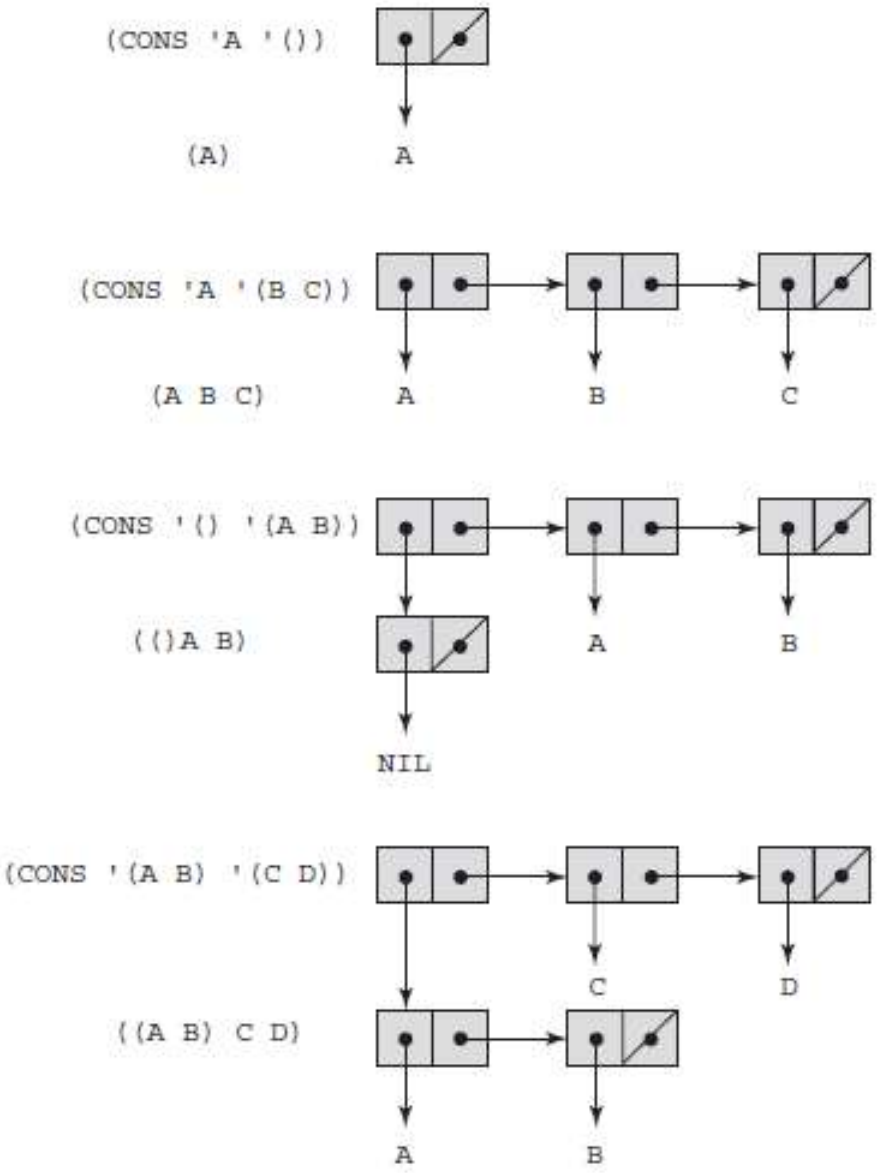
  returns a list with the same structure and same elements as a_list.
- LIST is a function that constructs a list from a variable number of parameters. It is a shorthand version of nested CONS functions, as illustrated in the following:

  (LIST 'apple 'orange 'grape)

  Returns (apple orange grape)

Figure 15.2

The result of several CONS operations



Predicate Functions for Symbolic Atoms and Lists

- Scheme has three fundamental predicate functions, EQ?, NULL?, and LIST?, for symbolic atoms and lists.
- The EQ? function takes two expressions as parameters, although it is usually used with two symbolic atom parameters.
- It returns #T if both parameters have the same pointer value—that is, they point to the same atom or list; otherwise, it returns #F.
- If the two parameters are symbolic atoms, EQ? returns #T if they are the same symbols (because Scheme does not make duplicates of symbols); otherwise #F.
- Consider the following examples:

```
(EQ? 'A 'A) returns #T
(EQ? 'A 'B) returns #F
(EQ? 'A '(A B)) returns #F
(EQ? '(A B) '(A B)) returns #F or #T
(EQ? 3.4 (+ 3 0.4)) returns #F or #T
```

- As the fourth example indicates, the result of comparing lists with EQ? is not consistent. The reason for this is that two lists that are exactly the same often are not duplicated in memory. At the time the Scheme system creates a list, it checks to see whether there is already such a list. If there is, the new list is nothing more than a pointer to the existing list. In these cases, the two lists will be judged equal by EQ?. However, in some cases, it may be difficult to detect the presence of an identical list, in which case a new list is created. In this scenario, EQ? yields #F.
- The last case shows that the addition may produce a new value, in which case it would not be equal (with EQ?) to 3.4, or it may recognize that it already has the value 3.4 and use it, in which case EQ? will use the pointer to the old 3.4 and return #T.
- EQ? works for symbolic atoms but does not necessarily work for numeric atoms. The = predicate works for numeric atoms but not symbolic atoms.

- Scheme has a different predicate, EQV?, which works on both numeric and symbolic atoms.
- Consider the following examples:

```
(EQV? 'A 'A) returns #T
(EQV? 'A 'B) returns #F
(EQV? 3 3) returns #T
(EQV? 'A 3) returns #F
(EQV? 3.4 (+ 3 0.4)) returns #T
(EQV? 3.0 3) returns #F
```

- EQV? is not a pointer comparison, it is a value comparison.
- The LIST? predicate function returns #T if its single argument is a list and #F otherwise, as in the following examples:

```
(LIST? '(X Y)) returns #T
(LIST? 'X) returns #F
(LIST? '()) returns #T
```

- The NULL? function tests its parameter to determine whether it is the empty list and returns #T if it is.
- Consider the following examples:

```
(NULL? '(A B)) returns #F
(NULL? '()) returns #T
(NULL? 'A) returns #F
(NULL? '(())) returns #F
```

- The last call yields #F because the parameter is not the empty list. Rather, it is a list containing a single element, the empty list.
- Consider the problem of membership of a given atom in a given list that does not include sublists. Such a list is called a **simple list**. If the function is named member, it could be used as follows:

```
(member 'B '(A B C)) returns #T
(member 'B '(A C D E)) returns #F
```

LET

- LET is a function that creates a local scope in which names are temporarily bound to the values of expressions.
- It is often used to factor out the common subexpressions from more complicated expressions.
- These names can then be used in the evaluation of another expression, but they cannot be rebound to new values in LET.
- The following example illustrates the use of LET.
- It computes the roots of a given quadratic equation, assuming the roots are real.
- The mathematical definitions of the real (as opposed to complex) roots of the quadratic equation ax2 + bx + c are as follows:

  root1 = (-b + sqrt(b2 - 4ac))/2a and

  root2 = (-b - sqrt(b2 - 4ac))/2a

```
(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))
))
```

- LET is actually shorthand for a LAMBDA expression applied to a parameter.
- The following two expressions are equivalent:

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

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

  - In the first expression, 7 is bound to alpha with LET;
  - in the second, 7 is bound to alpha through the parameter of the LAMBDA expression.

Tail Recursion in Scheme

- A function is **tail recursive** if its recursive call is the last operation in the function.
- This means that the return value of the recursive call is the return value of the nonrecursive call to the function.
- For example,

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

- This function can be automatically converted by a compiler to use iteration, resulting in faster execution than in its recursive form.

Functional Forms

- This section describes two common mathematical functional forms that are provided by Scheme: composition and apply-to-all.

1. *Functional Composition*

- Functional composition is the only primitive functional form provided by the original LISP.
- Function composition is a functional form that takes two functions as parameters and returns a function that first applies the second parameter function to its parameter and then applies the first parameter function to the return value of the second parameter function.
- In other words, the function h is the composition function of f and g if h(x) = f(g(x)).
- For example, consider the following example:

  (DEFINE (g x) (* 3 x))

  (DEFINE (f x) (+ 2 x))
- Now the functional composition of f and g can be written as follows:

  (DEFINE (h x) (+ 2 (* 3 x)))

- In Scheme, the functional composition function compose can be written as follows:

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

- For example, we could have the following:

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

  - This call would yield c.
  - This is an alternative, though less efficient, form of CADR.

- Now consider another call to compose:

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

  - This call would yield (b). This is an alternative to CDAR.

2. An Apply-to-All Functional Form

- The most common functional forms provided in functional programming languages are variations of mathematical apply-to-all functional forms.
- The simplest of these is **map**, which has two parameters: a function and a list.
- map applies the given function to each element of the given list and returns a list of the results of these applications.
- A Scheme definition of map follows:

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

- As an example of the use of map, suppose we want all of the elements of a list cubed.
- We can accomplish this with the following:

  (map (LAMBDA (num) (* num num num)) '(3 4 2 6))
- This call returns (27 64 8 216).

Comparison of Functional and Imperative Languages

- **Functional Languages:**

- Simple semantics

- Simple syntax

- Inefficient execution

- Programs can automatically be made concurrent

- **Imperative Languages:**

- Complex semantics

- Complex syntax

- Efficient execution

- Concurrency is programmer designed

- Functional languages can have a very simple **syntactic structure**. The syntax of the imperative languages is much more complex. This makes them more difficult to learn and to use.
- The **semantics** of functional languages is also simpler than that of the imperative languages.
- **Execution efficiency** is another basis for comparison. When functional programs are interpreted, they are of course much slower than their compiled imperative counterparts.
- Considering the relative efficiency of functional and imperative programs, it is reasonable to estimate that an average functional program will execute in about twice the time of its imperative counterpart.
- Another source of the difference in execution efficiency between functional and imperative programs is the fact that imperative languages were designed to run efficiently on von Neumann architecture computers, while the design of functional languages is based on mathematical functions. This gives the imperative languages a large advantage.

- Functional languages have a potential advantage in **readability**. In many imperative programs, the details of dealing with variables obscure the logic of the program.
- **Concurrent execution** in the imperative languages is difficult to design and difficult to use.
  - In an imperative language, the programmer must make a static division of the program into its concurrent parts, which are then written as tasks, whose execution often must be synchronized. This can be a complicated process.
  - Programs in functional languages are naturally divided into functions. In a pure functional language, these functions are independent in the sense that they do not create side effects and their operations do not depend on any nonlocal or global variables. Therefore, it is much easier to determine which of them can be concurrently executed.