Module 1

CHAPTER 1 Introduction to Programming Paradigms & Core Language Design Issues

Syllabus

Introduction to different programming paradigms : Names, Scopes, and Bindings, Scope Rules, Storage Management. Type Systems, Type Checking, Equality Testing and Assignment. Subroutine and Control Abstraction: Stack Layout, Calling sequence, parameter passing Generic subroutines and modules. Exception handling, Coroutines and Events.

Self-Learning Topic : Implementation of basic concepts using programming language.

1.1 INTRODUCTION TO PROGRAMMING PARADIGMS

GQ. 1.1.1 Explain the concept of programming paradigms.

☞ **What is a Program ?**

- At the most primitive level, a program is a sequence of characters chosen from an alphabet. But not all sequences are legal: there are *syntactic* restrictions (e.g., identifiers in C cannot begin with a digit).
- We also ascribe meaning to programs using the *semantics* of the language, which further restricts the legal sequences (e.g., declarations must precede uses in Ada).
- A *Programming Language* specifies what character sequences are legal and what these sequences mean (i.e., it specifies the *syntax* and *semantics*).

☞ **Why So Many Languages?**

- *Evolution*. First of all there are languages that are successors of others.
- B2 begot ABC begot Python

  Second, new concepts arise and sometimes go out of style. Recently, concurrency (threads) and object orientation have become popular.
- *Special Purpose*. There are languages tailored for hardware design (e.g. Verilog), manipulating character strings (Icon), logic programming (Prolog), etc.
- *Personal Preference*. Language designers who don't like existing languages create new ones.

1.1.1 The Programming Language Spectrum

GQ. 1.1.2 What is Programming Language Spectrum?

☞ **1. Imperative vs. Declarative Languages**

- An imperative language programmer essentially tells the computer exactly how to solve the problem at hand. In contrast a declarative language is higher-level: The programmer describes only what is to be done.
- For example, malloc/free (especially free) is imperative, but garbage collection is normally found in declarative languages. The Prolog example below illustrates the declarative style.
- There are broad spectrum languages like Ada that try to provide both low-level and high-level features. By necessity, such languages are large and complex.

☞ **2. von Neumann : Fortran, Pascal, C, Ada 83**

- This is the most common programming paradigm and largely subsumes the Object-Oriented class described next. It is the prototypical imperative language style.
- The defining characteristic is that the state of a program (very roughly the values of the variables) changes during execution. Very often the change is the result of executing an assignment statement.
- Recently, the term von Neumann is used to refer to serial execution of a program (i.e., no concurrency).

☞ **3. Object-Oriented : Simula 67, Smalltalk, Ada 95, Java, C#, C++**

- Languages that emphasize information hiding and (especially) inheritance. Data structures are bundled together with the operators using them.
- Most are von Neumann, but pure object-oriented languages like Smalltalk have a different viewpoint of computation, which is viewed as occurring at the objects themselves.

☞ **4. Functional (a.k.a. Applicative) : Scheme, ML, Haskell**

- These languages are based on the lambda calculus. They emphasize functions (without side effects) and discourage assignment statements and other forms of mutable state.
- Functions are first-class objects; new functions can be constructed *while the program is running*.

1.1.2 Characteristics of Modern Languages

- Modern general-purpose languages such as Ada, C++, and Java share many characteristics.
- Large and complex: Hugh manuals and grammars.
- Complex type system.
- Procedures and functions.
- Object-oriented facilities.
- Abstraction mechanisms with information hiding.
- Multiple storage-allocation mechanisms.
- Support for concurrency (not in C++).
- Support for generic programming (new in Java).
- Large, standard libraries.

1.3 SCOPE

GQ. 1.3.1 Explain the term 'Scope' in programming language ?

- A scope is the context within a computer program in which a variable name or other identifier is valid and can be used, or within which a declaration has effect.
- The scope of a binding is also known as the visibility of an entity.
- Static scope: Scoping follows the stricture of the program. C is said to be statically scoped
- Dynamic scope, where scoping follows the execution path. the languages, including APL, Snobol, and early versions of Lisp, are dynamically scoped: their bindings depend on the flow of execution at at run time.

Example

```
int i = 1; //global variable
void printdata()
{
cout << i <<endl;
}
int main ( )
{
int i = 2;
printdata( );
return 0;
}
```

- If static scoping used, the result will be 1
- If dynamic scoping used, this would print out 2.
- The lexical scope *of* a variable's definition is resolved by searching its containing block or function, then if that fails searching the outer containing block, and so on.
- Whereas with dynamic scope the calling function is searched, then the function which called that calling functions, and so on, progressing up the call stack. Of course, in both rules, we first look for a local definition of a variable.

1.4 BINDING

GQ. 1.4.1 Write short note on Binding in programming language With Example

- In general a binding is as association of two things. We will be interesting in binding a name to the thing it names and will see that a key question is when does this binding occur. The answer to that question is called the binding time.
- There is quite a range of possibilities. The following list is ordered from early binding to late binding.
- **Language design time :** The designers bind keywords to their meaning.
- **Language implementation time :** Some parts of a language are normally left for the implementer to decide (i.e., they are **not** the same across implementations). One common example is the number of bits in an integer; another is the order in which the additions are performed in A+B+C.
- **Program writing time :** Clearly, programmers choose many names.
- **Compile time :** The compiler binds high-level constructs, such as control structures (if, while, etc.) to machine code and also may bind static objects (e.g., global data) to specific machine locations.
- **Link time :** The linker combines separately compiled object modules into a final load module ready to be loaded and run. During this process, the overall layout of the modules is determined and inter-module references are resolved. The technical terminology is that relative addresses are relocated and external references are resolved.
- **Load time :** In older operating systems, jobs did not move once started. With such systems the loader bound virtual addresses (i.e., the addresses in the program) to physical addresses (i.e., addresses in the machine. Today it is more complicated: Very little of the job need be loaded prior to execution and once loaded, parts of the job can be moved. That is, today the binding of virtual address to physical addresses changes during execution.
- **Run time :** This is a large class and actually covers a range of times, for example
  - Program start time.
  - Module entry time.
  - Declaration elaboration time, when a declaration is first encountered.
  - Function/procedure call time.
  - Block entry time.
  - Statement execution time.

Let's look at an example from ada

```
procedure Demo is
  X : Integer;
begin
  X := 3;
  declare
    type Color is (Red, Blue, Green, Purple, White, Black);
    type ColorMatrix is array (Integer range <>, Integer range <>) of Color;
    subtype ColorMatrix10 is ColorMatrix(0..9,0..9);
    A : ColorMatrix10;
    B : ColorMatrix(X..8, 6..12); -- could not be bound at start
  begin
    A(2,3) := Red;
    A(2,X) := Red;        -- Same as previous statement
    B(4,11) := Blue;
  end;
  X := 4;
end Demo;
```

- Static binding refers to binding performed prior to run time.
- Dynamic binding refers to binding performed during run time.
- These terms are not very precise since there are many times prior to run time, and run time itself covers several times.

1.5 SCOPE RULE

GQ. 1.5.1 *Write the short note on 'scope rule'.*

- The region of program text where a binding is active is called the scope of the binding.
- Note that this can be different from the lifetime.
- The lifetime of the outer x in the example on the right is all of procedure f, but the scope of that x has a hole where the inner x hides the binding.

- We shall see below that in some languages, the hole can be filled.

Static vs Dynamic Scoping

```
procedure main is
  x : integer := 1;
  procedure f is
  begin
    put(x);
  end f;
  procedure g is
    x : integer := 2;
  begin
    f;
  end g;
begin
  g;
end main;
```

- Before we begin in earnest, I thought a short example might be helpful. What is printed when the procedure main on the right is run?
- That looks pretty easy, main just calls g, g just calls f, and f just prints (put is ada-speak for print) x. So x is printed.

1.5.1 Static Scoping

- In static scoping, the binding of a name can be determined by reading the program text; it does not depend on the execution.
- Thus it can be determined at compile time.
- **The simplest situation is the one in early versions of Basic :** There is only one scope, the whole program. Recall, that early basic was intended for tiny programs.
- I believe variable names were limited to two characters, a letter optionally follow by a digit. For large programs a more flexible approach is needed, as given in the next section.

1.2 NAME

GQ. 1.2.1 Explain the term 'NAME' in programming language ?

- A name is an identifier, i.e., a string of characters (with some restrictions) that represents something else.
- Many different kinds of things can be named, for example.
  - Variables
  - Constants
  - Functions/Procedures
  - Types
  - Classes
  - Labels (i.e., execution points)
  - Continuations (i.e., execution points with environments)
  - Packages/Modules

  Names are an important part of abstraction.
- Abstraction eases programming by supporting information hiding, that is by enabling the suppression of details.
- Naming a procedure gives a control abstraction.
- Naming a class or type gives a data abstraction. Consider the following Ada program.

```
procedure Demo is
type Color is (Red, Blue, Green, Purple, White, Black);
   type ColorMatrix is array (Integer range <>, Integer range <>) of Color;
   subtype ColorMatrix10 is ColorMatrix(0..9,0..9);
   A : ColorMatrix10;
   B : ColorMatrix(3..8, 6..12);
 begin
   A(2,3) := Red;
   B(4,11) := Blue;
 end Demo;
```

1.5.2 Nested Subroutines, i.e., Nested Scopes

- The typical situation is that the relevant binding for a name is the one that occurs in the smallest containing block and the scope of the binding is that block.
- So the rule for finding the relevant binding is to look in the current block. If the name is found, that is the binding. If the name is not found, look in the immediately enclosing scope and repeat until the name is found. If the name is not found at all, the program is erroneous.
- What about built in names such as type names (Integer, etc), standard functions (sin, cos, etc), or I/O routines (Print, etc)? It is easiest to consider these as defined in an invisible scope enclosing the entire program.
- Given a binding B in scope S, the above rules can be summarized by the following two statements.
  1. B is available in scopes nested inside S, unless B is overridden, in which case it is hidden.
  2. B is not available in scopes enclosing S.
- Some languages have facilities that enable the programmer to reference bindings that would otherwise be hidden by statement 1 above. For example
- In Ada a reference S.Y can be used to access the binding of Y declared in scope S even if there is a lexically closer scope.
- In C++ S::Y can be used to access the binding of Y is class S. Similarly ::Y can be used to refer to the binding of Y in the global scope (outside all classes).

```
procedure outer is
 x : integer := 6;
  procedure inner is
  begin
          put(x);
     end inner;
 begin
      inner;
 end outer;
```

```
procedure outer is
    x : integer := 6;
      procedure inner is
           x : integer := 88;
           begin
                put(x,outer.x);
      end inner;
           begin
           inner;
 end outer2;
```

1.5.3 Declaration Order

GQ. 1.5.2 Explain the Declaration order with example.

There are several questions here.

1. Must all declarations for a block precede all other statements?
   In Ada the answer is yes. Indeed, the declarations are set off by the syntax.

```
procedure <declarations> begin <statements> end;

declare <declarations> begin <statements> end;
```

In C, C++, and java the answer is no. The following is legal.

```
int x;  x=4;  int y;
```

1. Can one declaration refer to a previous declaration in the same block? Yes for Ada, C, C++, Java.

```
int z=4;  int y=z;
```

2. Do declarations take affect where the block begins (i.e., they apply to the entire block except for holes due to inner scopes) or do they start only at the declaration itself?

```
int y;  y=z;  int z=4;
```

- In Java, Ada, C, C++ they start at the declaration so the example above is illegal.
- In JavaScript and Modula3 they start at the beginning of the block so the above is legal.
- In Pascal the declaration starts at block beginning, but can't be used before it is declared. This has a weird effect: In inner declaration hides an outer declaration but can't be used in earlier statements of the inner.

1.6 STORAGE MANAGEMENT

GQ. 1.6.2 Explain the concept of Storage management.

1.6.1 Lifetimes

- The term *lifetime* to refer to the interval between creation and destruction.
- For example, the interval between the binding's creation and destruction is the binding's lifetime. For another example, the interval between the creation and destruction of an object is the object's lifetime, the binding lifetime differ from the object lifetime

- Pass-by-reference calling semantics. At the time of the call the parameter in the called procedure is bound to the object corresponding to the called argument. Thus the binding of the parameter in the called procedure has a shorter lifetime that the object it is bound to.
- Dangling References. Assume there are two pointers P and Q. An object is created using P, then P is assigned to Q; and finally the object is destroyed using Q. Pointer P is still bound to the object after the latter is destroyed, and hence the lifetime of the binding to P exceeds the lifetime of the object it is bound to. Dangling references like this are nearly always a bug and argue against languages permitting explicit object de-allocation (rather than automatic deallocation via garbage collection).

1.6.2 Storage Allocation Mechanisms

GQ. 1.6.2 Explain Storage Allocation Mechanisms.

There are three primary mechanisms used for storage allocation:

1. *Static* objects maintain the same (virtual) address throughout program execution.
2. *Stack* objects are allocated and deallocated in a last-in, first-out (LIFO, or stack-like) order. The allocations/deallocations are normally associated with procedure or block entry/exit.
3. *Heap* objects are allocated/deallocated at arbitrary times. The price for this flexibility is that the memory management operations are more expensive.

1. Static Allocation

- This is the simplest and least flexible of the allocation mechanisms. It is designed for objects whose lifetime is the entire program execution.
- The obvious examples are global variables. These variables are bound once at the beginning and remain bound until execution ends; that is their object and binding lifetimes are the entire execution.
- Static binding permits slightly better code to be compiled (for some architectures and compilers) since the addresses are computable at compile time.

Using Static Allocation for all Objects

- In a (perhaps overzealous) attempt to achieve excellent run time performance, early versions of the Fortran language were designed to permit static allocation of all objects.

  The price of this decision was high.

  - Recursion was not supported.
  - Arrays had to be declared of a fixed size.

- Before condemning this decision, one must remember that, at the time Fortran was introduced (mid 1950s), it was believed by many to be impossible for a compiler to turn out high-quality machine code. The great achievement of Fortran was to provide the first significant counterexample to this mistaken belief.

2. Local Variables

- For languages supporting recursion (which includes recent versions of Fortran), the same local variable name can correspond to multiple objects corresponding to the multiple instantiations of the recursive procedure containing the variable.
- Thus a static implementation is not feasible and stack-based allocation is used instead.
- These same considerations apply to compiler-generated temporaries, parameters, and the return value of a function.

3. Constants

- If a constant is constant throughout execution (what??, see below), then it can be stored statically, even if used recursively or by many different procedures.
- These constants are often called *manifest constants* or *compile time constants*.
- In some languages a constant is just an object whose value doesn't change (but whose lifetime can be short). In ada

```
loop
 declare
  v : integer;
 begin
  get(v);                -- input a value for v
  declare
   c : constant integer := v;  -- c is a "constant"
  begin
   v := 5;               -- legal; c unchanged.
   c := 5;               -- illegal
  end;
 end;
end loop;
```

For these constants static allocation is again not feasible.

1.6.3 Stack-Based Allocation

- This mechanism is tailored for objects whose lifetime is the same as the block/procedure in which it is declared.
- Examples include local variables, parameters, temporaries, and return values.
- The key observation is that the lifetimes of such objects obey a LIFO (stack-like) discipline: If object A is created prior to object B, then A will be destroyed no earlier than B.
- When procedure P is invoked the local variables, etc for P are allocated together and are pushed on a stack.
- This stack is often called the *control stack* and the data pushed on the stack for a single invocation of a procedure/block is called the *activation record* or *frame* of the invocation.
- When P calls Q, the frame for Q is pushed on to the stack, right after the frame for P and the LIFO lifetime property guarantees that we will remove frames from the stack in the safe order (i.e., will always remove (pop) the frame on the top of the stack.

1.6.4 Heap-Based Allocation

- What if we don't have LIFO lifetimes and thus cannot use stack-based allocation methods? Then we use heap-based allocation, which just means we can allocate and destroy objects in any order and with lifetimes unrelated to program/block entry and exit.
- A heap is a region of memory from which allocations and destructions can be performed at arbitrary times.

Remark

Please do not confuse these heaps with the heaps you may have learned in a data structures course. Those (data-structure) heaps are used to implement priority queues; they are not used to implement our heaps.

What objects are heap allocated ?

- Objects allocated with malloc() in C, or new in Java, or cons in Scheme.
- Objects whose size may change during execution, for example, extendable arrays and strings.

Implementing Heaps

- The question is how do respond to allocate/destroy commands? Looked at from the memory allocators viewpoint, the question is how to implement requests and returns of memory blocks (typically, the block returned must be one of those obtained by a request, not just a portion of a requested block).
- Note that, since blocks are not necessarily returned in LIFO order, the heap will have not simply be a region of allocated memory and another region of available memory. Instead it will have free regions interleaved with allocated regions.
- So the first question becomes, when a request arrives, which region should be (partially) uses to satisfy it. Each algorithmic solution to this question (e.g., first fit, best fit, worst fit, circular first fit, quick fit, buddy, Fibonacci) also includes a corresponding algorithm for processing the return of a block.
- **First Fit :** Choose the first free block big enough to satisfy the request. A right-size piece of this block is used to satisfy the request and the remaining piece is a new, smaller free block. If the entire block was used to satisfy the request, the unnecessary portion is wasted. This is called internal fragmentation since the wasted space is inside (internal to) an allocated block.
- **Best Fit :** Similar, but return the smallest free block that is big enough.
- **Worst Fit :** Similar, but return the largest free block.
- **Circular First Fit :** The same as first fit, but start the next search where the previous one left off.
- **Quick Fit :** Keep, in addition, lists of blocks of commonly needed size. For example, cons cells in Scheme are might all be the same size (two pointers).
- *Buddy* and *Fibonacci* are more complicated.
- What happens when the user no longer needs the heap-allocated space?
- **Manual deallocation :** The user issues a command like free or delete to return the space (C, Pascal). When a block (including any internal wasted space) is returned, it is coalesced, if possible, with any adjacent free blocks.
- Automatic deallocation via garbage collection (Java, C#, Scheme, ML, Perl).
- Semi-automatic deallocation using destructors (C++, Ada). The destructor is called automatically by the system, but the programmer writes the destructor code.
- Poorly done manual deallocation is a common programming error.
- If an object is deallocated and subsequently used, we get a *dangling reference*.

- If an object is never deallocated, we get a *memory leak*.
- We can run out of heap space for at least three different reasons.

Q. 1 What if there is not enough free space in total for the new request and all the currently allocated space is needed ?

☑ **Solution : Abort.**

Q. 2 What if we have several, non-contiguous free blocks, none of which are big enough for the new request, but in total they are big enough? This is called external fragmentation since the wasted space is outside (external to) all allocated blocks.

☑ **Solution : Compactify.**

Q. 3 What if some of the allocated blocks are no longer accessible by the user, but haven't been returned ?

☑ **Solution: Garbage Collection (next section).**

1.6.5 Garbage Collection

GQ. 1.6.3 Explain the concept of Garbage Collection in storage management. List its advantage and disadvantage.

- A garbage collection algorithm is one that automatically deallocates heap storage when it is no longer needed.
- It should be compared to manual deallocation functions such as free(). There are two aspects to garbage collection : first, determining automatically what portions of heap allocated storage will (definitely) not be used in the future, and second making this unneeded storage available for reuse.
- After describing the pros and cons of garbage collection, we describe several of the algorithms used.

Advantages and Disadvantages of Garbage Collection

- We start with the negative. Providing automatic reclamation of unneeded storage is an extra burden for the language implementer.
- More significantly, when the garbage collector is running, machine resources are being consumed. For some programs the garbage collection overhead can be a significant portion of the total execution time.
- If, as is often the case, the programmer can easily tell when the storage is no longer needed, it is normally much quicker for the programmer to free it manually than to have a garbage collector do it.

1.7 TYPE SYSTEM

GQ. 1.7.1 Explain the Concept of Type System.

A type system consists of :

- A mechanism ofr defining types and associating them with language constructs
- A set of rules for
  - (i) **Type equivalence :** When do two objects have the same type?
  - (ii) **Type compatibility :** Where can objects of a given type be used?
  - (iii) **Type synthesis/inference :** How do you determine the type of an expression from its parts or, in some cases, from its context (i.e., the type of the whole is used to determine the type of the parts).
- The synthesis/inference terminology is not standardized. Some texts, e.g., 3e, use type inference both for determining the type of the whole from the type of its parts, and for determining the type of the parts from the type of the whole. Other texts, e.g., the Dragon book, use type synthesis for the former and type inference for the latter.
- Some languages are untyped (e.g., B the predecessor of C); we will have little to say about those beyond saying that B actually had one datatype, the computer word.
- Types must be assigned to those constructs that can have values or that can refer to objects that have values. These include.
  - Literal constants (e.g., 5.8, "hello").
  - Named constants (e.g. static final int x = 3).
  - Variables.
  - Subroutines (not always, but having signatures is nice)
  - More complicated expressions made up of these.

1.7.1 Type Equivalence

- When are two types equivalent? There are two schools: name equivalence and structural equivalence. In (strict) name equivalence two type definitions are always distinct; Thus the four types on the right T1,...,T4 are all distinct.
- In structural equivalence, types are equivalent if they have the same structure so types T3 and T4 are equivalent and aggregates of those two types could be assigned to each other. Similarly, T1, T2, and integer are equivalent under structural equivalence.

- type T1 is new integer;
- type T2 is new integer;
- type T3 is record x:integer; y:integer; end record;
- type T4 is record x:integer; y:integer; end record;
- subtype S1 is integer;

Variants of Name Equivalence

- In addition to strict name equivalence as used above, there is also a concept of loose name equivalence where one type can be considered an alias of another. For example in Modula-2, which has loose name equivalence,

```
TYPE T5 = INTEGER;
```

- would be considered an alias of INTEGER and variables of type T5 could be assigned to variables of type INTEGER.

1.7.2 Type Compatibility

- A value must have a type compatible with the context in which the value is used. For most languages this notion is significantly weaker than equivalence; that is, there may be several non-equivalent types that can legally occur at a given context.
- We first need to ask what are the contexts in which only a type compatible with the expected type can be used. Three important contexts are
- **Assignment statements :** The type of the left hand side (lhs) must be compatible with the type of the value computed by the rhs.
- **Subroutine calls :** The types of the arguments must be compatible with the types of the corresponding parameters.
- **Built-in operations :** This situation is very similar to a subroutine call. For example, the operands of and must have types compatible with Boolean. Many of these operators, e.g., +, are overloaded (defined below) so there may be several types with which the operand types may be compatible. For example the operand types of + may be compatible with integer or with real

1.7.3 Type Inference/Synthesis

- In this section we emphasize type synthesis, i.e, determining the type of an expression given the type of its constituents. Next lecture, when we study ML, we will encounter type inference, where the type of the constituents in determined by the type of the expression.

Easy Cases

Sometimes it is easy to synthesize the type of the expression. For example,

- The result of a comparison is Boolean.
- The result of a function call has the type declared for that function.
- The built-in functions, e.g., arithmetic, usually have the same type as their operands (after any necessary coercions have been performed).
- For languages like C where assignments are expressions, the result of an assignment is the type of the left hand side.

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Ttada is
 subtype T1 is Integer range  0..20;
 subtype T2 is Integer range 10..20;
 A1 : T1;
 A2 : T2;
begin
 Get (A1); Get (A2);
 Put (A1+A2);
end Ttada;
```

Subranges

- What would be the type of A1 + A2 in the Ada program on the right? Two possibilities come to mind.
- The simplest soln is to view each value as an integer (the base type of T1 and T2.
- More complicated would be to do a bounds analysis and conclude that A1 + A2 is Integer range 10..40.
- Ada chooses the first approach. Indeed, in Ada values are associated with types, not subtypes. It is variables that can be associated with subtypes. As a result in Ada, assigning to a subtype that has a range constraint may require a run-time check.

1.7.4 Strong vs Weak Typing

GQ. 1.7.2 Explain Strong vs Weak Typing.

- The key attribute of strongly typed languages is that variables, constants, etc can be used only in manners consistent with their types. In contrast weakly typed languages offer many ways to bypass the type system.
- Compare the three programs on the right. The C program compiles and runs without errors. The Scheme define is accepted, but (test) gives an error report. The Ada program doesn't even compile.

```
int main (void) {
  int *x;
  int y;
  int *z;
  z = x + y;
  return 0;
}
```

```
(define test
 (lambda ()
  (let ((x 5) (y '(4)))
   (+ x y))))
```

```
procedure Ttada is
  X : Integer := 1;
  type T is access Integer;
  Y : T;
begin
  X := X + Y;
end Ttada;
```

1.7.5 Static vs Dynamic (Strong) Type Systems

GQ. 1.7.3 Explain Static vs Dynamic (Strong) Type Systems.

- Static and Dynamic strongly typed systems both prevent type clashes, but the prevention is done at different times and by different methods.
- In a static type system
  - Variables and values are typed.
  - The compiler (interpreter) enforces type rules at compile (definition) time.
- In a dynamic type system
  - Variables are not typed; values are typed.
  - The compiler (interpreter) enforces type rules at run time.
- Ada, Pascal, and ML have static type systems.
- Scheme (Lisp), Smalltalk, and scripting languages (if strongly typed) have dynamic type systems. These systems typically have late binding as well.
- A mixture is possible as well. Ada has a very few run-time checks; Java a few more.
- Static type systems have the following advantages.
  - Faster execution: The run-time checks are avoided.
  - Typically, better (more extensive) error checking, with errors reported sooner.
  - The resulting program is easier to read and maintain.
- Dynamic type systems have the following advantages.
  - They are more flexible.
  - Programs are easier to write.

1.8 TYPE CHECKING

GQ. 1.8.1 Define Type checking with example.

Definition : Type checking is the process of ensuring that a program obeys the type system's type compatibility rules.

Definition : A violation of the type checking rules is called type clash.

Definition : A language is called strongly typed if it prevents operations on inappropriate types.

Definition : *A strongly typed language is called statically typed if the necessary checks can be performed at compile time.*

- Type checking means checking that each operation should receive proper number of arguments and of proper data type.

```
A=B*j+d;
```

- *** and -** are basically **int and float data types** based operations and if any variable in this **A=B*j+d;**Is of other than int and float then compiler will generate type error.

1.8.1 Two ways of Type Checking

1.8.1(1) Dynamic Type Checking

- It is done at runtime.
- It uses concept of type tag which is stored in each data objects that indicates the data type of the object.

Example

- An integer data object contains its 'type' and 'values' attribute.
- so Operation only be performed after type checking sequence in which type tag of each argument is checked. If the types are not correct then error will be generated.
- Perl and Prolog follow basically dynamically type checking because data type of variables A+B in this case may be changed during program execution.
- So that type checking must be done at runtime.

1.8.1(2) Advantages of Dynamic Type

1. It is much flexible in designing programs or we ca say that the flexibility in program design.
2. In this no declarations are required.
3. In this type may be changed during execution.
4. In this programmerare free from most concern about data type.

1.8.1(3) Disadvantage of Dynamic Type

1. **Difficult to debug :** We need to check program execution paths for testing and in dynamic type checking, program execution path for an operation is never checked.
2. **Extra storage :** Dynamic type checking need extra storage to keep type information during execution.

3. **Seldom hardware support :** As hardware seldom support the dynamic type checking so we have to implement in software which reduces execution speed.

1.8.1(4) Type checking : Static Type Checking

- Static Type Checking is done at complete time.
- Information needed at compile time is provided- by declaration- by language structures.

The information required includes :

1. **for each operation :** The number, order, and data type, of its arguments.
2. **For each variables :** Name and data type of data object.

☞ **Example**

```
A+B
```

in this type of A and B variables must not be changed.

3. **for each constant :** Name and data type and value

```
const int x=28;

const float x=2.087;
```

- In this data type, the value and name is specified and in further if checked value assigned should match its data type.

1.8.1(5) Advantages of Static Type Checking

1. **Compiler saves information :** if that type of data is according to the operation then compiler saves that information for checking later operations which further no need of compilation.
2. **Checked execution paths :** As static type checking includes all operations that appear in any program statement, all possible execution paths are checked, and further testing for type error is not needed. So no type tag on data objects at run-time are not required, and no dynamic checking is needed.

1.8.1(6) Disadvantages of Static Type Checking

It affects many aspects of languages

1. declarations
2. data control structures
3. provision of compiling separately some subprograms.

1.9 EQUALITY TESTING AND ASSIGNMENT

GQ. 1.9.1 Explain Equality Testing and Assignment with example.

- For simple, primitive data types such as integers, floating-point numbers, or characters, equality testing and assignment are relatively straightforward operations, with obvious semantics and obvious implementations (bit-wise comparison or copy). For more complicated or abstract data types, however, both semantic and implementation subtleties arise.
- Consider for example the problem of comparing two character strings. Should the expression s = t determine whether s and t
  - (a) are aliases for one another?
  - (b) occupy storage that is bit-wise identical over its full length?
  - (c) contain the same sequence of characters?
  - (d) would appear the same if printed?
- The second of these tests is probably too low level to be of interest in most programs; it suggests the possibility that a comparison might fail because of garbage in currently unused portions of the space reserved for a string. The other three alternatives may all be of interest in certain circumstances, and may generate different results.
- In many cases the definition of equality boils down to the distinction between l-values and r-values: in the presence of references, should expressions be considered equal only if they refer to the same object, or also if the objects to which they refer are in some sense equal? The first option (refer to the same object) is known as a shallow comparison. The second (refer to equal objects) is called a deep comparison. For complicated data structures (e.g., lists or graphs) a deep comparison may require recursive traversal.
- In imperative programming languages assignment operations may also be deep or shallow. Under a reference model of variables, a shallow assignment a := b will make a refer to the object to which b refers.
- A deep assignment will create a copy of the object to which b refers, and make a refer to the copy. Under a value model of variables, a shallow assignment will copy the value of b into a, but if that value is a pointer (or a record containing pointers), then the objects to which the pointer(s) refer will not be copied.

= (Assignment) and == (Equal to) operators

1. = operator

The "=" is an assignment operator is used to assign the value on the right to the variable on the left.

For example

```
a = 10;
b = 20;
ch = 'y';
```

Example

```
filter_none
edit
play_arrow
brightness_4
// C program to demonstrate
// working of Assignment operators

#include <stdio.h>

int main()
{

  // Assigning value 10 to a
  // using "=" operator
  int a = 10;
  printf("Value of a is %d\n", a);
   return 0;
}
```

Output

```
Value of a is 10
```

☞ 2. == operator

The '==' operator checks whether the two given operands are equal or not. If so, it returns true. Otherwise it returns false.

For example

5==5

This will retuen true.

☞ **Example**

```
// C program to demonstrate
// working of relational operators
#include <stdio.h>
int main()
{
  int a = 10, b = 4;
  // equal to
  if (a == b)
    printf("a is equal to b\n");
  else
    printf("a and b are not equal\n");
  return 0;
}
```

Output

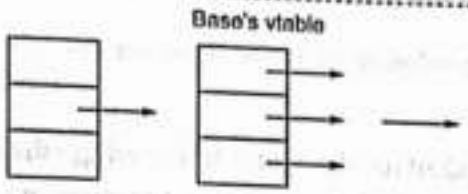
```
a and b are not equal
```

The differences can be shown in tabular form as follows:

=	==
It is an assignment operator.	It is a relational or comparison operator.
It is used for assigning the value to a variable.	It is used for comparing two values. It returns 1 if both the values are equal otherwise returns 0.
Constant term cannot be placed on left hand side. **Example:** 1=x; is invalid.	Constant term can be placed in the left hand side. **Example:** 1==1 is valid and returns 1.

1.10 REVIEW OF STACK LAYOUT

GQ. 1.10.1 Explain the Review of Stack Layout.



- As mentioned previously, stack-based storage is very efficient and is used for data whose lifetime is that of the procedure in which it is declared. Since procedures have a LIFO-like behavior (last one called is the first to return) a stack is an ideal storage model.
- Whenever a subroutine is called, it is given an activation record (a.k.a. frame, or AR) on the stack.
- There are some subtleties to enable support for varargs in languages like C. These are mentioned later.
- There are normally two pointers into the stack.
  - One points to the actual top of stack. When a new AR is created it starts here. The 3e calls this the stack pointer sp
  - The other points into the middle. The 3e calls it the frame pointer fp; the dragon book calls it the top_sp.
  - The reason for the second pointer is that the objects of unknown-at-compile-time size are all allocated at the top of the AR.
  - Thus, the displacement from the value of sp to fixed size objects is not known at compile time. But the fp points below the varying size objects so its displacement to each fixed size object is known at compile time.

1.11 CALLING SEQUENCE

GQ. 1.11.1 Write a short note on calling sequence.

- The caller and the callee cooperate to create a new AR when a subroutine is called. The exact protocol used is system dependent.
- There are two parts to the calling sequence: the prologue is executed at the time of the call, the epilogue is executed at the time of the return.

Saving and Restoring Registers

- Some systems use caller-save; others use callee-save; still others have some registers saved by the caller, other registers saved by the callee.

Maintaining the Static Chain

- In addition to the dynamic link, another inter-AR pointer is kept: the static link. The static link points to the AR of the most recent activation of the lexically enclosing block.
- The static link is needed so that non-local objects can be referenced (local objects are in the current AR).
- It is not hard to calculate the static link during (the prologue of) the calling sequence unless the language supports passing subroutines as arguments. We will not cover it; the simple case is again in the compiler notes.

A Typical Calling Sequence

As mentioned previously, the exact details are system dependent, what follows is a reasonable approximation.

The calling sequence begins with the caller :

1. Pushes some registers on to the stack (modifying sp).
2. Pushes arguments.
3. Computes and pushes the static link.
4. Executes a call machine instruction and pushes the return address.

Next the callee

1. Pushes the old frame ptr and calculates the new one.
2. Pushes some registers.
3. Allocates space for temporaries and local variables.
4. Begins execution of the subroutine.

When the subroutine completes, the callee:

1. Stores the return value (perhaps on the stack).
2. Restores some registers.
3. Restores the sp.
4. Restores the fp.
5. Jumps to the return address, resuming the caller.

Finally, the caller :

1. Restores some registers.
2. The return value to its destination.

1.11.1 Parameter Modes

GQ. 1.11.2 Explain Various parameter mode.

- The mode of a parameter determines the relation between the actual and corresponding formal parameter. For example, do changes to the latter affect the former.
- There are a number of modes including.
  - ***Call-by-value*** : The formal is bound to the value of the actual. The value of the actual is copied to the formal, but the locations are distinct. Hence changes to the formal do not affect the actual. But see below for the effect when the formal and actual are pointers.
  - ***Call-by-reference*** : The formal is bound to the location of the actual. Most languages require the actual to be an l-value. Changes to the former directly affect the latter.
  - ***Call-by-value/result*** : The formal is bound to the value of the actual. When the routine returns, the actual is bound to the (final) value of the formal. These semantics appear to have the same effect as call-by-reference, but see below. Call-by-value/result is also referred to as call-by-copy-return.
  - ***Call-by-name*** : The callee is macro-expanded substituting the actuals for the formals (see below).
  - ***Call-by-need*** : Similar to call-by-name (see below).
  - ***Call-by-sharing*** : Essentially call-by-reference (not discussed further).

```
int c = 1;
f(c);
printf ("%d\n");
...
void f(int x) { x = 5; }
```

1. Call-by-Value

- When using call-by-value semantics, the argument is evaluated (it might by an expression) and the value is assigned to the parameter. Changes to the parameter do not affect the argument, even when the argument is a variable. This is the mode used by C. Thus the C program on the right prints 1.

- As most C programmers know, g(&x): can change the value of x in the caller. This does not contradict the above. The value of &x can not be changed by g(). Naturally, the value of &x can be used by g(). Since that value points to x, g() can use &x to change x.
- As most C programmers know, if A is an array, h(A) can change the value of elements of A in the caller. This does not contradict the above. The extra knowledge needed is that in C writing an array without subscripts (as in h(A) above) is defined to mean the address of the first element. That is, h(A) is simply shorthand for h(&A[0]). Thus, the h(A) example is essentially the same as the g(&x) example preceding it.
- In Java, primitive types (e.g., int) are passed by value. Although the parameter can be assigned to, the argument is not affected.

☞ **2. Call-by-Reference**

- The location of the argument (the argument must be an l-value, but see the Fortran exception below) is bound the parameter. Thus changes to the parameter are reflected in the argument. Were C a call-by-reference language (it is not) then the example on the upper right would print 5.
- By default pascal uses call-by-value, but if a parameter is declared *var*, then call-by-reference is used.
- Fortran is a little weird in this regard. Parameters are normally call-by-reference. However, if the argument is not an l-value, a new variable is created, assigned the value of the argument, and is itself passed a call-by-reference.. Since this new variable is not visible to the programmer, changes by the callee to the corresponding parameter are not visible to the program.
- In Java, objects are passed by reference. If the parameter is declared final it is readonly.

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure ScalarParam is
  A : Integer := 10;
  B : Integer;
  Procedure F (X : in out Integer; Ans : out Integer) is
  begin
    X := X + A;
    Ans := X * A;
  end F;
begin
```

```
  F(A,B);
  Put (B);
end ScalarParam;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Ttada is
  type IntArray is array (0..2) of Integer;
  A : IntArray := (10, 10, 10);
  B : Integer;
  Procedure F (X : in out IntArray; Ans : out Integer) is
  begin
    X(0) := X(0) + A(0);
    Ans  := X(0) * A(0);
  end F;
begin
  F(A,B);
  Put (B);
end Ttada;
```

☞ **3. Call-by-Value/Result**

- With call-by-value/result the value in the argument is copied into the parameter during the call, and, at the return, the value in the parameter (the result) is copied back into the argument.
- Certainly this copying is different from call-by-reference, but the effect seems to be the same: changes to the parameter during the subroutine's execution are reflected in the argument after the return.

☞ **4. Call-by-Name**

- Call-by-name, made famous 40 years ago in Algol 60, will perhaps seem a little weird when compared to the currently more common modes above. In fact it is quite normal today, just not for subroutine invocation. One should compare it to macro expansion, for example #define in the C programming language. One should remember that in 1960, the most widely used programming languages were assembly languages and macro expansion was very common.

Module 2

CHAPTER 2 Imperative Paradigm : Data Abstraction in Object Orientation

Syllabus

Grouping of data and Operations : Encapsulation, Overloading, Polymorphism, Inheritance, Initialization and Finalization, Dynamic Binding.

Self-Learning Topic : Implementation of OOP concepts using preferrably C++ and Java language.

2.1 OBJECT-ORIENTED PROGRAMMING (OOP)

GQ. 2.1.1 Explain Object oriented programming.

- OOP is the term used to describe a programming approach based on **objects** and **classes**.
- The object-oriented paradigm allows us to organise software as a collection of objects that consist of both data and behaviour.
- This is in contrast to conventional functional programming practice that only loosely connects data and behaviour.
- Since the 1980s the word 'object' has appeared in relation to programming languages, with almost all languages developed since 1990 having object-oriented features.
- Some languages have even had object-oriented features retro-fitted. It is widely accepted that object-oriented programming is the most important and powerful way of creating software.

The object-oriented programming approach encourages :

- **Modularisation :** where the application can be decomposed into modules.

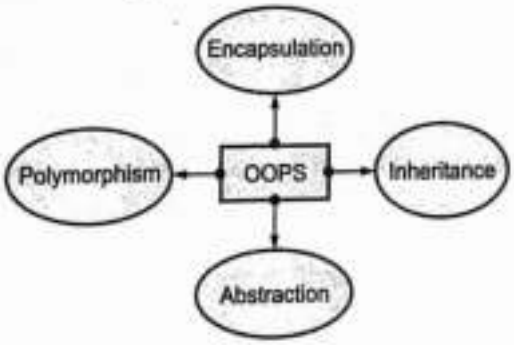


Fig. 2.1.1

- **Software re-use :** where an application can be composed from existing and new modules.

An object-oriented programming language generally supports five main features :

- Classes
- Objects
- Classification
- Polymorphism
- Inheritance

An Object-Oriented Class

If we think of a real-world object, such as a television (as in Fig. 1.1), it will have several features and properties :

- We do not have to open the case to use it.
- We have some controls to use it (buttons on the box, or a remote control).
- We can still understand the concept of a television, even if it is connected to a DVD player.
- It is complete when we purchase it, with any external requirements well documented.

A class should

- **Provide a well-defined interface :** such as the remote control of the television.
- **Represent a clear concept :** such as the concept of a television.
- **Be complete and well-documented :** the television should have a plug and should have a manual that documents all features.
- **The code should be robust :** it should not crash, like the television.
- **With a functional programming language (like C) we would have the component parts of the television scattered everywhere and we would be responsible for making them work correctly :** there would be no case surrounding the electronic components.
- Classes allow us a way to represent complex structures within a programming language. They have two components:
  - **States** - (or data) are the values that the object has.
  - **Methods** - (or behaviour) are the ways in which the object can interact with its data, the actions.

An Object

- An **object** is an instance of a class. You could think of a class as the description of a concept, and an object as the realisation of this description to create an independent distinguishable entity.
- For example, in the case of the Television, the class is the set of plans (or blueprints) for a generic television, whereas a television object is the realisation of these plans into a real-world physical television.
- So there would be one set of plans (the class), but there could be thousands of real-world televisions (objects).

- Objects can be concrete (a real-world object, a file on a computer) or could be conceptual (such as a database structure) each with its own individual identity

2.2 ENCAPSULATION

GQ. 2.2.1 Explain Data Encapsulation with diagram.

- Data encapsulation refers to the process of binding together data and functions or methods operating on this data into a single unit so that it is protected from outside interference and misuse.
- This is an important object-oriented programming concept and it leads to yet another OOP concept known as **"Data hiding"**. Encapsulation hides data and its members whereas abstraction expose only the necessary details or interfaces to the outside world.
- In a way, abstraction presents the "abstract view" of the hidden data to the outside world.
- Thus we already made a statement that encapsulation and abstraction go hand in hand.

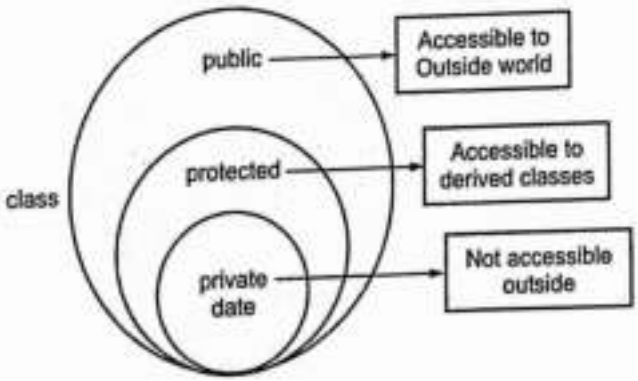


Fig. 2.2.1

- A class in C++ is the one where we bundle together data members and the functions operating on these data members along with access specifiers like private, public and protected represent encapsulation.
- We have already discussed access specifiers in our earlier tutorial on classes and objects.
- We also know that by default class members are private. When we declare class members as private and methods to access class members as public we are truly implementing encapsulation.
- At the same time, we provide an abstract view of data to the outside world in the form of public methods.

2.2.1 Implementation Of Encapsulation

GQ. 2.2.2 Explain Implementation Of Encapsulation with example.

- Encapsulation in C++ is implemented as a class that bundles data and the functions operating on this data together.
- Mostly data is declared as private so that it is not accessible outside the class.
- The methods or functions are declared as public and can be accessed using the object of the class.
- However, we cannot directly access private members and this is called data hiding.
- When this is done, data is secured and can be accessed only by functions of that particular class in which the data is declared.

```
// Example program
#include <iostream>
#include <string>
using namespace std;
//example class to demonstrate encapsulation
class sampleData{
  int num;
  char ch;

  public:
  //getter methods to read data values
  int getInt() const{
    return num;
}

char getCh() const{
  return ch;
```

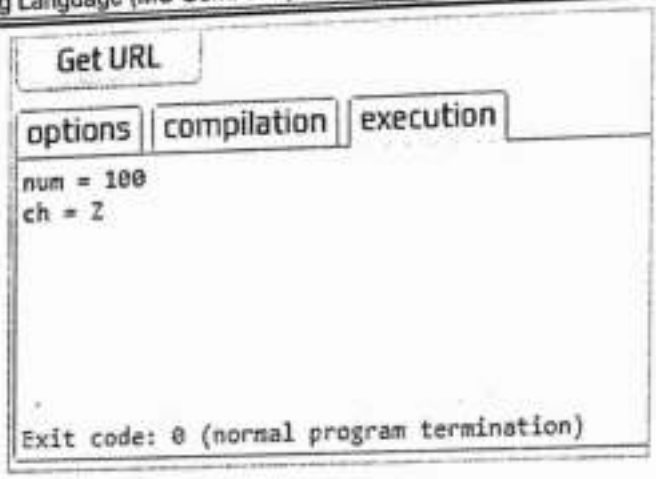
```
}

//setter methods to set data values
void setInt(int num) {
  this->num = num;
}
void setCh(char ch){
  this->ch = ch;
}
};
int main()
{
 sampleData s;
 s.setInt(100);
 s.setCh('Z');
 cout<<"num = "<<s.getInt()<<endl;
 cout<<"ch = "<<s.getCh();

 return 0;
}
```

Output

```
num = 100
ch = Z
```



- In the program above we have bundled two member variables along with the getter and setter methods into a class. and this is an example of encapsulation.
- We have declared two variables i.e. num and ch as private variables so that they are not accessible to the outside world.
- They are only accessible to the functions that we have declared as public.
- Thus we have hidden data members as private variables in a class.

2.2.2 Difference Between Encapsulation and Abstraction

Abstraction and encapsulation are closely bound together. Encapsulation aids in abstraction by bundling data and methods operating on that data together.

Encapsulation	Abstraction
Hides the data	Hides implementation
Bundles data and methods together	Provides an abstract interface to the user exposing only what is required
Aids in abstraction	Aids in reuse and security of the code.
Implemented as a class with access specifiers defining access to data members and methods	Implemented as an abstract class and interfaces that cannot be instantiated.

2.3 OVERLOADING

GQ. 2.3.1 Explain Overloading with suitable example.

- Within one class it is possible to define two or more methods that use the same name, but have different numbers of parameters.
- When this occurs, methods are called overloaded and such a process is referred to as method overloading.

- Method overloading is one of ways of polymorphism realization. Overloading of methods is performed according to the same rules as the function overloading.

If the called function has no exact match, the compiler searches for a suitable function on three levels sequentially:

1. search within class methods.
2. search within the base class methods, consistently from the nearest ancestor to the very first.
3. search among other functions.

- If there is no exact correspondence at all levels, but several suitable functions at different levels have been found, the function found at the least level is used. Within one level, there can't be more than one suitable function.
- Usually the function name tends to reflect its main purpose. As a rule, readable programs contain various well selected identifiers. Sometimes different functions are used for the same purposes.
- Let's consider, for example, a function that calculates the average value of an array of double precision numbers and the same function, but operating with an array of integers.

Both are convenient to be called AverageFromArray :

```
//+------------------------------------------------------------------+
//| The calculation of average for an array of double type           |
//+------------------------------------------------------------------+
double AverageFromArray(const double & array[],int size)
 {
  if(size<=0) return 0.0;
  double sum=0.0;
  double aver;
//---
  for(int i=0;i<size;i++)
   {
    sum+=array[i];   // Summation for the double
   }
  aver=sum/size;     // Just divide the sum by the number
//---
  Print("Calculation of the average for an array of double type");
  return aver;
 }
//+------------------------------------------------------------------+
//| The calculation of average for an array of int type              |
```

```
//+------------------------------------------------------------------+
double AverageFromArray(const int & array[],int size)
  {
   if(size<=0) return 0.0;
   double aver=0.0;
   int sum=0;
//---
   for(int i=0;i<size;i++)
     {
      sum+=array[i];    // Summation for the int
     }
   aver=(double)sum/size;// Give the amount of type double, and divide
//---
   Print("Calculation of the average for an array of int type");
   return aver;
  }
```

- Each function contains the message output via the Print() function;

```
Print("Calculation of the average for an array of int type");
```

- The compiler selects a necessary function in accordance with the types of arguments and their quantity.
- The rule, according to which the choice is made, is called the *signature matching algorithm.*
- A signature is a list of types used in the function declaration.

▸ **Example**

```
//+------------------------------------------------------------------+
//| Script program start function                                    |
//+------------------------------------------------------------------+
void OnStart()
  {
//---
   int    a[5]={1,2,3,4,5};
   double b[5]={1.1,2.2,3.3,4.4,5.5};
   double int_aver=AverageFromArray(a,5);
   double double_aver=AverageFromArray(b,5);
   Print("int_aver = ",int_aver,"  double_aver = ",double_aver);
  }
//--- Result of the script
// Calculate the average for an array of int type
```

```
// Calculate the average for an array of double type
// int_aver= 3.00000000  double_aver= 3.30000000
```

- Function overloading is a process of creating several functions with the same name, but different parameters. This means that in overloaded variants of a function, the number of arguments and/or their type must be different.
- A specific function variant is selected based on the correspondence of the list of arguments when calling the function, to the list of parameters in the function declaration.
- When an overloaded function is called, the compiler must have an algorithm to select the appropriate function.
- The algorithm that performs this choice depends on castings of what types are present.
- The best correspondence must be unique. An overloaded function must be the best match among all the other variants for at least one argument.
- At the same time it must match for all other arguments not worse than other variants.
- Below is a matching algorithm for each argument.

2.3.1 Algorithm of Choosing an Overloaded Function

1. Use strict matching (if possible).
2. Try standard type increase.
3. Try standard typecasting.

- The standard type increase is better than other standard conversions.
- Increase is the conversion of float to double, of bool, char, short or enum to int.
- Typecasting of arrays of similar integer types also belongs to typecasting. Similar types are: bool, char, uchar, since all the three types are single-byte integers; double-byte integers short and ushort; 4-byte integers int, uint, and color; long, ulong, and datetime.
- Of course, the strict matching is the best. To achieve such a consistency typecasting can be used.
- The compiler cannot cope with ambiguous situations. Therefore you should not rely on subtle differences of types and implicit conversions that make the overloaded function unclear.
- If you doubt, use explicit conversion to ensure strict compliance.
- Examples of overloaded functions in MQL4 can be seen in the example of ArrayInitialize() functions.
- Function overloading rules apply to overload of class methods.

- Overloading of system functions is allowed, but it should be observed that the compiler is able to accurately select the necessary function. For example, we can overload the system function MathMax() in 4 different ways, but only two variants are correct.

▶ **Example**

```
// 1. overload is allowed - function differs from built-in MathMax() function in the number of parameters
double MathMax(double a,double b,double c);
// 2. overload is not allowed!
// number of parameters is different, but the last has a default value
// this leads to the concealment of the system function when calling, which is unacceptable
double MathMax(double a,double b,double c=DBL_MIN);
// 3. overload is allowed - normal overload by type of parameters a and b
double MathMax(int a,int b);
// 4. overload is not allowed!
// the number and types of parameters are the same as in original double MathMax(double a,double b)
int MathMax(double a,double b);
```

2.3.2 Operation Overloading

GQ. 2.3.2 Explain Operation Overloading with suitable example.

- For ease of code reading and writing, overloading of some operations is allowed.
- Overloading operator is written using the keyword operator.

The following operators can be overloaded :

binary +,-,/,*,%,<<,>>,==,!=,<,>,<=,>=,=,+=,-=,/=,*=,%=,&=,|=,^=,<<=,>>=,&&,||,&,|,^

unary +,-,++,--,!,~

assignment operator =

indexing operator []

- Operation overloading allows the use of the operating notation (written in the form of simple expressions) for complex objects - structures and classes.
- Writing expressions using overloaded operations simplifies the view of the source code, because a more complex implementation is hidden.
- For example, consider complex numbers, which consist of real and imaginary parts.
- They are widely used in mathematics. The MQL4 language has no data type to represent complex numbers, but it is possible to create a new data type in the form of a **structure or class.**

Declare the complex structure and define four methods that implement four arithmetic operations :

```
//+------------------------------------------------------------------+
//| A structure for operations with complex numbers                  |
//+------------------------------------------------------------------+
struct complex
 {
 double        re; // Real part
 double        im; // Imaginary part
 //--- Constructors
               complex():re(0.0),im(0.0) { }
               complex(const double r):re(r),im(0.0) { }
               complex(const double r,const double i):re(r),im(i) { }
               complex(const complex &o):re(o.re),im(o.im) { }
 //--- Arithmetic operations
 complex       Add(const complex &l,const complex &r) const; // Addition
 complex       Sub(const complex &l,const complex &r) const; // Subtraction
 complex       Mul(const complex &l,const complex &r) const; // Multiplication
 complex       Div(const complex &l,const complex &r) const; // Division
 };
```

- Now, in our code we can declare variables representing complex numbers, and work with them.

For example

```
void OnStart()
 {
//--- Declare and initialize variables of a complex type
  complex a(2,4),b(-4,-2);
  PrintFormat("a=%.2f+i*%.2f,  b=%.2f+i*%.2f",a.re,a.im,b.re,b.im);
//--- Sum up two numbers
  complex z;
  z=a.Add(a,b);
  PrintFormat("a+b=%.2f+i*%.2f",z.re,z.im);
//--- Multiply two numbers
  z=a.Mul(a,b);
  PrintFormat("a*b=%.2f+i*%.2f",z.re,z.im);
//--- Divide two numbers
  z=a.Div(a,b);
  PrintFormat("a/b=%.2f+i*%.2f",z.re,z.im);
//---
 }
```

Module 2

- But it would be more convenient to use usual operators "+", "-", "*" and "/" for ordinary arithmetic operations with complex numbers.
- Keyword operator is used for defining a member function that performs type conversion.
- Unary and binary operations for class object variables can be overloaded as non-static member functions. They implicitly act on the class object.
- Most binary operations can be overloaded like regular functions that take one or both arguments as a class variable or a pointer to an object of this class.

For our type complex, overloading in the declaration will look like this :

```
//--- Operators
complex operator+(const complex &r) const { return(Add(this,r)); }
complex operator-(const complex &r) const { return(Sub(this,r)); }
complex operator*(const complex &r) const { return(Mul(this,r)); }
complex operator/(const complex &r) const { return(Div(this,r)); }
```

The full example of the script :

```
//+------------------------------------------------------------------+
//| Script program start function                                    |
//+------------------------------------------------------------------+
void OnStart()
 {
//--- Declare and initialize variables of type complex
  complex a(2,4),b(-4,-2);
  PrintFormat("a=%.2f+i*%.2f,  b=%.2f+i*%.2f",a.re,a.im,b.re,b.im);
  //a.re=5;
  //a.im=1;
  //b.re=-1;
  //b.im=-5;
//--- Sum up two numbers
  complex z=a+b;
  PrintFormat("a+b=%.2f+i*%.2f",z.re,z.im);
//--- Multiply two numbers

  z=a*b;
  PrintFormat("a*b=%.2f+i*%.2f",z.re,z.im);
//--- Divide two numbers
  z=a/b;
  PrintFormat("a/b=%.2f+i*%.2f",z.re,z.im);
//---
 }
```

```
//+------------------------------------------------------------------+
//| A structure for operations with complex numbers                  |
//+------------------------------------------------------------------+
struct complex
  {
   double         re; // Real part
   double         im; // Imaginary part
   //--- Constructors
                  complex():re(0.0),im(0.0) { }
                  complex(const double r):re(r),im(0.0) { }
                  complex(const double r,const double i):re(r),im(i) { }
                  complex(const complex &o):re(o.re),im(o.im) { }
   //--- Arithmetic operations
   complex        Add(const complex &l,const complex &r) const;  // Addition
   complex        Sub(const complex &l,const complex &r) const;  // Subtraction
   complex        Mul(const complex &l,const complex &r) const;  // Multiplication
   complex        Div(const complex &l,const complex &r) const;  // Division
   //--- Binary operators
   complex operator+(const complex &r) const { return(Add(this,r)); }
   complex operator-(const complex &r) const { return(Sub(this,r)); }
   complex operator*(const complex &r) const { return(Mul(this,r)); }
   complex operator/(const complex &r) const { return(Div(this,r)); }
  };
//+------------------------------------------------------------------+
//| Addition                                                         |
//+------------------------------------------------------------------+
complex complex::Add(const complex &l,const complex &r) const
  {
   complex res;
//---
   res.re=l.re+r.re;
   res.im=l.im+r.im;
//--- Result
   return res;
  }
//+------------------------------------------------------------------+
//| Subtraction                                                      |
//+------------------------------------------------------------------+
complex complex::Sub(const complex &l,const complex &r) const
  {
   complex res;
```

```
   //---
     res.re=l.re-r.re;
     res.im=l.im-r.im;
   //--- Result
     return res;
    }
   //+------------------------------------------------------------------+
   //| Multiplication                                                   |
   //+------------------------------------------------------------------+
   complex complex::Mul(const complex &l,const complex &r) const
    {
     complex res;
  //---
     res.re=l.re*r.re-l.im*r.im;
     res.im=l.re*r.im+l.im*r.re;
  //--- Result
    return res;
   }
 //+------------------------------------------------------------------+
 //| Division                                                         |
 //+------------------------------------------------------------------+
 complex complex::Div(const complex &l,const complex &r) const
  {
 //--- Empty complex number
   complex res(EMPTY_VALUE,EMPTY_VALUE);
 //--- Check for zero
  if(r.re==0 && r.im==0)
    {
     Print(__FUNCTION__+": number is zero");
     return(res);
    }
 //--- Auxiliary variables
  double e;
  double f;
 //--- Selecting calculation variant
  if(MathAbs(r.im)<MathAbs(r.re))
   {
   e = r.im/r.re;
   f = r.re+r.im*e;
   res.re=(l.re+l.im*e)/f;
   res.im=(l.im-l.re*e)/f;
```

```
  }
 else
  {
   e = r.re/r.im;
   f = r.im+r.re*e;
   res.re=(l.im+l.re*e)/f;
   res.im=(-l.re+l.im*e)/f;
  }
//--- Result
 return res;
}
```


- Most unary operations for classes can be overloaded as ordinary functions that accept a single class object argument or a pointer to it. Add overloading of unary operations "-" and "!".

```
//+------------------------------------------------------------------+
//| Structure for operations with complex numbers                    |
//+------------------------------------------------------------------+
struct complex
 {
  double         re;     // Real part
  double         im;     // Imaginary part
...
  //--- Unary operators
  complex operator-() const; // Unary minus
  bool    operator!() const; // Negation
 };
//+------------------------------------------------------------------+
//| Overloading the "unary minus" operator                           |
//+------------------------------------------------------------------+
complex complex::operator-() const
 {
  complex res;
//---
  res.re=-re;
  res.im=-im;
//--- Result
  return res;
 }
//+------------------------------------------------------------------+
//| Overloading the "logical negation" operator                      |
```

```
//+------------------------------------------------------------------+
bool complex::operator!() const
 {
//--- Are the real and imaginary parts of the complex number equal to zero?
  return (re!=0 && im!=0);
 }
```

Now we can check the value of a complex number for zero and get a negative value :

```
//+------------------------------------------------------------------+
//| Script program start function                                    |
//+------------------------------------------------------------------+
void OnStart()
 {
//--- Declare and initialize variables of type complex
  complex a(2,4),b(-4,-2);
  PrintFormat("a=%.2f+i*%.2f,  b=%.2f+i*%.2f",a.re,a.im,b.re,b.im);
//--- Divide the two numbers
  complex z=a/b;
  PrintFormat("a/b=%.2f+i*%.2f",z.re,z.im);
//--- A complex number is equal to zero by default (in the default constructor re==0 and im==0)
  complex zero;
  Print("!zero=",!zero);
//--- Assign a negative value
  zero=-z;
  PrintFormat("z=%.2f+i*%.2f,  zero=%.2f+i*%.2f",z.re,z.im, zero.re,zero.im);
  PrintFormat("-zero=%.2f+i*%.2f",-zero.re,-zero.im);
//--- Check for zero once again
  Print("!zero=",!zero);
//---
 }
```

- Note that we did not have to overload the assignment operator "=", as structures of simple types can be directly copied one into each other.
- Thus, we can now write a code for calculations involving complex numbers in the usual manner.
- Overloading of the indexing operator allows to obtain the values of the arrays enclosed in an object, in a simple and familiar way, and it also contributes to a better readability of the source code.
- For example, we need to provide access to a symbol in the string at the specified position.

A string in MQL4 is a separate type string, which is not an array of symbols, but with the help of an overloaded indexing operation we can provide a simple and transparent work in the generated CString class :

```
//+------------------------------------------------------------------+
//| Class to access symbols in string as in array of symbols         |
//+------------------------------------------------------------------+
class CString
 {
  string        m_string;

public:
                CString(string str=NULL):m_string(str) { }
  ushort operator[] (int x) { return(StringGetCharacter(m_string,x)); }
 };
//+------------------------------------------------------------------+
//| Script program start function                                    |
//+------------------------------------------------------------------+
void OnStart()
 {
//--- An array for receiving symbols from a string
  int    x[]={ 19,4,18,19,27,14,15,4,17,0,19,14,17,27,26,28,27,5,14,
               17,27,2,11,0,18,18,27,29,30,19,17,8,13,6 };
  CString str("abcdefghijklmnopqrstuvwxyz[ ]CS");
  string  res;
//--- Make up a phrase using symbols from the str variable
  for(int i=0,n=ArraySize(x);i<n;i++)
    {
     res+=ShortToString(str[x[i]]);
    }
//--- Show the result
  Print(res);
 }
```

2.4 POLYMORPHISM

GQ 2.4.1 Explain Polymorphism with example.

- Polymorphism is an opportunity for different classes of objects, related through inheritance, to respond in various ways when calling the same function element.
- It helps to create a universal mechanism describing the behavior of not only the base class, but also descendant classes.

Module 2

- Let's continue to develop a base class CShape, and define a member function GetArea() designed to calculate the area of a shape.
- In all the descendant classes, produced by inheritance from the base class, we redefine this function in accordance with rules of calculating the area of a particular shape.
- For a square (class CSquare), the area is calculated through its sides, for a circle (class CCircle), area is expressed through its radius etc.
- We can create an array to store objects of CShape type, in which both objects of a base class and those of all descendant classes can be stored.
- Further we can call the same function for each element of the array.

▶ **Example**

```
//--- Base class
class CShape
 {
protected:
  int       m_type;          // Shape type
  int       m_xpos;          // X - coordinate of the base point
  int       m_ypos;          // Y - coordinate of the base point
public:
  void      CShape(){m_type=0;};  // constructor, type=0
  int       GetType(){return(m_type);};// returns type of the shape
virtual
  double    GetArea(){return (0); }// returns area of the shape
 };
```

- Now, all of the derived classes have a member function getArea(), which returns a zero value.
- The implementation of this function in each descendant will vary.

```
//--- The derived class Circle
class CCircle : public CShape          // After a colon we define the base class
 {                              // from which inheritance is made
private:
  double    m_radius;          // circle radius
public:
  void      CCircle(){m_type=1;}; // constructor, type=1
  void      SetRadius(double r){m_radius=r;};
  virtual double GetArea(){return (3.14*m_radius*m_radius);}// circle area
};
```

For the class Square the declaration is the same :

```
//--- The derived class Square
class CSquare : public CShape          // After a colon we define the base class
 {                                     // from which inheritance is made
private:
  double       m_square_side;         // square side
 public:
  void         CSquare(){m_type=2;}; // constructor, type=1
  void         SetSide(double s){m_square_side=s;};
  virtual double  GetArea(){return (m_square_side*m_square_side);}// square area
 };
```


- For calculating the area of the square and circle, we need the corresponding values of m_radius and m_square_side, so we have added the functions SetRadius() and SetSide() in the declaration of the corresponding class.
- It is assumed that object of different types (CCircle and CSquare) derived from one base type CShape are used in our program.
- Polymorphism allows creating an array of objects of the base CShape class, but when declaring this array, these objects are yet unknown and their type is undefined.
- The decision on what type of object will be contained in each element of the array will be taken directly during program execution.
- This involves the dynamic creation of objects of the appropriate classes, and hence the necessity to use object pointers instead of objects.
- The new operator is used for dynamic creation of objects.
- Each such object must be individually and explicitly deleted using the delete operator.
- Therefore we will declare an array of pointers of CShape type, and create an object of a proper type for each element (**new Class_Name**),

2.5 INHERITANCE

GQ. 2.5.1 Explain the process of Inheritance with suitable example .

- The characteristic feature of OOP is the encouragement of code reuse through inheritance.
- A new class is made from the existing, which is called the base class.
- The derived class uses the members of the base class, but can also modify and supplement them.
- Many types are variations of the existing types.

- It is often tedious to develop a new code for each of them. In addition, the new code implies new errors.
- The derived class inherits the description of the base class, thus any re-development and re-testing of code is unnecessary.
- The inheritance relationships are hierarchical.
- Hierarchy is a method that allows to copy the elements in all their diversity and complexity.
- It introduces the objects classification. For example, the periodic table of elements has gases.
- They possess to properties inherent to all periodic elements.
- Inert gases constitute the next important subclass.
- The hierarchy is that the inert gas, such as argon is a gas, and gas, in its turn, is part of the system.
- Such a hierarchy allows to interpret behaviour of inert gases easily.
- We know that their atoms contain protons and electrons, that is true for all other elements.
- We know that they are in a gaseous state at room temperature, like all the gases.
- We know that no gas from inert gas subclass enters usual chemical reaction with other elements, and it is a property of all inert gases.
- Consider an example of the inheritance of geometric shapes.
- To describe the whole variety of simple shapes (circle, triangle, rectangle, square etc.), the best way is to create a base class (ADT), which is the ancestor of all the derived classes.
- Let's create a base class CShape, which contains just the most common members describing the shape.
- These members describe properties that are characteristic of any shape - the type of the shape and main anchor point coordinates.

Example

```
//--- The base class Shape
class CShape
 {
protected:
  int    m_type;          // Shape type
  int    m_xpos;          // X - coordinate of the base point
  int    m_ypos;          // Y - coordinate of the base point
public:
      CShape(){m_type=0; m_xpos=0; m_ypos=0;} // constructor
```

```
 void    SetXPos(int x){m_xpos=x;} // set X
 void    SetYPos(int y){m_ypos=y;} // set Y
};
```

- Next, create new classes derived from the base class, in which we will add necessary fields, each specifying a certain class.
- For the Circle shape it is necessary to add a member that contains the radius value.
- The Square shape is characterized by the side value.

Module 2

Therefore, derived classes, inherited from the base class CShape will be declared as follows :

```
//--- The derived class circle
class CCircle : public CShape      // After a colon we define the base class
 {                                 // from which inheritance is made
private:
  int        m_radius;            // circle radius
 public:
           CCircle(){m_type=1;}// constructor, type 1
 };
```

For the Square shape class declaration is similar :

```
//--- the derived class Square
class CSquare : public CShape      // After a colon we define the base class
 {                                 // from which inheritance is made
private:
  int        m_square_side;       // square side
 public:
           CSquare(){m_type=2;} // constructor, type 2
 };
```

- It should be noted that while object is created the base class constructor is called first, and then the constructor of the derived class is called.
- When an object is destroyed first the destructor of the derived class is called, and then a base class destructor is called.
- Thus, by declaring the most general members in the base class, we can add an additional members in derived classes, which specify a particular class. Inheritance allows creating powerful code libraries that can be reused many times.

The syntax for creating a derived class from an already existing one is as follows:

```
class class_name :
      (public | protected | private) opt base_class_name
{
  class members declaration
};
```

- One of aspects of the derived class is the visibility (openness) of its members successors (heirs).
- The public, protected and private keywords are used to indicate the extent, to which members of the base class will be available for the derived one.
- The public keyword after a colon in the header of a derived class indicates that the protected and public members of the base class CShape should be inherited as protected and public members of the derived class CCircle.
- The private class members of the base class are not available for the derived class.
- The public inheritance also means that derived classes (CCircle and CSquare) are CShapes.
- That is, the Square (CSquare) is a shape (CShape), but the shape does not necessarily have to be a square.
- The derived class is a modification of the base class, it inherits the protected and public members of the base class.
- The constructors and destructors of the base class cannot be inherited. In addition to members of the base class, new members are added in a derivative class.
- The derived class may include the implementation of member functions, different from the base class.
- It has nothing common with an overload, when the meaning of the same function name may be different for different signatures.
- In protected inheritance, public and protected members of base class become protected members of derived class.
- In private inheritance, the public and protected members of base class become private members of the derived class.
- In protected and private inheritance, the relation that "the object of a derivative class is object of a base class" is not true.
- The protected and private inheritance types are rare, and each of them needs to be used carefully.

- It should be understood that the type of inheritance (public, protected or private) does not affect the ways of **accessing the members of base classes in the hierarchy of inheritance from a derived class.**

With any type of inheritance, only base class members declared with public and protected access specifiers will be available out of the derived classes. Let's consider it in the following example:

```
#property copyright "Copyright 2011, MetaQuotes Software Corp."
#property link      "https://www.mql5.com"
#property version   "1.00"
//+------------------------------------------------------------------+
//| Example class with a few access types                            |
//+------------------------------------------------------------------+
class CBaseClass
 {
private:          //--- The private member is not available from derived classes
  int             m_member;
protected:        //--- The protected method is available from the base class and its derived classes
  int             Member(){return(m_member);}
public:           //--- Class constructor is available to all members of classes
                  CBaseClass(){m_member=5;return;};
private:          //--- A private method for assigning a value to m_member
  void            Member(int value) { m_member=value;};
  };
//+------------------------------------------------------------------+
//| Derived class with errors                                        |
//+------------------------------------------------------------------+
class CDerived: public CBaseClass // specification of public inheritance can be omitted, since it is default
 {
public:
  void Func() // In the derived class, define a function with calls to base class members
   {
   //--- An attempt to modify a private member of the base class
   m_member=0;       // Error, the private member of the base class is not available
   Member(0);        // Error, the private method of the base class is not available in derived classes
   //--- Reading the member of the base class
   Print(m_member);  // Error, the private member of the base class is not available
   Print(Member());  // No error, protected method is available from the base class and its derived classes
   }
 };
```

- In the above example, CBaseClass has only a public method — the constructor. Construc
  are called automatically when creating a class object.

- Therefore, the private member m_member and the protected methods Member() cannot be called from the outside. But in case of public inheritance, the Member() method of the base class will be available from the derived classes.
- In case of protected inheritance, all the members of the base class with public and protected access become protected. It means that if public data members and methods of the base class were accessible from the outside, with protected inheritance they are available only from the classes of the derived class and its further derivatives.

```
//+------------------------------------------------------------------+
//| Example class with a few access types                            |
//+------------------------------------------------------------------+
class CBaseMathClass
 {
private:        //--- The private member is not available from derived classes
  double        m_Pi;
public:         //--- Getting and setting a value for m_Pi
  void          SetPI(double v){m_Pi=v;return;};
  double        GetPI(){return m_Pi;};
public:         // The class constructor is available to all members
                CBaseMathClass() {SetPI(3.14); PrintFormat("%s",__FUNCTION__);};
 };
//+------------------------------------------------------------------+
//| A derived class, in which m_Pi cannot be modified                |
//+------------------------------------------------------------------+
class CProtectedChildClass: protected CBaseMathClass // Protected inheritance
 {
private:
  double        m_radius;
public:         //--- Public methods in the derived class
  void          SetRadius(double r){m_radius=r; return;};
  double        GetCircleLength(){return GetPI()*m_radius;};
 };
//+------------------------------------------------------------------+
//| Script starting function                                         |
//+------------------------------------------------------------------+
void OnStart()
```

```
{
//--- When creating a derived class, the constructor of the base class will be called automatically
  CProtectedChildClass pt;
//--- Specify radius
  pt.SetRadius(10);
  PrintFormat("Length=%G",pt.GetCircleLength());
//--- If we uncomment the line below, we will get an error at the stage of compilation, since SetPI() is now protected
// pt.SetPI(3);
//--- Now declare a variable of the base class and try to set the Pi constant equal to 10
  CBaseMathClass bc;
  bc.SetPI(10);
//--- Here is the result
  PrintFormat("bc.GetPI()=%G",bc.GetPI());
}
```


- The example shows that methods SetPI() and GetPi() in the base class CBaseMathClass are open and available for calling from any place of the program.
- But at the same time, for CProtectedChildClass which is derived from it these methods can be called only from the methods of the CProtectedChildClass class or its derived classes.
- In case of private inheritance, all the members of the basic class with the public and protected access become private, and calling them becomes impossible in further inheritance.

2.6 INITIALIZATION AND FINALIZATION

GQ. 2.6.1 Disuses important issues in Initialization and Finalization

- Most object-oriented languages provide some sort of special mechanism to *initialize* an object automatically at the beginning of its lifetime.
- When written in the form of a subroutine, this mechanism is known as a *constructor*.
- Though the name might be thought to imply otherwise, a constructor does not allocate space; it initializes space that has already been allocated.
- A few languages provide a similar *destructor* mechanism to *finalize* an object automatically at the end of its lifetime.

☞ **Several important issues arise.**

1. **Choosing a constructor :** An object-oriented language may permit a class to have zero, one, or many distinct constructors. In the latter case, different constructors may have different names, or it may be necessary to distinguish among them by number and types of arguments.

2. **References and values :** If variables are references, then every object must be created explicitly, and it is easy to ensure that an appropriate constructor is called. If variables are values, then object creation can happen implicitly as a result of elaboration. In this latter case, the language must either permit objects to begin their lifetime uninitialized, or it must provide a way to choose an appropriate constructor for every elaborated object.

3. **Execution order :** When an object of a derived class is created in C++, the compiler guarantees that the constructors for any base classes will be executed, outermost first, before the constructor for the derived class. Moreover, if a class has members that are themselves objects of some class, then the constructors for the members will be called before the constructor for the object in which they are contained.

☞ 1 Choosing a Constructor

- C++, Java, and C# all allow the programmer to specify more than one constructor for a given class.
- In C++, Java, and C#, the constructors behave like overloaded subroutines: they must be distinguished by their numbers and types of arguments.
- In Smalltalk and Eiffel, different constructors can have different names; code that creates an object must name a constructor explicitly. In Eiffel one might say.

```
class COMPLEX creation
new_cartesian, new_polar feature {ANY}
x, y : REAL
new_cartesian(x_val, y_val : REAL) is do
x := x_val; y := y_val
end
new_polar(rho, theta : REAL) is do
x := ro * cos(theta) y := ro * sin(theta)
end
--other public methods
feature {NONE}
--private methods
end -- class COMPLEX
...
```

```
a, b : COMPLEX
...
!!b.new_cartesian(0, 1) !!a.new_polar(pi/2, 1)
```

- The !! operator is Eiffel's equivalent of new. Because class COMPLEX specified constructor ("creator") methods, the compiler will insist that every use of !! specify a constructor name and arguments.
- There is no straightforward analog of this code in C++; the fact that both constructors take two real arguments means that they could not be distinguished by overloading.

2 References and Values

GQ. 2.6.2 Explain References and Values with example.

- Several object-oriented languages, including Simula, Smalltalk, Python, Ruby, and Java, use a programming model in which variables refer to objects. Other languages, including C++, Modula-3, Ada 95, and Oberon, allow a variable to have a value that *is* an object.
- Eiffel uses a reference model by default, but allows the programmer to specify that certain classes should be expanded, in which case variables of those classes will use a value model.
- In a similar vein, C# uses struct to define types whose variables are values, and class to define types whose variables are references.

1. Declarations and constructors in C++

- If a C++ variable of class type foo is declared with no initial value, then the compiler will call foo's zero-argument constructor (if no such constructor exists, but other constructors do, then the declaration is a static semantic error—a call to a nonexistent subroutine):

Copy constructors

```
foo b;    // calls foo::foo()
```

If the programmer wants to call a different constructor, the declaration must specify constructor arguments to drive overload resolution :

```
foo b(10, 'x'); // calls foo::foo(int, char)
```

2. Copy constructors

- The most common argument list consists of a single object, of the same or different class:

```
foo a;
bar b;
...
```

```
foo c(a);  //  calls foo::foo(foo&)
foo d(b);  //  calls foo::foo(bar&)
```

- Usually the programmer's intent is to declare a new object whose initial value is "the same" as that of the existing object.
- In this case it is more natural to write

```
foo a;      // calls foo::foo()
bar b;      // calls bar::bar()
...
foo c = a; // calls foo::foo(foo&)
foo d = b;// calls foo::foo(bar&)
```

- In recognition of this intent, a single-argument constructor in C++ is called a *copy constructor*.
- It is important to realize here that the equals sign (=) in these declarations indicates initialization, not assignment.
- The effect is *not* the same as that of the similar code fragment.

```
foo a, c, d;// calls foo::foo() three times
bar b;      // calls bar::bar()
c   =   a;  //      calls  foo::operator=(foo&)
d   =   b;  //      calls  foo::operator=(bar&)
```

- Here c and d are initialized with the zero-argument constructor, and the later use of the equals sign indicates *assignment*, not initialization.
- The distinction is a common source of confusion in C++ programs.
- It arises from the combination of a value model of variables and an insistence that every elaborated object be initialized by a constructor.

3 Execution Order

- If the object's class (call it *B*) is derived from some other class (call it *A*), C++ insists on calling an *A* constructor before calling a *B* constructor, so the derived class is guaranteed never to see its inherited fields in an inconsistent state.
- When the programmer creates an object of class *B* (either via declaration or with a call to new), the creation operation specifies arguments for a *B* constructor.

- These arguments allow the C++ compiler to resolve overloading when multiple constructors exist.
- Adding them to the creation syntax (as Simula does) would be a clear violation of abstraction.

The answer adopted in C++ is to allow the header of the constructor of a derived class to specify base class constructor arguments :

foo::foo(*foo params*) : bar(*bar args*) {...

- Here foo is derived from bar. The list *foo params* consists of formal parameters for this particular foo constructor.
- Between the parameter list and the opening brace of the subroutine definition is a "call" to a constructor for the base class bar.
- The arguments to the bar constructor can be arbitrarily complicated expressions involving the foo parameters.
- The compiler will arrange to execute the bar constructor before beginning execution of the foo constructor.
- Similar syntax allows the C++ programmer to specify constructor arguments or initial values for members of the class.

For example, we could have used this syntax to initialize prev, next, head_node, and val in the constructor for list_node :

```
list_node() : prev(this), next(this), head_node(this), val(0) {
// empty body -- nothing else to do
}
```

Given that all of these members have simple (pointer or integer) types, there will be no significant difference in the generated code.

But suppose we have members that are themselves objects of some nontrivial class :

```
class foo : bar {
mem1_t member1;   // mem1_t and
mem2_t member2;   // mem2_t are classes
...
}
foo::foo(foo params ) : bar( bar args ), member1( mem1 args ), member2( mem2 args ) {
```

- Here the use of embedded calls in the header of the foo constructor causes the compiler to call the copy constructors for the member objects, rather than calling the default (zero-argument) constructors, followed by operator=. Both semantics and performance may be different as a result.

4 Garbage Collection

- When a C++ object is destroyed, the destructor for the derived class is called first, followed by those of the base class(es), in reverse order of derivation. By far the most common use of destructors in C++ is manual storage reclamation.
- Suppose, for example, that we were to create a list or queue of character-string names :

```
class name_list_node : public gp_list_node {
char *name;   // pointer to the data in a node
public:
name_list_node() {
name = 0;     // empty string
}
name_list_node(char *n)   {
name = new char[strlen(n)];
strcpy(name, n);      // copy argument into member
}
```

2.7 DYNAMIC BINDING

- Dynamic method binding is central to object-oriented programming.
- Imagine, for example, that our administrative computing program has created a list of persons who have overdue library books.
- The list may contain both students and professors.
- If we traverse the list and print a mailing label for each person, dynamic method binding will ensure that the correct printing routine is called for each individual.
- In this situation the definitions in the derived classes are said to *override* the definition in the base class.

2.7.1 Semantics and Performance

- The principal argument against static method binding-and thus in favor of dynamic binding based on the type of the referenced object-is that the static approach denies the derived class control over the consistency of its own state.
- Suppose, for example, that we are building an I/O library that contains a

The need for dynamic

binding

text_file class:

```
class text_file {
char *name;
long position; // file pointer
public:
void seek(long whence);
...
};
```

Now suppose we have a derived class read_ahead_text_file:

```
class read_ahead_text_file : public text_file {
char *upcoming_characters;
public:
void seek(long whence); // redefinition
...
};
```

- The code for read_ahead_text_file::seek will undoubtedly need to change the value of the cached upcoming_characters.
- If the method is not dynamically dispatched, however, we cannot guarantee that this will happen : if we pass a read_ahead_text_file reference to a subroutine that expects a text_file reference as argument, and if that subroutine then calls seek.

2.7.2 Virtual and Nonvirtual Methods

- In Simula, C++, and C#, which use static method binding by default, the programmer
- can specify that particular methods should use dynamic binding by labeling them as virtual. Calls to virtual methods are *dispatched* to the appropriate implementation at run time, based on the class of the object, rather than the type of the reference. In C++ and C#, the keyword virtual prefixes the
- Virtual methods in C++ and C# subroutine declaration

```
class person {
public:
virtual void print_mailing_label();
```

2.7.3 Abstract Classes

- In most object-oriented languages it is possible to omit the body of a virtual method in a base class.
- In Java and C#, one does so by labeling both the class Abstract methods in Java

and C# and the missing method as abstract:

```
abstract class person {
...
public abstract void print_mailing_label();
...
```

The notation in C++ is somewhat: one follows the subroutine Abstract methods in C++ declaration with an "assignment" to zero:

```
class person {
...
public:
virtual void print_mailing_label() = 0;
...
```

- C++ refers to abstract methods as *pure virtual* methods. Regardless of declaration syntax, a *class* is said to be abstract if it has at least one abstract method.

- It is not possible to declare an object of an abstract class, because it would be missing at least one member. The only purpose of an abstract class is to serve as a base for other, *concrete* classes.
- A concrete class (or one of its intermediate ancestors) must provide a real definition for every abstract method it inherits.

2.7.4 Member Lookup

GQ. 2.7.1 Explain concept of Member lookup in dynamic hiding with example.

- With dynamic method binding, however, the object referred to by Vtables a reference or pointer variable must contain sufficient information to allow the code generated by the compiler to find the right version of the method at run time.
- The most common implementation represents each object with a record whose first field contains the address of a *virtual method table* (*vtable*) for the object's class .
- The vtable is an array whose *i*th entry indicates the address of the code for the object's *i*th virtual method.
- All objects of a given concrete class share the same vtable. _
- Suppose that the this (self) pointer for methods is passed in register r1, Implementation of a virtual method call that m is the third method of class foo, and that f is a pointer to an object of class foo. Then the code to call f->m() looks something like this:

```
class foo {
int a;
double b;
char c;
public:
virtual void k( ...
virtual int l( ...
virtual void m();
virtual double n( ...
} F;
```

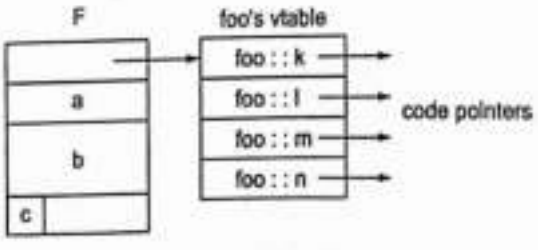


Fig. 2.7.1 : Implementation of virtual methods.

```
class bar : public foo {
int w;
public:
void m() override;
virtual double s( ...
virtual char *t( ...
} B;
```

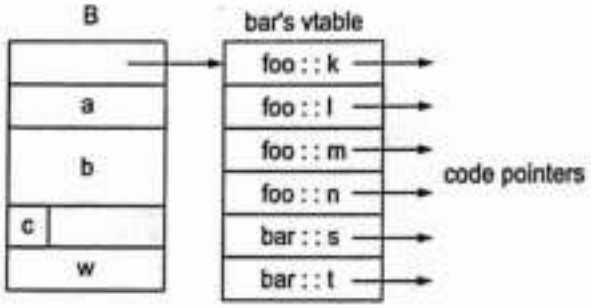


Fig. 2.7.2 : Implementation of single inheritance

2.7.5 OBJECT CLOSURES

- Object closures can be used in an object-oriented language to achieve roughly the same effect as subroutine closures in a language with nested subroutines-namely, to encapsulate a method with *context* for later execution.
- It should be noted that this mechanism relies, for its full generality, on dynamic method binding.
- Recall the plus_x object closure, here adapted to the apply_to_A code, and rewritten in generic form:

```
template<typename T>
class un_op {
public:
virtual T operator()(T i) const = 0;
};
class plus_x : public un_op<int> {
const int x;
public:
```

```
plus_x(int n) : x(n) { }
virtual int operator()(int i) const { return i + x; }
};
void apply_to_A(const un_op<int>& f, int A[], int A_size) {
int i;
for (i = 0; i < A_size; i++) A[i] = f(A[i]);
}
int A[10];
apply_to_A(plus_x(2), A, 10);
```


- Any object derived from un_op<int> can be passed to apply_to_A.
- The "right" function will always be called because operator() is virtual.
- A particularly useful idiom for many applications is to encapsulate a method *and its arguments* in an object closure for later execution Object passes in that first parameter position, would be declared as

```
std::function<void()> cf = std::bind(foo, 3, 3.14, 'x');
```

- The bind routine (an automatically instantiated generic function) encapsulates its first parameter (a function) together with the arguments that should eventually be passed to that function.
- The standard library even provides a "placeholder" mechanism (not shown here) that allows the programmer to bind only a subset of the function's parameters, so that parameters eventually passed to the function object can be used to fill in the remaining positions. _
- Object closures are commonly used in Java (and several other languages) to encapsulate start-up arguments for newly created threads of control .

  They can also be used to implement iterators via the *visitor pattern*.

Chapter Ends...

□□□

Module 3

CHAPTER 3 Declarative Programming Paradigm: Functional Programming

Syllabus

Introduction to Lambda Calculus, Functional Programming Concepts, Evaluation order, Higher order functions, I/O-Streams and Monads.

Self-Learning Topic : Implementation of programs using functional programming Language Haskel can refer to hacker rank website for problem statements.

3.1 INTRODUCTION TO LAMBDA CALCULUS

3.1.1 Introduction

GQ. 3.1.1 What is Lambda Calculus? Discuss it in details with its syntax.

- Lambda calculus is a framework developed by Alonzo Church in 1930s to study computations with functions. The lambda calculus was an attempt to formalise functions as a means of computing. It is a way of expressing computation through the use of functions we call Lambdas. If a problem is computable then it means we can build an algorithm to solve it and thus it can be expressed through the use of Lambda Calculus. Any of the computer programs we have ever written and any of the ones that are still unwritten can be expressed using either Lambda Calculus.
- The Lambda calculus can be called the smallest universal programming language of the world. The Lambda calculus calculus consists of a single transformation rule (variable substitution) and a single function definition scheme.
    - The lambda calculus is equivalent in definitional power to that of Turing machines.
    - The lambda calculus serves as the theoretical model for functional programming languages and has applications to artificial intelligence, proof systems, and logic.
    - Lisp was developed by John McCarthy at MIT in 1958 around the lambda calculus.
    - ML, a general-purpose functional language, was developed by Robin Milner at the University of Edinburgh in the early 1970s. Caml and OCaml are dialects of ML developed at INRIA in 1985 and 1996, respectively.
    - Haskell, considered by many as one of the purest functional programming languages, was developed by Simon Peyton Jones, Paul Houdak, Phil Wadler and others in the late 1980s and early 90s.
    - Many legacy languages including C++ and Java have incorporated features from the lambda calculus such as lambda expressions.
    - Because of its simplicity, the lambda calculus is a useful tool for the study and analysis of programming languages.

Module 3

3.1.2 CFG(Context Free Grammer) for the Lambda Calculus

- The central concept in the lambda calculus is an expression which we can think of as a program that when evaluated returns a result consisting of another lambda calculus expression.

- That means, everything in Lambda Calculus is an expressions, which means that everything must evaluate to a value.

Here is the grammar for lambda expressions :

expr → λ variable . expr | expr expr | variable | (expr) | constant

- A variable is an identifier.
- A constant is a built-in function such as addition or multiplication, or a constant such as an integer or boolean.
- However, as we shall see, all programming language constructs can be represented as functions with the pure lambda calculus so these constants are unnecessary. However, we will use some constants for notational simplicity.

3.1.3 Function Abstraction

GQ. 3.1.2 Write short notes on function abstraction with its example.

- Abstractions are perhaps the most iconic kind of lambda expression, they define what we call functions or, more adequately, lambdas: which are just anonymous functions.
- A function abstraction, often called a **lambda abstraction**, is a lambda expression that defines a function. A function abstraction consists of four parts: a lambda followed by a variable, a period, and then an expression as in *λx.expr*.
- In the function abstraction *λx.expr* the variable x is the formal parameter of the function and expr is the body of the function.
- For example, the function abstraction *λx. + x 1* defines a function of *x* that adds *x* to 1. Parentheses can be added to lambda expressions for clarity. Thus, we could have written this function abstraction as *λx.(+ x 1)* or even as *(λx. (+ x 1))*.
- In C this function definition might be written as

```
int addOne (int x)
{
 return (x + 1);
}
```

- Note that unlike C the lambda abstraction does not give a name to the function. The lambda expression itself is the function. We can say that *λx.expr binds* the variable *x* in *expr* and that *expr* is the scope of the variable.

3.1.4 Function Application

GQ. 3.1.3 What is Lambda Application? Discuss

- A function application, often called a **lambda application**, consists of an expression followed by an expression : *expr expr*.
- The first expression is a function abstraction and the second expression is the argument to which the function is applied. All functions in lambda calculus have exactly one argument. Multiple-argument functions are represented by currying, which we will explain after this topic.
- For example, the lambda expression $\lambda x.\ (+\ x\ 1)\ 2$ is an application of the function $\lambda x.\ (+\ x\ 1)$ to the
  argument 2.
  - This function application $\lambda x.\ (+\ x\ 1)\ 2$ can be evaluated by substituting the argument 2 for the formal parameter x in the body $(+\ x\ 1)$. Doing this we get $(+\ 2\ 1)$. This substitution is called a **beta reduction.**
  - Beta reductions are like macro substitutions in C. To do beta reductions correctly we may need to rename bound variables in lambda expressions to avoid name clashes.
  - Function application associates left-to-right; thus, $f\ g\ h = (f\ g)h$.
  - Function application binds more tightly than λ; thus, $\lambda x.\ f\ g\ x = (\lambda x.\ (f\ g)x)$.
  - Functions in the lambda calculus are first-class citizens; that is to say, functions can be used as arguments to functions and functions can return functions as results.

Module 3

3.1.5 Free and Bound Variables

GQ. 3.1.4 Show how all bound variables in a lambda expression can be given distinct names.

- In the function definition $\lambda x.x$ the variable x in the body of the definition (the second x) is bound because its first occurrence in the definition is λx. A variable that is not bound in *expr* is said to be free in *expr*. In the function $(\lambda x.xy)$, the variable x in the body of the function is bound and the variable y is free.
- Every variable in a lambda expression is either bound or free. Bound and free variables have quite a different status in functions.

☞ **In the expression** $(\lambda x.x)(\lambda y.yx)$

- The variable x in the body of the leftmost expression is bound to the first lambda.
- The variable y in the body of the second expression is bound to the second lambda.

- The variable x in the body of the second expression is free.
- Note that x in second expression is independent of the x in the first expression.

☞ **In the expression $(\lambda x.xy)(\lambda y.y)$**

- The variable y in the body of the leftmost expression is free.
- The variable y in the body of the second expression is bound to the second lambda.

☞ **Given an expression e, the following rules define FV(e), the set of free variables in e:**

- If e is a variable x, then $FV(e) = \{x\}$.
- If e is of the form $\lambda x.y$, then $FV(e) = FV(y) - \{x\}$.
- If e is of the form xy, then $FV(e) = FV(x) \cup FV(y)$.

– An expression with no free variables is said to be *closed*.

3.1.6 Beta Reductions

– A function application $\lambda x.e$ f is evaluated by substituting the argument f for all free occurrences of the formal parameter x in the body e of the function definition.

– We will use the notation $[f/x]e$ to indicate that f is to be substituted for all free occurrences of x in the expression e.

▶ **Examples**

- $(\lambda x.x)y \rightarrow [y/x]x = y$.
- $(\lambda x.xzx)y \rightarrow [y/x]xzx = yzy$.
- $(\lambda x.z)y \rightarrow [y/x]z = z$ since the formal parameter x does not appear in the body z.

– This substitution in a function application is called a beta reduction and we use a right arrow to indicate it.

– If $expr1 \rightarrow expr2$, we say expr1 reduces to $expr2$ in one step.

– In general, $(\lambda x.e)f \rightarrow [f/x]e$ means that applying the function $(\lambda x.e)$ to the argument expression f reduces to the expression $[f/x]e$ where the argument expression f is substituted for the function's formal parameter x in the function body e.

– A lambda calculus expression is "run" by computing a final result by repeatedly applying beta reductions.

– We use $\rightarrow^*$ to denote the reflexive and transitive closure of $\rightarrow$; that is, zero or more applications of beta reductions.

Examples

- $(\lambda x.x)y \rightarrow y$ (illustrating that $\lambda x.x$ is the identity function).
- $(\lambda x.xx)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow (\lambda y.y)$; thus, we can write $(\lambda x.xx)(\lambda y.y) \rightarrow^* (\lambda y.y)$.

Note that here we have applied a function to a function as an argument and the result is a function.

3.1.7 Evaluating a Lambda Expression

GQ. 3.1.5 Write short note on Lambda Expression with example.

- A lambda calculus expression can be thought of as a program which can be executed by evaluating it. Evaluation is done by repeatedly finding a reducible expression (called a redex) and reducing it by a function evaluation until there are no more redexes.
  - **Example 1 :** The lambda expression $(\lambda x.x)y$ in its entirety is a redex that reduces to y.
  - **Example 2 :** The lambda expression (+ (* 1 2) (– 4 3)) has two redexes : (* 1 2) and (– 4 3). If we choose to reduce the first redex, we get (+ 2 (– 4 3)). We can then reduce (+ 2 (– 4 3)) to get (+ 2 1). Finally we can reduce (+ 2 1) to get 3.

Module 3

Note that if we had chosen the second redex to revaluate first, we would have ended up with the same result :

- (+ (* 1 2) (- 4 3)) → (+ (* 1 2) 1) → (+ 2 1) → 3.

3.1.8 Currying

GQ. 3.1.6 Write short notes on Currying.

- All functions in the lambda calculus are prefix and take exactly one argument. If we want to apply a function to more than one argument, we can use a technique called *currying* that treats a function applied to more than one argument to a sequence of applications of one-argument functions.

For example,

- To express the sum of 1 and 2 we can write, (+ 1 2) as ((+ 1) 2), where the expression (+ 1) denotes the function that adds 1 to its argument.

Thus ((+ 1) 2) means the function + is applied to the argument 1 and the result is a function (+ 1) that adds 1 to its argument :

(+ 1 2) = ((+ 1) 2) 3

3.1.9 Renaming Bound Variables by Alpha Reduction

- The name of a formal parameter in a function definition is arbitrary. We can use any variable to name a parameter, so that the function $\lambda x.x$ is equivalent to $\lambda y.y$ and $\lambda z.z$. This kind of renaming is called *alpha reduction*.
- Note that we cannot rename free variables in expressions, also note that we cannot change the name of a bound variable in an expression to conflict with the name of a free variable in that expression.

3.1.10 Eta Conversion

- The two lambda expressions *(λx.+ 1 x)* and (+ 1) are equivalent in the sense that these expressions behave in exactly the same way when they are applied to an argument -- they add 1 to it. η-conversion is a rule that expresses this equivalence.
- In general, if x does not occur free in the function F, then $(\lambda x.F\ x)$ is η-convertible to F.

3.1.11 Substitutions

- For a beta reduction, we introduced the notation *[f/x]e* to indicate that the expression f is to be substituted for all free occurrences of the formal parameter x in the expression e:

  (x.e) f [f/x]e

- To avoid name clashes in a substitution *[f/x]e*, first rename the bound variables in e and f so they become distinct. Then perform the textual substituion of f for x in e.
  - For example, consider the substitution *[y(λx.x)/x] λy.(λx.x)yx.*
  - After renaming all the bound variables to make them all distinct we get *[y(λu.u)/x] λv.(λw.w)vx.*
  - Then doing the substitution we get *λv.(λw.w)v(y(λu.u)).*
- The rules for substitution are as follows. We assume x and y are distinct variables, and e, f and g are expressions.
  - For variables

    [e/x]x = e

    [e/x]y = y

- For function applications

  [e/x](f g) = ([e/x]f) ([e/x]g)

- For function abstractions

  [e/x](x.f) = x.f

  [e/x](y.f) = y.[e/x]f, provided y is not free in e (this is called the "freshness" condition).

Examples

1. [y/y](x.x) = x.x
2. [y/x]((x.y) x) = ([y/x](x.y)) ([y/x]x) = (x.y)y
3. Note that the freshness condition does not allow us to make the substitution [y/x](y.x) = y.([y/x]x) = y.y because y is free in the expression y.

3.1.12 Disambiguating Lambda Expressions

GQ. 3.1.7 Write rules to remove an unambiguous grammar for lambda calculus.

A few simple rules will remove the ambiguities.

Function application is left associative

f g h = ((f g) h)

Function application binds more tightly than lambda

$\lambda x.f\ g\ x = (\lambda x.(f\ g)\ x)$

The body in a function abstraction extends as far to the right as possible:

λx. + x 1 = λx. (+ x 1).

3.1.13 Normal Form

An expression containing no possible beta reductions is said to be in normal form. A normal form expression is one containing no redexes.

Examples of normal form expressions

- x where x is a variable.
- x e where x is a variable and e is a normal form expression.
- λx.e where x is a variable and e is a normal form expression.
- The expression *(λx.x x)(λx.x x)* does not have a normal form because it always evaluates to itself. We can think of this expression as a representation for an infinite loop.
- The expression *(λx. λy. y)((λz.z z)(λz.z z))* can be reduced to the normal form *λy.y*.

3.1.14 Evaluation Strategies

GQ. 3.1.8 Explain evaluation Strategy in Computer Programming.

An evaluation strategy specifies the order in which beta reductions for a lambda expression are made. Some reduction orders for a lambda expression may yield a normal form while other orders may not. For example, consider the given expression

$(\lambda x.1)((\lambda x.x\ x)(\lambda x.x\ x))$

This expression has two redexes

1. The entire expression is a redex in which we can apply the function (x.1) to the argument ((x.x x)(x.x x)) to yield the normal form 1. This redex is the leftmost outermost redex in the given expression.
2. The subexpression ((x.x x)(x.x x)) is also a redex in which we can apply the function (x.x x) to the argument (x.x x). Note that this redex is the leftmost innermost redex in the given expression. But if we evaluate this redex we get same subexpression: (x.x x)(x.x x) (x.x x)(x.x x). Thus, continuing to evaluate the leftmost innermost redex will not terminate and no normal form will result.

   There are two common reduction orders for lambda expressions:

   A. Normal order evaluation and

   B. applicative order evaluation.

Normal order evaluation

- In normal order evaluation we always reduce the leftmost outermost redex at each step.
- The first reduction order above is a normal order evaluation.
- A remarkable property of lambda calculus is that every lambda expression has a unique normal form if one exists. Moreover, if an expression has a normal form, then normal order evaluation will always find it.

Applicative order evaluation

- In applicative order evaluation we always reduce the leftmost innermost redex at each step.
- The second reduction order above is an applicative order evaluation.
- Thus, even though an expression may have a normal form, applicative order evaluation may fail to find it.

3.2 FUNCTIONAL PROGRAMMING CONCEPTS

3.2.1 Introduction

GQ. 3.2.1 Why functional Programming needed? Also categories its programming.

- Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Functional programming is a programming paradigm in which it is tried to bind each and everything in pure mathematical functions. It is a declarative type of programming style that focuses on what to solve rather than how to solve
- Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.

The programming paradigm is based on lambda calculus, which is briefly explained below :

Module 3

3.2.2 Lambda Calculus

- Instead of statements, It makes use of expressions. Unlike a statement, which is executed to assign variables, the evaluation of an expression produces a value. Lambda calculus forms the basis of almost all of the functional programming languages in use.
- Developed by Alonzo Church, Lambda Calculus is a framework for studying computations with functions. Anything that is computable using lambda calculus is computable. Amazingly, it can be labeled as the most succinct programming language of them all.
- In terms of its computational ability, lambda calculus is similar to the Turing machine that laid the foundation for the imperative style of programming. To put lambda calculus in simple words, it is a theoretical framework that describes functions and their evaluation.

Functional programming languages are categorized into two groups, i.e. –

- **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.

3.2.3 Concepts of Functional Programming

GQ. 3.2.2 Explain the details concept of Functional Programming Language along with it advantages and disadvantages.

It include five most important aspects,

A. Pure Functions

Pure functions have two important properties, they :

- Always produce the same output with the same arguments disregard of other factors. This property is also known as immutability
- Are deterministic. Pure functions either give some output or modify any argument or global variables i.e. they have no side effects.
- Because pure functions have no side-effects or hidden I/O, programs built using functional paradigm are easy to debug. Moreover, pure functions make writing concurrent applications easier.

When the code is written using the functional programming style, a capable compiler is able to :

- Memorize the results
- Parallelize the instructions
- Wait for evaluating results

B. Recursion

In the functional programming paradigm, there is no for and while loops. Instead, functional programming languages rely on recursion for iteration. Recursion is implemented using recursive functions, which repetitively call themselves until the base case is reached.

C. Referential Transparency

- Variables once defined in a functional programming language aren't allowed to change the value that they are holding throughout the execution of the program. This is known as referential transparency. It assures that the same language expression gives the same output.
- Functional programs don't have any assignment statements. For storing additional values in a program developed using functional programming, new variables must be defined. State of a variable in such a program is constant at any moment in time.
- Referential transparency eliminates even the slightest chances of any undesired effects due to the fact that any variable can be replaced with its actual value during any point in the program execution.

☞ **D. Functions are First-Class and can be Higher-Order**

- Functions in the functional programming style are treated as variables. Hence, they are first-class functions. These first-class functions are allowed to be passed to other functions as parameters or returned from functions or stored in data structures.
- A higher-order function is a function that takes other functions as arguments and/or returns functions. First-Class functions can be higher-order functions in functional programming languages.

☞ **E. Variables are Immutable**

- Variables are immutable i.e. it isn't possible to modify a variable once it has been initialized. Though we can create a new variable, modifying existing variables is not allowed.
- The immutable nature of variables in a functional programming language benefits in the form of preserving the state throughout the execution of a program.

Module 3

3.2.4 Characteristics of Functional Programming

GQ. 3.2.3 List out Characteristics of Functional Programming Languages.

The most prominent characteristics of functional programming are as follows :

- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming supports higher-order functions and lazy evaluation features.
- Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.

3.2.5 Advantages of Functional Programming

GQ. 3.2.4 Explain the advantages provided by functional programming.

Functional programming offers the following advantages :

- **Bugs-Free Code :** Pure functions take arguments once and produce unchangeable output. Hence, they don't produce any hidden output. They use immutable values, making debugging and testing easier. Functional programming does not support state, so there are no side-effect results and we can write error-free codes.

- **Efficient Parallel Programming :** Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.
- **Efficiency :** Due to the nature of pure functions to avoid changing variables or any data outside it, implementing concurrency becomes efficacious Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.
- **Supports Nested Functions :** Functional programming supports Nested Functions.
- **Lazy Evaluation :** It supports the concept of lazy evaluation, which means that the value is evaluated and stored only when it is required. Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.
- **Simple to understand :** Because pure functions don't change any states and are entirely dependent on the input, they are simple to understand. The return value given by such functions is the same as the output produced by them. The arguments and return type of pure functions are given out by their function signature.
- **Enhances the comprehension and readability :** It's style treats functions as values and passes the same to other functions as parameters. It enhances the comprehension and readability of the code.
- Functional Programming is used in situations where we have to perform lots of different operations on the same set of data.
  - Lisp is used for artificial intelligence applications like Machine learning, language processing, Modeling of speech and vision, etc.
  - Embedded Lisp interpreters add programmability to some systems like Emacs.

3.2.6 Disadvantages of Functional Programming

GQ. 3.2.5 Explain the downside of functional programming language.

- As a downside, functional programming requires a large memory space. As it does not have state, you need to create new objects every time to perform actions. It has following disadvantages:
- **Reduction in performance :** Immutable values combined with recursion might lead to a reduction in performance
- **Reduction in the readability of the code :** In some cases, writing pure functions causes a reduction in the readability of the code

- **I/O operations is tough :** Though writing pure functions is easy, combining the same with the rest of the application as well as the I/O operations is tough
- **Daunting task :** Writing programs in recursive style in place of using loops for the same can be a daunting task

3.2.7 Applications of Functional Programming Languages:

GQ. 3.2.6 Write in details applications of functional programming languages.

Followings all are the major and recent application of functional programming languages :

- Functional programming languages are preferred to be used for academic purposes rather than commercial software development.
- Nonetheless, several prominent programming languages following a functional programming paradigm, such as Clojure, Erlang, F#, Haskell, and Racket, are used widely for developing a variety of commercial and industrial applications.
- WhatsApp makes use of Erlang, a programming language following the functional programming paradigm, to enable its mere 100+ employees to manage the data belonging to over 1.5 billion people.
- Another important torchbearer of the functional programming style is Haskell. It is used by Facebook in its anti-spam system. Even JavaScript, one of the most widely used programming languages, flaunts properties of a dynamically typed functional language.
- Moreover, the functional style of programming is essential for various programming languages to lead in distinct domains. For example, R in statistics and J, K, and Q in financial analysis.
- Some elements of this programming paradigm are even used by domain-specific declarative languages such as Lex/Yacc and SQL for eschewing mutable values.

Generally, this paradigm is widely employed in :

- Applications aimed at concurrency or parallelism
- Carrying out mathematical computations.

3.2.8 Major highlights of functional programming Language

- Uses Immutable data.
- Follows Declarative Programming Model.
- Focus is on: "What you are doing"

- Supports Parallel Programming
- Its functions have no-side effects
- Flow Control is done using function calls & function calls with recursion
- It uses "Recursion" concept to iterate Collection Data.
- Execution order of statements is not so important.
- Supports both "Abstraction over Data" and "Abstraction over Behavior".

3.3 EVALUATION ORDER

GQ. 3.3.1 What is Evaluation order? Explain it in details.

- Order of evaluation of any part of any expression, including order of evaluation of function arguments is *unspecified* (with some exceptions listed below). The compiler can evaluate operands and other subexpressions in any order, and may choose another order when the same expression is evaluated again.
- There is no concept of left-to-right or right-to-left evaluation in C++. This is not to be confused with left-to-right and right-to-left associativity of operators: the expression a() + b() + c() is parsed as (a() + b()) + c() due to left-to-right associativity of operator+, but the function call to c may be evaluated first, last, or between a() or b() at run time:

```
#include <cstdio>
int a() { return std::puts("a"); }
int b() { return std::puts("b"); }
int c() { return std::puts("c"); }
void z(int, int, int) {}
int main() {
   z(a(), b(), c());          // all 6 permutations of output are allowed
   return a() + b() + c();    // all 6 permutations of output are allowed
}
```

Output

```
c
b
a
a
b
c
```

3.3.1 Sequenced Before Rule

- Evaluation of Expressions

Evaluation of each expression includes :

- Value computations: calculation of the value that is returned by the expression. This may involve determination of the identity of the object (glvalue evaluation, e.g. if the expression returns a reference to some object) or reading the value previously assigned to an object (prvalue evaluation, e.g. if the expression returns a number, or some other value)
- Initiation of side effects: access (read or write) to an object designated by a volatile glvalue, modification (writing) to an object, calling a library I/O function, or calling a function that does any of those operations.

Ordering

- "sequenced-before" is an asymmetric, transitive, pair-wise relationship between evaluations within the same thread.
- If A is sequenced before B, then evaluation of A will be complete before evaluation of B begins.
- If A is not sequenced before B and B is sequenced before A, then evaluation of B will be complete before evaluation of A begins.
- If A is not sequenced before B and B is not sequenced before A, then two possibilities exist:
- evaluations of A and B are unsequenced : they may be performed in any order and may overlap (within a single thread of execution, the compiler may interleave the CPU instructions that comprise A and B)
- evaluations of A and B are indeterminately sequenced: they may be performed in any order but may not overlap: either A will be complete before B, or B will be complete before A. The order may be the opposite the next time the same expression is evaluated.

Rules

1. Each value computation and side effect of a full expression, that is

Unevaluated Operands

Constant expression

Immediate Invocations

- an entire initializer, including any comma-separated constituent expressions
- the destructor call generated at the end of the lifetime of a non-temporary object

Module 3

- an expression that is not part of another full-expression (such as the entire expression statement, controlling expression of a for/while loop, conditional expression of if/switch, the expression in a return statement, etc), including implicit conversions applied to the result of the expression, destructor calls to the temporaries, default member initializers (when initializing aggregates), and every other language construct that involves a function call, is sequenced before each value computation and side effect of the next full expression.

1. The value computations (but not the side-effects) of the operands to any operator are sequenced before the value computation of the result of the operator (but not its side effects).
2. When calling a function (whether or not the function is inline, and whether or not explicit function call syntax is used), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function.
3. The value computation of the built-in post-increment and post-decrement operators is sequenced before its side-effect.
4. The side effect of the built-in post-increment and post-decrement operators is sequenced before its value computation (implicit rule due to definition as compound assignment)
5. Every value computation and side effect of the first (left) argument of the built in logical AND operator && and the built-in logical OR operator || is sequenced before every value computation and side effect of the second (right) argument.
6. Every value computation and side effect associated with the first expression in the conditional?: is sequenced before every value computation and side effect associated with the second or third expression.
7. The side effect (modification of the left argument) of the built-in assignment operator and of all built-in compound assignment operators is sequenced after the value computation (but not the side effects) of both left and right arguments, and is sequenced before the value computation of the assignment expression (that is, before returning the reference to the modified object).
8. Every value computation and side effect of the first (left) argument of the built-in comma operator is sequenced before every value computation and side effect of the second (right) argument.
9. In list-initialization, every value computation and side effect of a given initializer clause is sequenced before every value computation and side effect associated with any initializer clause that follows it in the brace-enclosed comma-separated list of initalizers.

10. A function call that is not sequenced before or sequenced after another function call is indeterminately sequenced
11. The call to the allocation function (operator new) is indeterminately sequenced with respect to sequenced before the evaluation of the constructor arguments in a new-expression
12. When returning from a function, copy-initialization of the temporary that is the result of evaluating the function call is sequenced-before the destruction of all temporaries at the end of the operand of the return statement, which, in turn, is sequenced-before the destruction of local variables of the block enclosing the return statement.
13. In a function-call expression, the expression that names the function is sequenced before every argument expression and every default argument.
14. In a function call, value computations and side effects of the initialization of every parameter are indeterminately sequenced with respect to value computations and side effects of any other parameter.
15. Every overloaded operator obeys the sequencing rules of the built-in operator it overloads when called using operator notation.
16. In a subscript expression E1[E2], every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2
17. In a pointer-to-member expression E1.*E2 or E1->*E2, every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2 (unless the dynamic type of E1 does not contain the member to which E2 refers).
18. In a shift operator expression E1<<E2 and E1>>E2, every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2.
19. In every simple assignment expression E1=E2 and every compound assignment expression E1@=E2, every value computation and side-effect of E2 is sequenced before every value computation and side effect of E1
20. Every expression in a comma-separated list of expressions in a parenthesized initializer is evaluated as if for a function call (indeterminately-sequenced)

Module 3

3.3.2 Undefined Behavior

1. If a side effect on a scalar object is unsequenced relative to another side effect on the same scalar object, the behavior is undefined.

```
i = ++i + 2;     // undefined behavior
i = i++ + 2;     // undefined behavior
```

```
f(i = -2, i = -2); // undefined behavior

f(++i, ++i);   // undefined behavior
i = ++i + i++;   // undefined behavior
```

2. If a side effect on a scalar object is unsequenced relative to a value computation using the value of the same scalar object, the behavior is undefined

```
cout << i << i++; // undefined behavior
a[i] = i++;     // undefined behavior
n = ++i + i;    // undefined behavior.
```

3.3.3 Sequenced Before Rule

- Evaluation of an expression might produce side effects, which are: accessing an object designated by a volatile lvalue, modifying an object, calling a library I/O function, or calling a function that does any of those operations.
- A sequence point is a point in the execution sequence where all side effects from the previous evaluations in the sequence are complete, and no side effects of the subsequent evaluations started.

Rules

1. There is a sequence point at the end of each full expression (typically, at the semicolon).
2. When calling a function (whether or not the function is inline and whether or not function call syntax was used), there is a sequence point after the evaluation of all function arguments (if any) which takes place before execution of any expressions or statements in the function body.
3. There is a sequence point after the copying of a returned value of a function and before the execution of any expressions outside the function.
4. Once the execution of a function begins, no expressions from the calling function are evaluated until execution of the called function has completed (functions cannot be interleaved).
5. In the evaluation of each of the following four expressions, using the built-in (non-overloaded) operators, there is a sequence point after the evaluation of the expression a.

```
a && b
a || b
a ? b : c
a , b
```

3.3.4 Undefined behavior

1. Between the previous and next sequence point a scalar object must have its stored value modified at most once by the evaluation of an expression, otherwise the behavior is undefined,

```
i = ++i + i++;          // undefined behavior
i = i++ + 1;            // undefined behavior
i = ++i + 1;            // undefined behavior
++ ++i;                 // undefined behavior
f(++i, ++i);            // undefined behavior
f(i = -1, i = -1);      // undefined behavior
```

2. Between the previous and next sequence point, the prior value of a scalar object that is modified by the evaluation of the expression, must be accessed only to determine the value to be stored. If it is accessed in any other way, the behavior is undefined.

Module 3

```
cout << i << i++;       // undefined behavior
a[i] = i++;             // undefined behavior
```

3.4 HIGHER ORDER FUNCTIONS

3.4.1 Introduction

GQ. 3.4.1 What is Higher Order Functions? Discuss it in details with its advantages.

- Generally while programming, we use first class functions which means that the programming language treats functions as values – that you can assign a function into a variable, pass it around. These functions do not take other functions as parameters and never has any function as their return type. To overcome all these shortcomings of first-class functions, the concept of Higher Order function was introduced.
- In Imperative programming languages like (C/C++), we use functions very often or actually we can say that functions play a major role when programming in these languages. A typical function can be defined by function name, it's return type and parameters it takes. We usually give int, char, pointer etc as parameter but is it possible to give one function as parameter to other function or can a function return another function as it's result? The answer is yes!
- The functions which take at least one function as parameter or returns a function as it results or performs both is called Higher Order Function.

- Many languages including- Javascript, Go, Haskell, Python, C++, C# etc, supports Higher Order Function. It is a great tool when it comes to functional programming.
- A function is said to be a *higher-order function* (also called a *functional form*) and it is a function that follows at least one of the following conditions
  - Takes on or more functions as argument
  - Returns a function as its result
- The Scheme version of map is slightly more general. Like for-each, it takes as argument a function and a *sequence* of lists. There must be as many lists as the function takes arguments, and the lists must all be of the same length. Map calls its function argument on corresponding sets of elements from the lists:

  (map * '(2 4 6) '(3 5 7)) ⟹ (6 20 42)

- Where for-each is executed for its side effects, and has an implementation dependent return value, map is purely functional: it returns a list composed of the values returned by its function argument.

Programmers in Scheme (or in ML, Haskell, or other functional languages) can easily define other higher-order functions. Suppose, for example, that we want to be able to "fold" the elements of a list together, using an associative binary operator :

```
(define fold (lambda (f i l)
(if (null? l) i ; i is commonly the identity element for f
(f (car l) (fold f i (cdr l))))))
```

- Now (fold + 0 '(1 2 3 4 5)) gives us the sum of the first five natural numbers, and (fold * 1 '(1 2 3 4 5)) gives us their product.
- One of the most common uses of higher-order functions is to build new functions from existing ones:

  (define total (lambda (l) (fold + 0 l)))

```
(total '(1 2 3 4 5)) ⟹ 15
(define total-all (lambda (l)
(map total l)))
(total-all '((1 2 3 4 5)
(2 4 6 8 10)
```

(3 6 9 12 15))) ⟹ (15 30 45)

(define make-double (lambda (f) (lambda (x) (f x x))))

(define twice (make-double +))

(define square (make-double *))

3.4.2 Currying

GQ. 3.4.2 What is Currying? Explain it with examples.

- Functions can be classified on the basis of the number of inputs they accept like binary function will take two inputs while a unary function will take only a single input.
- In Haskell, every function takes only one input that is every function in Haskell can be said unary.
- Then it is not possible to implement a function which take multiple parameters? Of course, it is possible, by a methodology called currying (named after Haskell Curry, scientist who made this methodology popular and invented Haskell).
- In currying every function takes only one argument and returns a function.
- While the last function in this series will return the desired output.
- Currying is the methodology of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

Module 3

A Simple Example of Currying:

Let's take an example, PLUS is a function which adds two number

1. We wish to add two numbers X and Y. X will be the input to PLUS function which returns a function named PLUS X.
2. PLUS X function takes one number and add X to it. Now input to this function will be Y. Final output will be X + Y.

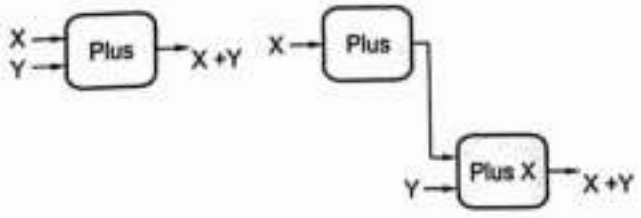


Fig. 3.4.1

3.4.3 Advantages of Higher Order Functions

- By use of higher order function, we can solve many problems easily.
- For example we need to build a function which takes a list and another function as it's input, applies that function to each element of that list and returns the new list.
- In Haskell, this could be done really easily using the inbuilt higher order function called map. Definition of map is :

```
map :: (a -> b) -> [a] -> [b]

map _ [ ] = [ ]

map f (x : xs) = f x : map f xs
```

Here,

- The first line is function initialisation.
- The :: symbol stands for "is of the type".
- **[a]** represents a list of similar element, entity written after last -> is always the return type of the function.A function in Haskell always return only one entity.
- **(a->b)** defines a function from 'a' to 'b'. We used recurrence to define map, [] denotes empty list and _ denotes "anything".
- Second line depicts that if empty list and any function is input then the output will be empty list.
- **x : xs** is used to take out elements one by one from list, x is the first element (head) and xs is remaining list (tail). : sign stands for concatenation. So in a nutshell, the third line is taking each element from the list and applying function 'f' on them and concatenating it with remaining list.

Example

map (+7) [1, 2, 3, 4, 5]

will return the list [8, 9, 10, 11, 12]. Here +7 is the function.

3.5 I/O- STREAMS AND MONADS

GQ. 3.5.1 What are the I/O –streams and monads? Discuss.

- **A major source of side effects can be found in traditional I/O :** an input routine will generally return a different value every time it is called, and multiple calls to an output routine, though they never return a value, must occur in the proper order if the program is to be considered correct.
- **One way to avoid these side effects is to model input and output as *streams* -** unbounded-length lists whose elements are generated lazily. Lists that obey lazy evaluation rules are often call streams.
- This name is suggestive of the infinite nature of list.
- A stream cab be through of as a partially computed list whose remaining elements can continue to be computed up to any desired number. Streams are an important mechanism in functional programming.
- To eliminate side effects completely, input and output must be introduced into functional languages as streams.
- Note that both Scheme and ML have adhoc stream constructions (besides the manual delay and force procedures mentioned earlier).
- More recent versions of Haskell employ a more general concept known as *monads*.
- Monads are drawn from a branch of mathematics known as *category theory*, but one doesn't need to understand the theory to appreciate their usefulness in practice.
- In Haskell, monads are essentially a clever use of higher-order functions, coupled with a bit of syntactic sugar, that allow the programmer to chain together a sequence of *actions* (function calls) that have to happen in order.
- The power of the idea comes from the ability to carry a hidden, structured value of arbitrary complexity from one action to the next.
- In many applications of monads, this extra hidden value plays the role of mutable state: differences between the values carried to successive actions act as side effects.
- Monads provide a more general solution to the problem of threading mutable state through a functional program. Following is the example use Haskell's standard IO monad, which includes a random number generator:

Module 3

```
twoMoreRandomInts :: IO [Integer]
-- twoMoreRandomInts returns a list of Integers. It also
-- implicitly accepts, and returns, all the state of the IO monad.
twoMoreRandomInts = do
rand1 <- randomIO
rand2 <- randomIO
return [rand1, rand2]
main = do
moreInts <- twoMoreRandomInts
print moreInts
(define output (squares input)))
```

- Streams formed the basis of the I/O system in early versions of Haskell.
- Unfortunately, while they successfully encapsulate the imperative nature of interaction at a terminal, streams don't work very well for graphics or random access to files.
- They also make it difficult to accommodate I/O of different kinds (since all elements of a list in Haskell must be of a single type).
- In most Haskell monads, hidden state can be explicitly extracted and examined.
- The IO monad, however, is abstract: only part of its state is defined in library header files; the rest is implemented by the language run-time system

Chapter Ends...

□□□

4.1 LOGIC PROGRAMMING WITH PROLOG

4.1.1 Logic Programming

GQ. 4.1.1 What is Logic Programming? Discuss its features also.

- Logic Programming is the name given to a distinctive style of programming, very different from that of conventional programming languages such as C++ and Java. As per programmer point of view Logic Programming is 'different' means clearer, simpler and generally better. Logic programming, allows a programmer to describe the logical structure of a problem rather than prescribe how a computer is to go about solving it. Based on their essential properties, languages for logic programming are sometimes called:

1. **Descriptive or Declarative Languages :** Programs are expressed as known facts and logical relationships about a problem that hypothesize the existence of the desired result; a logic interpreter then constructs the desired result by making inferences to prove its existence.

2. **Nonprocedural Languages :** The programmer states only what is to be accomplished and leaves it to the interpreter to determine how it is to be proved.

3. **Relational Languages :** Desired results are expressed as relations or predicates instead of as functions; rather than define a function for calculating the square of a number, the programmer defines a relation, say sqr(x,y), that is true exactly when y = x2. Imperative programming languages have a descriptive component, namely expressions: "3*p + 2*q" is a description of a value, not a sequence of computer operations; the compiler and the run-time system handle the details. High-level imperative languages, like Pascal, are easier to use than assembly languages because they are more descriptive and less prescriptive.

Module 4

- The goal of logic programming is for languages to be purely descriptive, specifying only what a program computes and not how. Correct programs will be easier to develop because the program statements will be logical descriptions of the problem itself and not of the execution process—the assumptions made about the problem will be directly apparent from the program text.

- Although there are other Logic Programming languages, by far the most widely used is Prolog, programs are written in the language of some logic. The name stands for Programming in Logic. Prolog is based on research by computer scientists in Europe in the 1960s and 1970s, notably at the Universities of Marseilles, London and Edinburgh. The first implementation was at the University of Marseilles in the early 1970s. Further development at the University of Edinburgh led to a de facto standard version, now known as Edinburgh Prolog. Prolog has been widely used for developing complex applications, especially in the

field of Artificial Intelligence. Although it is a general-purpose language, its main strengths are for symbolic rather than for numerical computation. Prolog (PROgramming in LOGic) is a representative logic language.

4.1.2 Advantages and Disadvantages of Logical Programming Paradigm

GQ. 4.1.2 List the advantages and limitations of Logic programming.

- Logical Languages use mathematical logic as a way to solve problems. Within Logical Languages, the programmer ensures that the choice declarative statements Logic Programming sets boundaries or constraints with which the program can 'reason' in order to an output.

Advantages of Logic Programming

The advantages of Logic Languages include :

- Logic programming proves that the validity of a given program is very simple and it appears that the system itself solve the problem because there is a minimum keeping programming steps.
- Logical Languages are quite reliable.
- Best suited for problems in which knowledge base can be established to come to a solution.
- Programs can be quickly developed as it uses true/false statements, rather than objects, etc.
- Logic programming can be used to express knowledge in a way that does not depend on its implementation, making programs more flexible, compressed and understandable.
- It enables knowledge to be separated from use, in the machine architecture can be changed without changing programs or their underlying code.
- It can be altered and extended in natural ways to support special forms of knowledge, such as meta-level or higher-order knowledge.
- It can be used in non-computational disciplines relying on reasoning and precise usage of expression.

Disadvantages of Logic Programming

The disadvantages of Logic Languages include :

- There is no suitable method of representing computational concepts originate in a built-in mechanism of state variables like it is found in conventional languages.
- Limited to which types of problems it can efficiently solve.

- The programs execution can be slow.
- True/false statements cannot solve most problems—at all.
- Initially, due to insufficient investment in complementary technologies, users were poorly served.
- In the beginning, poor facilities for supporting arithmetic, types, etc had a discouraging effect on the programming community.
- There is no adequate way of representing computational concepts found in built-in mechanisms of state variables (as is usually found in conventional languages).
- Some programmers always have, and always will prefer the overtly operational nature of machine operated programs, since they prefer the active control over the 'moving parts'.

4.1.3 PROLOG

GQ. 4.1.3 What is PROLOG? Write its syntax.

- PROLOG (PROgramming in LOGic) is a *logic programming* language. It has important role in artificial intelligence. Unlike many other programming languages, Prolog is intended primarily as a declarative programming language. It is based on the first-order predicate calculus. When you run a PROLOG program, a PROLOG interpreter systematically makes inferences from a set of *facts* and *rules* specified in *Horn* clause notation, a notation equivalent to first order predicate calculus. Core heart of prolog lies at the **logic** being applied. Formulation or Computation is carried out by running a query over these relations.

Module 4

- PROLOG is based on a proof procedure known as *resolution theorem proving*. Prolog implements a subset of predicate logic using the Resolution Principle, an efficient proof procedure for predicate logic developed by Alan Robinson. The first interpreter was written by Alain Colmerauer and Philippe Roussel at Marseilles, France, in 1972.

- PROLOG uses a *declarative programming* approach. Rather than describing *how* to do something, as is done in a procedural programming language (such as C), a PROLOG program describes *what* to do.

- The basic features of Prolog include a powerful pattern-matching facility, a backtracking strategy that searches for proofs, uniform data structures from which programs are built, and the general interchangeability of input and output.

- The Basic construct of logic programming term and statement are inherited from logic. There are three basic statements,

1. **Facts**: Facts are fundamental assertion about the problem domain, Facts are represented in the Prolog data by predicate clauses of the form.

2. **Rules:** A Prolog rule is a general statement about objects and their relationships. Rules are inferences about facts in the domain
3. **Queries :** Queries are question about the domain.

- Prolog program consists of a collection of facts and rules defined to constrain the Prolog interpreter in such a way that when we submit a query, the resulting answers solve the problems at hand. Facts, rules, and queries can all be entered interactively, but usually a Prolog programmer creates a file containing the facts and rules, and then after loading this file, enters only the queries interactively.

4.1.4 Prolog Syntax

- Prolog programs are constructed from terms that are **constants, variables,** or **structures**.
- Constants can be either atoms or numbers.
- **Atoms** are strings of characters starting with a lowercase letter or enclosed in apostrophes.
- **Numbers** are strings of digits with or without a decimal point and a minus sign.
- **Variables** are strings of characters beginning with an uppercase letter or an underscore.
- **Structures** consist of a functor or function symbol , which looks like an atom, followed by a list of terms inside parentheses, separated by commas.

4.1.5 Characteristics of the Prolog programming language

The main characteristics/features of the Prolog programming language are:

- It has based on logical programming with Horn clauses.
- Fully object oriented.
- object predicate values (delegates)
- Strongly typed.
- It has algebraic data types.
- Pattern matching and unification.
- Controlled non-determinism.
- Fully integrated fact databases

4.1.6 Data Objects in Prolog

Aim Discuss data objects in Prolog.

- Prolog Terms The data objects in Prolog are called terms. There are several different types of terms, which are listed below.

or (1) Numbers

- All versions of Prolog allow the use of integers (whole numbers). They are written as any sequence of numerals from 0 to 9, optionally preceded by a + or - sign, for example:

```
427
-67
+1
365
```

- Most versions of Prolog also allow the use of numbers with decimal points. They are written in the same way as integers, but contain a single decimal point, anywhere except before an optional + or - sign, e.g.

```
6.02
.141
+256.
```

or (2) Atoms

- Atoms are constants that do not have numerical values. There are three ways in which atoms can be written.

(a) Any sequence of one or more letters (upper or lower case), numerals and underscores, beginning with a lower case letter, e.g.

```
john
peter_a_Tuesday
x23_BCD
```

(b) Any sequence of characters enclosed in single quotes, including spaces and

```
more than letters, e.g.
'Today is Tuesday'
'John is 7 today'
'23abc'
```

Module 4

(c) Any sequence of one or more special characters from a list that includes the following + - * / > < = & # @ :

Examples

+++

>=

>

+-

(3) Variables

- In a query a variable is a name used to stand for a term that is to be determined, e.g. variable X may stand for atom dog, the number 12.3, or a compound term or a list.
- The name of a variable is denoted by any sequence of one or more letters (upper or lower case), numerals and underscores, beginning with an upper case letter or underscore, e.g.

X

Author

Person_A

_123A

Note : The variable _ which consists of just a single underscore is known as the anonymous variable and is reserved for a special purpose.

(4) Compound Terms

- Compound terms are of fundamental importance in writing Prolog programs.
- A compound term is a structured data type that begins with an atom, known here as a functor.
- The functor is followed by a sequence of one or more arguments, which are enclosed in brackets and separated by commas.
- The general form is functor(t1,t2, ... ,tn) n ≥ 1 If you are familiar with other programming languages, you may find it helpful to think of a compound term as representing a record structure.
- The functor represents the name of the record, while the arguments represent the record fields. The number of arguments a compound term has is called its arity.

Some examples of compound terms are:

likes(paul,prolog)

read(X)

dog(henry)

cat(X)

>(3,2)

person('john smith',32,doctor,london)

- Each argument of a compound term must be a term, which can be of any kind including a compound term. Thus some more complex examples of compound terms are:

likes(dog(henry),Y)

pred3(alpha,beta,gamma,Q)

pred(A,B,likes(X,Y),-4,pred2(3,pred3(alpha,beta,gamma,Q)))

(5) Lists

- A list is often considered to be a special type of compound term, but in this book it will be treated as a separate type of data object. Lists are written as an unlimited number of arguments (known as list elements) enclosed in square brackets and separated by commas, e.g. [dog,cat,fish,man].
- A list with no elements is known as the empty list. It is written as [].

4.1.7 Advantages of Prolog

UQ. 4.1.3 Write advantages of Prolog.

1. Easy to build database. Doesn't need a lot of programming effort.
2. Pattern matching is easy. Search is recursion based.
3. It has built in list handling. Makes it easier to play with any algorithm involving lists.

4.1.8 Disadvantages of Prolog

UQ. 4.1.4 What are the disadvantages of Prolog.

1. LISP (another logic programming language) dominates over prolog with respect to I/O features.
2. Sometimes input and output is not easy.
3. There are some features which have not been standardized, and differ between implementations. For example formatted input and output, file-handling, sorting predicates.

Module 4

4. You can't re-assign to parts of data structures. This makes it impossible to implement arrays. However, functional programmers have developed a number of fast-access data structures which do almost as good a job

4.1.9 Applications of Prolog

GQ. 4.1.7 List out Applications of Prolog.

- The main applications of the language can be found in the area of Artificial Intelligence, for PROLOG is being used in other areas in which symbol manipulation is of prime importance as well. Some application areas are:
  - Natural-language processing;
  - Compiler construction;
  - The development of expert systems;
  - Work in the area of computer algebra;
  - The development of (parallel) computer architectures;
  - Database systems
  - Prolog is highly used in artificial intelligence (AI).
  - Prolog is also used for pattern matching over natural language parse trees.
  - Modern Prolog environments support the creation of graphical user interfaces, as well as administrative and networked applications.
  - Prolog is well-suited for specific tasks that benefit from rule-based logical queries such as searching databases, voice control systems, and filling templates.

4.2 RESOLUTION AND UNIFICATION

GQ. 4.2.1 What is Resolution and Unification? Explain it in details.

- Prolog is a particularly simple programming language. It consists of a single data structure and a single operation on those data structures called unification.
- The deductive calculus that it implements is equally simple. There is one inference rule (resolution) and no axioms.
- Despite all this it is an extremely powerful programming language.
- It can be used to implement Turing Machines, or theorem provers for full First-Order Logic or Prolog itself.

4.2.1 Resolution

GQ. 4.2.2 What is Resolution? State the resolution principle.

- In simple words resolution is inference mechanism. Let's say we have clauses a :- b. and t :- p, m, r. So from that we can infer t :- p, b, r. - that is called resolution. Means, when you resolve two clauses you get one new clause.
- Another easy example, we have two sentences 1)) All women like shopping. 2)) Olivia is a woman. Now we ask query 'Who likes shopping'. So, by resolving above sentences we can have one new sentence Olivia likes shopping.
- Prolog execution is based on the Resolution proof method. Resolution is a technique of producing a new clause by resolving two clauses. Resolution involves combining information from separate rules. The fundamental operation of a resolution system takes a pair of clauses as input and produces a new clause, called their resolvent as output.
- The general idea:

  If we have rules of the form

  A :- B.

  C :- D.

  where A, B, C, D are expressions, and if B and C unify, say to U, then we have

  A :- U.

  C :- D.

  and hence we have

  A :- D.

- That is, to prove A, it suffices to prove D.
- So given a query Q, Prolog iterates through its rules until it finds one whose left side unifies with Q, and then it tries to prove such sub-query given by the right side of the rule.
- For example, given the two rules above, if query Q unifies with A, then Prolog will try to prove B (by, again, iterating through its rules and facts, looking for one whose left side unifies with B).

Module 4

4.2.2 Resolution Principle

The principle of resolution in propositional logic can be best described by the following theorem :

Resolution Theorem

For any three clauses p, q and r,

$p \vee r, \overline{q} \vee \neg r \quad p \vee q$

- **Proof :** The theorem, which can be proved by Wang's algorithm, is left as an exercise for the students.
- The resolution theorem can also be used for theorem proving and hence reasoning in propositional logic. The following steps should be carried-out in sequence to employ it for theorem proving. Resolution is a procedure used in proving that arguments which are expressible in predicate logic are correct.
- Resolution is a procedure that produces proofs by refutation or contradiction.
- Resolution leads to refute a theorem-proving technique for sentences in propositional logic and first-order logic, hence.
  1. Resolution is a rule of inference.
  2. Resolution is a computerized theorem prover.
  3. Resolution is so far only defined for Propositional Logic. The strategy is that the Resolution techniques of Propositional logic be adapted in Predicate Logic.
- Robinson in 1965 introduced the resolution principle. Which can be directly applied to any set of clauses. The principal is "Given any two clauses A and B, if there is a literal P1 in A which has a complementary literal P2 in B, delete P1 & P2 from A and B and construct a disjunction of the remaining clauses. The clause so constructed is called resolvent of A and B."

For example, consider the following clauses

A: P ∨ Q ∨ R

B: P' ∨ Q ∨ R

C: Q' ∨ R

- Clause A has the literal P which is complementary to P' in B Hence both of them deleted and a resolvent (disjunction of A and B after the complementary clauses are removed) is generated.

That resolvent has again a literal Q whose negation is available in C. Hence resolving these two, one has the final resolvent.

A: P ∨ Q ∨ R (given in the problem)

B: P' ∨ Q ∨ (given in the problem)

D: Q ∨ R (resolvent of A and B)

C: Q' ∨ R (given in the problem)

F: R (resolvent of C and D)

4.2.3 Unification

MQ. 4.2.4 Explain Unification in details also write its rules.

Unification is the process of matching two expressions by attempting to construct a set of bindings for the variables so that when the bindings are applied to the two expressions, they become syntactically equal.

The way in which Prolog matches two terms is called **unification**. If two expressions can be unified, then Prolog will return with corresponding bindings for any variables that occur in the expressions.

Unification is used as part of Prolog's computation rule. The following symbol is used for unification within a program.

=

- The idea is similar to that of unification in logic: we have two terms and we want to see if they can be made to represent the same structure.
- Unification succeeds if two expressions can be made to have "the same structure" through variable instantiation (giving a variable a value).
- An instantiated variable will not change its value. However, it can become uninstantiated when backtracking occurs.

Unification examples

- parent(X,Y) and parent(albert, edward)

  These unify: X=albert, Y=edward

- parent(X,edward) and parent(albert,Y)

  These unify: X=albert, Y=edward

- – parent(albert,edward) and parent(victoria,Y)
- These do not unify, since albert and victoria don't unify.
- – parent(X, edward) and parent(Y, edward)
- These unify: X=Y (or Y=X, or creating a variable Z and letting X=Z and Y=Z).

More unification examples:

- – parent(X,Y) and female(X)
- These do not unify, since parent and female don't unify.
- – parent(albert,edward) and parent(X,Y,Z)
- These do not unify, since the first expression has 2 arguments and the second expression has 3 arguments.
- Observe that in order for two expressions to unify, they must have the same functor and the same number of arguments (or else there's no way for them to have the same structure).

4.2.4. Unification Rules

GQ. 4.2.4 State the unification rules for Prolog.

The unification rules for Prolog state that

- A constant unifies only with itself.
- Two structures unify if and only if they have the same functor and the same arity, and the corresponding arguments unify recursively.
- A variable unifies with anything. If the other thing has a value, then the variable is instantiated. If the other thing is an uninstantiated variable, then the two variables are associated in such a way that if either is given a value later, that value will be shared by both.

4.3 LISTS

GQ. 4.3.1 Explain a Prolog List. How list can be splited.

- Lists are a special class of compound term. Lists are data structures essential to many programs. A Prolog list is a sequence of an arbitrary number of terms separated by commas and enclosed within square brackets. For example:
  - [a] is a list made of an atom.
  - [a, b] is a list made of two atoms.

- [a, X, father(X, telemachus)] is a list made of an atom, a variable, and a compound term.
- [[a, b], [[father(X, telemachus)]]] is a list made of two sublists.
- [] is the atom representing the empty list.

Although it is not obvious from these examples, Prolog lists are compound terms and the square bracketed notation is only a shortcut.

The list functor is a dot: "./2", and [a, b] is equivalent to the term .(a, .(b, [])). Computationally, lists are recursive structures. They consist of two parts: a head, the first element of a list, and a tail, the remaining list without its first element.

The head and the tail correspond to the first and second argument of the Prolog list Functor. Following figure shows the term structure of the list [a, b, c]. The tail of a list is possibly empty as in .(c, []). The notation "|" splits a list into its head and tail, and [H | T] is equivalent to .(H, T).

Splitting a list enables us to access any element of it and therefore it is a very frequent operation. Here are some examples of its use:

```
[1|T] = [1].
[1|T] = [1,2].
[1|T] = [1,2,3].
[1|T] = [1].
[1,2|T] = [1,2,3].
[1,2|T] = [1,2].
[X,Y|T] = [a,b,c,d].
[X,Y|T] = [a,b].
[H|T] = [a].
[H|T] = [a,b,c].
The query [] = [H|T] cannot be split.
[1|T] = [X|T].
Etc.
```

Fig. 4.3.1

4.4 ARITHMETIC EXECUTION ORDER

4.4.1 Operator Used in Prologs

GQ. 4.4.1 Describe the operators used in Prolog.
GQ. 4.4.2 What are Arithmetic Operators in Prolog?

- **Arithmetic Operators**
- In arithmetic expressions, + - * / symbols are special type of infix operator, and these operators are also known as arithmetic operators. In Prolog, operators are used as predicates but here operators are functions and these operators return a numerical value.
- Arithmetic expressions can include variables, numbers, operators, and arithmetic functions. These will be written in parentheses with their arguments. These will return numerical values just like the arithmetic operators.

Many of the usual arithmetic operators are available in Prolog :

Operator	Meaning	Example
+	addition	2 is 1 + 1.
-	subtraction	1 is 2 - 1.
-	unary minus	Try the query X is 1, Y is - X.
*	multiplication	4 is 2 * 2.
/	division	2 is 6 / 3
//	integer division	1 is 7 // 4.
mod	integer remainder	3 is 7 mod 4.
**	exponentiation	1.21 is 1.1 ** 2.

- Except in the context of an arithmetic comparison operator, arithmetic expressions need to be explicitly evaluated in Prolog, using the is built-in predicate.

Mostly one uses these operators in a goal more like this :

```
X is Y + 2
```

where Y is a variable already bound to a numeric value, or perhaps a arithmetic comparison goal like this:

```
X > Y * Z
```


Here's another example: a rule to tell if a (whole) number is odd:

```
odd(Num) :- Num is Num mod 2.
Another way to do this one is to use =:=:
odd(Num) :- Num mod 2 =:= 1.
If you suspect it be more cautious, you could first check that Num is in fact a whole number:
odd(Num) :- integer(Num), 1 is Num mod 2.
```

Prolog defines a set of prefix, infix, and postfix operators that includes the classical arithmetic symbols: "+", "-", "*", and "/". The Prolog interpreter considers operators as functions and transforms expressions into terms. Thus, 2*3+4*2 is equivalent to +(*(2,3), *(4, 2)).

The mapping of operators onto terms is governed by rules of priority and classes of associativity:

1. The priority of an operator is an integer ranging from 1 to 1200. It enables us to determine recursively the principal functor of a term. Higher-priority operators will be higher in the tree representing a term.
2. The associativity determines the bracketing of term A op B op C :
   (a) If op is left-associative, the term is read (A op B) op C.
   (b) If op is right-associative, the term is read A op (B op C).

Prolog defines an operator by its name, its specifier, and its priority. The specifier is a mnemonic to denote the operator class of associativity and whether it is infixed, prefixed, or postfixed.

Table 4.4.1 : shows the priority and specifier of predefined operators in Standard Prolog.

Table 4.4.1 : Operator Specifiers

Operator	Nonassociative	Right-associative	Left-associative
Infix	Xfx	Xfy	Yfx
Prefix	Fx	Fy	-
Postfix	Xf	-	yf

It is possible to declare new operators using the directive:

```
:- op(+Precedence, +Specifier, +Name).
```

Module 4

Table 4.4.2 : Priority and Specifier of operator in Standard Prolog

Precedence.	Type	Operator
1200	xfx	-->, :-
1200	fx	:-, ?-
1150	fx	dynamic, discontiguous, initialization, module_transparent, multifile, thread_local, volatile
1100	xfy	; |
1050	xfy	->, ;*->
1000	xfy	,
954	xfy	\
900	fy	\+
900	fx	-
700	xfx	<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is
600	xfy	:
500	yfx	+, -, /\, \/, xor
500	fx	+, -, ?, \
400	yfx	*, /, //, rdiv, <<, >>, mod, rem
200	xfx	**
200	xfy	^

4.4.2 Evaluation of Arithmetic Operations

GQ. 4.4.3 Write details note on evaluation of arithmetic operation in Prolog.

- The evaluation of an arithmetic expression uses the is/2 built-in operator is/2 computes the value of the Expression to the right of it and unifies it with Value:
- ?– value is Expression.
- Where Expression must be computable. Let us exemplify it. Recall first that "=" does not evaluate the arithmetic expression:

  ?-X = 1+1+1,

  X = 1+1+1 (or s = + (s (1, 1), 1)).
- To get a value, it is necessary to use is

  ?- X = 1+1+1+1, Y is X.

X= 1+1+1, Y = 3.

- If the arithmetic expression is not valid is/2 returns an error, as in:

  ?- X is 1+1+ a.

  Error

  Because a is not a number, or as in:

  ?- a is 1+1+Z.

  Error

  Because Z is not instantiated to a number. But

  ?- Z= 2, x is 1+1+Z.

  Z= 2, X=4

  is correct because Z has a numerical value when X is evaluated.

4.5 IMPERATIVE CONTROL FLOW

GQ. 4.5.1 Write a note on Imperative control flow.

- The ordering of clauses and of terms in Prolog is significant, with ramifications for efficiency, termination, and choice among alternatives.
- In addition to simple ordering, Prolog provides the programmer with several explicit control-flow features. The most important of these features is known as the cut.
- The cut is a zero-argument predicate written as an exclamation point: !. As a subgoal it always succeeds, but with a crucial side effect: it commits the interpreter to whatever choices have been made since unifying the parent goal with the left hand side of the current rule, including the choice of that unification itself.

  For example, recall our definition of list membership:

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
```

- If a given atom a appears in list L n times, then the goal ?- member(a, L) can succeed n times.
- These "extra" successes may not always be appropriate.
- They can lead to wasted computation, particularly for long lists, when member is followed by a goal that may fail :

```
prime_candidate(X) :- member(X, candidates), prime(X).
```

- Suppose that prime(X) is expensive to compute. To determine whether a is a prime candidate, we first check to see whether it is a member of the candidates list, and then check to see whether it is prime. If prime(a) fails, Prolog will backtrack and attempt to satisfy member(a, candidates) again. If a is in the candidates list more than once, then the subgoal will succeed again, leading to
- reconsideration of the prime(a) subgoal, even though that subgoal is doomed to fail. We can save substantial time by cutting off all further searches for a after the first is found:

```
member(X, [X | _]) :- !.
member(X, [_ | T]) :- member(X, T).
```

- The cut on the right-hand side of the first rule says that if X is the head of L, we should not attempt to unify member(X, L) with the left-hand side of the second rule; the cut commits us to the first rule.

4.6 DATABASE MANIPULATION

DQ. 4.6.1 What is Database Manipulation? Explain with the help of example.

- There is one key decision that we have to make before starting to implement the recognizer, and that is how we are going to represent the chart, and how we are going to work with it.
- Now, there are various options here. For example, one can represent the chart using some sort of structured list.
- But there is another, more attractive, way to go: represent the chart using the Prolog database, and manipulate the chart (that is, add new edges) using the database manipulation operations. As we are going to be making heavy use of Prolog's database manipulations, let's quickly run through them and remember how they worked.

There are five database manipulation commands that we will using :

```
assert
retract
asserta
assertz
retractall
```

- How are these used? Suppose we start with an empty database. So if we give the command:

```
?- listing
yes
```


we get a yes — and the listing (of course) is empty.

Suppose we now give this command:

```
?- assert(happy(mia)).
yes
```

- It succeeds (the assert commands always succeed). But what is important is not that it succeeds, but the side effect it has on the database, for if we now give the command:

```
?- listing
happy(mia).
yes
```

we get again yes,but now there is something in the database, namely the fact we asserted.

Suppose we then made four more assert commands :

```
?- assert(happy(vincent)).
yes
?- assert(happy(marsellus)).
yes
?- assert(happy(butch)).
yes

?- assert(happy(vincent)).
yes
```

and then we for a listing. Then we get:

```
?- listing.
happy(mia).
happy(vincent).
happy(marsellus).
happy(butch).
happy(vincent).
yes
```

There is an inverse predicate to assert, namely retract. For example, if we go straight on and give the command :

```
?- retract(happy(marsellus)).
yes
```

and then list the database we get:

```
?- listing.

happy(vina).
happy(vincent).
happy(butch).
happy(vincent).
yes
```

Suppose we go on further, and say

```
?- retract(happy(vincent)).
yes
```

and then ask for a listing:

```
?- listing.

happy(vina).
happy(butch).
happy(vincent).
yes
```

Note that the *first* occurrence of happy(vincent) was removed. Prolog removes the first thing in the database that matches the argument of retract.

```
?- retract(happy(_)).
yes
?- listing.
happy(butch).
happy(vincent).
yes
```

The predicate *retractall* removes everything that matches its argument :

```
?- retractall(happy(_)).
yes
?- listing.
yes
```

If we want more control over where the asserted material is placed, there are two variants of assert, namely :

- *assertz*, which places stuff at the *end* of the database, and
- *asserta*, which places stuff at the *beginning* of the database.

Prolog puts some restrictions on what can be asserted and retracted.

More precisely, other predicates are only allowed to remove or retract clauses of predicates that have been declared *dynamic* .

So, if you want to write a predicate which asserts or retracts clauses of the form happy(whom), you must have to put the following statement into your file:

```
:- dynamic happy/1.
```

Database manipulation is useful when we want to store the results of queries, so that if we need to ask the same question in future, we don't need to redo the work, we just look up the asserted fact.

This technique is called 'memoization', or 'caching'. And of course, this 'memoization' idea is the basic concept underlying chart-parsing: we want to store information about the syntactic structure we find, instead of having to re-compute it all the time.

4.7 PROLOG FACILITIES AND DEFICIENCIES

4.7.1 Write details note on PROLOG facilities and deficiencies.

4.7.1 Prolog Facilities

Prolog is a Logic Programming language. Logic languages have been proved adequate for facilities like deductive databases, expert sys- tems, rule-based AI applications, and more generally as general-purpose tools for declarative and non deterministic programming.

Module 4

1. It has important role in artificial intelligence. Unlike many other programming languages, Prolog is intended primarily as a declarative programming language.
2. In prolog, logic is expressed as relations (called as Facts and Rules). Core heart of prolog lies at the logic being applied.
3. Formulation or Computation is carried out by running a query over these relations.
4. Because Prolog is deceptively easy to understand. It's a small language (there are not a lot of constructs to learn) and the basic ideas are beautiful in their simplicity.
5. Modern Prolog environments support the creation of graphical user interfaces.
6. Its also supports administrative and networked applications.
7. Prolog is well-suited for specific tasks that benefit from rule-based logical queries such as searching databases, voice control systems, and filling templates.

4.7.2 Prolog Deficiencies

Prolog is not popular because there isn't much demand for Prolog in and of itself. Prolog has following deficiencies,

1. **Typelessness :** Prolog is untyped. In modern systems, this often becomes 'bolt on the type system you want', but if you're coding in vanilla prolog, you're typeless.
2. **Weak Encapsulation :** Originally Prolog was completely non modular. Modern Prologs have modules, but they still feel like bolt-ons.
3. **Lack of arrays :** Prolog programs sometimes suffer performance penalties for lack of a good O(1) array access structure.
4. **Impurity :** on a more theoretical level, Prolog is a compromise between the purely logical and the practical.
5. **Lack Of Standardization :** There's not just one prolog, there's a whole batch. It's the nature of programming in Prolog, where writing a metainterpreter is a normal thing to do. Fortunately, everyone seems to be focusing in on SWI-Prolog as a base for their experiments, so de-facto SWI is the base language and you build on that.
6. **Weak Organizational Support :** While Jan's amazing, he's only one guy. As SWI-Prolog grows, it's obvious we need more people actually working on the language.
7. **Poor GUI System :** xpce is obsolete and needs a facelift or replacement. This is becoming less of an issue as most UI has moved to the web. Most things that might be written as desktop applications in other languages become localhosted web apps in SWI-Prolog.

Chapter Ends...

□□□

5.0 CONCURRENCY

- In computing, a **process** is the instance of a **computer** program that is being executed by one or many threads. To run a program the operating system must allocate memory and creates a process.
- A processes have stack, heap, data segment and text segment in user space. The CPU context and file descriptor table are kept in kernel space.
- When executing the program, the program counter (PC) jumps around in the text segment.
- It contains the program code and its activity. Depending on the operating system (OS), a **process** may be made up of multiple threads of execution that execute instructions concurrently.
- *Concurrency* means multiple computations are happening at the same time. Concurrency is everywhere in modern programming, whether we like it or not :
  - Multiple computers in a network
  - Multiple applications running on one computer
  - Multiple processors in a computer (today, often multiple processor cores on a single chip)
- In fact, concurrency is essential in modern programming :
  - Web sites must handle multiple simultaneous users.
  - Mobile apps need to do some of their processing on servers ("in the cloud").
  - Graphical user interfaces almost always require background work that does not interrupt the user. For example, Eclipse compiles your Java code while you're still editing it.
- Being able to program with concurrency will still be important in the future. Processor clock speeds are no longer increasing.
- Instead, we're getting more cores with each new generation of chips. So in the future, in order to get a computation to run faster, we'll have to split up a computation into concurrent pieces.
- In computer science, ***concurrency*** refers to the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.
- This allows for parallel execution of the concurrent units, which can significantly improve overall speed of the execution in multi-processor and multi-core systems. In more technical

Module 5

terms, concurrency refers to the decomposability property of a program, algorithm, or problem into order-independent or partially-ordered components or units.

- A *concurrency model* specifies how threads in the system collaborate to complete the tasks they are given. Different concurrency models split the tasks in different ways, and the threads may communicate and collaborate in different ways.
- **Parallel computing** is a *type of computation in which many calculations or the execution of processes are carried out simultaneously.*
- Large problems can often be divided into smaller ones, which can then be solved at the same time.
- There are several different forms of parallel computing: *bit-level*, *instruction level*, *data*, and *task parallelism*.
- Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling.
- As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of *multi-core processors*.
- Parallel computing is closely related to concurrent computing-they are frequently used together, and often conflated, though the two are distinct : it is possible to have parallelism without concurrency (such as bit-level parallelism), and concurrency without parallelism (such as multitasking by time-sharing on a single-core CPU).
- A distributed system is a model in which components located on networked computers communicate and coordinate their actions by passing messages.
- The components interact with each other in order to achieve a common goal.
- **Three significant characteristics of distributed systems are :** concurrency of components, lack of a global clock, and independent failure of components.

or Background and Motivation

- Concurrent is not a new idea. Many theoretical foundations were laid in the 1960s, and Algol 68 included concurrent programming functions.
- However, widespread interest in concurrency is a relatively new phenomenon.
- This is partly due to the availability of low-cost multicore and multiprocessor machines, and partly due to the proliferation of graphics, multimedia, and Web-based applications, all of which are naturally represented by concurrent control threads.
- Concurrency has at least three important motivations :

Q5. Write the Concurrency important motivations.

1. Capture the logical structure of the problem. Many programs, especially servers and graphics applications, must track multiple independent "tasks" simultaneously. In general, the simplest and most logical way to construct such programs is to use a separate thread of control to represent each task.
2. Use additional processors to increase speed. As the main components of high-end servers and supercomputers, a variety of processors have recently been popularized in desktop and workbook computers. To use them effectively, programs must usually be written (or rewritten) with concurrency in mind.
3. Deal with separate physical devices. Applications running across the Internet or local computer groups are inherently concurrent. Similarly, many embedded applications (for example, the control system of a modern car) usually equip a separate processor for each of multiple devices.

Levels of Parallelism

Q6. Explain Levels of Parallelism.

In parallel computing, *granularity* is a qualitative measure of the ratio of computation to communication. Periods of computation are typically separated from periods of communication by synchronization events.

Fine-grained parallelism implies partitioning the application into small amounts of work done leading to a low computation to communication ratio. For example, if we partition a "for" loop into independent parallel computations by unrolling the loop, this would be an example of fine-grained parallelism.

One of the downsides to fine-grained parallelism is that there may be many synchronization points, for example, the compiler will insert synchronization points after each loop iteration that may cause additional overhead.

Also, many loop iterations would have to be parallelized in order to get decent speedup, but the developer has more control over load balancing the application.

Coarse-grained parallelism is where there is a high computation to communication ratio. For example, if we partition an application into several high level tasks that then get allocated to different cores, this would be an example of coarse-grained parallelism.

The advantage of coarse-grained parallelism is that there is more parallel code running at any point in time and there are fewer synchronizations required.

Module 5

- Pipelining can overlap the execution of instructions when they are independent of one another. This potential overlap among instructions is called instruction-level parallelism (ILP) since the instructions can be evaluated in parallel.
- The combination of deep, superscalar pipelines and aggressive speculation allow modern processors to track the dependencies between hundreds of "running" instructions, make progress on dozens of them, and complete multiple instructions in each cycle. Shortly after the turn, it was clear that the limit had been reached for a century: there was no limitation level parallelism available in traditional programs.
- Very long instruction word (VLIW) and hardware-managed superscalar execution both address extraction of independent instruction parallelism from unrelated instructions in an instruction stream. SIMD and vector parallelism directly allow the hardware instructions to target data-parallel execution.
- The lowest level of parallelism we can exploit is vector parallelism through the use of the new AVX-512 vector instructions found on Knights Landing processors.

5.1 LEVELS OF ABSTRACTION

UQ. 5.1.1 Write the short on Abstraction.

- Abstraction means forming simplified conceptual models of problems that enable general or reusable solutions. For example, functions are an abstraction with very general applicability to solving the problem of structuring and reasoning about code. Abstractions with high generality are called rich abstractions.
- Problems in complex domains like computing are solved using many stacked layers of abstraction. Program execution can broadly be decomposed into a computational *model* that is implemented in a *machine model* using an *execution model*, where the machine model is at the bottom of the stack. The layered architecture is relevant to understanding that contradictory or seemingly exclusive concepts can co-exist at different layers of abstraction, like concurrent execution on a serial machine. When the contradictions between an abstraction and the underlying model are exposed and the abstracted-away details can't be ignored, it's called a leak in the abstraction.
- With the spread of thread-level parallelism, different kinds of programmers will need to understand concurrency at different levels of detail, and use it in different ways. The simplest, most abstract case will arise when using "black box" parallel libraries. At the lowest level of abstraction, expert programmers may need to understand the hardware and run-time system in sufficient detail to implement synchronization mechanisms.

There are two common approaches to concurrent programming, multi-threading and multi-processing. Multi-processing is the easiest to comprehend since it just means having multiple instances of a process running to accomplish a task. The is pretty easy to do on UNIX based systems via calls to fork/join, but not so easy on Windows systems.

Multi-threading is probably the approach most people think of when talking about concurrency. It's not difficult to start multiple threads within an application, but the devil is in the details. You need to co-ordinate data sharing between thread (usually using locks) which can lead to deadlock or data in an invalid state. You also need to understand how to communicate between thread using concepts like semaphores, conditional variables etc.

The advantage to all this is that once you understand it you're able to more effectively utilize the underlying hardware. These days it's pretty much the norm for a processor to have multiple cores. By utilizing concurrent programming you can make these cores work for you, and your application will get a speed improvement.

The disadvantage is that you have to start thinking about how you'll split your application up into small parts that can be run on different threads. This is a lot harder than it sounds. Also, highly concurrent solutions can be awkward to unit test as the order of execution is less deterministic.

These days most languages ship with an abstraction over most concurrent primitives to make life a bit easier. For example, .NET 4 ships with the Task Parallel Library which make life a bit easier. So Java land they've got the Concurrency package.

5.2 MULTITHREADED PROGRAMS

UQ. 5.2.1 Explain the Multithreaded program with suitable example.

- A process can have one or more threads of execution. Threads share heap, data segment, text segment and file descriptor table but have separate stacks and CPU contexts. Each thread have a private program counter and threads executes concurrently. Depending on how threads are implemented, the CPU contexts can be stored in user space or kernel space. In the below figure a process with three threads is shown.

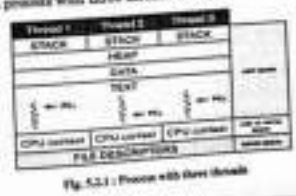


Fig. 5.2.1 : Process with three threads

Module 5

- Multithreading means that the code paths in your application are executed in parallel.
- You have to be aware, that the operating system will interrupt the flow of any one thread at any time, so that another thread will get CPU time.
- It doesn't matter if this is happening on a single or multi core CPU. What makes multithreaded programming difficult is that multiple threads in your application might have to run through the same sections of code, accessing the same data structures.

Example_1 : Multithreaded Web Browser

- In recent years, with the development of Web-based applications, the need for multithreaded programs has become particularly obvious in multi-threaded Web browsers.

```
procedure parse.page(address : url)
    connect server, request page contents
    parse.html.header
    while current.token in {"<p>", "<h1>", "<ul>", ...,
            "<background", "<image", "<table", "<frameset", ...}
        case current.token of
            "<p>"     : break.paragraph
            "<h1>"    : format.heading; match("</h1>")
            "<ul>"    : format.list; match("</ul>")
            ...
            "<background" :
                        a : attributes := parse.attributes
                        fork render.background(a)
            "<image"  : a : attributes := parse.attributes
                        fork render.image(a)
            "<table"  : a : attributes := parse.attributes
                        scan forward for "</table>" token
                        token.stream s := ...    -- table contents
                        fork format.table(s, a)
            "<frameset" :
                        a : attributes := parse.attributes
                        parse.frame.list(a)
                        match("</frameset>")
        ...
...
procedure parse.frame.list(a1 : attributes)
    while current.token in {"<frame", "<frameset", "<noframes>"}
        case current.token of
            "<frame"  : a2 : attributes := parse.attributes
                        fork format.frame(a1, a2)
```

Fig. 5.2.2 : Thread-based code from a hypothetical Web browser

- In browsers such as Firefox or Internet Explorer (see Figure 5.2), there are usually many different threads active at the same time, and each thread may communicate with the remote server (maybe very slow) before completing its task.
- When the user clicks the link, the browser will create a thread to request the specified document.
- For all pages except the smallest page, this thread will receive a series of message "data packets".
- When these packets begin to arrive, the thread must format them for display on the screen.
- The formatting task is similar to typesetting: the thread must access the font, combine the words and divide the words into several lines.
- For many special tags in the page, the formatting thread will spawn other threads: one for each image, one for the background (if any), one for formatting each table, and possibly for processing individually Frames.
- Each spawned thread communicates with the server to obtain the information required for its specific task (for example, the content of the image).
- At the same time, users can access items in the menu to create new browser windows, edit bookmarks, change preferences, etc., all of which are "parallel" with the presentation of page elements.
- To first approximation, the parse page subroutine is the root of a recursive descent parser for HTML. In several cases, however, the actions associated with recognition of a construct (background, image, table, frameset) proceed concurrently with continued parsing of the page itself.
- In this example, concurrent threads are created with the fork operation. An additional thread would likely execute in response to keyboard and mouse events.
- Using many threads ensures that relatively fast operations (for example, displaying text) do not wait for slower operations (for example, displaying large images).
- Whenever a thread is blocked (waiting for a message or I/O), the implementation will automatically switch to another thread.
- In the pre-emptive thread package, the implementation also switches between threads at other times (that is, it performs context switching) to prevent any thread from occupying the CPU.
- Any reader who remembers the earlier, more sequential browsers will appreciate the difference in perceived performance and responsiveness of multithreading.

Module 5

Example_2 : Dispatch Loop Web Browser

- In the absence of language or library support for threading, the browser must adopt a more sequential structure or concentrate all the processing of delayed trigger events in a dispatch loop (see Figure 5.3).
- The data structure associated with the dispatch loop keeps track of all tasks that the browser has not yet completed.
- The status of the task can be very complex.
- For the advanced task of rendering the page, the status must indicate which packets have been received and which have not yet been completed.
- It must also identify the various subtasks of the page (images, tables, frames, etc.) so that we can find all the subtasks and restore their state when the user clicks the "Stop" button.
- In order to ensure a good interactive response, we must ensure that any sub-actions of continue task do not take a long time to execute. Obviously, whenever we wait for a message, we must end the current operation.
- Whenever we read a file from a file, we must also end it, because disk operations are very slow.
- Finally, if the calculation time of any task requires more than a tenth of a second (a typical human perception threshold), then we must divide the task into multiple parts, save the state between these parts and return to the top of the loop.
- These considerations mean that the conditions at the top of the loop must cover the full range of asynchronous events, and the continued execution of any tasks that are subdivided due to continuous lengthy calculations must be interleaved to evaluate the conditions. (In practice, we may need more complex mechanisms than simple interleaving to ensure that neither input-driven tasks nor computationally bound tasks will waste more resources.)


```
type task_descriptor = record
    -- fields in lieu of thread-local variables, plus control-flow information
    ...
ready_tasks : queue of task_descriptor

procedure dispatch
    loop
        -- try to do something input-driven
        if a new event E (message, keystroke, etc.) is available
            if an existing task T is waiting for E
                continue_task(T, E)
            else if E can be handled quickly, do so
            else
                allocate and initialize new task T
                continue_task(T, E)
        -- now do something compute bound
        if ready_tasks is nonempty
            continue_task(dequeue(ready_tasks), 'ok')

procedure continue_task(T : task, E : event)
    if T is rendering an image
        and E is a message containing the next block of data
            continue_image_render(T, E)
    else if T is formatting a page
        and E is a message containing the next block of data
            continue_page_parse(T, E)
    else if T is formatting a page
        and E is 'ok'       -- we're compute bound
            continue_page_parse(T, E)
    else if T is reading the bookmarks file
        and E is an I/O completion event
            continue_goto_page(T, E)
    else if T is formatting a frame
        and E is a push of the "stop" button
            deallocate T and all tasks dependent upon it
    else if E is the "edit preferences" menu item
        edit_preferences(T, E)
    else if T is already editing preferences
        and E is a newly typed keystroke
            edit_preferences(T, E)
    ...
```

Fig. 5.3.3 : Dispatch loop from a hypothetical non-thread-based Web browser

Module 5

- The clauses in continue_task must cover all possible combinations of task status and kinds of events. The code in each clause executes the next consistent unit of work of its task and returns under the following conditions: (1) must wait for an event; (2) has consumed a significant amount of computing time; or (3) the task is completed. Before returning separately, the code (1) places tasks in a dictionary (used by dispatch), which maps waiting events to tasks waiting for events, (2) queues tasks to ready_tasks queue, or (3) records assignment task.
- The main problem with the dispatch loop (in addition to the complexity of subdividing tasks and saving state) is that it hides the algorithmic structure of the program. Standard control flow mechanisms can be used to elegantly describe each of the different tasks (retrieving pages, rendering images, browsing nested menus), if not because we have to return to the top of the dispatch loop in every operation that causes delays.
- In fact, the dispatch loop makes the program "from the inside out", so that task management becomes clear, while the control flow within the task becomes hidden. Like a true factor, the thread package processes the program "right side out", so that the management of tasks (threads) becomes implicit, and the control flow within the thread becomes clear.

5.3 CONCURRENT PROGRAMMING FUNDAMENTALS

Q.Q. 5.2.1 Write a short Concurrent Programming Fundamentals

- In concurrent programs, we will use the term thread to refer to active entities that the programmer thinks are running concurrently with other threads.
- Lightweight and heavy processes refer to the mechanisms of multiprocessing systems.
- In a lightweight process, threads are used to distribute the workload. Here, you will see a process executed for the application or service in the OS. The process will have 1 or more threads. Each thread in this process shares the same address space. Because threads share their address space, communication between threads is simple and efficient. Each thread can be compared with the process in the heavyweight scheme.
- In the heavyweight process, new processes will be created to perform work in parallel. Here (for the same application or service), you will see multiple processes running. Each heavyweight process contains its own address space. The communication between these processes will involve other communication mechanisms, such as sockets or pipes.
- The benefit of the lightweight process comes from saving resources. Since threads use the same code segment, data segment and OS resources, they are fewer overall resources. The disadvantage is that now you must ensure that the system is thread-safe. You must ensure

that the threads do not step on each other. Fortunately, Java provides the necessary tools to allow you to perform this operation.

- Sometimes we use the term task to refer to a well-defined unit of work that must be executed by a thread.
- In a common programming habit, a set of threads share a common "task package", that is, a list of tasks to be completed.
- Each thread repeatedly takes a task from the package, executes the task, and then returns to another task. Sometimes, the work of a task requires adding a new task to the bag.
- Unfortunately, the terminology between the system and the author is inconsistent. Several languages call it a threaded process. Ada calls them tasks.
- Some operating systems call lightweight process threads. Mach OS, derived from OSF UNIX and Mac OS X, calls the address space shared by lightweight processes a task.

or Communication and Synchronization

- In any concurrent programming model, the two most critical problems to be solved are communication and synchronization.
- Communication refers to any mechanism that allows one thread to obtain information generated by another thread.
- The communication mechanism of imperative programs is usually based on shared memory or message passing.
- In the shared memory programming model, multiple threads can access some or all variables of the program. For a pair of threads to communicate, one thread writes a value to a variable, and the other thread simply reads it.
- In the message-passing programming model, threads have no public state. For a pair of threads to communicate, one of them must perform an explicit send operation to transfer data to another.
- Synchronization refers to any mechanism that allows the programmer to control the relative order in which operations occur in different threads.
- Synchronization is often implicit in the messaging model: a message must be sent before it can be received. If the thread tries to receive a message that has not been sent, it will wait for the sender to catch up.
- In the shared memory model, synchronization is usually not implicit: unless we do something special, the "receiving" thread can read its "old" value before the variable "sender" writes it.

Module 5

- In shared memory programs and message-based programs, synchronization can be achieved through spinning (also known as busy-waiting) or blocking.
- In busy waiting synchronization, the thread runs a loop in which the thread continuously re-evaluates a certain condition, until the condition becomes true. (for example, until the message queue becomes non-empty or the shared variable reaches a certain value). It is probably the result of some operations that other threads run on other processors.
- Note that there is no point in waiting busy on a single processor, when we monopolize the resources (processors) needed to achieve this condition, we cannot expect the condition to become true. (Threads on a single processor may sometimes be busy waiting for I/O to complete, but this is another case: I/O devices run in parallel with the processor.)
- In blocking synchronization (also called scheduler-based synchronization), the waiting thread automatically gives up its processor to other threads. Before doing so, it will leave comments in certain data structures associated with the synchronization conditions.
- A thread that corrects the situation at some point in the future will find the note and take action to replay the blocked thread.

or Languages and Libraries

- Thread-level concurrency are often provided to programmers within the sort of explicit concurrency languages, traditional sequential language extensions supported by the compiler, or library packages outside of appropriate languages.
- Although shared memory languages are more common at the "low end" (for multi-core and small multi-processor machines), and message passing libraries are more common at the "high end" (for massively parallel supercomputers), all three options are has been widely used.
- For many years, almost all parallel programming has adopted traditional sequential languages (mainly C and FORTRAN), and libraries for synchronization or message passing have been added. This method is still dominant today.
- In the UNIX world, the shared memory parallel mechanism has been greatly integrated in the POSIX threads standard, which includes mechanisms for creating, destroying, scheduling and synchronizing threads.
- Although the standard does not require changes to the syntax of languages that use the standard, POSIX-compliant compilers must avoid performing optimizations that may introduce race in programs with multiple threads.
- Microsoft's Win32 thread package and compiler provide similar functionality for the Windows platform.

- For high-end scientific computing, message-based parallelism has also converged to the MPI (Message Passing Interface) standard.
- It is suitable for open source and commercial implementation of almost every platform.
- Although the language support for concurrency can be traced back to Algol 68 (and the coroutine for Simula), and Ada received this support widely in the late 1980s, it was not until the mid-1990s that people really became interested in these features. The explosive growth of the planet Wide Web began to drive the event of parallel servers and concurrent client programs. This development
- It happened to coincide with the introduction of Java, and Microsoft followed C# a few years later.
- In comparison to library packages, an explicitly concurrent programming language has the advantage of compiler support. It can make use of syntax aside from subroutine calls, and may integrate communication and thread management more tightly with such concepts as type checking, scoping, and exceptions.
- At an equivalent time, since most programs have historically been sequential, concurrent languages are slow to realize widespread acceptance, particularly as long as the presence of concurrent features can sometime make the sequential case harder to know.

5.4 THREAD CREATION SYNTAX

MQ. 5.4.1. Explain thread Creation syntax with suitable example.

- Almost every concurrent system allows threads to be created (and destroyed) dynamically. Syntactic and semantic details vary considerably from one language or library to another but most conforms to one of six principal options: co-begin, parallel loops, launch-at-elaboration, fork (with optional join), implicit receipt, and early reply.
- The first two options delimit threads with special control-flow constructs. The others use syntax resembling (or identical to) subroutines.
- At least one language (SR) provides all six options. Most others pick and choose. Most libraries use fork/join, as do Java and C#. Ada uses both launch-at-elaboration and fork. OpenMP uses co-begin and parallel loops. RPC systems are typically based on the implicit receipt.

Module 5

Co-begin

- The usual semantics of a compound statement (sometimes delimited with begin... end) involve sequential execution of the constituent statements. A co-begin construct calls instead for concurrent execution:

```
co-begin -- all a statements run concurrently
stmt 1
stmt 2
...
stmt n
end
```

- Each statement can itself be a sequential or parallel compound, or (commonly) a subroutine call.
- Co-begin was the principal means of creating threads in Algol-68. It appears in Co-begin in OpenMP a variety of other systems as well, including OpenMP:

```
#pragma omp sections
{
# pragma omp section
{
printf("thread 1 here\n");
}
# pragma omp section
{
printf("thread 2 here\n");
}
}
```

- In C, OpenMP directives all begin with #pragma omp. (The # sign must appear in column 1.) Most directives, like those shown here, must appear immediately before a loop construct or a compound statement delimited with curly braces.

Parallel loop

- Many concurrent systems, including OpenMP, several dialects of Fortran, and the recently announced .NET parallel FX library, provide a loop whose iterations will be executed simultaneously. In OpenMP in C, we can say

```
#pragma omp parallel for
for (int i = 0; i < 3; i++)
{
printf("thread %d here\n", i);
}
```

In C# with Parallel FX, the equivalent code looks like this :

```
ParallelFor(0, 3, i => {
Console.WriteLine("Thread" + i + "here");
});
```

- The third parameter of ParallelFor is a delegate, in this case lambda expression. The similar Foreach method requires two parameters iterator and delegate.
- In many systems, the programmer's responsibility is to ensure that it is safe to execute loop iterations concurrently. In a sense, correctness will never depend on the result of a race condition.
- For example, it is usually necessary to synchronize access to global variables to ensure that iterates with one another.
- In several languages (such as Occam), language rules prohibit conflicting access. The compiler checks to ensure that variables written by one thread will not be read or written by any concurrently active thread.

Fork/coinvoke

- Co-begin, parallel loops, and detached start all lead to concurrent co-begin vs fork/join control flow mode, in which thread execution is currently nested.
- The fork operation is more general: it makes the creation of a thread an explicit executable operation. When providing peer join operations, it allows threads to wait for the completion of previously branched threads.
- Because fork and join are not bound to nested structures, they can lead to arbitrary patterns of concurrent control flow.

Implicit Receipt

- We have assumed in all our examples so far that newly created threads will run in the address space of the creator.
- In RPC systems it is often desirable to create a new thread automatically in response to an incoming request from some other address space.
- Rather than have an existing thread execute a receive operation, a server can bind a communication channel to a local thread body or subroutine.
- When a request comes in, a new thread springs into existence to handle it. In effect, the bind operation grants remote clients the ability to perform a fork within the server's address space, though the process is often less than fully automatic.

Early Reply

- We normally think of sequential subroutines in terms of a single thread, which Modelling subroutines with fork/join saves its current context (its program counter and registers), executes the subroutine, and returns to what it was doing before.
- The effect is the same, however, if we have two threads—one that executes the caller and another that executes the callee.
- In this case, the call is essentially a fork/join pair. The caller waits for the callee to terminate before continuing execution. Nothing dictates, however, that the callee has to terminate in order to release the caller; all it really has to do is complete the portion of its work on which result parameters depend.

Implementing threads

- A thread library provides programmers with an API for creating and managing threads. Support for threads must be provided either at the user level or by the kernel.
  - Kernel level threads are supported and managed directly by the operating system.
  - User level threads are supported above the kernel in user space and are managed without kernel support.

5.5 KERNEL LEVEL THREADS

GQ. 5.5.1 Explain kernel level thread

- Kernel level threads are supported and managed directly by the operating system.
- The kernel knows about and manages all threads.
- One process control block (PCP) per process.

- One thread control block (TCB) per thread in the system.
- Provide system calls to create and manage threads from user space.

Advantages

1. The kernel has full knowledge of all threads.
2. Scheduler may decide to give more CPU time to a process having a large number of threads.
3. Good for applications that frequently block.

Disadvantages

1. Kernel manage and schedule all threads.
2. Significant overhead and increase in kernel complexity.
3. Kernel level threads are slow and inefficient compared to user level threads.
4. Thread operations are hundreds of times slower compared to user level threads.

5.6 USER LEVEL THREADS

GQ. 5.6.1 Explain user level threads

- User level threads are supported above the kernel in user space and are managed without kernel support.
  - Threads managed entirely by the run-time system (user-level library).
  - Ideally, thread operations should be as fast as a function call.
  - The kernel knows nothing about user-level threads and manage them as if they where single-threaded processes.

Advantages

1. Can be implemented on an OS that does not suport kernel level threads.
2. Does not require modifications of the OS.
3. Simple representation: PC, registers, stack and small thread control block all stored in the user-level process address space.
4. Simple management: Creating, switching and synchronizing threads done in user-space without kernel intervention.
5. Fast and efficient: switching threads not much more expensive than a function call.

Module 5

Disadvantages

1. Not a perfect solution (a tradeoff).
2. Lack of coordination between the user-level thread manager and the kernel.
3. OS may make poor decisions like:
   - scheduling a process with idle threads
   - blocking a process due to a blocking thread even though the process has other threads that can run
   - giving a process as a whole one time slice irrespective of whether the process has 1 or 1000 threads
   - unscheduled a process with a thread holding a lock.
4. May require communication between the kernel and the user-level thread manager (scheduler activations) to overcome the above problems.

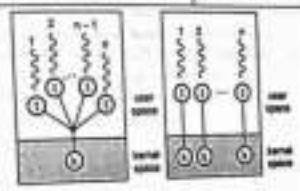
5.7 USER-LEVEL THREAD MODELS

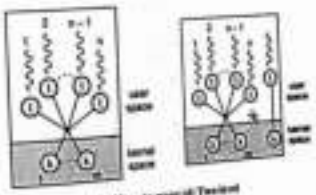
GQ. 5.7.1 Explain User-level thread models in brief

- In general, user-level threads can be implemented using one of four models.
  - Many-to-one
  - One-to-one
  - Many-to-many
  - Two-level
- All models maps user-level threads to kernel-level threads. A kernel thread is similar to a process in a non-threaded (single-threaded) system. The kernel thread is the unit of execution that is scheduled by the kernel to execute on the CPU. The term virtual processor is often used instead of kernel thread.

Table 5.7.1

Sr. No.	Many-to-one	One-to-one
1.	In the many-to-one model all user level threads execute on the same kernel thread. The process can only run one user-level thread at a time because there is only one kernel-level thread associated with the process.	In the one-to-one model every user-level thread execute on a separate kernel-level thread.
2.	The kernel has no knowledge of user-level threads. From its perspective, a process is an opaque black box that occasionally makes system calls.	In this model the kernel must provide a system call for creating a new kernel thread.
	Many-to-many	**Two-level**
1.	In the many-to-many model the process is allocated m number of kernel-level threads to execute n number of user-level thread.	The two-level model is similar to the many-to-many model but also allows for certain user-level threads to be bound to a single kernel-level thread.





(c) Many-to-many (d) Two-level

Fig. 5.7.1

Module 5

5.7.1 Scheduler Activations

- In both the many-to-many model and the two-level model there must be some way for the kernel to communicate with the user level thread manager to maintain an appropriate number of kernel threads allocated to the process. This mechanism is called scheduler activations.
- The kernel provides the application with a set of kernel threads (virtual processors), and then the application has complete control over what threads to run on each of the kernel threads (virtual processors). The number of kernel threads (virtual processors) in the set is controlled by the kernel, in response to the competing demands of different processes in the system.
- The kernel notify the user-level thread manager of important kernel events using upcalls from the kernel to the user-level thread manager. Examples of such events includes a thread making a blocking system call and the kernel allocating a new kernel thread to the process.

Example

- Let's study an example of how scheduler activations can be used. The kernel has allocated one kernel thread (1) to a process with three user-level threads (2). The three user level threads take turn executing on the single kernel-level thread.
- The executing thread makes a blocking system call (3) and the the kernel blocks the calling user-level thread and the kernel-level thread used to execute the user-level thread (4). Scheduler activation: the kernel decides to allocate a new kernel-level thread to the process (5). Upcall: the kernel notifies the user-level thread manager which user-level thread that is now blocked and that a new kernel-level thread is available (6). The user-level thread manager move the other threads to the new kernel thread and resumes one of the ready threads (7).
- While one user-level thread is blocked (8) the other threads can take turn executing on the new kernel thread (9).

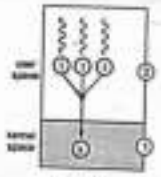


Fig. 5.7.2(a)

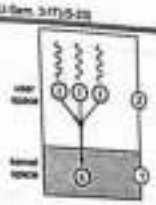


Fig. 5.7.2(a)

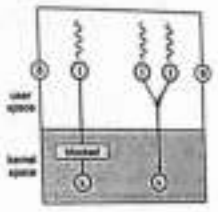


Fig. 5.7.2(b)

Fig. 5.7.2 : steps involved in scheduler activations process

5.7.2 User-level thread scheduling

GQ. 5.7.2 Explain cooperative and pre-emptive thread scheduling in User-level thread scheduling.

- Scheduling of the user-level threads among the available kernel-level threads is done by a thread scheduler implemented in user space. There are two main methods: cooperative and pre-emptive thread scheduling.

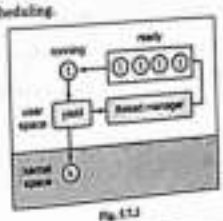


Fig. 5.7.3

Module 5

Cooperative scheduling of user-level threads

- The cooperative model is similar to multiprogramming where a process executes on the CPU until making a I/O request. Cooperative user-level threads execute on the assigned kernel-level thread until they voluntarily give back the kernel thread to the thread manager.
- In the cooperative model, threads yield to each other, either explicitly (e.g., by calling a yield() provided by the user-level thread library) or implicitly (e.g., requesting a lock held by another thread). In the below figure a many-to-one system with cooperative user-level threads is shown.

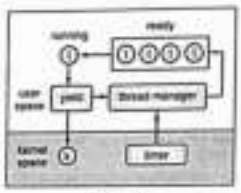


Fig. 5.7.4

Preemptive scheduling of user-level threads

- The pre-emptive model is similar to multitasking (aka time sharing).
- Multitasking is a logical extension of multiprogramming where a timer is set to cause an interrupt at a regular time interval and the running process is replaced if the job requests I/O or if the job is interrupted by the timer.
- This way, the running job is given a time slice of execution that cannot be exceeded.
- In the pre-emptive model, a timer is used to cause execution flow to jump to a central dispatcher thread, which chooses the next thread to run.
- In the below figure a many-to-one system with pre-emptive user-level threads is shown.

Cooperative and preemptive (hybrid) scheduling of user-level threads

- A hybrid model between cooperative and pre-emptive scheduling is also possible where a running thread may yield() or pre-empted by a timer.

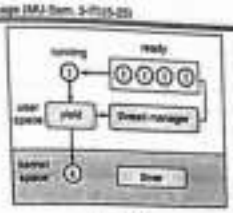


Fig. 5.7.5

5.7.3 Implementing Synchronization

MQ. 5.7.3 Write a short note on Synchronization

- Before the study of synchronization, let us understand the common nomenclature used for different activities in the computer system.
  - **Concurrent tasks :** If not explicitly stated otherwise, the term task will be used to refer to a concurrent unit of execution such as a thread or a process.
  - **Atomic operations :** In concurrent programming, an operation (or set of operations) is atomic if it appears to the rest of the system to occur at once without being interrupted. Other words used synonymously with atomic are: linearizable, indivisible or uninterruptible.
  - **Non-atomic operations :** Incrementing and decrementing a variable are examples of non-atomic operations. The following C expression, X++; is translated by the compiler to three instructions:
- load X from memory into a CPU register.
- increment X and save result in a CPU register.
- store result back to memory.
  - **Race condition :** A race condition or race hazard is the behaviour of an electronic, software or other system where the output is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when events do not happen in the intended order. The term originates with the idea of two signals racing each other to influence the output first.
  - **Data race :** A data race occurs when two instructions from different threads access the same memory location, and at least one of these accesses is a write and there is no synchronization that is mandating any particular order among those accesses.

Module 5

- **Critical section :** Concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed are protected. This protected section is the critical section or critical region. Typically, the critical section accesses a *shared resource*, such as a data structure, a peripheral device, or a network connection, that would not operate correctly in the context of multiple concurrent accesses.
- **Mutual exclusion (mutex) :** In computer science, mutual exclusion is a property of concurrency control, which is instituted for the purpose of preventing race conditions; it is the requirement that one thread of execution never enter its critical section at the same time that another concurrent thread of execution enters its own critical section. Often the word mutex is used as a short form of mutual exclusion.

- Typically, synchronization serves either to make some operation atomic or to delay that operation until some necessary precondition holds. Atomicity is most commonly achieved with mutual exclusion locks. Mutual exclusion ensures that only one thread is executing some critical section of code at a given point in time.
- Critical sections typically transform a shared data structure from one consistent state to another.
- Condition synchronization allows a thread to wait for a precondition, often expressed as a predicate on the value(s) in one or more shared variables.
- It is tempting to think of mutual exclusion as a form of condition synchronization (don't proceed until no other thread is in its critical section), but this sort of condition would require consensus among all extant threads, something that condition synchronization doesn't generally provide.
- Atomicity of operations on the ready list and related data structures ensures that they always satisfy a set of logical invariants: the lists are well-formed, each thread is either running or resides in exactly one list, and so forth. Condition synchronization appears in the requirement that a process in need of a thread to run must wait until the ready list is nonempty.
- It is worth emphasizing that we do not in general want to overly synchronize programs. To do so would eliminate opportunities for parallelism, which we generally want to maximize in the interest of performance.
- Moreover, not all races are bad. If two processes are racing to dequeue the last thread from the ready list, we don't really care which succeeds and which waits for another thread.
- We do care that the implementation of dequeue does not have internal, instruction-level races that might compromise the ready list's integrity.

- In general, our goal is to provide only as much synchronization as is necessary to eliminate "bad" races—those that might otherwise cause the program to produce incorrect results.
- When you create code that is thread safe but still benefits from sharing data or resources between threads, the most important aspect of programming becomes the ability to synchronize threads.
- Synchronization is the cooperative act of two or more threads that ensures that each thread reaches a known point of operation in relationship to other threads before continuing. Attempting to share resources without correctly using synchronization is the most common cause of damage to application data.
- Typically, synchronizing two threads involves the use of one or more synchronization primitives. Synchronization primitives are low-level functions or application objects that your application uses or creates to provide the synchronization behaviour that the application requires.
- Here are the most common synchronization primitives in order of least to most computationally expensive:
  - Compare and swap
  - Mutual exclusion (mutexes) and threads
  - Semaphores and threads
  - Condition variables and threads
  - Threads as synchronization primitives
  - Spare location locks
  - Object locks

5.7.4 Busy-Wait Synchronization

Module 5

SQ. 17.4 What is Busy-Wait Synchronization?

- Busy-waiting is defined as live code-code being run on the CPU that is continually executed without the process making any progress. In both solutions, the while loop, where tests are continually being executed, is a **spinlock**. For the most part, busy-waiting has been defined as a bogging mechanism that can starve other processes from the CPU.
- However, this issue has been complicated with the introduction of multiprocessor machines. "Although busy-waiting is less efficient when all processes execute on the same processor, it is efficient when each process executes on a different processor."

- Nevertheless, busy-waiting still should not be used in general. Even with multiprocessor systems, it can't be assumed that there are enough processors for all processes.
- Additionally, such process might contain a number of threads that are seeking their own CPU time within the context of their parent process.
- There might be hundreds of processes and threads running, and if many of those are busy-waiting, very little progress would be made overall, causing low CPU utilization.
- Data-parallel algorithms are often structured as a series of high-level steps, or phases, typically expressed as iterations of some outermost loop.
- Correctness often depends on making sure that every thread completes the previous step before any moves on to the next. A barrier serves to provide this synchronization.
- Some programming languages supports the basic barrier mechanism, which keeps threads blocked until a certain party threshold has been reached. For this implementation, an additional requirement must be met before the threads are released.
- A release call has to be made explicitly by a controlling thread. This allows the *barrier* to be monitored from a context not blocked by the barrier, providing a way to determine exactly when the threads on the barrier are released. The barrier is also designed to prevent missing barrier calls.
- This helps make sure that threads do not break from one barrier and then enter another, only to conflict with the previous barrier.
- When a thread is released from a barrier, it cannot enter another barrier using the same instance without all threads having first been released by the previous barrier.

or Blocking synchronization

- Blocking synchronization is the status quo for synchronizing shared memory across processes or threads on a single system.
- It puts a process to sleep in order to prevent a busy-wait or spinlock. This conserves CPU cycles, allowing other progress bound processes to move forward.
- Lock-based synchronization ensures that the *critical sections* are executed in a mutually exclusive manner so that no data becomes corrupted or inconsistent.
- Blocking synchronization is easy to code with since it can be encapsulated in objects such as semaphores, mutexes, monitors, and reader-writer locks. These objects also provide reusability, which is often paramount to a programmer's solution choice.
- Lastly, blocking synchronization provides the option of a fair policy implementation, in which processes are ordered fairly and executed based upon their arrival times.

- Nevertheless, blocking has some disadvantages. It requires a great deal of overhead.
- The blocking and wakeup calls, which involve interacting with the operating system, require time to process. The maintenance of the fair policy requires additional time and space.
- Any locked-based algorithm must deal with the problem of deadlock. If deadlock occurs, the process can fail.
- Lastly, due to the overhead, the requirement of mutual exclusion, and the property of fairness blocking synchronization can be quite slow.

or Non-blocking synchronization

- Non-blocking synchronization is a newer alternative synchronization that seeks to resolve some of the issues inherent in lock-based algorithms.
- Non-blocking has little overhead since it does not worry about blocking, waking, or maintaining a mechanism for a fair policy. Without that mechanism, less time and space is used.
- Non-blocking algorithms are a subset of lock-free algorithms, and hence, they do not have to deal with deadlock. If a process fails in the critical section, the other processes do not also fail. Instead, they continue with progress, as intended.
- Under the assumption that conflicts between processes are actually rare, non-blocking algorithms do not waste time with cautionary measures.
- Despite the apparent speed of non-blocking synchronization, it has some severe limitations. First, although both blocking and non-blocking do not satisfy starvation freedom, non-blocking does not guarantee fairness, whereas blocking does.

or The Livelock

- Second, live lock is an issue of non-blocking lock-free algorithms. Instead of getting stuck in the inactive state of deadlock, live lock can potentially be more hazardous since it is an indefinite state where live code is being executed.
- However, it should be noted that, in practice, the situation does not occur often, and even then, live lock is not permanent.
- Wait-free algorithms require that every process will make progress within a finite number of steps. This eliminates the possibility of unbounded waiting, as wait-free algorithms are starvation-free.
- By contrast, non-blocking algorithms require only that some process will make progress within a finite number of steps. This leaves the possibility that some processes might become starved, which means that they will loop continually within the contending operation. This

Module 5

indefinite state, called deadlock with lock-based algorithms, is instead called livelock with lock-free algorithms.

- Whereas deadlock is a sleeping state that does not affect the CPU performance, livelock entails live code that is being looped upon continually, which potentially can waste CPU cycles in a busy-wait fashion.
- While this result can be hazardous, the trade-off is that at least some process is making progress, and as long as that property holds, the process in livelock might eventually get out of it.
- Livelock is potentially resolvable, whereas deadlock is a permanent halt to progress.

The ABA Problem

- Third, there is the ABA problem, which is inherent to any system architecture using the atomic compare-and-swap operation.
- No system fully supports an operation such as double-compare-and-swap or LL/SC/VL to facilitate a complete implementation of many non-blocking algorithms.
- In multithreaded computing, the **ABA problem** occurs during synchronization, when a location is read twice, has the same value for both reads, and "value is the same" is used to indicate "nothing has changed".
- However, another thread can execute between the two reads and change the value, do other work, then change the value back, thus fooling the first thread into thinking "nothing has changed" even though the second thread did work that violates that assumption.
- The ABA problem occurs when multiple threads (or processes) accessing shared data interleave. A common case of the ABA problem is encountered when implementing a lock-free data structure.
- There are a few solutions to the ABA problem, but none are fully usable because of inherent difficulties in implementing such operations in hardware architecture.
- A well-known solution is called DCAS, or double-compare-and-swap.
- Along with variable V, a counter or tag is attached to V to keep track of the number of times that V has been altered.
- DCAS, unlike CAS (compare-and-swap), will make two comparisons, one with the actual value of V, and then one with the expected value of its tag.

Memory consistency model

- A memory consistency model, or memory model, for a shared-memory multiprocessor specifies how memory behaves with respect to read and write operations from multiple processors.
- From the programmer's point of view, the model enables correct reasoning about the memory operations in a program.
- From the system designer's point of view, the model specifies acceptable memory behaviors for the system.
- As such, the memory consistency model influences many aspects of system design, including the design of programming languages, compilers, and the underlying hardware.
- The definition of the memory model is often incorporated into the semantics of the programming language used at a given interface.
- Therefore, the memory model may vary for different interfaces or for different languages at the same interface.
- An intermediate system component such as the compiler is responsible for correctly mapping the memory semantics of its input language to that of its output language.
- The majority of programmers deal with the highest interface, using a high-level programming language for specifying programs.
- At this level, a programmer typically has the choice of using either a traditional sequential language or an explicitly parallel language.
- The sequential language interface provides the familiar uniprocessor semantics for memory, thus relieving the programmer from reasoning about the multiprocessor aspects of the system.
- This interface is typically supported either by a traditional sequential compiler that generates a single thread of control to execute on the multiprocessor or by a parallelizing compiler that deals with the multiprocessor issues transparently to the programmer.
- On the other hand, programmers who opt to express their algorithms more directly using an explicitly parallel language are exposed to the multiprocessor semantics for memory operations in one form or another.
- A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program.

Module 5

- In the uniprocessor, operations executed in order specified by the program whereas in the multiprocessor, all operations executed in order, and the operations of each individual one appear in program order.

5.7.5 Semaphores

GQ. 5.7.5 Explain the term semaphore in operating system.

- Semaphores (sometimes referred to as counting semaphores) can be used to control access to shared resources. A semaphore can be thought of as an intelligent counter. Every semaphore has a current count, which is greater than or equal to 0.
- Any thread can decrement the count to lock the semaphore (this is also called waiting on the semaphore). Attempting to decrement the count past 0 causes the thread that is calling to wait for another thread to unlock the semaphore.
- Any thread can increment the count to unlock the semaphore (this is also called posting the semaphore). Posting a semaphore might wake up a waiting thread if there is one present.
- In their simplest form (with an initial count of 1), semaphores can be thought of as a mutual exclusion (mutex). The important distinction between semaphores and mutexes is the concept of ownership. No ownership is associated with a semaphore. Unlike mutexes, it is possible for a thread that never waited for (locked) the semaphore to post (unlock) the semaphore. This can cause unpredictable application behavior. You must avoid this if possible.
- The operating system provides the following additional capabilities of some semaphore APIs :
  - More complete management capabilities, including permissions on semaphores that are similar to file permissions
  - The ability to group semaphores in sets and perform atomic operations on the group
  - The ability to do multicount wait and post operations
  - The ability to wait for a semaphore to have a count of 0
  - The ability to undo operations that were done by another thread under certain conditions

5.7.6 Message passing

GQ. 5.7.6 Write the brief concept of message passing.

In the message-passing model, concurrent modules interact by sending messages to each other through a communication channel. Modules send off messages, and incoming messages to each module are queued up for handling. Examples include :

- A and B might be two computers in a network, communicating by network connections.
- A and B might be a web browser and a web server – A opens a connection to B, asks for a web page, and B sends the web page data back to A.
- A and B might be an instant messaging client and server.
- A and B might be two programs running on the same computer whose input and output have been connected by a pipe, like ls | grep typed into a command prompt.

Fig. 5.7.6

- Issues in message based computing

Naming Communication Partners

- To send or receive a message, one must generally specify where to send it to, or where to receive it from–communication partners need names for (or references to) one another. Names may refer directly to a thread or process. Alternatively, they may refer to an entry or port of a module, or to some sort of socket or channel abstraction.
- The first naming option–addressing messages to processes–appears in Hoare's original CSP proposal, and in PVM and MPI. Each PVM or MPI process has a unique id (an integer), and each send or receive operation specifies the id of the communication partner.
- MPI implementations are required to be reentrant; a process can safely be divided into multiple threads, each of which can send or receive messages on the process's behalf.
- PVM has hidden state variables that are not automatically synchronized, making threaded PVM programs problematic.
- The second naming option—addressing messages to ports—appears in Ada. An Ada entry call of the form t.foo (args) sends a message to the entry named foo in task (thread) t (t may be either a task name or the name of a variable whose value is a pointer to a task). Among all the message-passing mechanisms we have considered, datagrams are the only one that does not provide some sort of ordering constraint.
- In general, most message passing systems guarantee that messages sent over the same "communication path" arrive in order. When naming processes explicitly, a path links a single sender to a single receiver.
- All messages from that sender to that receiver arrive in the order sent. When naming ports, a path links an arbitrary number of senders to a single receiver. Messages that arrive at a port in a given order will be seen by receivers in that order.

- Note, however, that while messages from the same sender will arrive at a port in order, messages from different senders may arrive in different orders. When naming channels, a path links all the senders that can use the channel to all the receivers that can use it.

Sending

One of the most important issues to be addressed when designing a send operation is the extent to which it may block the caller: once a thread has initiated a send operation, when is it allowed to continue execution? Blocking can serve at least three purposes:

- **Resource management :** A sending thread should not modify outgoing data until the underlying system has copied the old values to a safe location. Most systems block the sender until a point at which it can safely modify its data, without danger of corrupting the outgoing message.
- **Failure semantics :** Particularly when communicating over a long-distance network, message passing is more error-prone than most other aspects of computing. Many systems block a sender until they are able to guarantee that the message will be delivered without error.
- **Return parameters :** In many cases a message constitutes a request, for which a reply is expected. Many systems block a sender until a reply has been received. When deciding how long to block, we must consider synchronization semantics, buffering requirements, and the reporting of run-time errors.

Receiving

- Probably the most important dimension on which to categorize mechanisms for receiving messages is the distinction between explicit receive operations and the implicit receipt. The SR language provides an implicit receipt.
- With implicit receipt, every message that arrives at a given port (or over a given channel) will create a new thread of control, subject to resource limitations (any implementation will have to stall incoming requests when the number of threads grows too large).
- With explicit receipt, a message must be queued until some already-existing thread indicates a willingness to receive it.
- At any given point in time, there may be a potentially large number of messages waiting to be received. Most languages and libraries with explicit receipt allow a thread to exercise some sort of selectivity with respect to which messages it wants to consider.

5.7.7 Remote Procedure Call

MU 5.7.7 What is Remote Procedure Call? Explain.

- Any of the three principal forms of send (no-wait, synchronization, remote invocation) can be paired with either of the principal forms of receive (explicit or implicit).
- The combination of remote invocation send with explicit receipt (e.g., as in Ada) is sometimes known as *rendezvous*. The combination of remote invocation send with implicit receipt is usually known as *remote procedure call*.
- For communication based on requests from clients to servers, remote procedure calls (RPCs) provide an attractive interface to message passing. Rather than talk to a server directly, an RPC client calls a local stub procedure, which packages its parameters into a message, sends them to a server, and waits for a response, which it returns to the client in the form of result parameters.
- Several vendors provide tools that will generate stubs automatically from a formal description of the server interface.
- RPC is available in several concurrent languages (SR obviously among them). It is also supported on many systems by augmenting a sequential language with a stub compiler. The stub compiler is independent of the language's regular compiler. It accepts as input a formal description of the subroutines that are to be called remotely.
- The description is roughly equivalent to the subroutine headers and declarations of the types of all parameters. Based on this input the stub compiler generates source code for client and server stubs.
- A client stub for a given subroutine marshals request parameters and an indication of the desired operation into a message buffer, sends the message to the server, waits for a reply message, and unmarshals that message into result parameters.
- A server stub takes a message buffer as parameter, unmarshals request parameters, calls the appropriate local subroutine, marshals return parameters into a reply message, and sends that message back to the appropriate client. Invocation of a client stub is relatively straightforward.

Module 5

Remote Procedure Call (RPC) merges aspects of events and threads: from the server's point of view, an RPC is an event executed by a separate thread in response to a request from a client. Whether built into the language or implemented via a stub compiler, it requires a run-time system (dispatcher) with detailed knowledge of calling conventions, concurrency, and storage management.

Module 6

CHAPTER 6 Alternative Paradigms: Scripting Languages

Syllabus

Common characteristic , Different problem domain for using scripting , Use of Scripting in Web development – server and client side scripting , Innovative features of scripting language - Names and Scopes, string and pattern manipulation, data types, object orientation

6.1 COMMON CHARACTERISTICS

GQ. 6.1.1 Explain the common characteristics of scripting language.

While it is difficult to define scripting languages precisely, there are several characteristics that they tend to have in common.

1. **Both batch and interactive use :** A few scripting languages (notably Perl) have a compiler that insists on reading the entire source program before it produces any output.Most other languages, however, are willing to compile or interpret their input line by line. Rexx, Python, Tcl, Guile, and (with a short helper script) Ruby will all accept commands from the keyboard.

2. **Economy of expression :** To support both rapid development and interactive use, scripting languages tend to require a minimum of "boilerplate." Some make heavy use of punctuation and very short identifiers (Perl is notorious for this), while others (e.g., Rexx, Tcl, and AppleScript) tend to be more "Englishlike," with lots of words and not much punctuation. All attempt to avoid the extensive declarations and top-level structure common to conventional languages. Where a trivial program looks like this in Java,

```
class Hello {
public static void main(String[] args) {
System.out.println("Hello, world!");
}
}
```

3. **Lack of declarations** ; simple scoping rules. Most scripting languages dispense with declarations, and provide simple rules to govern the scope of names. In some languages (e.g., Perl) everything is global by default; optional declarations can be used to limit a variable to a nested scope. In other languages (e.g., PHP and Tcl), everything is local by default; globals must be explicitly imported.

   Python adopts the interesting rule that any variable that is assigned a value is local to the block in which the assignment appears. Special syntax is required to assign to a variable in a surrounding scope.

4. **Flexible dynamic typing :** In keeping with the lack of declarations, most scripting languages are dynamically typed. In some (e.g., PHP, Python, Ruby, and Scheme), the type of a variable is checked immediately prior to use. In others (e.g., Rexx, Perl, and Tcl), a variable will be interpreted differently in different contexts.

Module 6

In Perl, for example, the program

```
$a = "4";
print $a . 3 . "\n"; # '.' is concatenation
print $a + 3 . "\n"; # '+' is addition
```

will print

43

7

- This contextual interpretation is similar to coercion, except that there isn't necessarily a notion of "natural" type from which an object must be converted; the various possible interpretations may all be equally "natural."

5 **Easy access to system facilities :** Most programming languages provide a way to ask the underlying operating system to run another program, or to performsome operation directly. In scripting languages, however, these requests are much more fundamental, and have much more direct support. Perl, for one, provides well over 100 built-in commands that access operating system functions for input and output, file and directory manipulation, process management, database access, sockets, interprocess communication and synchronization, protection and authorization, time-of-day clock, and network communication.

These built-in commands are generally a good bit easier to use than corresponding library calls in languages like C.

6. **Sophisticated pattern-matching and string manipulation :** In keeping with their text processing and report generation ancestry, and to facilitate the manipulation of textual input and output for external programs, scripting languages tend to have extraordinarily rich facilities for pattern matching, search, and string manipulation.

7. **High-level data types :** High-level data types like sets, bags, dictionaries, lists, and tuples are increasingly common in the standard library packages of conventional programming languages. A few languages (notably C++) allow users to redefine standard infix operators to make these types as easy to use as more primitive, hardware-centric types. Scripting languages go one step further by building high-level types into the syntax and semantics of the language itself.

In most scripting languages, for example, it is commonplace to have an"array" that is indexed by character strings, with an underlying implementation based on hash tables. Storage is invariably garbage collected.

6.2 DIFFERENT PROBLEM DOMAIN FOR USING SCRIPTING

- Some general-purpose languages-Scheme and Visual Basic in particular-are widely used for scripting. Conversely, some scripting languages, including Perl, Python, and Ruby, are intended by their designers for general-purpose use, with features intended to support "programming in the large": modules, separate compilation, reflection, program development environments, and so on.
- For the most part, however, scripting languages tend to see their principal use in well-defined problem domains.
- We consider some of these in the following subsections.

6.2.1 Shell (Command) Languages

GQ. 6.2.1 Explain shell language in brief

- They have features designed for interactive use : Provide a wealth of mechanisms to manipulate file names, arguments, and commands, and to glue together other programs: Most of these features are retained by more general scripting languages.

We consider a few of them :

(a) Filename and Variable Expansion

(b) Tests, Queries, and Conditions

(c) Pipes and Redirection

(d) Quoting and Expansion

(e) Functions

(f) The #! Convention

a. Filename and Variable Expansion

- Most users of a Unix shell are familiar with "wildcard" expansion of filenames. The following command 3 will list all files in the current directory whose names" end in .pdf:

```
ls *.pdf
```

- The shell expands the pattern *.pdf into a list of all matching names. If there are three of them (say fig1.pdf, fig2.pdf,and fig3.pdf), the result is equivalent to ls fig1.pdf fig2.pdf fig3.pdf.

- Filename expansion is sometimes called "globbing," after the original Unix glob command that implemented it.
- Filename expansion is particularly useful in loops. Such loops may be typed directly from the keyboard, or embedded in scripts intended for later execution.
- Suppose, for example, that we wish to create PDF versions of all our EPS Fig. 3

  For loops in the shell :

```
for fig in *.eps
do
ps2pdf    $fig
done
```

- Multiple commands can be entered on a single line if they are separated by semicolons. The following, for example, is equivalent to the loop in the previous

A whole loop on one line:

```
for fig in *.eps; do ps2pdf    $fig; done
```

☞ b. Tests, Queries, and Conditions

- The loop above will execute ps2pdf for every EPS file in the current directory. Suppose, however, that we already have some PDF files, and only want to create the ones that are missing

  Conditional tests in the shell :

```
for fig in *.eps
do
target=${fig%.eps}.pdf
if [ $fig -nt $target ]
then
ps2pdf    $fig
fi
done
```

- The third line of this script is a variable assignment. The expression ${fig%.eps} within the right-hand side expands to the value of fig with any trailing .eps removed.

Pipes and Redirection

- One of the principal innovations of Unix was the ability to chain commands together, "piping" the output of one to the input of the next.
- Like most shells, bash uses the vertical bar character (|) to indicate a pipe. To count the number Pipes of figures in our directory,without distinguishing between EPS and PDF versions, we might type.

```
for fig in *; do echo ${fig%.*}; done | sort -u | wc –l
```

- Here the first command, a for loop, prints the names of all files with extensions (dot-suffixes) removed. The echo command inside the loop simply prints its arguments. The sort –u command after the loop removes duplicates, and the wc -l command counts lines.
- Like most shells, bash also allows output to be directed to a file, or input read from a file. To create a list of figures, we might type.

```
for fig in *; do echo ${fig%.*}; done | sort -u > all_figs
```

- The "greater than" sign indicates output redirection. If doubled (sort -u >> all_figs) it causes output to be appended to the specified file, rather than overwriting the previous contents.
- In a similar vein, the "less than" sign indicates input redirection. Suppose we want to print our list of figures all on one line, separated by spaces, instead of on multiple lines. On a Unix system we can type.

```
tr '\n' ' ' < all_figs
```

- This invocation of the standard tr command converts all newline characters to spaces. Because tr was written as a simple filter, it does not accept a list of files on the command line; it only reads standard input.

c. Quoting and Expansion

- Shells typically provide several quoting mechanisms to group words together into strings.
- Single (forward) quotes inhibit filename and variable expansion in the quoted text, and cause it to be treated as a single word, even if it contains white space.
- Double quotes also cause the contents to be treated as a single word, but do not inhibit expansion.

Thus

```
foo=bar
single='$foo'
```

Module 6

```
double="$foo"
echo $single $double
```

will print "$foo bar". _

- Several other bracketing constructs in bash group the text inside, for various purposes.
- Command lists enclosed in parentheses are passed to a subshell for evaluation.

If the opening parenthesis is preceded by a dollar sign, the output of the nested command list is expanded into the surrounding context :

```
for fig in $(cat my_figs); do ps2pdf ${fig}.eps; done
```

- Here cat is the standard command to print the content of a file. Most shells use backward single quotes for the same purpose ('cat my_figs'); bash supports this syntax as well, for backward compatibility. _
- Command lists enclosed in braces are treated by bash as a single unit.

They can be used, for example, to redirect the output of a sequence of commands :

```
{ date; ls; } >> file_list
```

☞ d. Functions

- Users can define functions in bash that then work like built-in commands.Many users, for example, define ll as a shortcut for ls -l,which lists files in the current directory in "long format."

```
function ll () {
ls -l "$@"
}
```

- Within the function, $1 represents the first parameter, $2 represents the second, and so on. In the definition of ll, $@ represents the entire parameter list.
- Functions can be arbitrarily complex. In particular, bash supports both local variables and recursion.
- Shells in the csh family provide a more primitive alias mechanism that works via macro expansion. _

☞ e. The #! Convention

As noted above, shell commands can be read froma script file. To execute them in the current shell, one uses the "dot" command :

```
./my_script
```

- where my_script is the name of the file. Many operating systems, including most versions of Unix, allow one to make a script function as an executable program, so that users can simply type

```
my_script
```

- Two steps are required. First, the file must be marked executable in the eyes of the operating system.
- On Unix one types chmod +x my_script. Second, the filemust be self-descriptive in a way that allows the operating system to tell which shell (or other interpreter) will understand the contents.
- Under Unix, the file must begin with the characters #!, followed by the name of the shell. The typical bash script thus begins with

```
#!/bin/bash
```

- Specifying the full path name is a safety feature: it anticipates the possibility that the user may have a search path for commands on which some other program named bash appears before the shell.

6.2.2 Text Processing and Report Generation

GQ. 6.2.2 Explain Text Processing and Report Generation in scripting language with example.

- Shell languages tend to be heavily string-oriented. Commands are strings, parsed into lists of words. Variables are string-valued.
- Variable expansion mechanisms allow the user to extract prefixes, suffixes, or arbitrary substrings.
- Concatenation is indicated by simple juxtaposition. There are elaborate quoting conventions. Few more conventional languages have similar support for strings.
- Tools to accomplish this task constitute the second principal class of ancestors for modern scripting languages.

a. Sed

- SED command in UNIX is stands for stream editor and it can perform lot's of function on file like, searching, find and replace, insertion or deletion.
- Though most common use of SED command in UNIX is for substitution or for find and replace. By using SED you can edit files even without opening it, which is much quicker way

to find and replace something in file, than first opening that file in VI Editor and then changing it.

- SED is a powerful text stream editor. Can do insertion, deletion, search and replace(substitution).
- SED command in unix supports regular expression which allows it perform complex pattern matching.

Syntax

```
sed OPTIONS... [SCRIPT] [INPUTFILE...]
```

Example

Consider the below text file as an input.

```
$cat > geekfile.txt
unix is great os. unix is opensource. unix is free os.
learn operating system.
unix linux which one you choose.
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

Sample Commands

1. **Replacing or substituting string :** Sed command is mostly used to replace the text in a file. The below simple sed command replaces the word "unix" with "linux" in the file.

```
$sed 's/unix/linux/' geekfile.txt
```

Output

```
linux is great os. unix is opensource.
unix is free os. learn operating system.
linux linux which one you choose.
linux is easy to learn.unix is a multiuser os.
Learn unix .unix is a powerful.
```

- Here the "s" specifies the substitution operation. The "/" are delimiters. The "unix" is the search pattern and the "linux" is the replacement string.
- By default, the sed command replaces the first occurrence of the pattern in each line and it won't replace the second, third...occurrence in the line.

2. **Replacing the nth occurrence of a pattern in a line :** Use the /1, /2 etc flags to replace the first, second occurrence of a pattern in a line. The below command replaces the second occurrence of the word "unix" with "linux" in a line.

```
$sed 's/unix/linux/2' geekfile.txt
```

Output

```
unix is great os. linux is opensource. unix is free os.
learn operating system.
unix linux which one you choose.
unix is easy to learn.linux is a multiuser os.Learn unix .unix is a powerful.
```

b. Awk

GQ. 6.2.3 What Can We Do With Awk ?

- Awk is in some sense an evolutionary link between stream editors like sed and full-fledged scripting languages. It retains sed's line-at-a-time filter model of computation, but allows the user to escape this model when desired, and replaces single-character editing commands with syntax reminiscent of C.
- Awk provides (typeless) variables and a variety of control-flow constructs, including subroutines.
- An awk program consists of a sequence of patterns, each of which has an associated action. For every line of input, the interpreter executes, in order, the actions whose patterns evaluate to true.
- Awk is a scripting language used for manipulating data and generating reports. The awk command programming language requires no compiling, and allows the user to use variables, numeric functions, string functions, and logical operators.
- Awk is a utility that enables a programmer to write tiny but effective programs in the form of statements that define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line. Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then performs the associated actions.

1. AWK Operations

(a) Scans a file line by line.

(b) Splits each input line into fields.

Module 6

(c) Compares input line/fields to pattern.

(d) Performs action(s) on matched lines.

☞ **2. Useful For**

(a) Transform data files.

(b) Produce formatted reports.

☞ **3. Programming Constructs**

(a) Format output lines.

(b) Arithmetic and string operations.

(c) Conditionals and loops.

▸ **Syntax**

```
awk options 'selection _criteria {action }' input-file > output-file
```

▸ **Options**

```
-f program-file : Reads the AWK program source from the file
                  program-file, instead of from the
                  first command line argument.
-F fs           : Use fs for the input field separator
```

☞ **Sample Commands**

➢ **Example : Consider the following text file as the input file for all cases below.**

```
$cat > employee.txt
ajay manager account 45000
sunil clerk account 25000
varun manager sales 50000
amit manager account 47000
tarun peon sales 15000
deepak clerk sales 23000
sunil peon sales 13000
satvik director purchase 80000
```

1. **Default behavior of Awk :** By default Awk prints every line of data from the specified file.

```
$ awk '{print}' employee.txt
```

Output

```
ajay manager account 45000
sunil clerk account 25000
varun manager sales 50000
amit manager account 47000
tarun peon sales 15000
deepak clerk sales 23000
sunil peon sales 13000
satvik director purchase 80000
```

- In the above example, no pattern is given. So the actions are applicable to all the lines.
- Action print without any argument prints the whole line by default, so it prints all the lines of the file without failure.

2. Print the lines which matches with the given pattern.

```
$ awk '/manager/ {print}' employee.txt
```

Output

```
ajay manager account 45000
varun manager sales 50000
amit manager account 47000
```

In the above example, the awk command prints all the line which matches with the 'manager'.

c. Perl

GQ. 6.2.4 Explain the purpose of Perl language in brief

- Perl is a general-purpose, high level interpreted and dynamic programming language. It was developed by Larry Wall, in 1987.
- There is no official Full form of the Perl, but still, the most used expansion is **"Practical Extraction and Reporting Language"**.

Module 6

- Some of the programmers also refer Perl as the "Pathologically Eclectic Rubbish Lister" Or "Practically Everything Really Likable".
- The acronym "Practical Extraction and Reporting Language" is used widely because Perl was originally developed for the text processing like extracting the required information from a specified text file and for converting the text file into a different form.
- Perl supports both the procedural and Object-Oriented programming. Perl is a lot similar to C syntactically and is easy for the users who have knowledge of **C, C++**.

☞ **Evolution of Perl**

- It all started when Larry Wall was working on a task to generate the reports from a lot of text files which have cross-references.
- Then he started to use awk for this task but soon he found that it is not sufficient for this task. So instead of writing a utility for this task, he wrote a new language i.e. Perl and also wrote the interpreter for it.
- He wrote the language Perl in C and some of the concepts are taken from awk, sed, and LISP etc.
- At the beginning level, Perl was developed only for the system management and text handling but in later versions, Perl got the ability to handle regular expressions, and network sockets etc.
- In present Perl is popular for its ability to handling the Regex(Regular Expressions). The first version of Perl was **1.0** which released on December 18, 1987.
- The **latest version of Perl is 5.28. Perl 6** is different from Perl 5 because it is a fully object-oriented reimplementation of Perl 5.

Perl has many reasons for being popular and in demand. Few of the reasons are mentioned below :

- **Easy to start :** Perl is a high-level language so it is closer to other popular programming languages like C, C++ and thus, becomes easy to learn for anyone.
- **Text-Processing :** As the acronym "Practical Extraction and Reporting Language" suggest that Perl has the high text manipulation abilities by which it can generate reports from different text files easily. Also, it can convert the files into some another form.
- **Contained best Features :** Perl contains the features of different languages like C, sed, awk, and sh etc. which makes the Perl more useful and productive.
- **System Administration :** Due to having the different scripting languages capabilities Perl make the task of system administration very easy. Instead of becoming dependent on many

languages, just use Perl to complete out the whole task of system administration. In Spite of this Perl also used in web programming, web automation, GUI programming etc.

- **Web and Perl :** Perl can be embedded into web servers to increase its processing power and it has the DBI package, which makes web-database integration very easy.

☞ **Beginning with Perl Programming**

- **Finding a Interpreter :** There are various online IDEs which can be used to run Perl programs without installing.
- **Windows :** There are various IDEs to run Perl programs or scripts : **Padre, Eclipse with EPIC plugin** etc.

☞ **Programming in Perl**

- Since the Perl is a lot similar to other widely used languages syntactically, it is easier to code and learn in Perl.
- Programs can be written in Perl in any of the widely used text editors like **Notepad++, gedit** etc.
- After writing the program save the file with the extension **.pl** or **.PL** To run the program use **perl file_name.pl** on the command line.

➢ **Example : A simple program to print Welcome to GFG!**

```
filter_none

edit

play_arrow

brightness_4

# Perl program to print Welcome to GFG!

#!/usr/bin/perl

 # Below line will print "Welcome to GFG!"

print "Welcome to GFG!\n";
```

Output

```
Welcome to GFG!
```

- **Comments :** Comments are used for enhancing the readability of the code. The interpreter will ignore the comment entries and does not execute them. Comments can be of the single line or multiple lines.

Module 6

☞ **Single line Comment**

▸ **Syntax**

```
# Single line comment
```

☞ **Multi-line comment**

▸ **Syntax**

```
= Multi line comments
Line start from  = is interpreted as the
starting of multiline comment and =cut is
consider as the end of multiline comment
=cut
```

- **Print :** It is a function in Perl to show the result or any specified output on the console.
- **Quotes :** In Perl, you can use either single quotes(') or double quotes(""). Using single quotes will not interpolate any variable or special character but using double quotes will interpolates.
- **\n :** It is used for the new line character which uses the backslash (\) character to escape any type of character.
- **/usr/bin/perl :** It is actual Perl interpreter binary which always starts with #!. This is used in the Perl Script Mode Programming.

☞ **Advantages of Perl**

- Perl Provides supports for cross platform and it is compatible with mark-up languages like HTML, XML etc.
- It is very efficient in text-manipulation i.e. Regular Expression. It also provides the socket capability.
- It is free and a Open Source software which is licensed under Artistic and GNU General Public License (GPL).
- It is an embeddable language that's why it can embed in web servers and database servers.
- It supports more than 25, 000 open source modules on **CPAN(Comprehensive Perl Archive Network)** which provide many powerful extensions to the standard library. For example, XML processing, GUI(Graphical User Interface) and DI(Database Integration) etc.

Disadvantages of Perl

- Perl doesn't supports portability due to **CPAN** modules.
- Programs runs slowly and program needs to be interpreted each time when any changes are made.
- In Perl, the same result can be achieved in several different ways which make the code untidy as well as unreadable.
- Usability factor is lower when compared to other languages.

6.2.3 Mathematics and Statistics

- Anyone who owns a programmable calculator realizes that similar needs arise in mathematics and statistics.
- And just as shell and report generation tools have evolved into powerful languages for general purpose computing, so too have notations and tools for mathematical and statistical computing.
- APL, one of the more unusual languages of the 1960s. Originally conceived as a pen-and-paper notation for teaching applied mathematics, APL retained its emphasis on the concise, elegant expression of mathematical algorithms when it evolved into a programming language.
- The modern successors to APL include a trio of commercial packages for mathematical computing:Maple,Mathematica, andMatlab.
- Though their design philosophies differ, each provides extensive support for numerical methods, symbolic mathematics (formula manipulation), data visualization, and mathematical modeling.
- All three provide powerful scripting languages,with a heavy orientation toward scientific and engineering applications.
- As the "3 Ms" are to mathematical computing, so the S and R languages are to statistical computing. Originally developed at Bell Labs by John Chambers and colleagues in the late 1970s, S is a commercial package widely used in the statistics community and in quantitative branches of the social and behavioral sciences.
- R is an open-source alternative to S that is largely though not entirely compatible with its commercial cousin.
- Among other things, R supports multidimensional array and list types, array slice operations, user-defined infix operators, call-by-need parameters, first-class functions, and unlimited extent.

6.2.4 "Glue" Languages and General-Purpose Scripting

- "Glue" Languages and General Purpose Scripting

(a) Perl

(b) Python

(c) Ruby

- Yukihiro Matsumoto wanted a language "more powerful than Perl, and more object-oriented than Python".
- Success helped by Ruby on Rails web-development framework which was adopted by Apple, Twitter, and others .
- Ousterhout joined Sun Microsystems in 1994, where for three years he led a multiperson team devoted to Tcl development
- In comparison to Perl, Tcl is somewhat more verbose It makes less use of punctuation, and has fewer special cases
- As noted, Rexx is generally considered the first of the general purpose scripting languages, predating Perl and Tcl by almost a decade.
- Perl and Tcl are roughly contemporaneous: both were initially developed in the late 1980s .
- Perl was originally intended for glue and text processing applications.
- Tcl was originally an extension language, but soon grew into glue applications.
- Python was originally developed by Guido van Rossum at CWI in Amsterdam, the Netherlands, in the early 1990s.
  - He continued his work at CNRI in Reston, Virginia, beginning in 1995.
  - In 2000 the Python team moved to BeOpen.com, and to Digital Creations
  - Recent versions of the language are owned by the Python Software –All releases are Open Source.
- **Ruby** : As the popularity of scripting grew in the 1990s, users were motivated to develop additional languages, to provide additional features, address the needs of specific application domains or support a style of programming.

6.2.5 Extension Languages

- Most applications accept some sort of commands these commands are entered textually or triggered by user interface events such as mouse clicks, menu selections, and keystrokes.
- Commands in a graphical drawing program might save or load a drawing; select, insert, delete, or modify its parts; choose a line style, weight, or color; zoom or rotate the display; or modify user preferences.

- An extension language serves to increase the usefulness of an application by allowing the user to create new commands, generally using the existing commands as primitives.
- Extension languages are increasingly seen as an essential feature of sophisticated tools.
  - Adobe's graphics suite (Illustrator, Photoshop, InDesign, etc.) can be extended (scripted) using JavaScript, Visual Basic (on Windows), or AppleScript.
  - AOLserver, an open-source web server from America On-Line, can be scripted using Tcl. Disney and Industrial Light and Magic use Python to extend their internal (proprietary) tools.
- To admit extension, a tool must
  - incorporate, or communicate with, an interpreter for a scripting language.
  - provide hooks that allow scripts to call the tool's existing commands.
  - allow the user to tie newly de.ned commands to user interface events.
- With care, these mechanisms can be made independent of any particular scripting language .
- One of the oldest existing extension mechanisms is that of the emacs text editor,
  - An enormous number of extension packages have been created for emacs; many of them are installed by default in the standard distribution.
  - The extension language for emacs is a dialect of Lisp called Emacs Lisp.

6.3 USE OF SCRIPTING IN WEB DEVELOPMENT

- From a programming languages point of view, simple playback of recorded audio or video is not particularly interesting. We therefore focus our attention here on content that is generated on the fly by a program-a script-associated with an Internet URI (uniform resource identifier).

- Suppose we type a URI into a browser on a client machine, and the browser sends a request to the appropriate web server.
- If the content is dynamically created, an obvious first question is: does the script that creates it run on the server or the client machine? These options are known as server-side and client-side web scripting, respectively.
- Server-side scripts are typically used when the service provider wants to retain complete control over the content of the page, but can't (or doesn't want to) create the content in advance. Examples include the pages returned by search engines, Internet retailers, auction sites, and any organization that provides its clients with on-line access to personal accounts.
- Client-side scripts are typically used for tasks that don't need access to proprietary information, and are more efficient if executed on the client's machine. Examples include interactive animation, error-checking of fill-in forms, and a wide variety of other self-contained calculations.

6.4 SERVER AND CLIENT SIDE SCRIPTING

GQ. 6.4.1 Explain Server and client side scripting with suitable diagram.

- **web page** is a document available on world wide web. Web Pages are stored on web server and can be viewed using a web browser.
- A web page can cotain huge information including text, graphics, audio, video and hyper links. These hyper links are the link to other web pages.
- Collection of linked web pages on a web server is known as **website.** There is unique **Uniform Resource Locator (URL)** is associated with each web page.

6.4.1 Static Web page

- **Static web pages** are also known as flat or stationary web page. They are loaded on the client's browser as exactly they are stored on the web server. Such web pages contain only static information. User can only read the information but can't do any modification or interact with the information.
- Static web pages are created using only HTML. Static web pages are only used when the information is no more required to be modified.

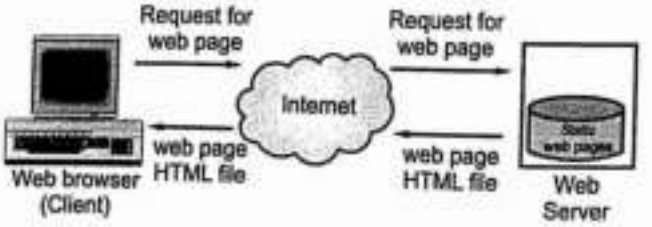


Fig. 6.4.1

6.4.2 Dynamic Web page

- **Dynamic web page** shows different information at different point of time. It is possible to change a portaion of a web page without loading the entire web page. It has been made possible using **Ajax** technology.

a. Server-side dynamic web page

- It is created by using server-side scripting. There are server-side scripting parameters that determine how to assemble a new web page which also include setting up of more client-side processing.

b. Client-side dynamic web page

- It is processed using client side scripting such as JavaScript. And then passed in to **Document Object Model (DOM).**

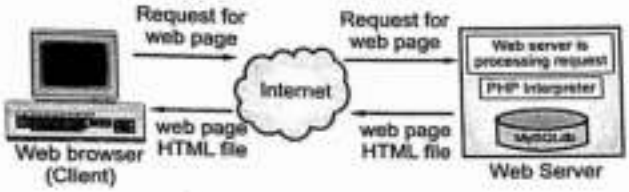


Fig. 6.4.2

6.4.3 Scripting Languages

- Scripting languages are like programming languages that allow us to write programs in form of script. These scripts are interpreted not compiled and executed line by line.
- Scripting language is used to create dynamic web pages.

1. Client-side Scripting

- **Client-side scripting** refers to the programs that are executed on client-side. Client-side scripts contains the instruction for the browser to be executed in response to certain user's action.
- Client-side scripting programs can be embedded into HTML files or also can be kept as separate files.

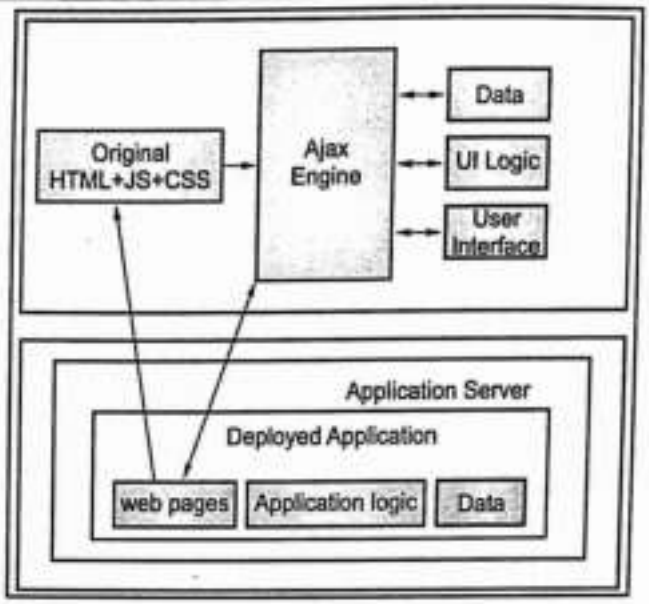


Fig. 6.4.3

Following table describes commonly used Client-Side scripting languages :

Sr. No.	Scripting Language Description
1.	**JavaScript** It is a prototype based scripting language. It inherits its naming conventions from java. All java script files are stored in file having .js extension.
2.	**ActionScript** It is an object oriented programming language used for the development of websites and software targeting Adobe flash player.
3.	**Dart** It is an open source web programming language developed by Google. It relies on source-to-source compiler to JavaScript.
4.	**VBScript** It is an open source web programming language developed by Microsoft. It is superset of JavaScript and adds optional static typing class-based object oriented programming.

2. Server-side Scripting

- **Sever-side scripting** acts as an interface for the client and also limit the user access the resources on web server. It can also collects the user's characteristics in order to customize response.

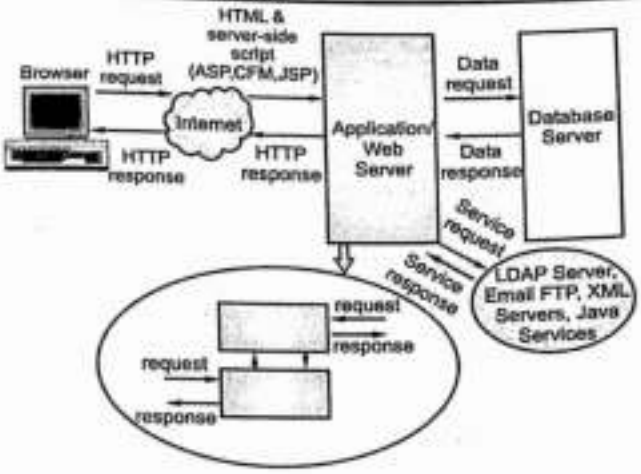


Fig. 6.4.4

Following table describes commonly used Server-Side scripting languages :

Sr. No.	Scripting Language Description
1.	**ASP** Active Server Pages (ASP)is server-side script engine to create dynamic web pages. It supports Component Object Model (COM) which enables ASP web sites to access functionality of libraries such as DLL.
2.	**ActiveVFP** It is similar to PHP and also used for creating dynamic web pages. It uses native Visual Foxpro language and database.
3.	**ASP.net** It is used to develop dynamic websites, web applications, and web services.
4.	**Java** Java Server Pages are used for creating dynamic web applications. The Java code is compiled into byte code and run by Java Virtual Machine (JVM).
5.	**Python** It supports multiple programming paradigms such as object-oriented, and functional programming. It can also be used as non-scripting language using third party tools such as Py2exe or Pyinstaller.
6.	**WebDNA** It is also a server-side scripting language with an embedded database system.

6.5 INNOVATIVE FEATURES OF SCRIPTING LANGUAGE

GQ. 6.5.1 Discuses the Innovative features of scripting language.

- Conventional languages tend to stress efficiency, maintainability, portability, and the static detection of errors.
- Their type systems tend to be built around such hardware-level concepts as fixed size integers, floating-point numbers, characters, and arrays.
- Scripting languages tend to stress flexibility, rapid development, local customization, and dynamic (run-time) checking.
- Their type systems, likewise, tend to embrace such high-level concepts as tables, patterns, lists, and files.
- They were originally designed to -**glue**‖ existing programs together to build a larger system.
- Ex: general-purpose scripting languages like **Perl** and **Python**
- Modern scripting languages have two principal sets of ancestors.
  - command interpreters or " shells" of traditional batch and " terminal" (command-line) computing.
  - IBM's JCL, MS-DOS command interpreter, Unix sh and csh.
  - various tools for text processing and report generation.
  - IBM's RPG, and Unix's sed and awk.

6.5.1 Common Characteristics

- Both batch and interactive use.
  - While a few languages (e.g. Perl) have a compiler that requires the entire source program, almost all scripting languages either compile or interpret line by line.
  - Many "compiled" versions are actually completely equivalent to the interpreter running behind the scenes (like in Python).
- Economy of expression.
  - Two variants: some make heavy use of punctuation and short identifiers (like Perl), while others emphasize "English-like" functionality.

Either way, things get shorter. Java versus Python (or Ruby or Perl) :

```
class Hello {
public static void main(String[] args) {
System.out.println("Hello, world!");
}
```

☞ **Output print "Hello, world!"**

- Lack of declarations; simple scoping rules.
  - While the rules vary, they are generally fairly simple and additional syntax is necessary to alter them.
  - In **Perl**, everything is of global scope by default, but optional parameters can limit the scope to local
  - In **PHP**, everything is local by default, and any global variables must be explicitly imported.
  - In **Python**, everything is local to the block in which the assignment appears, and special syntax is required to assign a variable in a surrounding scope.

☞ **Flexible dynamic typing**

- In PHP, Python and Ruby, the type of a variable is only checked right before use
- In Perl, Rexx, or Tcl, things are even more dynamic :

```
$a = "4"
print $a . 3 . "\n"
print $a + 3 . "\n"
```

Perl

Outputs the following:

```
43
7
```

- Easy access to system facilities
  - While all languages provide support for OS functionality, scripting languages generally provide amazing and much more fundamental built-in support.

Module 6

- Examples include directory and file manipulation, I/O modules, sockets, database access, password and authentication support, and network communications.

– Sophisticated pattern matching and string manipulation

- Perl is perhaps the master of this, but it traces back to the text processing sed/awk ancestry.
- These are generally based on extended regular expression

☞ **High level data types**

– In general, scripting languages provide support for sets, dictionaries, lists and tuples (at a minimum).

– While languages like C++ and Java have these, they usually need to be imported separately.

– Behind the scenes, optimizations like arrays indexed using hash tables are quite common.

– Garbage collection is always automatic, so user never has to deal with heap/stack issues.

6.5.2 Scope and Names

GQ. 6.5.2 Explain scope and name in scripting language.

– Most scripting languages (Scheme is the obvious exception) do not require variables to be declared

- Perl and JavaScript permit optional declarations - sort of compiler-checked documentation
- Perl can be run in a mode (use strict 'vars') that requires declarations

– With or without declarations, most scripting languages use dynamic typing

- The interpreter can perform type checking at run time, or coerce values when appropriate
- Tcl is unusual in that all values-even lists-are represented internally as strings

– Nesting and scoping conventions vary quite a bit

- Scheme, Python, JavaScript provide the classic combination of nested subroutines and static (lexical) scope
- Tcl allows subroutines to nest, but uses dynamic scope
- Named subroutines (methods) do not nest in PHP or Ruby

- Perl and Ruby join Scheme, Python, and JavaScript in providing first class anonymous local subroutines
- Nested blocks are statically scoped in Perl
- In Ruby, they are part of the named scope in which they appear
- Scheme, Perl, Python provide for variables captured in closures
- PHP and the major glue languages (Perl, Tcl, Python, Ruby) all have sophisticated namespace rules
- mechanisms for information hiding and the selective import of names from separate modules

– What is the scope of undeclared variable?

- In Perl, all variables are global unless otherwise specified.
- In PHP, local unless explicitly imported.
- Ruby has only two levels: $foo is global, foo is local; @foo is instance of current object, and @@foo is instance variable of current object's class

In Python, all variables are local by default, unless explicitly imported :

```
i=1; j=3
def outer( )
```

– Scope in Python

In Python, all variables are local by default, unless explicitly imported :

```
i=1; j=3
def outer():
  def middle(k):
    def inner( ):
          global i              #from main program, not outer
      i = 4
        inner( )
        return i,j,k            #3 element tuple
      i=2
      return middle(j)          #old (global) j
  print outer( )
  print i,j
```

- This prints: (2,3,3)

4 3

- Scope in Python
  - By default, there is no way for a nested scope to write to a non-local or non-global scope - so in previous example, inner could not modify outer's i variable.

R has an interesting convention :

Normal assignment puts value into the local variable :

```
i <- 4
```

Super assignment puts value into whatever variable would be found under normal (static) scoping rules :

```
i <<- 4
```

Tcl uses dynamic scoping, but in an odd way - the programmer must request other scopes explicitly :

```
upvar i j ;                #j is the local name for caller's I
uplevel 2   {puts [expr $a + $b] }
#executes 'puts' two scopes up on dynamic chain
```

6.5.3 Pattern Matching

GQ. 6.5.3 Explain the different Pattern matching concept with example.

- Regular expressions are present in many scripting languages and related tools employ extended versions of the notation.
  - extended regular expressions – in sed and awk, Perl, Tcl, Python, and Ruby.
  - grep, the stand-alone Unix is a pattern-matching tool, is another useful program that you might be familiar with.
- In general, two main groups.
  - The first group includes awk, egrep, the regex routines of the C standard library, and older versions of Tcl.
  - These implement REs as defined in the **POSIX standard.**
  - Languages in the second group follow the lead of Perl, which provides a large set of extensions, sometimes referred to as "**advanced REs**".

1. Pattern matching : POSIX Res.

Basic operations are familiar :

/ab(cd|ef)g*/ - Matches abcd, abcdg, abefg, abefgg, etc.

Other quantifiers :

?: 0 or 1 repetitions

+: 1 or more repetitions

{n} : exactly n repetitions

{n,} : at least n repetitions

{n,m} : between n and m repetitions

^ and $ force the match to be at the beginning or end of the line

Brackets can indicate a character class: [aeiou] - any vowel

Ranges: [0-9]

A dot . matches any single character

^ before a character class is negation

2. Pattern matching: extended REs

- Perl adds on to this extensively.

Example

```
$_ = "albatross";
if (/ba.*s+/) ... #true
if (/^ba.*s+/) ... #false - no match at start
```

=~ tests if it matches, !~ tests if it does not (or defaults to checking against $_, if not specified)

Substitution is done by s///:

```
$foo = "albatross";
$foo =~ s/lbat/c; #now across
```

- Variations on normal Res
  - Trailing i makes the match case insensitive.

```
$foo = "Albatross";
if ($foo =~ /^al/i) ... #true
```

- Trailing g will replace all occurances.

```
$foo = "albatross";

$foo =~ s/[aciou]/-/g ... # "-lb-tr-ss"
```

- Trailing **x** causes Perl to ignore all comments and embedded white space in the pattern, so that you can break up long patterns into multiple lines.

4. Pattern matching: greedy matches

- If multiple matches are possible, it will take the —left-most longest‖ possible one. For example, in the string abcbcbcde, the pattern /(bc)+/ will match abcbcbcde.
- This is known as the greedy match.

Other options

- *? matches the smallest number of instances of the preceding subexpression that will allow it to succeed.
- +? matches at least one instance, but no more than necessary.

  ?? matches either 0 or 1 instance, with a preference for 0 .

6.5.4 Data Types

GQ. 6.5.4 Explain the data types.

- As we have seen, scripting languages don't generally require (or even permit) the declaration of types for variables
- Most perform extensive run-time checks to make sure that values are never used in inappropriate ways
- Some languages (e.g., Scheme, Python, and Ruby) are relatively strict about this checking
  - When the programmer wants to convert from one type to another, it must say so explicitly
- Perl (and likewise Rexx and Tcl) takes the position that programmers should check for the errors they care about
  - in the absence of such checks the program should do something reasonable

- Numeric types have a bit more variation across languages, but emphasis is universally that the programmer shouldn't worry about the issue unless necessary.
- Won't say too much here, except be cautious about arithmetic if it matters to your program.
- Some of these even store numbers as strings, so calculations may not always be what you expect, although most do a good job of auto-converting if needed.
- For composite types, a heavy emphasis is on mappings (also called dictionaries, hashes, or associated arrays).
  - Generally these are similar to arrays, but access time depends upon a hash function.

Example

```
director = {}
director["Star Wars"] = "George Lucas"
director["The Princess Bride"] = "Rob Reiner"
print director["Star Wars"]
print "Buffy" in director
```

- Behind the scenes, this is actually using a hash function. Still O(1) access time (mostly), but the constant is not nearly as fast as normal array access.

6.5.5 Object Orientation

- Perl 5 has features that allow one to program in an object-oriented style
- PHP and JavaScript have cleaner, more conventional-looking object-oriented features
  - both allow the programmer to use a more traditional imperative style
- Python and Ruby are explicitly and uniformly object-oriented
- Perl uses a value model for variables; objects are always accessed via pointers

Module 6

- In PHP and JavaScript, a variable can hold either a value of a primitive type or a reference to an object of composite type.
    - In contrast to Perl, however, these languages provide no way to speak of the reference itself, only the object to which it refers
- Python and Ruby use a uniform reference model
- Classes are themselves objects in Python and Ruby, much as they are in Smalltalk.
- They are types in PHP, much as they are in C++, Java, or C#
- Classes in Perl are simply an alternative way of looking at packages (namespaces)
- JavaScript, remarkably, has objects but no classes
    - its inheritance is based on a concept known as prototypes
- Both PHP and JavaScript are more explicitly object oriented

Chapter Ends...

□□□