PRINCIPLES OF PROGRAMMING LANGUAGES

UNIT-1

Introduction

The concept of Programming Languages: Syntax, Semantics and pragmatics

There are 3 things that relate to the concept pemrogramanan language: syntax, semantiks and pragmatic. 3 in teaching this concept uses the analogy of the ordinary language we use everyday.

Syntax

The syntax of a language related to the structure of language. For example, to form a valid sentence in the language we use the structure: [subject] + [verb] + [noun]. By using this structure, we can form a sentence, for example: I eat rice. In connection with the programming language, we must meet the syntax (read: the rules of language structure) so that programs can run. For example, in the BASIC language, to mengassign a variable with a value, we use the Operands '=', but if in Pascal, we use':='. Example in BASIC: a = 1, but in the language Pascal, a: = 1.

Semantic

Semantics of a language describes the relationship between syntax and computational models. Simply put, semantic explain the meaning of the program.
Following analogy. If we use the syntax [subject] + [verb] + [noun], we can produce sentences.
If we generate the sentence I eat rice, so this sentence to meet the rules of syntax.
But, if I make a sentence I eat stones, the syntax of this sentence is correct. However, semantically, this sentence does not contain significant meaning.
In connection with the programming language, sometimes there are times when a programmer can not associate syntax computing model. Logic errors can easily occur.

As with the example there is a programming language as follows:

```
if (a = 5) (
echo 'The value a = 5?;
)
```

If the program is run, what happens? Depending of what language is used. If the language used is C, then the output comes out is always value a = 5, although a previous variable value other than 5. Why did that happen? That's because the operator '=' in C means mengassign a variable that is to the left with the existing value on the right. In C, the syntax of this operation is correct.
But, if that meant the programmer wants to evaluate the value of a variable, then the

logic operators should wear'==='. Thus, the actual program to be

```
if (a == 5) (
echo 'The value a = 5?;
)
```

Pragmatic

Pragmatics associated with the ease of implementation and efficiency. In analogy with the language, we can just tell the person "Do not smoke" when there are regulations that prohibit a person smoking in a room. Such a short sentence is enough efficient. But, in other occasions we might use the phrase "Please you do not smoke in here because, according to government regulations in a number XXX XXX stated that smoking in public places will lead to violations of regulations, other than that from the health side ... blah blah blah".

In relation to the programming language, a programmer should be able to ensure efficiency in doing lawyer-coding-an. In the C language, programmers are given the power to allocate memory. As a result, if the programmer negligent in mengontorl variables resulting from the assignment of a pointer, it will be a memory leak. This is caused when a programmer mengcreate a pointer variable, and then delete them, the information is still there in memory, it's just not accessible anymore.

3.4 Translation Models

This section discusses translation models under the following subheadings:

- BNF Grammars
- Syntax Trees
- Finite State Machines
- Other Methodologies

3.4.1 Grammars

A grammar is defined completely by 3 finite sets and a start symbol. Mathematically, we may represent a grammar as follows:

G[Z] = {N, T, P, Z} where
N is the set of non-terminal symbols;
T is the set of terminal symbols;
P is the set of production rules that ultimately connect expressions with non-terminal symbols to expressions with terminal symbols;
Z is the start symbol such that Z Є N.

From the above definition, the following constraints are normally applied:

- The intersection of sets N and T is the empty set, i.e., $N \cap T = \{\}$.
- The alphabet of a grammar is comprised of all terminal and non-terminal symbols, i.e., $N \cup T$ = alphabet.
- The language of the grammar is the set of all acceptable strings for that grammar.

Reputed linguist Noam Chomsky describes four types of grammar. With some modification, we use them in the study of programming languages to explain these languages are developed. The languages are

- Phrase Structure Grammar
- Context Sensitive Grammar
- Context Free Grammar
- Regular Grammar

3.4.2 Phrase Structure Grammar

Phrase structure grammars (PSG) are used to describe natural languages. There are different dialects of PSG, for instance head-driven phrase structure grammar (HPSG), the lexical functional grammar (LFG), and the generalized phrase structure grammar (GPSG).

In a PSG, the productions are of the following form:

B ::= V where
B $\in$ {N $\cup$ T) and B is not null; alternately expressed as B $\in$ {N $\cup$ T}+
V $\in$ {N $\cup$ T) and V can be null; alternately expresses as V $\in$ {N $\cup$ T}*

To put it in words, a non-null notation (including terminal and/or non-terminal symbols) on the left can be replaced by any valid combination of symbols (terminal and/or non-terminal) on the right, including the empty set.

Example 1:

Consider the PSG given by the following sets:
G[Z] = {(Z, A, B, C), (a, b, c), P, Z} where P consists of the following rules:
R1. Z ::= aZBC | aBC
R2. CB ::= BC
R3. aB ::= ab
R4. bB ::= bc
R5. bC ::= bc
R6. cC ::= cc
R7. BC ::= cC

Q1a. We may derive the string *abc* from the grammar (thus showing that it is valid) as follows:
By R1: Z → aBC By R3: aBC → abC By R5: abC → abc

Q1b. We may show that a^2bc^3 is a valid string as follows:
By R1a: Z → aZBC By R1b: aZBC → aaBCBC By R3: aaBCBC → aabCBC
By R5: aabCBC → aabcBC By R7: aabcBC → aabccC By R6: aabccC → aabccc

Notice from the forgoing example, that in order to determine that a string pattern or phrase is valid, it has to be derived. Each term in a derivation is called a *sentential form*. Alternately, a sentential form is a term that is derivable from the start symbol of a grammar. Formally, a language may be defined as a set of sentential forms (each consisting of only terminal symbols) that can be derived from the start symbol of the grammar.

3.4.3 Context-Sensitive Grammar

In a context-sensitive grammar (CSG), either sides of any given production rule may be surrounded by a context of a set of terminal and/or non-terminal symbol(s). Formally, we say that productions are of the following form:

aAy ::= xay where
A Є N
x, y Є {N Ụ T} or x, y Є { }; alternately expressed as x, y Є {N Ụ T}*
a Є {N Ụ T) and a is not null; alternately expresses as a Є {N Ụ T}+

To further paraphrase, a string may replace a non-terminal A in the context of x and y, where x and y represent valid sentential forms of the grammar. One application of this grammar is in programming languages that require variables to be declared prior to their usage (for example, Pascal, C, C++, Java, COBOL, etc.). Apart from this, the CSG is not widely used for programming languages.

3.4.4 Context-Free Grammar

In a context-free grammar (CFG), a non-terminal symbol, A, may be replaced by a string (i.e. sentential form) in any context. The production rules are typically used to recursively generate string patterns from a start symbol. CFGs appear in most programming languages. Formally, we say that productions are of the following form:

A ::= a where
A Є N
a Є {N Ụ T} or a Є { }; alternately expressed as a Є {N Ụ T}*

Example 2:

Referring to example 1, Rule 1 is also context-free. We may therefore define a grammar as follows:
Consider the CFG given by the following sets:
G2[Z] = {(Z, A, B, C), (a, b, c), P, Z} where P consists of the following rules:
R1. Z ::= aZBC | aBC | abc

Example 3:

We may define a grammar for numeric data as follows:
G[Number] = {(Number, Digit), (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .), P, Number} where P consists of the following:
R1. Number ::= <Digit> | <Digit> <Number>
R2. Number ::= <Number>.<Number>
R3. Digit ::= 0|1|2|3|4|5|6|7|8|9

3.4.5 Regular Grammar

In a regular grammar (RG), productions are of the following format:

A ::= a | aB | Ba where
A, B Є N and a Є T

Example 4:

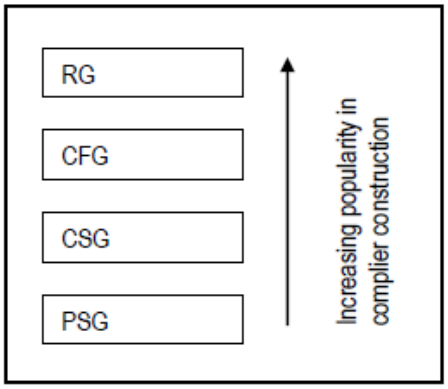
Consider the RG given by the following sets:
G3[Z] = {(Z, B), (a, b), P, Z} where P consists of the following rules:
R1. Z ::= Zb | Bb
R2. B ::= Ba | a

Q4a. We may derive the string a^3b^2 from the grammar (thus showing that it is valid) as follows:
By R1a: Z → Zb By R1b: Zb → Bbb By R2a: Bbb → Babb By R2a: Babb → Baabb
By R2b: Baabb → aaabb

3.4.6 Chomsky Hierarchy

The grammars may be organized in a hierarchy (known as the Chomsky hierarchy) as shown in figure 3.2. Of the four grammars, CFG and RG are more widely used in the design of programming languages.

Figure 3.2: Chomsky Hierarchy



3.4.7 Other Notations

Two additional notations worth remembering are as follows:

- A →+ B means that B is derivable from A in one or more steps.
- A →* B means that B is derivable from A in zero or more steps.
- Let G[Z] be a grammar, and let xβy be a sentential form of G. Then β is called a *phrase* of sentential form xβy for non-terminal B if Z →* xβy and B →+ β. Moreover, β is called a *simple phrase* of sentential form xβy for non-terminal B if Z →* xβy and B → β (i.e., β is derivable from B in one step).

  In other words, if β is derivable form a non-terminal symbol in one step, and β appears as part of a sentential form, S, that is derivable in zero or more steps, then β is a simple phrase of sentential form S.

Example 5: Figure 3.3 shows an example of an RG that defines an integer.

Figure 3.3: Grammar for Integer Definition

Consider the grammar given by the following sets:
G4[<Integer>] = {(<Integer>, <Digit>), (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), P, <Integer>} where P consists of the following rules:
R1. <Integer> ::= <Integer> <Digit> | <Digit>
R2. <Digit> ::= 0|1|2|3|4|5|6|7|8|9

Q5a. The BNF notation for this grammar could simply be expressed as follows:
<Integer> ::= <Integer> <Digit> | <Digit>
<Digit> ::= 0|1|2|3|4|5|6|7|8|9

Q5b. We can show that <Integer>1 is a phrase as follows:
<Integer> → <Integer><Digit> → <Integer>1
So <Integer>1 is a phrase of itself for non-terminal <Integer>.

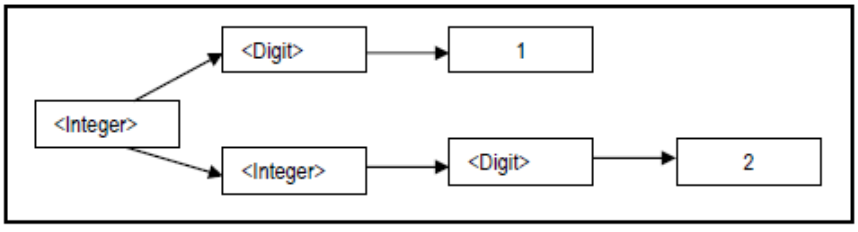
Q5c. We can also easily show that 1 is a simple phrase of sentential form <Integer>1 for non-terminal <Digit>.

3.4.8 Syntax Trees

A *syntax tree* (also called *derivation tree*) is a graphical representation used to illustrate the derivation of a sentential form from a grammar.

Example 6: Figure 3.4 shows how we may use a derivation tree to show that 21 is a valid integer of grammar in figure 3.3.

Figure 3.4: Syntax Tree to Show that 21 is Valid Based on Grammar G3 of Figure 3.3



Ambiguity

If two or more derivation trees exist for the same sentential form of a grammar G, then G is said to be ambiguous. The effect of ambiguity is to cause confusion: given an input, it is not known for certain which interpretation the computer will take.

Example 7:

The following grammar is used for the **If-Statement** in languages such as Pascal and Algol:

<If-Statement> ::= If <Condition> Then <Statement> [Else <Statement>]
<Statement> ::= <IfStatement> | <WhileStatement> | <ForStatement> | AssignmentStatement> | . . .
<Condition> ::= [NOT] <Comparison> | <Variable> <Operator> <Variable> | <BooleanVariable> |
<Comparison> <Connector> <Comparison>
<Connector> ::= AND | OR
<Operator> ::= < | <= | = | <> | > | >=

Now consider the pseudo-statement, and show that it is ambiguous: If C1 Then If C2 Then S1 Else S2:
The derivation trees are shown in figure 3.5.

Figure 3.5: Syntax Trees for Ambiguous If-Statement

Figure 3.5a: Ambiguity — the Else is Associated with the Second If-Statement

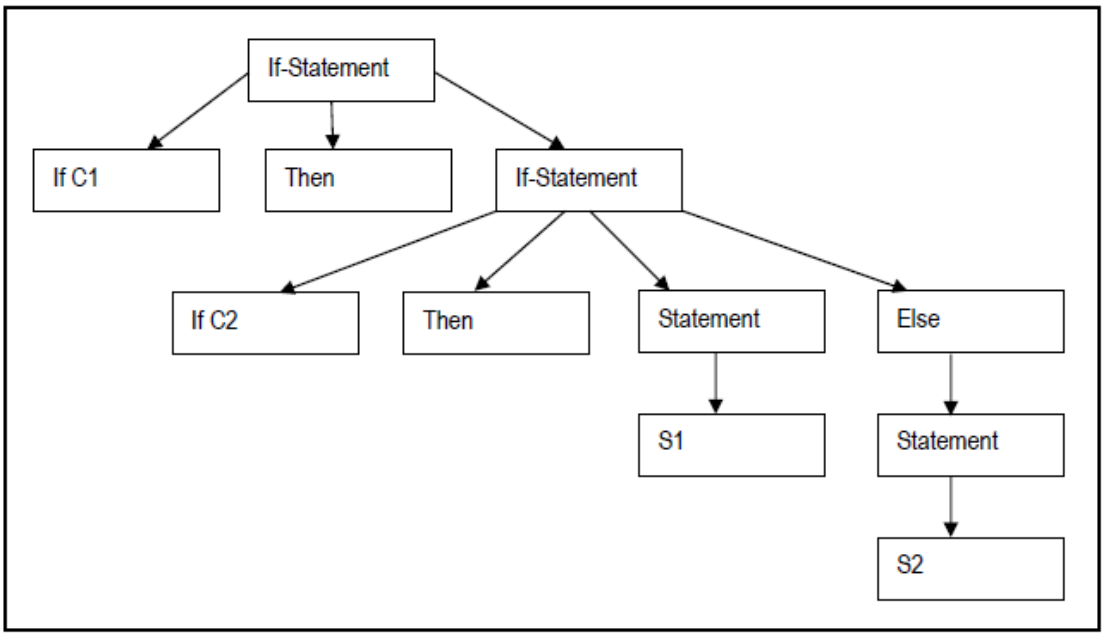
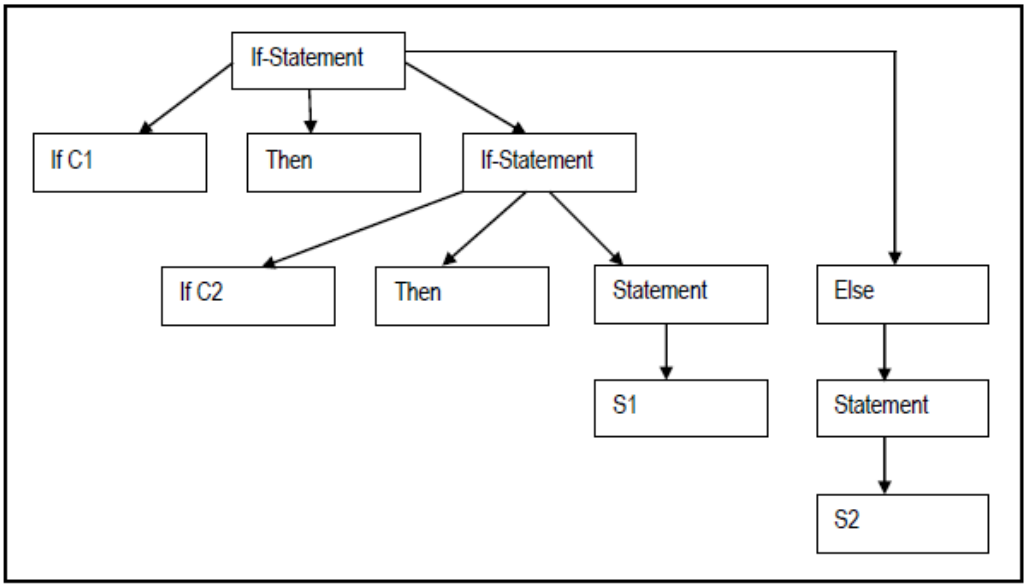


Figure 3.5b: Ambiguity — the Else is Associated with the First If-Statement



By observation, a grammar is ambiguous whenever any of the following conditions hold:

a. The grammar contains a self-embedded term, and there is a (left or right) recursion on that term
b. The grammar contains circulations of the form A →+ A

Most programming languages exhibit ambiguity in some aspect of their grammar. Of more importance is whether and how the language allows the programmer to avoid ambiguities. For instance, in many languages, you can use blocking (i.e., compound statement) to avoid ambiguity when using nested if-statements.

3.4.9 Finite State Machines

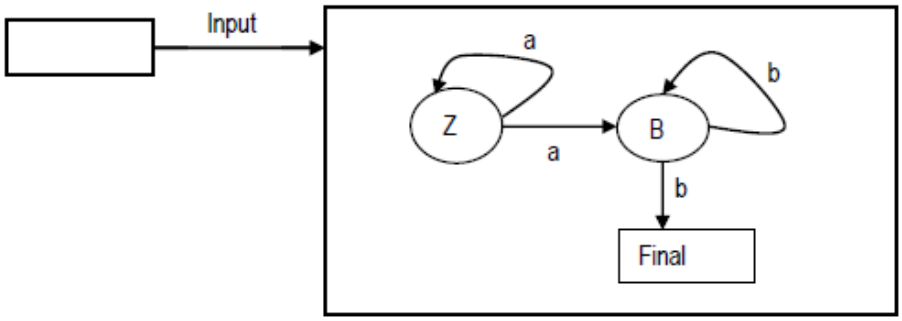
A *finite state machine* (FSM) is a graphical representation of the states and transitions among states for a software component. In an FSM (also referred to as *state diagram* or *finite state automaton*) nodes are states; arcs are transitions labeled by event names (the label on a transition arc is the event name causing the transition). The state-name is written inside the node. When used in the context of programming language design, the FSM is employed to represent the production rules of a grammar.

Example 8:

Consider the grammar given by the following sets:
G5[<Z>] = {G<Z> ::= {(Z, B), (a, b), P, Z} where P consists of the following rules:
R1. Z ::= aZ | aB
R2. B ::= bB | b

Then the language for this grammar may be expressed as follow: L<G5> = {$a^m b^n$ where m, n >= 1}.
Figure 3.6 shows the FSM for this grammar.

Figure 3.6: FSM for the Grammar of Example 8



The FSM is useful to syntax analysis in the following way: To test the validity of an input string, the final state must be reached. If a final state cannot be reached, then the input string is invalid.

A finite state machine of this form (illustrated in figure 3.6) is said to be non-deterministic. The reason for this is that it is impossible to determine which path to follow in the FSM without looking ahead. If we assume that there is no way of looking ahead to the next symbol, then only the current symbol can be considered.

Formally, we may define a *non-deterministic finite state machine* (NDFSM) as an FSM with the following properties:

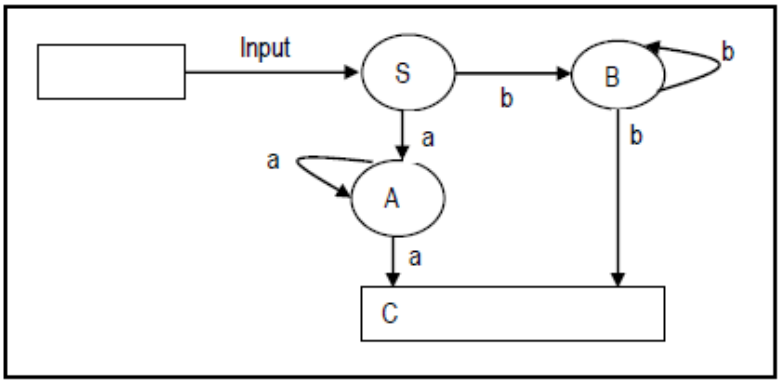
- A finite set of states (nodes)
- A finite input alphabet
- A start state (one of the nodes)
- A finite set of final states which is a subset of the set of states
- A set of transition functions (arcs) form node to node, each being labeled by an element of the input alphabet
- Given a state and an input symbol, more than one resultant states may be possible

What would be more desirable is a *deterministic finite state machine* (DFSM) — where each transition is predictable. To obtain a DFSM from an NDFSM, you represent the non-deterministic transitions as transitions to new states, as shown in figure 3.7.

Formally, we may define a deterministic finite state machine (DFSM) as an FSM with the following properties:

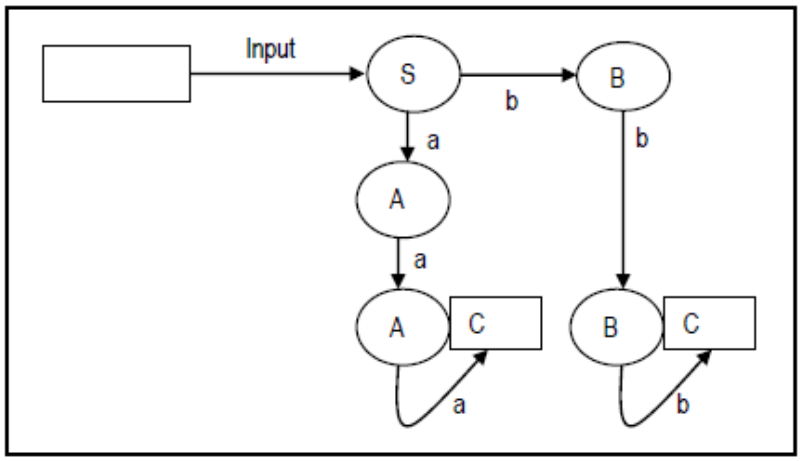
- A finite set of states (nodes)
- A finite input alphabet
- A start state (one of the nodes)
- A finite set of final states which is a subset of the set of states
- A set of transition functions (arcs) form node to node, each being labeled by an element of the input alphabet, where given a state and an input symbol, only one resultant state transition is possible

Figure 3.7: Replacing an NDFSM with a DFSM



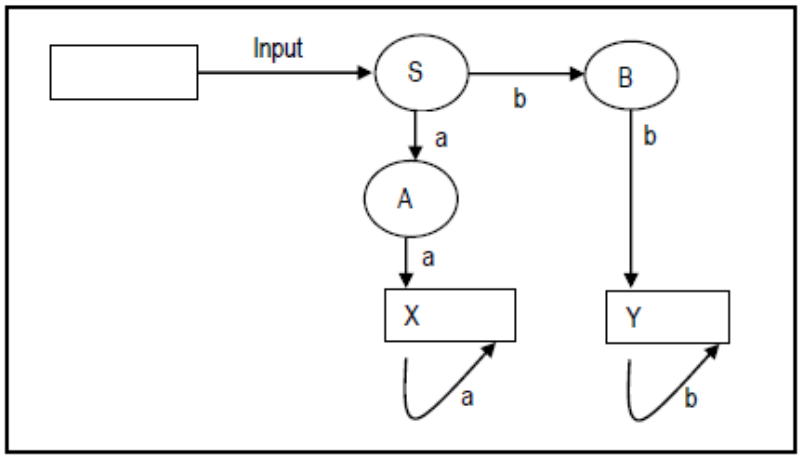
Related production rules are
S ::= aA | bB
A ::= aA | aC
B ::= bB | bC
C ::= 0

The above NDFSM can be replaced by the following DFSM:



In the hybrid state AC: If the next input is an **a**, it is pushed to state A; if the next input is a final symbol, it is pushed to C.

In the hybrid state BC: If the next input is a **b**, it is pushed to state A; if the next input is a final symbol, it is pushed to C.



Referring to the diagram above, if we replace the hybrid states as new final states, then we obtain a DFSM as shown.

3.4.9 Finite State Machines (continued)

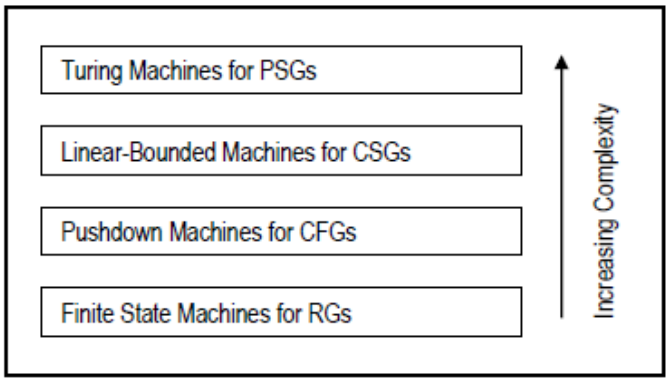
FSMs relate to the translation process in the following way:

- Only regular grammars (RGs) are represented by FSMs.
- Only DFSMs are useful for automatic translation. The input string must lead to a final state in a deterministic way; otherwise it is invalid.
- The DFSM may be represented internally by software.
- The DFSM is equivalent to the derivation tree, and may therefore be considered as an alternative.
- Due to all of the above, most programming languages are based on RGs.

3.4.10 Other Methodologies

Apart from FSMs, and syntax trees, other methodologies for recognizing syntax include *pushdown machines* (PM), *linear-bounded machines* (LBM), and *turing machines* (TM). Figure 3.8 provides a listing showing the relative complexity of each notation. However, knowledge of these other techniques is not required for this course.

Figure 3.8: Methodologies for Syntax Recognition



Please note:

- Turing machines ideally have infinite storage; in practice, computers qualify as Turing machines.
- A linear-bounded machine can be considered to be a Turing machine with finite storage.
- A pushdown machines can be considered to be a finite state machine with a stack.

Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value.

The **assignment statement** gives a value to a variable:

```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.14159
```

This example makes three assignments. The first assigns the string value `"What's up, Doc?"` to a variable named `message`. The second gives the integer `17` to `n`, and the third assigns the floating-point number `3.14159` to a variable called `pi`.

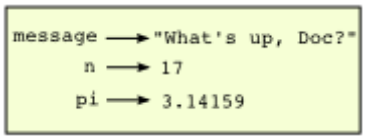
The **assignment token**, `=`, should not be confused with *equals*, which uses the token `==`. The assignment statement binds a *name*, on the left-hand side of the operator, to a *value*, on the right-hand side. This is why you will get an error if you enter:

```
>>> 17 = n
File "<interactive input>", line 1
SyntaxError: can't assign to literal
```

Tip

> When reading or writing code, say to yourself "n is assigned 17" or "n gets the value 17". Don't say "n equals 17".

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state snapshot** because it shows what state each of the variables is in at a particular instant in time. (Think of it as the variable's state of mind). This diagram shows the result of executing the assignment statements:



If you ask the interpreter to evaluate a variable, it will produce the value that is currently linked to the variable:

```
>>> message
'What's up, Doc?'
>>> n
17
>>> pi
3.14159
```

We use variables in a program to "remember" things, perhaps the current score at the football game. But variables are *variable*. This means they can change over time, just like the scoreboard at a football game. You can assign a value to a variable, and later assign a different value to the same variable. (*This is different from maths. In maths, if you give `x` the value 3, it cannot change to link to a different value half-way through your calculations!*)

```
>>> day = "Thursday"
>>> day
'Thursday'
>>> day = "Friday"
>>> day
```

```
'Friday'
>>> day = 21
>>> day
21
```

You'll notice we changed the value of `day` three times, and on the third assignment we even made it refer to a value that was of a different type.

A great deal of programming is about having the computer remember things, e.g. *The number of missed calls on your phone*, and then arranging to update or change the variable when you miss another call.

Statements

A **statement** is an instruction that the Python interpreter can execute. We have only seen the assignment statement so far. Some other kinds of statements that we'll see shortly are `while` statements, `for` statements, `if` statements, and `import` statements. (There are other kinds too!)

When you type a statement on the command line, Python executes it. Statements don't produce any result.

Expressions

An **expression** is a combination of values, variables, operators, and calls to functions. If you type an expression at the Python prompt, the interpreter **evaluates** it and displays the result:

```
>>> 1 + 1
2
>>> len("hello")
5
```

In this example `len` is a built-in Python function that returns the number of characters in a string. We've previously seen the `print` and the `type` functions, so this is our third example of a function!

The *evaluation of an expression* produces a value, which is why expressions can appear on the right hand side of assignment statements. A value all by itself is a simple expression, and so is a variable.

```
>>> 17
17
>>> y = 3.14
```

```
>>> x = len("hello")
>>> x
5
>>> y
3.14
```

Binding Time Spectrum

A source file has many names whose properties need to be determined. The meaning of these properties might be determined at different phases of the life cycle of a program. Examples of such properties include the set of values associated with a type; the type of a variable; the memory location of the compiled function; the value stored in a variable, and so forth. Binding is the act of associating properties with names. Binding time is the moment in the program's life cycle when this association occurs.

Many properties of a programming language are defined during its creation. For instance, the meaning of key words such as while or for in C, or the size of the integer data type in Java, are properties defined at language design time. Another important binding phase is the language implementation time. The size of integers in C, contrary to Java, were not defined when C was designed. This information is determined by the implementation of the compiler. Therefore, we say that the size of integers in C is determined at the *language implementation time*.

Many properties of a program are determined at compilation time. Among these properties, the most important are the types of the variables in statically typed languages. Whenever we annotate a variable as an integer in C or Java, or whenever the compiler infers that a variable in Haskell or SML has the integer data type, this information is henceforward used to generate the code related to that variable. The location of statically allocated variables, the layout of the [activation records] of function and the control flow graph of statically compiled programs are other properties defined at compilation time.

If a program uses external libraries, then the address of the external functions will be known only at link time. It is in this moment that the runtime environment finds where is located the `printf` function that a C program calls, for instance. However, the absolute addresses used in the program will only be known at loading time. At that moment we will have an image of the executable program in memory, and all the dependences will have been already solved by the loader.

Finally, there are properties which we will only know once the program executes. The actual values stored in the variables is perhaps the most important of these properties. In dynamically typed languages we will only know the types of variables during the execution of the program. Languages that provide some form of late binding will only lets us know the target of a function call at runtime, for instance.

As an example, let us take a look at the program below, implemented in C. In line 1, we have defined three names: `int`, `i` and `x`. One of them represents a type while the others represent the declaration of two variables. The specification of the C language defines the meaning of the keyword `int`. The properties related to this specification are bound when the language is defined. There are other properties that are left out of the language definition. An example of this is the range of values for the `int` type. In this way, the implementation of a compiler can choose a particular range for the `int` type that is the most natural for a specific machine. The type of variables `i` and `x` in the first line is bound at compilation time. In line 4, the program calls the function `do_something` whose definition can be in another source file. This reference is solved at

link time. The linker tries to find the function definition for generating the executable file. At loading time, just before a program starts running, the memory location for `main`, `do_something`, `i` and `x` are bound. Some bindings occur when the program is running, i.e., at runtime. An example is the possible values attributed to `i` and `x` during the execution of the program.

```
1 int i, x = 0;
2 void main() {
3   for (i = 1; i <= 50; i++)
4     x += do_something(x);
5 }
```

The same implementation can also be done in Java, which is as follows:

```
1 public class Example {
2  int i, x = 0;
3
4     public static void main(String[] args) {
5     for (i = 1; i <= 50; i++) {
6        x += do_something(x);
7        }
8     }
9 }
```

Assignment

In computer programming, an **assignment statement** sets and/or re-sets the value stored in the storage location(s) denoted by a variable name; in other words, it copies a value into the variable. In most imperative programming languages, the assignment statement (or expression) is a fundamental construct.

Today, the most commonly used notation for this basic operation has come to be `x = expr` (originally Superplan 1949–51, popularized by Fortran 1957 and C) followed by [1] `x := expr` (originally ALGOL 1958, popularised by Pascal),[2] although there are many other notations in use. In some languages the symbol used is regarded as an operator (meaning that the assignment statement as a whole returns a value) while others define the assignment as a statement (meaning that it cannot be used in an expression).

Assignments typically allow a variable to hold different values at different times during its life-span and scope. However, some languages (primarily strictly functional) do not allow that kind of "destructive" reassignment, as it might imply changes of non-local state. The purpose is to enforce referential transparency, i.e. functions that do not depend on the state of some variable(s), but produce the same results for a given set of parametric inputs at any point in time. Modern programs in other languages also often use similar strategies, although less strict, and only in certain parts, in order to reduce complexity, normally in conjunction with complementing methodologies such as data structuring, structured programming and object orientation.

l-values and r-values

L-value: "l-value" refers to memory location which identifies an object. l-value may appear as either left hand or right hand side of an assignment operator(=). l-value often represents as identifier.
Expressions referring to modifiable locations are called "**modifiable l-values**". A modifiable l-value cannot have an array type, an incomplete type, or a type with the **const** attribute. For structures and unions to be modifiable **lvalues**, they must not have any members with the **const** attribute. The name of the identifier denotes a storage location, while the value of the variable is the value stored at that location. An identifier is a modifiable **lvalue** if it refers to a memory location and if its type is arithmetic, structure, union, or pointer. For example, if ptr is a pointer to a storage region, then ***ptr** is a modifiable l-value that designates the storage region to which **ptr** points.

In C, the concept was renamed as **"locator value"**, and referred to expressions that locate (designate) objects. The l-value is one of the following:

1. The name of the variable of any type i.e, an identifier of integral, floating, pointer, structure, or union type.
2. A subscript ([ ]) expression that does not evaluate to an array.
3. A unary-indirection (*) expression that does not refer to an array
4. An l-value expression in parentheses.
5. A **const** object (a nonmodifiable l-value).
6. The result of indirection through a pointer, provided that it isn't a function pointer.
7. The result of member access through pointer(-> or .)

```
// declare a an object of type 'int'
int a;

// a is an expression referring to an
// 'int' object as l-value
a = 1;

int b = a; // Ok, as l-value can appear on right

// Switch the operand around '=' operator
9 = a;

// Compilation error:
// as assignment is trying to change the
// value of assignment operator
```

R-value: r-value" refers to data value that is stored at some address in memory. A r-value is an expression that can't have a value assigned to it which means r-value can appear on right but not on left hand side of an assignment operator(=).

```
// declare a, b an object of type 'int'
```

```
int a = 1, b;

a + 1 = b; // Error, left expression is
           // is not variable(a + 1)

// declare pointer variable 'p', and 'q'
int *p, *q; // *p, *q are lvalue

*p = 1; // valid l-value assignment

// below is invalid - "p + 2" is not an l-value
// p + 2 = 18;

q = p + 5; // valid - "p + 5" is an r-value

// Below is valid - dereferencing pointer
// expression gives an l-value
*(p + 2) = 18;

p = &b;

int arr[20]; // arr[12] is an lvalue; equivalent
              // to *(arr+12)
              // Note: arr itself is also an lvalue

struct S { int m; };

struct S obj; // obj and obj.m are lvalues

// ptr-> is an lvalue; equivalent to (*ptr).m
// Note: ptr and *ptr are also lvalues
struct S* ptr = &obj;
```

Note: The unary & (address-of) operator requires an lvalue as its operand. That is, &n is a valid expression only if n is an lvalue. Thus, an expression such as &12 is an error. Again, 12 does not refer to an object, so it's not addressable. For instance,

```
// declare a as int variable and
// 'p' as pointer variable
int a, *p;

p = &a; // ok, assignment of address
        // at l-value

&a = p;    // error: &a is an r-value

int x, y;

(  x < y ? y : x) = 0; // It's valid because the ternary
                  // expression preserves the "lvalue-ness"
                 // of both its possible return values
```

*Remembering the mnemonic, that **lvalues** can appear on the left of an assignment operator while **rvalues** can appear on the right*

Environments and stores

Though Environment Setup is not an element of any Programming Language, it is the first step to be followed before setting on to write a program.

When we say Environment Setup, it simply implies a base on top of which we can do our programming. Thus, we need to have the required software setup, i.e., installation on our PC which will be used to write computer programs, compile, and execute them. For example, if you need to browse Internet, then you need the following setup on your machine −

- A working Internet connection to connect to the Internet
- A Web browser such as Internet Explorer, Chrome, Safari, etc.

If you are a PC user, then you will recognize the following screenshot, which we have taken from the Internet Explorer while browsing tutorialspoint.com.

Similarly, you will need the following setup to start with programming using any programming language.

- A text editor to create computer programs.
- A compiler to compile the programs into binary format.
- An interpreter to execute the programs directly.

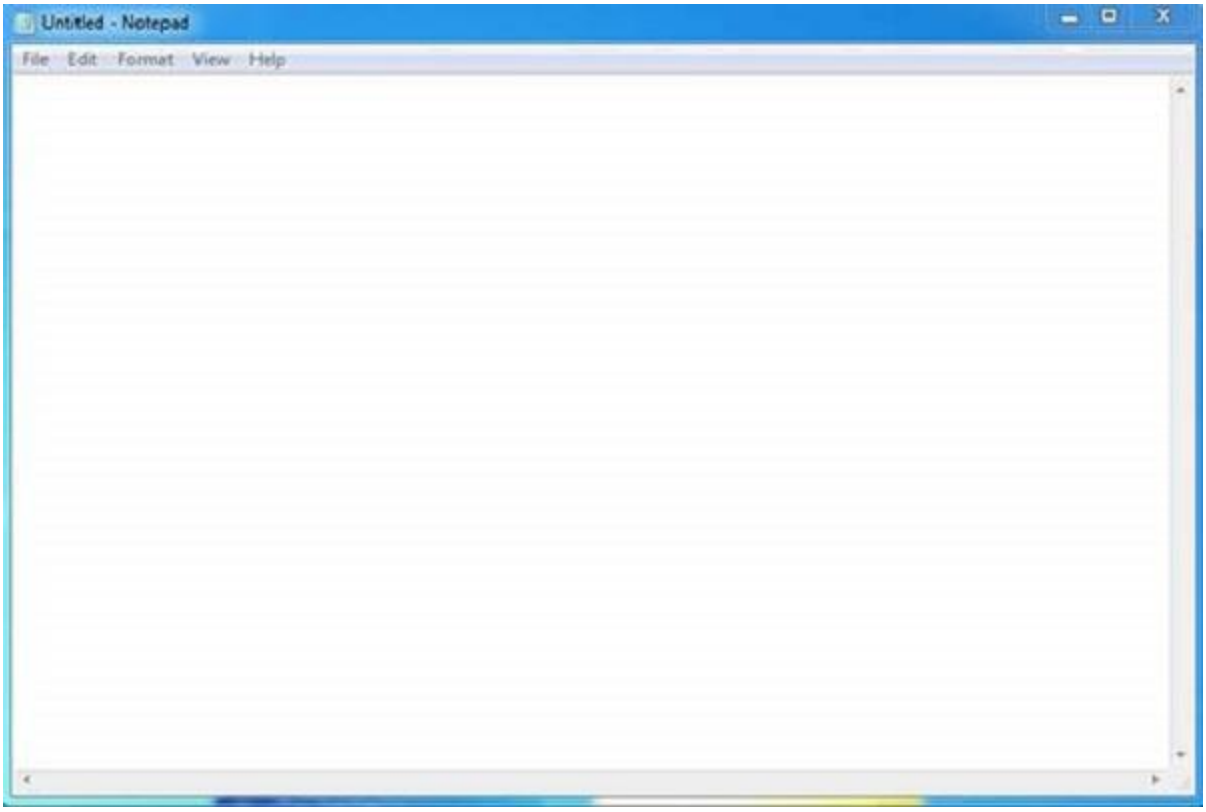
In case you don’t have sufficient exposure to computers, you will not be able to set up either of these software. So, we suggest you take the help from any technical person around you to set up the programming environment on your machine from where you can start. But for you, it is important to understand what these items are.

Text Editor

A text editor is a software that is used to write computer programs. Your Windows machine must have a Notepad, which can be used to type programs. You can launch it by following these steps −

```
Start Icon → All Programs → Accessories → Notepad → Mouse Click on
Notepad
```

It will launch Notepad with the following window −



You can use this software to type your computer program and save it in a file at any location. You can download and install other good editors like **Notepad++**, which is freely available.

If you are a Mac user, then you will have **TextEdit** or you can install some other commercial editor like **BBEdit** to start with.

Compiler?

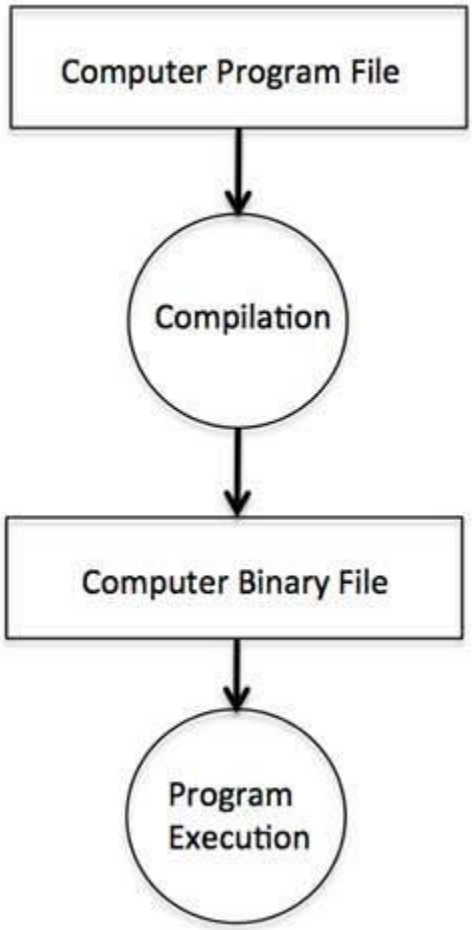
You write your computer program using your favorite programming language and save it in a text file called the program file.

Now let us try to get a little more detail on how the computer understands a program written by you using a programming language. Actually, the computer cannot understand your program directly given in the text format, so we need to convert this program in a binary format, which can be understood by the computer.

The conversion from text program to binary file is done by another software called Compiler and this process of conversion from text formatted program to binary format file is called program compilation. Finally, you can execute binary file to perform the programmed task.

We are not going into the details of a compiler and the different phases of compilation.

The following flow diagram gives an illustration of the process −

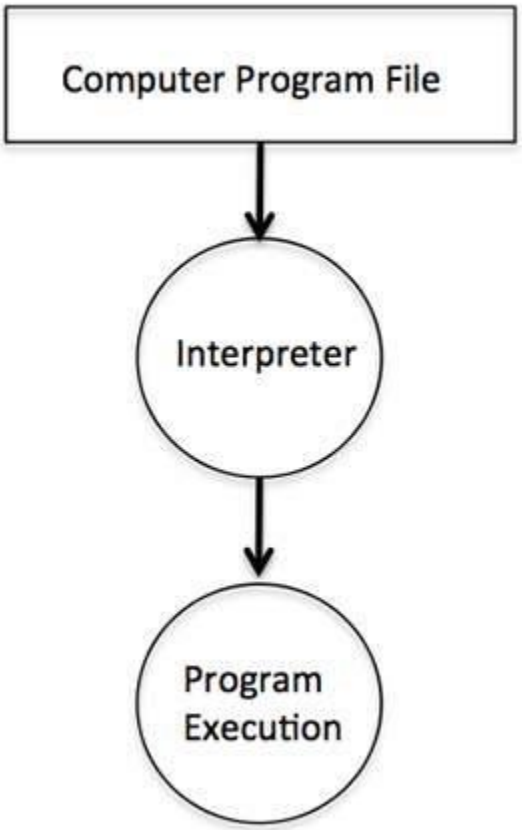


So, if you are going to write your program in any such language, which needs compilation like C, C++, Java and Pascal, etc., then you will need to install their compilers before you start programming.

Interpreter

We just discussed about compilers and the compilation process. Compilers are required in case you are going to write your program in a programming language that needs to be compiled into binary format before its execution.

There are other programming languages such as Python, PHP, and Perl, which do not need any compilation into binary format, rather an interpreter can be used to read such programs line by line and execute them directly without any further conversion.



So, if you are going to write your programs in PHP, Python, Perl, Ruby, etc., then you will need to install their interpreters before you start programming.

Online Compilation

If you are not able to set up any editor, compiler, or interpreter on your machine, then *tutorialspoint.com* provides a facility to compile and run almost all the programs online with an ease of a single click.

So do not worry and let's proceed further to have a thrilling experience to become a computer programmer in simple and easy steps.

Storage Allocation

In computer science, storage allocation is the assignment of particular areas of a magnetic disk to particular data or instructions.

The different ways to allocate memory are:

1. Static storage allocation
2. Stack storage allocation
3. Heap storage allocation

Static storage allocation

- In static allocation, names are bound to storage locations.
- If memory is created at compile time then the memory will be created in static area and only once.
- Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion.
- The drawback with static storage allocation is that the size and position of data objects should be known at compile time.
- Another drawback is restriction of the recursion procedure.

Stack Storage Allocation

- In static storage allocation, storage is organized as a stack.
- An activation record is pushed into the stack when activation begins and it is popped when the activation end.
- Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.
- It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.

Heap Storage Allocation

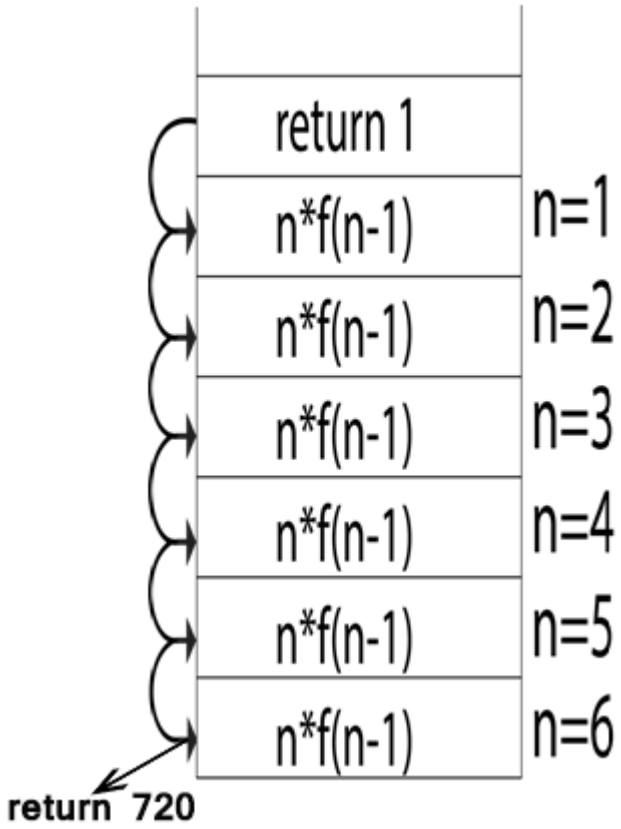
- Heap allocation is the most flexible allocation scheme.
- Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
- Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.

- Heap storage allocation supports the recursion process.

Example:

```
1. fact (int n)
2. {
3.    if (n<=1)
4.        return 1;
5.    else
6.        return (n * fact(n-1));
7. }
8. fact (6)
```

The dynamic allocation is as follows:



Constants

In computer programming, a **constant** is a value that cannot be altered by the program during normal execution, i.e., the value is *constant*.[a] When associated with an identifier, a constant is said to be "named," although the terms "constant" and "named constant" are often used interchangeably. This is contrasted with a *variable*, which is an identifier with a value that can be changed during

normal execution, i.e., the value is *variable.* Constants are useful for both programmers and compilers: For programmers they are a form of self-documenting code and allow reasoning about correctness, while for compilers they allow compile-time and run-time checks that verify that constancy assumptions are not violated, and allow or simplify some compiler optimizations.

There are various specific realizations of the general notion of a constant, with subtle distinctions that are often overlooked. The most significant are: compile-time (statically-valued) constants, run-time (dynamically-valued) constants, immutable objects, and constant types (const).

Typical examples of compile-time constants include mathematical constants, values from standards (here maximum transmission unit), or internal configuration values (here characters per line), such as these C examples:

```
const float PI = 3.1415927;  // maximal single float precision
const unsigned int MTU = 1500;  // Ethernet v2, RFC 894
const unsigned int COLUMNS = 80;
```

Typical examples of run-time constants are values calculated based on inputs to a function, such as this C++ example:

```
void f(std::string s) {
  const size_t l = s.length();
  // ...
}
```

constant initialization

Sets the initial values of the static variables to a compile-time constant.

Syntax

static *T* **&** *ref* **=** ***constexpr*****;**	(1)
static *T object* **=** ***constexpr*****;**	(2)

Explanation

Constant initialization is performed after (until C++14)instead of (since C++14) zero initialization of the static and thread-local objects and before all other initialization. Only the following variables are constant initialized:

1) Static or thread-local references, if it is bound to static glvalue, to a temporary object (or its subobject), or to a function, and if every expression (including implicit conversions) in the initializer of the reference is a constant expression.
2) Static or thread-local object of class type that is initialized by a constructor call, if the constructor is constexpr and all constructor arguments (including implicit conversions) are constant expressions, and if the initializers in the constructor's initializer list and the brace-or-equal initializers of the class members only contain constant expressions.

3) Static or thread-local object (not necessarily of class type), that is not initialized by a constructor call, if the object is value-initialized or if every expression in its initializer is a constant expression.

The effects of constant initialization are the same as the effects of the corresponding initialization, except that it's guaranteed that it is complete before any other initialization of a static or thread-local object begins, and it may be performed at compile time.

Notes

The compiler is permitted to initialize other static and thread-local objects using constant initialization, if it can guarantee that the value would be the same as if the standard order of initialization was followed.

In practice, constant initialization is performed at compile time, and pre-calculated object representations are stored as part of the program image (e.g. in the `.data` section). If a variable is both `const` and constant-initialized, its object representation may be stored in a read-only section of the program image (e.g. the `.rodata` section)

Example

```
#include <iostream>
#include <array>

struct S {
    static const int c;
};
const int d = 10 * S::c; // not a constant expression: S::c has no preceding
                         // initializer, this initialization happens after const
const int S::c = 5;      // constant initialization, guaranteed to happen first
int main()
{
    std::cout << "d = " << d << '\n';
    std::array<int, S::c> a1; // OK: S::c is a constant expression
//  std::array<int, d> a2;    // error: d is not a constant expression
}
```

Output:

```
d = 50
```

Statement-level control structure

Introduction

- A control structure is a control statement and the statements whose execution it controls.
  - Selection Statements
  - Iterative Statements
  - Unconditional Branching
- Overall Design Question:
  - What control statements should a language have, beyond selection and pretest logical loops?

Selection Statements

- A selection statement provides the means of choosing between two or more paths of execution.
- Two general categories:
  - Two-way selectors
  - Multiple-way selectors

Two-Way Selection Statements

- The general form of a two-way selector is as follows:

if control_expression
  then clause
  else clause

Design issues

- What is the form and type of the control expression?
- How are the then and else clauses specified?
- How should the meaning of nested selectors be specified?

The control Expression

- Control expressions are specified in parenthesis if the then reserved word is not used to introduce the then clause, as in the C-based languages.
- In C89, which did not have a Boolean data type, arithmetic expressions were used as control expressions.
- In contemporary languages, such as Java and C#, **only Boolean expressions** can be used for control expressions.

- Fortan includes a **three-way** selector named the arithmetic IF that uses an arithmetic expression for control. It causes control to go to one of three different labeled statements, depending on whether the value of its control expression is **negative, zero, or greater than zero**. This statement is on the obsolescent feature list of Fortran 95.

Clause Form

- In most contemporary languages, the then and else clauses either appear as single statements or compound statements.
- C-based languages use **braces** to form compound statements.
- In Ada the last clause in a selection construct is terminated with **end** and **if**.
- One exception is **Perl**, in which all then and else clauses must be **compound statements**, even if they contain single statements.

Nesting Selectors

- In Java and contemporary languages, the static semantics of the language specify that the **else** clause is always paired with the nearest unpaired **then** clause.

```
if (sum == 0)
   if (count == 0)
        result = 0;
else
        result = 1;
```

- A rule, rather than a syntactic entity, is used to provide the disambiguation.
- So in the above example, the **else** clause would be the alternative to the second then clause.
- To force the alternative semantics in Java, a different syntactic form is required, in which the inner **if-then** is put in a compound, as in

```
if (sum == 0) {
   if (count == 0)
        result = 0;
}
else
result = 1;
```

- C, C++, and C# have the same problem as Java with selection statement nesting.

Multiple Selection Constructs

- The **multiple selection** construct allows the selection of one of any number of statements or statement groups.

Design Issues

- What is the form and type of the control expression?
- How are the selectable segments specified?
- Is execution flow through the structure restricted to include just a single selectable segment?
- What is done about unrepresented expression values?

Examples of Multiple Selectors

- The C, C++, and Java switch

```
switch (expression)      {
  case constant_expression_1 : statement_1;
  ...
  case constant_expression_n : statement_n;
  [default: statement_n+1]
}
```

- The control expression and the constant expressions are integer type.
- The **switch** construct does not provide implicit branches at the end of those code segments.
- Selectable segments can be statement sequences, blocks, or compound statements.
- Any number of segments can be executed in one execution of the construct (a trade-off between reliability and flexibility—convenience.)
- To avoid it, the programmer must supply a break statement for each segment.
- **default** clause is for unrepresented values (if there is no **default**, the whole statement does nothing.)
- C# switch statement rule disallows the implicit execution of more than one segment. The rule is that every selectable segment must end with an explicit unconditional branch statement, either a **break**, which transfers control out of the switch construct, or a **goto**, which can transfer control to on of the selectable segments. C# switch statement example:

```
switch (value)  {
  case -1: Negatives++;
        break;
  case  0: Positives++;
        goto case 1;
  case  1: Positives ++;
  default: Console.WriteLine("Error in switch \n");
 }
```

Multiple Selection Using if

- Early Multiple Selectors:
  - FORTRAN arithmetic IF (a **three-way** selector)
    IF (arithmetic expression) N1, N2, N3
- Bad aspects:
  - Not encapsulated (selectable segments could be anywhere)
  - Segments require GOTOs
  - FORTRAN computed GOTO and assigned GOTO
- To alleviate the poor readability of deeply nested two-way selectors, Ada has been extended specifically for this use.

```
If     Count < 10      then Bag1 := True;
elsif Count < 100      then Bag2 := True;
elsif Count < 1000     then Bag3 := True;
end if;
```

which is equivalent to the following:

```
if Count < 10 then
   Bag1 := True;
else
   if Count < 100 then
      Bag2 := True;
else
     if Count < 1000 then
        Bag3 := True;
     end if;
   end if;
end if;
```

- The **elsif** version is the **more readable** of the two.
- This example is **not easily** simulated with a **switch-case** statement, because each selectable statement is chosen on the basis of a Boolean expression.
- In fact, none of the multiple selectors in contemporary languages are as **general** as the if-then-elsif construct.

Iterative Statement

- The repeated execution of a statement or compound statement is accomplished by iteration zero, one, or more times.
- Iteration is the very essence of the power of computer.
- The repeated execution of a statement is often accomplished in a functional language by recursion rather then by iteration.
- General design issues for iteration control statements:
    1. How is iteration controlled?
    2. Where is the control mechanism in the loop?
- The primary possibilities for iteration control are logical, counting, or a combination of the two.
- The main choices for the location of the control mechanism are the top of the loop or the bottom of the loop.
- The **body** of a loop is the collection of statements whose execution is controlled by the iteration statement.
- The term **pretest** means that the loop completion occurs before the loop body is executed.
- The term **posttest** means that the loop completion occurs after the loop body is executed.
- The iteration statement and the associated loop body together form an **iteration construct**.

Counter-Controlled Loops

- A counting iterative control statement has a var, called the **loop var**, in which the count value is maintained.
- It also includes means of specifying the **intial** and **terminal** values of the loop var, and the difference between sequential loop var values, called the **stepsize**.
- The intial, terminal and stepsize are called the **loop parameters**.
- Design Issues:
  - What are the type and scope of the loop variable?
  - What is the value of the loop variable at loop termination?
  - Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?
  - Should the loop parameters be evaluated only once, or once for every iteration?
- FORTRAN 90's **DO** Syntax:

```
[name:]  DO label variable = initial, terminal [, stepsize]
                ...
         END DO [name]
```

- The label is that of the last statement in the loop body, and the stepsize, when absent, defaults to **1**.
- Loop variable **must** be an **INTEGER** and may be either negative or positive.
- The loop params are allowed to be expressions and can have negative or positive values.
- They are evaluated at the beginning of the execution of the DO statement, and the value is used to compute an iteration count, which then has the number of times the loop is to be executed.
- The loop is controlled by the iteration count, not the loop param, so even if the params are changed in the loop, which is legal, those changes cannot affect loop control.
- The iteration count is an internal var that is inaccessible to the user code.
- The DO statement is a single-entry structure.

The for Statement of the C-Based Languages

- Syntax:

```
for ([expr_1] ; [expr_2] ; [expr_3])
    loop body
```

- The loop body can be a single statement, a compound statement, or a null statement.

```
for (i = 0, j = 10; j == i;  i++)
    ...
```

- All of the expressions of C's for are optional.
- If the second expression is absent, it is an **infinite** loop.
- If the first and third expressions are absent, no assumptions are made.
- The C for design choices are:

  1. There are no explicit loop variable or loop parameters.
  2. All involved vars can be changed in the loop body.
  3. It is legal to branch into a for loop body despite the fact that it can create havoc.

- C's for is **more flexible** than the counting loop statements of Fortran and Ada, because each of the expressions can comprise multiple statements, which in turn allow multiple loop vars that can be of any type.
- Consider the following for statement:

```
for (count1 = 0, count2 = 1.0;
    count1 <= 10 && count2 <= 100.0;
    sum = ++count1 + count2, count2 *= 2.5)
        ;
```

- The operational semantics description of this is:

```
count1 = 0
count2 = 1.0
loop:
   if count1 > 10 goto out
   if count2 > 100.0 goto out
   count1 = count1 + 1
   sum = count1 + count2
   count2 = count2 * 2.5
   goto loop
out…
```

- The loop above does not need and thus **does not** have a loop body.
- C99 and C++ differ from earlier version of C in two ways:
  1. It can use a Boolean expression for loop control.
  2. The first expression can include var definitions.

Logically Controlled Loops

- Design Issues:

    1. Pretest or posttest?
    2. Should this be a special case of the counting loop statement (or a separate statement)?

- C and C++ also have both, but the control expression for the posttest version is treated just like in the pretest case (**while - do** and **do - while**)
- These two statements forms are exemplified by the following C# code:

```
sum = 0;
indat = Console.ReadLine( );
while (indat >= 0) {
   sum += indat;
   indat = Console.ReadLine( );
}

value = Console.ReadLine( );
do {
   value /= 10;
   digits ++;
} while (value > 0);
```

- The only real difference between the do and the while is that the do always causes the loop body to be executed **at least once**.
- Java does not have a **goto**, the loop bodies cannot be entered anywhere but at their beginning.

User-Located Loop Control Mechanisms

- It is sometimes convenient for a programmer to choose a location for loop control other than the top or bottom of the loop.
- Design issues:
  1. Should the conditional be part of the exit?
  2. Should control be transferable out of more than one loop?
- C and C++ have unconditional unlabeled exits (**break**).
- Java, Perl, and C# have unconditional labeled exits (**break** in Java and C#, **last** in Perl).
- The following is an example of nested loops in C#:

```
OuterLoop:
      for (row = 0; row < numRows; row++)
         for (col = 0; col < numCols; col++) {
            sum += mat[row][col];
            if (sum > 1000.0)
                  break outerLoop;
      }
```

- C and C++ include an unlabeled control statement, **continue**, that transfers control to the control mechanism of the smallest enclosing loop.
- This is not an exit but rather a way to skip the rest of the loop statements on the current iteration without terminating the loop structure. Ex:

```
while (sum < 1000) {
   getnext(value);
   if (value < 0) continue;
   sum += value;
}
```

- A negative value causes the assignment statement to be skipped, and control is transferred instead to the conditional at the top of the loop.
- On the other hand, in

```
while (sum < 1000) {
   getnext(value);
   if (value < 0) break;
   sum += value;
}
```

A negative value terminates the loop.

- Java, Perl, and C# have statements similar to continue, except they can include labels that specify which loop is to be continued.
- The motivation for user-located loop exits is simple: They fulfill a common need for goto statements through a highly restricted branch statement.
- The target of a goto can be many places in the program, both above and below the goto itself.

- However, the targets of user-located loop exits must be below the exit and can only follow immediately the end of a compound statement.

Iteration Based on Data Structures

- Concept: use order and number of elements of some data structure to control iteration.
- C#'s **foreach** statement iterates on the elements of array and other collections.

```
String[ ] strList = {"Bob", "Carol", "Ted", "Beelzebub"};
…
foreach (String name in strList)
   Console.WriteLine("Name: {0}", name);
```

- The notation `{0}` in the parameter to `Console.WriteLine` above indicates the position in the string to be displayed where the value of the first named variable, `name` in this example, is to be placed.
- Control mechanism is a call to a function that returns the next element in some chosen order, if there is one; else exit loop
- C's **for** can be used to build a user-defined **iterator**.

```
for (p=root; p==NULL; traverse(p)) {
    …
}
```

- The iterator is called at the beginning of each iteration, and each time it is called, the iterator returns an element from a particular data structure in some specific order.

Unconditional Branching

- An unconditional branch statement transfers execution control to a specified place in the program.

Problems with Unconditional Branching

- The unconditional branch, or **goto**, is the most powerful statement for controlling the flow of execution of a program's statements.
- However, using the goto carelessly can lead to **serious problems**.
- Without restrictions on use, imposed either by language design or programming standards, goto statements can make programs virtually unreadable, and as a result, **highly unreliable** and **difficult to maintain**.
- There problems follow directly from a goto's capability of forcing any program statement to follow any other in execution sequence, regardless of whether the statement proceeds or follows the first in textual order.
- **Java doesn't have a goto**. However, most currently popular languages include a goto statement.
- **C#** uses goto in the **switch** statement.

PRINCIPLES OF PROGRAMMING LANGUAGES

UNIT-2

Primitive Types

Pointers

In computer science, a **pointer** is a programming language object that stores a memory address. This can be that of another value located in computer memory, or in some cases, that of memory mapped computer hardware. A pointer *references* a location in memory, and obtaining the value stored at that location is known as *dereferencing* the pointer. As an analogy, a page number in a book's index could be considered a pointer to the corresponding page; dereferencing such a pointer would be done by flipping to the page with the given page number and reading the text found on that page. The actual format and content of a pointer variable is dependent on the underlying computer architecture.

Using pointers significantly improves performance for repetitive operations like traversing iterable data structures, e.g. strings, lookup tables, control tables and tree structures. In particular, it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point.

Pointers are also used to hold the addresses of entry points for called subroutines in procedural programming and for run-time linking to dynamic link libraries (DLLs). In object-oriented programming, pointers to functions are used for binding methods, often using what are called virtual method tables.

A pointer is a simple, more concrete implementation of the more abstract *reference* data type. Several languages, especially low-level languages, support some type of pointer, although some have more restrictions on their use than others. While "pointer" has been used to refer to references in general, it more properly applies to data structures whose interface explicitly allows the pointer to be manipulated (arithmetically via *pointer arithmetic*) as a memory address, as opposed to a magic cookie or capability which does not allow such.[*citation needed*] Because pointers allow both protected and unprotected access to memory addresses, there are risks associated with using them, particularly in the latter case. Primitive pointers are often stored in a format similar to an integer; however, attempting to dereference or "look up" such a pointer whose value is not a valid memory address will cause a program to crash. To alleviate this potential problem, as a matter of type safety, pointers are considered a separate type parameterized by the type of data they point to, even if the underlying representation is an integer. Other measures may also be taken (such as validation & bounds checking), to verify that the pointer variable contains a value that is both a valid memory address and within the numerical range that the processor is capable of addressing.

Structured types

A structured data type is a compound data type which falls under user-defined category and used for grouping simple data types or other compound data types. This contains a

sequence of member variable names along with their type/attributes and they are enclosed within curl brackets.

```
struct < struct name > {
  < type > < member >;
};
```

Need for Struct data types

There are some situations when we need to group different types of variables in one group. Let's see one situation here- we want to store the name, roll and age of a student.

```
unsigned int student_roll;
char student_name [MAX_STRING];
unsigned int student_age;
```

Here we have a logical grouping between there three variables but still these three variables are scattered. We are accessing three different variables for storing attribute values of a single student. Now how to group this inside one logical entity, like three variables for a single student grouped inside one variable. This type of grouping is called structure. One point to note here is that array can group only same type elements and here structure has different types.

Example of Struct data types

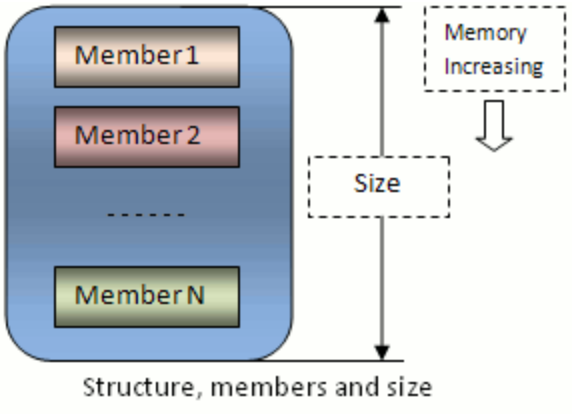
Let's define these again with a structure type

```
struct student_t {
  unsigned int roll;
  char name[MAX_STRING];
  unsigned int age;
};
struct student_t student1;
student1.roll = <roll value>;
strcpy (student1.name, <name>);
student1.age = <age>
```

Structure size and memory layout

Structure is user defined type to group of different type of variables of either compiler defined legacy types or other user defined types or mixed. Individual entity of a

structure element is called member. Members inside a structure are placed sequentially next to next in the memory layout. Thus minimum size of a structure is the sum total of all sizes of members, with considering padding.



Structure, members and size

Struct data types syntax

```
struct <name> {
  <type> <member name 1>;
  <type> <member name 2>;
  ...
  <type> <member name N>;
};
```

Structure usage & applications

Structures are the most applicable field in any C program. Practical use of structure data type is count less. However below are some frequent applicable fields-

1. Student, Employee, Customer etc data records,
2. Objects displayed in Video games,
3. Nodes in linked list, queue, LIFO,
4. Binary tree and other Tree type objects
5. File, Folders, volumes, partitions in file system and explorer nodes,
6. All the GUI elements, Window, buttons, menu items,
7. Header attributes of data files like, images (bitmaps, PNG, JPEG, GIF), sounds(wav, mp3, midi), video(mp4, avi, DIVX),
8. Data structures used in OS (TASK, IPC, locks, device, driver etc),
9. Frames, Packets,etc in TCP IP and OSI communication layers
10. Any application or services handing objects.

Structure demo example

```c
/* Structure type demo example program */
#include<stdio.h>
/* Structure type student */
struct student {
  char name[100];
  char dept[100];
  int rollno;
  float marks;
};
/* Structure type main routine */
int main (int argc, char *argv[])
{
  /* declare struct variable */
  struct student s1;
  printf("\nEnter the name, dept, roll number and marks of
student:\n");
  scanf("%s %s %d %f", s1.name, s1.dept, &s1.rollno, &s1.marks);
  printf("\nThe name, dept, roll number and marks of the student
are:");
  printf("\n%s %s %d %.2f",s1.name,s1.dept,s1.rollno,s1.marks);
}
```

Program output

```
Enter the name, dept, roll number and marks of student:

Student1 ECE 1 96.5


The name, dept, roll number and marks of the student are:

Student1 ECE 1 96.50
```

Coercion

Many programming languages support the conversion of a value into another of a different data type. This kind of type conversions can be implicitly or explicitly made. Implicit conversion, which is also called coercion, is automatically done. Explicit conversion, which is also called casting, is performed by code instructions. This code treats a variable of one data type as if it belongs to a different data type. The languages that support implicit conversion define the rules that will be automatically applied when primitive compatible values are involved. The C code below illustrates implicit and explicit coercion. In line 2 the int constant 3 is automatically converted to double before assignment (implicit coercion). An explicit coercion is performed by involving the destination type with parenthesis, which is done in line 3.

```
1 double x, y;
2 x = 3;             // implicitly coercion (coercion)
3 y = (double) 5;    // explicitly coercion (casting)
```

A function is considered a polymorphic one when it is permited to perform implicit or explicit parameter coercion. If the same is valid for operands, the related operator is considered a polymorphic operator. Below, a piece of C++ code exemplifies these polymorphic expressions.

```
#include <iostream>
void f(double x) {      // polymorphic function
  std::cout << x << std::endl;
}

int main() {
  double a = 5 + 6.3;  // polymorphic operator
  std::cout << a << std::endl;

  f(5);
  f((double) 6);
}
```

Notion of type equivalence

A structural type system (or property-based type system) is a major class of type system, in which type compatibility and equivalence are determined by the type's structure, and not by other characteristics such as its name or place of declaration. Structural systems are used to determine if types are equivalent and whether a type is a subtype of another. It contrasts with nominative systems, where comparisons are based on explicit declarations or the names of the types, and duck typing, in which only the part of the structure accessed at runtime is checked for compatibility.

A type-checker for a statically typed language must verify that the type of any expression is consistent with the type expected by the context in which that expression appears. For instance, in an assignment statement of the form x := e, the inferred type of the expression e must be consistent with the declared or inferred type of the variable x. This notion of consistency, called compatibility, is specific to each programming language.

TYPE CHECKING RULES usually have the form

> **if** two type expressions are equivalent
> **then** return a given type
> **else** return **type_error**

KEY IDEAS. The central issue is then that we have to define when two given type expressions are equivalent.

- The main difficulty arises from the fact that most modern languages allow the naming of user-defined types.
- For instance, in C and C++ this is achieved by the `typedef` statement.
- When checking equivalence of named types, we have two possibilities.

  Name equivalence.
  Treat named types as basic types. Therefore two type expressions are *name equivalent* if and only if they are identical, that is if they can be represented by the same syntax tree, with the same labels.
  Structural equivalence.
  Replace the named types by their definitions and recursively check the substituted trees.

STRUCTURAL EQUIVALENCE. If type expressions are built from basic types and constructors (without type names, that is in our example, when using products instead of records), structural equivalence of types can be decided easily.

- For instance, to check whether the constructed types `array(n1,T1)` and `array(n2,T2)` are equivalent
  - we can check that the integer values `n1` and `n2` are equal and recursively check that `T1` and `T2` are equivalent,
  - or we can be less restrictive and check only that `T1` and `T2` are equivalent.
- Compilers use representations for type expressions (trees or dags) that allow type equivalence to be tested quickly.

RECURSIVE TYPES. In PASCAL a *linked list* is usually defined as follows.

```
type link = ^ cell;
     cell = record
               info: type;
               next: link;
            end;
```

The corresponding type graph has a cycle. So to decide structural equivalence of two types represented by graphs PASCAL compilers put a *mark* on each visited node (in order not to visit a node twice). In C, a *linked list* is usually defined as follows.

```
struct cell {
    int info;
    struct cell *next;
};
```

To avoid cyclic graphs, C compilers

- require type names to be declared before they are used, except for pointers to records.
- use structural equivalence except for records for which they use name equivalence.

Polymorphism

Polymorphism is an object-oriented programming concept that refers to the ability of a variable, function or object to take on multiple forms. A language that features polymorphism allows developers to program in the general rather than program in the specific.

In a programming language that exhibits polymorphism, objects of classes belonging to the same hierarchical tree (inherited from a common base class) may possess functions bearing the same name, but each having different behaviors.

As an example, assume there is a base class named Animals from which the subclasses Horse, Fish and Bird are derived. Also assume that the Animals class has a function named Move, which is inherited by all subclasses mentioned. With polymorphism, each subclass may have its own way of implementing the function. So, for example, when the Move function is called in an object of the Horse class, the function might respond by displaying trotting on the screen. On the other hand, when the same function is called in an object of the Fish class, swimming might be displayed on the screen. In the case of a Bird object, it may be flying.

In effect, polymorphism cuts down the work of the developer because he can now create a sort of general class with all the attributes and behaviors that he envisions for it. When the time comes for the developer to create more specific subclasses with certain unique attributes and behaviors, the developer can simply alter code in the specific portions where the behaviors differ. All other portions of the code can be left as is.

Overloading

Overloading refers to the ability to use a single identifier to define multiple methods of a class that differ in their input and output parameters. Overloaded methods are generally used when they conceptually execute the same task but with a slightly different set of parameters.

Overloading is a concept used to avoid redundant code where the same method name is used multiple times but with a different set of parameters. The actual method that gets called during runtime is resolved at compile time, thus avoiding runtime errors. Overloading provides code clarity, eliminates complexity, and enhances runtime performance.

Overloading is used in programming languages that enforce type-checking in function calls during compilation. When a method is overloaded, the method chosen will be selected at compile time. This is not the same as virtual functions where the method is defined at runtime.

Unlike Java, C# allows operators to be overloaded, in addition to methods, by defining static members using the operator keyword. This feature helps to extend and customize the semantics of operators relevant to user-defined types so that they can be used to manipulate object instances with operators.

The overload resolution in C# is the method by which the right function is selected on

the basis of arguments passed and the list of candidate function members that have the same name. The different contexts in which the overload resolution is used include:

- Invocation of a method in an expression
- Constructor during object creation
- Indexer accessor through an element access and predefined or user-defined operator expression

It is recommended to avoid overloading across inheritance boundaries because it can cause confusion. Overloading can become cumbersome to developers if it is used excessively and with user-defined types as parameters because it can reduce the readability and maintainability of code.

Inheritance

In object-oriented programming, **inheritance** is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation. Also defined as deriving new classes (sub classes) from existing ones (super class or base class) and forming them into a hierarchy of classes. In most class-based object-oriented languages, an object created through inheritance (a "child object") acquires all the properties and behaviors of the parent object (except: constructors, destructor, overloaded operators and friend functions of the base class). Inheritance allows programmers to create classes that are built upon existing classes,[1] to specify a new implementation while maintaining the same behaviors (realizing an interface), to reuse code and to independently extend original software via public classes and interfaces. The relationships of objects or classes through inheritance give rise to a directed graph. Inheritance was invented in 1969 for Simula.[2]

An inherited class is called a **subclass** of its parent class or super class. The term "inheritance" is loosely used for both class-based and prototype-based programming, but in narrow use the term is reserved for class-based programming (one class *inherits from* another), with the corresponding technique in prototype-based programming being instead called *delegation* (one object *delegates to* another).

Inheritance should not be confused with subtyping.[3][4] In some languages inheritance and subtyping agree,[a] whereas in others they differ; in general, subtyping establishes an is-a relationship, whereas inheritance only reuses implementation and establishes a syntactic relationship, not necessarily a semantic relationship (inheritance does not ensure behavioral subtyping). To distinguish these concepts, subtyping is also known as *interface inheritance*, whereas inheritance as defined here is known as *implementation inheritance* or *code inheritance*.[5] Still, inheritance is a commonly used mechanism for establishing subtype relationships.[6]

Inheritance is contrasted with object composition, where one object *contains* another object (or objects of one class contain objects of another class); see composition over inheritance. Composition implements a has-a relationship, in contrast to the is-a relationship of subtyping.

Type Parameterization

Type parameterization allows you to write generic classes and traits.

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods −

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example

Following example illustrates how we can print an array of different type using a single Generic method −

Live Demo

```
public class GenericMethodTest {
   // generic method printArray
   public static < E > void printArray( E[] inputArray ) {
      // Display array elements
      for(E element : inputArray) {
         System.out.printf("%s ", element);
      }
      System.out.println();
   }
```

```
   public static void main(String args[]) {
      // Create arrays of Integer, Double and Character
      Integer[] intArray = { 1, 2, 3, 4, 5 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
      Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      System.out.println("Array integerArray contains:");
      printArray(intArray);   // pass an Integer array

      System.out.println("\nArray doubleArray contains:");
      printArray(doubleArray);   // pass a Double array

      System.out.println("\nArray characterArray contains:");
      printArray(charArray);   // pass a Character array
   }
}
```

This will produce the following result −

Output

```
Array integerArray contains:
1 2 3 4 5

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O
```

Bounded Type Parameters

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Example

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects −

Live Demo

```
public class MaximumTest {
   // determines the largest of three Comparable objects
```

```
   public static <T extends Comparable<T>> T maximum(T x, T y, T z)
{
      T max = x;   // assume x is initially the largest

      if(y.compareTo(max) > 0) {
         max = y;   // y is the largest so far
      }

      if(z.compareTo(max) > 0) {
         max = z;   // z is the largest now
      }
      return max;   // returns the largest object
   }

   public static void main(String args[]) {
      System.out.printf("Max of %d, %d and %d is %d\n\n",
         3, 4, 5, maximum( 3, 4, 5 ));

      System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",
         6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));

      System.out.printf("Max of %s, %s and %s is %s\n","pear",
         "apple", "orange", maximum("pear", "apple", "orange"));
   }
}
```

This will produce the following result −

Output

```
Max of 3, 4 and 5 is 5

Max of 6.6,8.8 and 7.7 is 8.8

Max of pear, apple and orange is pear
```

Generic Classes

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example

Following example illustrates how we can define a generic class −

Live Demo

```
public class Box<T> {
   private T t;

   public void add(T t) {
      this.t = t;
   }

   public T get() {
      return t;
   }

   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      Box<String> stringBox = new Box<String>();

      integerBox.add(new Integer(10));
      stringBox.add(new String("Hello World"));

      System.out.printf("Integer Value :%d\n\n", integerBox.get());
      System.out.printf("String Value :%s\n", stringBox.get());
   }
}
```

This will produce the following result −

Output

```
Integer Value :10
String Value :Hello World
```

Abstract data types

The Data Type is basically a type of data that can be used in different computer program. It signifies the type like integer, float etc, the space like integer will take 4-bytes, character will take 1-byte of space etc.

The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword “Abstract” is used as we can use these datatypes, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive datatypes, but operation logics are hidden.

Some examples of ADT are Stack, Queue, List etc.

Let us see some operations of those mentioned ADT −

- Stack −
  - isFull(), This is used to check whether stack is full or not
  - isEmpry(), This is used to check whether stack is empty or not
  - push(x), This is used to push x into the stack

    - pop(), This is used to delete one element from top of the stack
    - peek(), This is used to get the top most element of the stack
    - size(), this function is used to get number of elements present into the stack
- Queue −
    - isFull(), This is used to check whether queue is full or not
    - isEmpry(), This is used to check whether queue is empty or not
    - insert(x), This is used to add x into the queue at the rear end
    - delete(), This is used to delete one element from the front end of the queue
    - size(), this function is used to get number of elements present into the queue
- List −
    - size(), this function is used to get number of elements present into the list
    - insert(x), this function is used to insert one element into the list
    - remove(x), this function is used to remove given element from the list
    - get(i), this function is used to get element at position i
    - replace(x, y), this function is used to replace x with y value

Information Hiding

Information hiding is the process of hiding the details of an object or function. The hiding of these details results in an abstraction, which reduces the external complexity and makes the object or function easier to use. In addition, information hiding effectively decouples the calling code from the internal workings of the object or function being called, which makes it possible to change the hidden portions without having to also change the calling code. Encapsulation is a common technique programmers use to implement information hiding.

Abstraction

Abstraction (from the Latin *abs*, meaning *away from* and *trahere* , meaning *to draw*) is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics. In object-oriented programming, abstraction is one of three central principles (along with encapsulation and inheritance). Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency. In the same way that abstraction sometimes works in art, the object that remains is a representation of the original, with unwanted detail omitted. The resulting object itself can be referred to as an abstraction, meaning *a named entity made up of selected*

attributes and behavior specific to a particular usage of the originating entity. Abstraction is related to both encapsulation and data hiding.

In the process of abstraction, the programmer tries to ensure that the entity is named in a manner that will make sense and that it will have all the relevant aspects included and none of the extraneous ones. A real-world analogy of abstraction might work like this: You (the object) are arranging to meet a blind date and are deciding what to tell them so that they can recognize you in the restaurant. You decide to include the information about where you will be located, your height, hair color, and the color of your jacket. This is all data that will help the procedure (your date finding you) work smoothly. You should include all that information. On the other hand, there are a lot of bits of information about you that aren't relevant to this situation: your social security number, your admiration for obscure films, and what you took to "show and tell" in fifth grade are all irrelevant to this particular situation because they won't help your date find you. However, since entities may have any number of abstractions, you may get to use them in another procedure in the future.

Scope

The scope of an identifier is a part of the program in which the identifier can be used to access its object. There are different categories of scope: block (or local), function, function prototype, and file. These categories depend on how and where identifiers are declared.

- **Block**: The scope of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such block is known as the enclosing block). Parameter declarations with a function definition also have block scope, limited to the scope of the function body.
- **File**: File scope identifiers, also known as *globals*, are declared outside of all blocks; their scope is from the point of declaration to the end of the source file.
- **Function**: The only identifiers having function scope are statement labels. Label names can be used with goto statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing `label_name:` followed by a statement. Label names must be unique within a function.
- **Function prototype**: Identifiers declared within the list of parameter declarations in a function prototype (not as a part of a function definition) have

a function prototype scope. This scope ends at the end of the function prototype.

Visibility

The visibility of an identifier is a region of the program source code from which an identifier's associated object can be legally accessed.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily hidden by the appearance of a duplicate identifier: the object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier ends.

Technically, visibility cannot exceed a scope, but a scope *can* exceed visibility. See the following example:

```
void f (int i) {
 int j;           // auto by default
 j = 3;           // int i and j are in scope and visible

   {              // nested block
    double j;     // j is local name in the nested block
    j = 0.1;      // i and double j are visible;
                  // int j = 3 in scope but hidden
   }
                  // double j out of scope
 j += 1;          // int j visible and = 4
}
// i and j are both out of scope
```

Procedures

In computer programming, a **procedure** is a set of coded instructions that tell a computer how to run a program or calculation. Many different types of programming languages can be used to build a procedure. Depending on the programming language, a procedure may also be called a subroutine, subprogram or function.

Modules

A module is a software component or part of a program that contains one or more routines. One or more independently developed modules make up a program. An enterprise-level software application may contain several different modules, and each module serves unique and separate business operations.

Modules make a programmer's job easy by allowing the programmer to focus on only one area of the functionality of the software application. Modules are typically incorporated into the program (software) through interfaces.

Software applications include many different tasks and processes that cohesively serve all paradigms within a complete business solution. Early software versions were gradually built from an original and basic level, and development teams did not yet have the ability to use prewritten code.

The introduction of modularity allowed programmers to reuse prewritten code with new applications. Modules were created and bundled with compilers, in which each module performed a business or routine operation within the program.

For example, Systems, Applications and Products in Data Processing (SAP) - an enterprise resource planning (ERP) software - is comprised of several large modules (for example, finance, supply chain and payroll, etc.), which may be implemented with little or no customization. A classic example of a module-based application is Microsoft Word, which contains modules incorporated from Microsoft Paint that help users create drawings or figures.

Classes

In object-oriented programming, a **class** is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).[1][2] In many languages, the class name is used as the name for the class (the template itself), the name for the default constructor of the class (a subroutine that creates objects), and as the type of objects generated by instantiating the class; these distinct concepts are easily conflated.[2]

When an object is created by a constructor of the class, the resulting object is called an instance of the class, and the member variables specific to the object are called instance variables, to contrast with the class variables shared across the class.

In some languages, classes are only a compile-time feature (new classes cannot be declared at run-time), while in other languages classes are first-class citizens, and are generally themselves objects (typically of type `Class` or similar). In these languages, a class that creates classes is called a metaclass.

Packages

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or

"API" for short. Its packages represent the tasks most commonly associated with general-purpose programming. For example, a `String` object contains state and behavior for character strings; a `File` object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem; a `Socket` object allows for the creation and use of network sockets; various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

The Java Platform API Specification contains the complete listing for all packages, interfaces, classes, fields, and methods supplied by the Java SE platform. Load the page in your browser and bookmark it. As a programmer, it will become your single most important piece of reference documentation.

Objects

In computer science, an **object** can be a variable, a data structure, a function, or a method, and as such, is a value in memory referenced by an identifier.

In the class-based object-oriented programming paradigm, *object* refers to a particular instance of a class, where the object can be a combination of variables, functions, and data structures.

In relational database management, an object can be a table or column, or an association between data and a database entity (such as relating a person's age to a specific person).

Object-Oriented Programming

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as *attributes* or *properties*), and code, in the form of procedures (often known as *methods*). A feature of objects is an object's procedures that can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another.[1][2] OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

Many of the most widely used programming languages (such as C++, Java, Python, etc.) are multi-paradigm and they support object-oriented programming to a greater or lesser degree, typically in combination with imperative, procedural programming. Significant object-oriented languages include Java, C++, C#, Python, PHP, JavaScript, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Scala, Common Lisp, MATLAB, and Smalltalk.

PRINCIPLES OF PROGRAMMING LANGUAGES

UNIT-3

Storage Management

Storage Management is defined as it refers to the management of the data storage equipment's that are used to store the user/computer generated data. Hence it is a tool or set of processes used by an administrator to keep your data and storage equipment's safe.
Storage management is a process for users to optimize the use of storage devices and to protect the integrity of data for any media on which it resides and the category of storage management generally contain the different type of subcategories covering aspects such as security, virtualization and more, as well as different types of provisioning or automation, which is generally made up the entire storage management software market.

Storage management key attributes:
Storage management has some key attribute which is generally used to manage the storage capacity of the system. These are given below:
1. Performance
2. Reliability
3. Recoverability
4. Capacity

Feature of Storage management:
There is some feature of storage management which is provided for storage capacity. These are given below:

1. Storage management is a process that is used to optimize the use of storage devices.
2. Storage management must be allocated and managed as a resource in order to truly benefit a corporation.
3. Storage management is generally a basic system component of information systems.
4. It is used to improve the performance of their data storage resources.

Advantage of storage management:
There are some advantage of storage management which are given below:

- It is very simple to managed a storage capacity.
- It is generally take a less time.
- It is improve the performance of system.
- In virtualization and automation technologies can help an organization improve its agility.

Static and dynamic memory management

Static Memory Allocation: Memory is allocated for the declared variable by the compiler. The address can be obtained by using 'address of' operator and can be assigned to a pointer. The memory is allocated during compile time. Since most of the declared variables have static memory, this kind of assigning the address of a variable to a pointer is known as static memory

allocation.

Dynamic Memory Allocation: Allocation of memory at the time of execution (run time) is known as dynamic memory allocation. The functions calloc() and malloc() support allocating of dynamic memory. Dynamic allocation of memory space is done by using these functions when value is returned by functions and assigned to pointer variables.

Tabular Difference Between Static and Dynamic Memory Allocation in C:

S.No	Static Memory Allocation	Dynamic Memory Allocation
1	In the static memory allocation, variables get allocated permanently.	In the Dynamics memory allocation, variables get allocated only if your program unit gets active.
2	Static Memory Allocation is done before program execution.	Dynamics Memory Allocation is done during program execution.
3	It uses stack for managing the static allocation of memory	It uses heap for managing the dynamic allocation of memory
4	It is less efficient	It is more efficient
5	In Static Memory Allocation, there is no memory re-usability	In Dynamics Memory Allocation, there is memory re-usability and memory can be freed when not required
6	In static memory allocation, once the memory is allocated, the memory size can not change.	In dynamic memory allocation, when memory is allocated the memory size can be changed.
7	In this memory allocation scheme, we cannot reuse the unused memory.	This allows reusing the memory. The user can allocate more memory when required. Also, the user can release the memory when the user needs it.
8	In this memory allocation scheme, execution is faster than dynamic memory allocation.	In this memory allocation scheme, execution is slower than static memory allocation.
9	In this memory is allocated at compile time.	In this memory is allocated at run time.
10	In this allocated memory remains from start to end of the program.	In this allocated memory can be released at any time during the program.

11	**Example:** This static memory allocation is generally used for array.	**Example:** This dynamic memory allocation is generally used for linked list.

Stack based memory management

Stack Allocation : The allocation happens on contiguous blocks of memory. We call it stack memory allocation because the allocation happens in function call stack. The size of memory to be allocated is known to compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is deallocated. This all happens using some predefined routines in compiler. Programmer does not have to worry about memory allocation and deallocation of stack variables.

Heap based memory management

Heap Allocation : The memory is allocated during execution of instructions written by programmers. Note that the name heap has nothing to do with heap data structure. It is called heap because it is a pile of memory space available to programmers to allocated and de-allocate. If a programmer does not handle this memory well, memory leak can happen in the program.

Key Differences Between Stack and Heap Allocations

1. In a stack, the allocation and deallocation is automatically done by whereas, in heap, it needs to be done by the programmer manually.
2. Handling of Heap frame is costlier than handling of stack frame.
3. Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
4. Stack frame access is easier than the heap frame as the stack have small region of memory and is cache friendly, but in case of heap frames which are dispersed throughout the memory so it cause more cache misses.
5. Stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.
6. Accessing time of heap takes is more than a stack.

Comparison Chart:

PARAMETER	STACK	HEAP
Basic	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and Deallocation	Automatic by compiler instructions.	Manual by programmer.
Cost	Less	More
Implementation	Hard	Easy

PARAMETER	STACK	HEAP
Access time	Faster	Slower
Main Issue	Shortage of memory	Memory fragmentation
Locality of reference	Excellent	Adequate
Flexibility	Fixed size	Resizing is possible
Data type structure	Linear	Hierarchical

Sequence Control

The control of execution of the operations, both primitive and user defined, is termed as sequence control.
Sequence control structures are categorized into following four groups:

1. Expressions: These form the basic building blocks for statements and express how at are manipulated and changed by a program. Properties such as precedence rules and parentheses determine how expressions become evaluated.
2. Statement: Statements such as conditional or iterative statements, determine how control flows from one segment of a program to another.

3. Declarative programming: It is an execution model that does not depend on statements, but nevertheless causes execution to proceed through a program.

4. Subprograms: Subprograms such as subprogram calls and coroutines, form a way to transfer control from one segment of a program to another.

Implicit and Explicit Sequence Control:
Sequence control structures may be either implicit or explicit.

i) Implicit sequence-control structures:
These are defined by language to be in effect 'un1es'smodified by the programmer through some explicit structure. For example, most languages define the physical sequence of statements in a program as controlling the sequence in which statements are executed, unless modified by an explicit sequence-control statement.

ii) Explicit sequence-control structure
These are the sequence-control structures that the programmer may optionally use to modify the implicit sequence of operations defined by the language.

Implicit and explicit sequencing with arithmetic expressions

- **Prefix Notation**
  - Operators come first, then the operands
  - e.g. (a + b) * (c - a) => * + a b - c a
  - No ambiguity and no parenthesis needed
  - Number of arguments for an operator must be known a priori
  - Relatively easy to decode using stack mechanism
- **Postfix notation**
  - An operator follows its operands
  - e.g. (a+b) *(c-a) => a b + c a - *
  - No ambiguity and no parenthesis needed

- Relatively easy to decode using stack mechanism

➢ **Infix notation**
- Only suitable for binary operations (Thus used with prefix)
- For more than one infix operator, it is inherently ambiguous
- Parenthesis is used to explicitly annotate the order

➢ **Implicit control rules**
- ***Hierarchy of operations (precedence) e.g. Ada***
  - **, abs, not : Exponential absolute negation
  - * / mod : Multiplication, division,
  - + - : Unary addition and subtraction
  - + - & : Binary addition and subtraction
  - = < > : Relational
  - and or xor : Boolean operatins
- ***Associative***
  - left-right associativity (+, -, others)
  - right-left associativity (exponential)

➢ **Issues in Evaluating Expressions**
- **Uniform Evaluation rules (eager and lazy)**
  - ***Eager Evaluation:*** evaluate the operands as soon as they appear (parallel processing SNOBOL)
  - ***Lazy Evaluation:*** Delay evaluation of operands as late as possible (LISP)

  - ```
    foo( x )
    { ...
    if ( a > 0 )
    a = x;
    else a = a + 1;
    }
    foo( b/a );
    ```

- **Side effects**
  - a * foo(x) + a; say a= 1; foo(x) generates 3
    - if each term is evaluated 1 * 3 + 2 = 5
    - if a is evaluated only once 1 * 3 + 1 = 4
    - if evaluate foo(x) first 2 * 3 + 2 = 8
- **Error Condition:** No solution other than exceptional statements
- **Short-circuit expression:** Use the characteristics of “and” or “or” operation

- a b - a is true b is not evaluated
- a && b - a is false b is not evaluated
- e.g.
  ```
  b = 9;
  if ( a = 4 b = 3 )
  printf (" a = %d b = %d\n");
  ```
  ```

Implicit and explicit sequencing with non-arithmetic expressions

- **Pattern Matching**
  Pattern matching operation: An operation matches and assigns a set of variables to a predefined template

  e.g. (palindromes)
  A -> 0A0 1A1 0 1
  matches 00100 -> 00A00 -> 0A0 -> A
  applied term rewriting 00A00 is a term rewrite of 00100

  term rewriting: A restricted form of pattern match

  e.g. Given string a1a2a3a4 and α -> β.
  if α = a3 then a1a2βa4
  we say a1a2βa4 is term rewrite of a1a2a3a4

  e.g.) in ML
  fun fact(1) = 1
  fact(N:int) = N * fact(N - 1);
  fun length(nil) = 0
  length(a :: y) = 1 + length(y)

  Unification and substitution: In Prolog, Database consists of facts and rules
  Substitution: replacement of a string to another one
  Unification: A pattern match to determine if the query has a valid substitution consistent with the rules and facts in the database

  Fact : ParentOf(Ann, John) ParentOf(Sam, Ann)
  Fact : ParentOf(Tom, John)
  Query : ParentOf(x, John)
  Þ X = Ann, x = Tom
  Rule : GrandOarentOf(X,Y) :- ParentOf(X,Z), ParentOf(Z,Y)
  then query : GrandParentOf (x,John) => x = sam

- **Backtracking**
  Backup to the previous subgoal that matched and try another possible goal for it to match when current subgoal is failed.

Sequence control between statements

Basic Statements
Basic statements are assignments, subprogram calls, and input and output statements. Within a basic statement, sequences of operations may be invoked by using expressions.

Assignments to Data Objects
Change the computation by assigning values to data objects is major mechanism that affects the state of computation by programs.

Assignment statements
The primary purpose of an assignment is to assign to the L-value of a data object, the R-value of some expression.

The assignment operations are
A:=B // Pascal, Ada
A=B // C, FORTRAN, PL/I, Prolog, ML, SNOBOL4
MOVE B TO A // COBOL
A<- B // APL

And also
A=B;
A+=B
++A;
A++; these are all the assignment statements are available.

Input statement
All the languages include a statement form for reading data from the user at a terminal, from files, or from a communications line. Such statements also change the values of variables through assignments.

Other assigning operations
Parameter transmission is often defined as assignment of the argument value to the formal parameter. Various forms of implicit are found as well;

Forms of Statement-Level Sequence Control
The three main statement-level sequence controls are:

1. **Composition**
   Statement may be placed in a textual sequence so that they are executed in order whenever the larger program structure containing the sequence is executed.

2. **Alternation**
   Two sequences of statements may form alternatives so that one or the other sequence is executed, but not both.
   Eg., if statements.

3. **Iteration**
   All looping statements are iterative statements. The control repetitively executes a statement at, ‘n' number of times.

Explicit Sequence Control

1. **GOTO**
   The programmer can able to transfer the control explicitly. ‘Goto' the best example for explicit sequence controls.

2. **Break statement.**
   Usually the break causes control to move forward in the program to an explicit point at the end of

a given control structure. Thus, break in C causes control to exit the immediately enclosing while, for, or switch statement.

Structured programming Design

The Goto and Break statement spoils the hierarchy of programming structure.

Disadvantages of GOTO statements

1. Lack of hierarchical program structure.
2. Order of statements in the program text need not correspond to the order of execution.
3. Group of statements may serve multiple purposes.

Structured Sequence Control

1. **Compound Statements**
   Is a sequence of statements that may be treated as a single statement in the construction of larger statements.

   e.g., begin
   Statement 1.
   Statement 2 // IN Pascal

   Statement n
   End;

   And similarly

   {
   // Braces In C and C++ and JAVA
   }

   Within the compound statement, statements are written in the order in which they are to be executed.

2. **Conditional Statements**

   A conditional statement is one that expresses alternation of two or more statements, or optional execution of a single statement;

   // IF statements and IF then Else Statements.

3. **Case statements**

   Iteration Statements
   Simple repetition.
   In COBOL
   Perform body K times

   Here, the body of the statements are executes ‘K' times.
   Repetition while condition holds

Eg., while (x!='y'||x!='Y')
{
Statements;
}

4. **Repetition while incrementing a counter**

   E.g., for (I=1;I<=n;I++)
   The I++ is a counter.

5. **Indefinite repetition**
   The loop will execute infinite number of times.
   E.g.,

   N=2;
   For (I=1;I<=n; n++)
   {
   Statements;
   }

6. **Implementation of loop statements**
   Implementation of loop-control statements using the hardware branch/jump instruction is straightforward. To implement a for loop, the expressions in the loop head defining the final value and increment must be evaluated on initial entry to the loop and saved in special temporary storage areas.

Subprogram control

Subprogram Sequence Control

Now we are going to discuss about the simple mechanisms for sequencing between tow subprograms, how on subprogram invokes another and called subprogram returns to the first. The simple Call and return statements structure is common for all the subprograms
Simple subprograms call return.

1. Subprograms cannot be recursive

A subprogram is directly recursive if it contains a call on itself; it is indirectly recursive if it calls another subprogram that calls the original subprogram or that initiates a further chain of subprogram calls that eventually leads back to a call of the original subprogram.

1. Explicit call statements are required.
2. Subprograms must execute completely at each call.
3. Immediate transfer of control at point of call.
4. Single execution sequence.

Simple Call-Return Subprograms

Implementation
Current-instruction pointer.
Current-environment pointer

Stack-Based Implementation

Recursive Subprograms
Specification
Implementation

The Pascal Forward Declaration

Data Control and referencing environments

ATTRIBUTES OF DATA CONTROL

Names and Referencing Environments
Program Elements That may be named

Associations and Referencing Environments

Referencing environments
1. Local referencing environment

2. Nonlocal referencing environment
3. Global referencing environment
4. Predefined referencing environment

Visibility
Dynamic scope
Referencing operations
Local, nonlocal, and global references.
Aliases for Data Objects
Static and Dynamic Scope
The importance of static scope

Block Structure
Local Data and Local Referencing Environments
Implementation

Deletion
1. Individual variables
2. A subprogram name
3. A formal-parameter
4. recursive subprogram
Advantages and Disadvantages

Parameter transmission/Parameter passing

Actual and Formal Parameter

Establishing the Correspondence
Positional correspondence
Correspondence by explicit name
Method for Transmitting Parameters

Call by name
Call by reference
Call by value
Call by value-result
Call by constant value
Call by result
Transmission Semantics
Elementary data types
Composite data types
Explicit Function Values
Implementation of Parameter Transmission
Parameter Transmission Examples
Simple variables and constants
Data structures
Components of data structures

Aliases and Parameters
Subprograms as Parameters
Statement Labels as Parameters

static and dynamic scope

Static Scope. A non-local variable of a subprogram refers to the variable of that name from the enclosing unit (or from that unit's enclosing unit, etc.).

Dynamic Scope. A non-local variable of a subprogram refers to the variable of that name from the subprogram's caller (or from the caller's caller, etc.).

Advantages of dynamic scope:

1. Easier to implement in interpreters.
   Many dynamically-scoped languages (e.g., APL, older Lisps, SNOBOL) are interpreter-based.
2. Programming convenience:
   - A routine (or set of routines) can be set up to depend on a global variable with a default value.
   - A user of the routine may *temporarily* override the global default by declaring a variable of the same name. (When the user routine terminates, the global variable is restored.)

Disadvantages of dynamic scope:

1. Readability: it is hard to analyze routines with non-local variables that may be redefined by any caller of the routine.
2. Security: A local variable declared by a user may accidentally override a global variable used by a routine that the user calls (either directly or indirectly).

Block structure

In computer programming, a **block** or **code block** is a lexical structure of source code which is grouped together. Blocks consist of one or more declarations and statements. A programming language that permits the creation of blocks, including blocks nested within other blocks, is called

a **block-structured programming language**. Blocks are fundamental to structured programming, where control structures are formed from blocks.

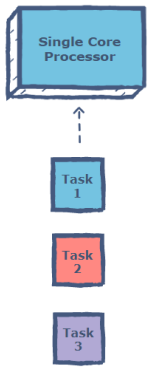
The function of blocks in programming is to enable groups of statements to be treated as if they were one statement, and to narrow the lexical scope of objects such as variables, procedures and functions declared in a block so that they do not conflict with those having the same name used elsewhere. In a block-structured programming language, the objects named in outer blocks are visible inside inner blocks, unless they are masked by an object declared with the same name.

PRINCIPLES OF PROGRAMMING LANGUAGES

UNIT-4

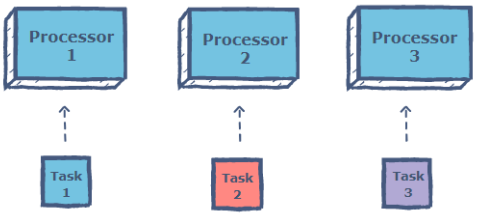
Concurrent Programming

Concurrency generally refers to *events or circumstances* that are *happening or existing* at the **same time**.



In programming terms, **concurrent programming** is a technique in which two or more processes start, run in an *interleaved fashion* through **context switching** and complete in an overlapping time period by managing access to *shared resources* e.g. on a single core of CPU.

This doesn't necessarily mean that multiple processes will be *running at the* **same instant** – even if the results might make it seem like it.

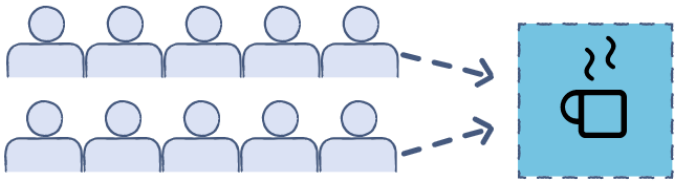


Difference between Concurrent & Parallel programming

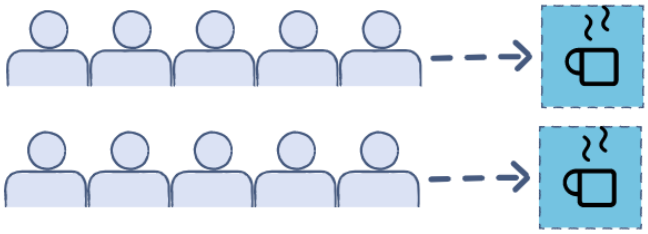
In *parallel programming*, parallel processing is achieved through hardware parallelism e.g. executing two processes on two separate CPU cores simultaneously.

A Real World Example

Concurrent: *Two Queues* & a *Single* Espresso machine.



Parallel: *Two Queues* & *Two* Espresso machines.



Concepts and communication

Concurrency is a property of systems (program, network, computer, etc.) in which several computations are executing simultaneously, and potentially interacting with each other. The computations start, run, and complete in overlapping time periods; they can run at the exact same instant (e.g. **parallelism**), but are not required to.

Concurrency in Programming

Concurrency is implemented in programming logic by explicitly giving computations or processes within a system their own separate execution point or *thread of control*. This allows these computations to avoid waiting for all other computations to complete – as is the case in sequential programming.

Concurrent Computing vs. Parallel Computing

Although concurrent computing is considered a parent category that encompasses parallel programming, they share some distinct differences.

In **parallel computing**, execution occurs at the exact same instant typically with the goal of optimizing modular computations. This forces parallel computing to utilize more than one processing core because each thread of control is running simultaneously and takes up the core's entire clock cycle for the duration of execution – and thus parallel computing is impossible on a single-core machine. This differs from concurrent computing which focuses on the *lifetime* of the computations overlapping and not necessarily their moments of execution. For example, the execution steps of a process can be broken up into **time slices**, and if the entire process doesn't finish during its time slice then it can be paused while another process begins.

Why use Concurrent Programming?

The ultimate benefit of concurrent programming is to utilize the resources of the executing machine to the fullest extent. This typically results in a speed boost in execution

time because the program is no longer subject to normal sequential behavior. Starting in the early 2000's, a common trend in personal computers has been to use multi-core processing units instead of a single omni-powerful CPU core. This helps to optimize the total execution time of a process with multiple threads by spreading the load across all cores. How processes and threads get scheduled is best left to its own discussion and getting down to that level of task scheduling is OS-specific, so I won't dig deeper into how processes and threads are scheduled – but feel free to read up on some of the **synchronization** patterns employed.

The concept of concurrency is employed in modern programming languages typically through a process called **multithreading**. Multithreading allows a program to run on multiple threads while still under the same parent process, thus providing the benefits of parallelization (faster execution, more efficient use of the computer's resources, etc.) but also carrying with it the problems of parallelization too (discussed more below), which is why some languages make use of a mechanism called the **Global Interpreter Lock** (GIL). The GIL is found most commonly in the standard implementations of Python and Ruby (CPython and Ruby MRI, respectively), and prevents more than one thread from executing at a time – even on multi-core processors. This might seem like a massive design flaw, but the GIL exists to prevent any **thread-unsafe** activities – meaning that all code executing on a thread does not manipulate any shared data structures in a manner that risks the safe execution of the other threads. Typically language implementations with a GIL increase the speed of single-threaded programs and make integrations with C libraries easier (because they are often not thread-safe), but all at the price of losing multithreading capabilities.

However, if concurrency through a language implementation with a GIL is a strong concern, there are usually ways around this hindrance. While multithreading is not an option, applications interpreted through a GIL can still be designed to run on different *processes* entirely – each one with their own GIL.

Problems with Concurrent Programming

It has been said that the first rule of concurrent programming is *it's hard*. Because the idea behind concurrency is to execute computations simultaneously, the potential exists for these separate tasks to access and unintentionally distort shared resources among them (e.g. thread-unsafe behavior). When shared resources are accessed, a programmatic **arbiter** is typically involved which handles the allocation of those resources – but this type of activity can ultimately create indeterminacy and lead to issues such as **deadlock** (where multiple computations are waiting on each other to finish, and thus never do), and **starvation** (where resources are constantly denied to a certain task).

This makes coordination when executing concurrent tasks extremely important because even areas where the developer has little control – such as memory allocation on **the stack or the heap** – can become indeterminate.

The Future of Concurrent Programming

Concurrent programming is incredibly powerful even given its obvious flaws, and thus it has been a very active field in theoretical computer science research. In programming, several languages have provided phenomenal support to give developers the tools to take advantage of concurrent behavior; this is perhaps most evident in the API behind **Go**, one of the newest popular languages created by developers at Google. Other languages, such as Python and Ruby, see the negative power of concurrency and thus default to using a GIL.

Countless models have been built to better understand concurrency and describe various theories, such as the Actor Model, CSP Model, and Disruptor Model. However, just as with most programming concepts, there is no silver bullet for concurrent programming. If you build an application that employs concurrency, be sure to plan it carefully and know what's going on at all times – or else you risk the cleanliness of your data.

Deadlocks

In concurrent computing, a **deadlock** is a state in which each member of a group is waiting for another member, including itself, to take action, such as sending a message or more commonly releasing a lock. Deadlock is a common problem in multiprocessing systems, parallel computing, and distributed systems, where software and hardware locks are used to arbitrate shared resources and implement process synchronization.

In an operating system, a deadlock occurs when a process or thread enters a waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process. If a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process, then the system is said to be in a deadlock.

In a communications system, deadlocks occur mainly due to lost or corrupt signals rather than resource contention.

Semaphores

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows −

- **Wait**

  The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(S)
{
```

```
   while (S<=0);

   S--;
}
```

- **Signal**

  The signal operation increments the value of its argument S.

```
signal(S)
{
   S++;
}
```

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −

- **Counting Semaphores**

  These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

- **Binary Semaphores**

  The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages of Semaphores

Some of the advantages of semaphores are as follows −

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows −

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.

- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

Monitors

Monitors are a synchronization construct that were created to overcome the problems caused by semaphores such as timing errors.

Monitors are abstract data types and contain shared data variables and procedures. The shared data variables cannot be directly accessed by a process and procedures are required to allow a single process to access the shared data variables at a time.

This is demonstrated as follows:

```
monitor monitorName
{
   data variables;

   Procedure P1(....)
   {

   }

   Procedure P2(....)
   {

   }

   Procedure Pn(....)
   {

   }

   Initialization Code(....)
   {

   }
}
```

Only one process can be active in a monitor at a time. Other processes that need to access the shared variables in a monitor have to line up in a queue and are only provided access when the previous process release the shared variables.

Threads

In general, as we know that thread is a very thin twisted string usually of the cotton or silk fabric and used for sewing clothes and such. The same term thread is also used in the world of computer programming. Now, how do we relate the thread used for sewing clothes and the thread used for computer programming? The roles performed by the two threads is similar here. In

clothes, thread hold the cloth together and on the other side, in computer programming, thread hold the computer program and allow the program to execute sequential actions or many actions at once.

Thread is the smallest unit of execution in an operating system. It is not in itself a program but runs within a program. In other words, threads are not independent of one other and share code section, data section, etc. with other threads. These threads are also known as lightweight processes.

States of Thread

To understand the functionality of threads in depth, we need to learn about the lifecycle of the threads or the different thread states. Typically, a thread can exist in five distinct states. The different states are shown below −

New Thread

A new thread begins its life cycle in the new state. However, at this stage, it has not yet started and it has not been allocated any resources. We can say that it is just an instance of an object.

Runnable

As the newly born thread is started, the thread becomes runnable i.e. waiting to run. In this state, it has all the resources but still task scheduler have not scheduled it to run.

Running

In this state, the thread makes progress and executes the task, which has been chosen by task scheduler to run. Now, the thread can go to either the dead state or the non-runnable/ waiting state.

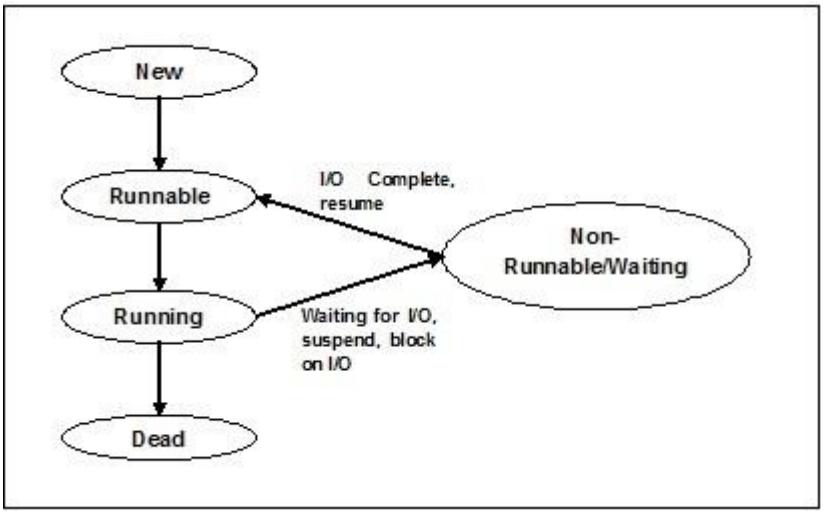
Non-running/waiting

In this state, the thread is paused because it is either waiting for the response of some I/O request or waiting for the completion of the execution of other thread.

Dead

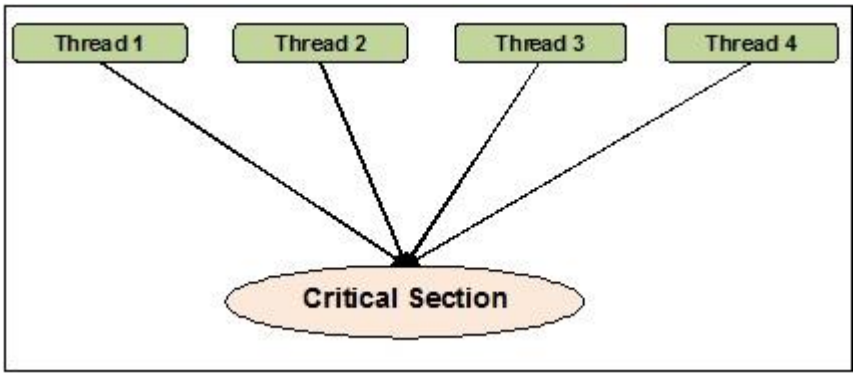
A runnable thread enters the terminated state when it completes its task or otherwise terminates.

The following diagram shows the complete life cycle of a thread −



Synchronization

Thread synchronization may be defined as a method with the help of which we can be assured that two or more concurrent threads are not simultaneously accessing the program segment known as critical section. On the other hand, as we know that critical section is the part of the program where the shared resource is accessed. Hence we can say that synchronization is the process of making sure that two or more threads do not interface with each other by accessing the resources at the same time. The diagram below shows that four threads trying to access the critical section of a program at the same time.



To make it clearer, suppose two or more threads trying to add the object in the list at the same time. This act cannot lead to a successful end because either it will drop one or all the objects or it will completely corrupt the state of the list. Here the role of the synchronization is that only one thread at a time can access the list.

Issues in thread synchronization

We might encounter issues while implementing concurrent programming or applying synchronizing primitives. In this section, we will discuss two major issues. The issues are −

- Deadlock

- Race condition

Race condition

This is one of the major issues in concurrent programming. Concurrent access to shared resources can lead to race condition. A race condition may be defined as the occurring of a condition when two or more threads can access shared data and then try to change its value at the same time. Due to this, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.

Deadlock

Deadlock is a troublesome issue one can face while designing the concurrent systems. We can illustrate this issue with the help of the dining philosopher problem as follows −

Edsger Dijkstra originally introduced the dining philosopher problem, one of the famous illustrations of one of the biggest problem of concurrent system called deadlock.

In this problem, there are five famous philosophers sitting at a round table eating some food from their bowls. There are five forks that can be used by the five philosophers to eat their food. However, the philosophers decide to use two forks at the same time to eat their food.

Now, there are two main conditions for the philosophers. First, each of the philosophers can be either in eating or in thinking state and second, they must first obtain both the forks, i.e., left and right. The issue arises when each of the five philosophers manages to pick the left fork at the same time. Now they all are waiting for the right fork to be free but they will never relinquish their fork until they have eaten their food and the right fork would never be available. Hence, there would be a deadlock state at the dinner table.

Logic Programming

Introduction

Logic programming is a computer programming paradigm
where program statements express facts and rules about problems within a system of formal logic. Rules are written as logical clauses with a head and a body; for instance, "H is true if B1, B2, and B3 are true." Facts are expressed similar to rules, but without a body; for instance, "H is true."

Some logic programming languages, such as Datalog and ASP (Answer Set Programming), are purely declarative. They allow for statements about what the program should accomplish, with no explicit step-by-step instructions on how to do so. Others, such as Prolog, are a combination of declarative and imperative. They may also include procedural statements, such as "To solve H, solve B1, B2, and B3."

Rules

- Everything not defined is undefined. This may be a tautology, but it is a useful one. Many of the rules below are just special cases of this rule.
- All parameters must be valid. The contract for a function applies only when the caller adheres to the conditions, and one of the conditions is that the parameters are actually what they claim to be. This is a special case of the "everything not defined is undefined" rule.
    - Pointers are not `NULL` unless explicitly permitted otherwise.
    - Pointers actually point to what they purport to point to. If a function accepts a pointer to a `CRITICAL_SECTION`, then you really have to pass pointer to a valid `CRITICAL_SECTION`.
    - Pointers are properly aligned. Pointer alignment is a fundamental architectural requirement, yet something many people overlook having been pampered by a processor architecture that is very forgiving of alignment errors.
    - The caller has the right to use the memory being pointed to. This means no pointers to memory that has been freed or memory that the caller does not have control over.
    - All buffers are valid to the size declared or implied. If you pass a pointer to a buffer and say that it is ten bytes in length, then the buffer really needs to be ten bytes in length.
    - Handles refer to valid objects that have not been destroyed. If a function wants a window handle, then you really have to pass a valid window handle.
- All parameters are stable.
    - You cannot change a parameter while the function call is in progress.

    - If you pass a pointer, the pointed-to memory will not be modified by another thread for the duration of the call.
    - You can't free the pointed-to memory either.
- The correct number of parameters is passed with the correct calling convention. This is another special case of the "everything not defined is undefined" rule.
    - Thank goodness modern compilers refuse to pass the wrong number of parameters, though you'd be surprised how many people manage to sneak the wrong number of parameters past the compiler anyway, usually by devious casting.
    - When invoking a method on an object, the `this` parameter is the object. Again, this is something modern compilers handle automatically, though people using COM from C (and yes they exist) have to pass the `this` parameter manually, and occasionally they mess up.
- Function parameter lifetime.
    - The called function can use the parameters during the execution of the function.
    - The called function cannot use the parameters once the function has returned. Of course, if the caller and the callee have agreed on a means of extending the lifetime, then those rules apply.
        - The lifetime of a parameter that is a pointer to a COM object can be extended by the use of the `IUnknown::AddRef` method.
        - Many functions are passed parameters with the express intent that they be used after the function returns. It is then the caller's responsibility to ensure that the lifetime of the parameter is at least as long as the function needs it. For example, if you register a callback function, then the callback function needs to be valid until you deregister the callback function.
- Input buffers.
    - A function is permitted to read from the full extent of the buffer provided by the caller, even if not all of the buffer is required to determine the result.
- Output buffers.
    - An output buffer cannot overlap an input buffer or another output buffer.
    - A function is permitted to write to the full extent of the buffer provided by the caller, even if not all of the buffer is required to hold the result.
    - If a function needs only part of a buffer to hold the result of a function call, the contents of the unused portion of the buffer are undefined.
    - If a function fails and the documentation does not specify the buffer contents on failure, then the contents of the output buffer are undefined. This is a special case of the "everything not defined is undefined" rule.
    - Note that COM imposes its own rules on output buffers. COM requires that all output buffers be in a marshallable state even on failure. For objects that require nontrivial marshalling (interface pointers and BSTRs being the most common examples), this means that the output pointer must be `NULL` on failure.

Structured Data and Scope of the variables

Structured data is the data which conforms to a data model, has a well define structure, follows a consistent order and can be easily accessed and used by a person or a computer program.
Structured data is usually stored in well-defined schemas such as Databases. It is generally tabular with column and rows that clearly define its attributes.

SQL (Structured Query language) is often used to manage structured data stored in databases.

Characteristics of Structured Data:

- Data conforms to a data model and has easily identifiable structure
- Data is stored in the form of rows and columns
  Example : Database
- Data is well organised so, Definition, Format and Meaning of data is explicitly known
- Data resides in fixed fields within a record or file
- Similar entities are grouped together to form relations or classes
- Entities in the same group have same attributes
- Easy to access and query, So data can be easily used by other programs
- Data elements are addressable, so efficient to analyse and process

The **scope of a variable** is the region of your program in which it is defined. Traditionally, JavaScript defines only two scopes-global and local.

- **Global Scope** − A variable with global scope can be accessed from within any part of the JavaScript code.
- **Local Scope** − A variable with a local scope can be accessed from within a function where it is declared.

Example : Global vs. Local Variable

The following example declares two variables by the name **num** - one outside the function (global scope) and the other within the function (local scope).

```
var num = 10
function test() {
  var num = 100
  console.log("value of num in test() "+num)
}
console.log("value of num outside test() "+num)
test()
```

The variable when referred to within the function displays the value of the locally scoped variable. However, the variable **num** when accessed outside the function returns the globally scoped instance.

The following output is displayed on successful execution.

```
value of num outside test() 10
value of num in test() 100
```

Operators and Functions

In programming, a **function** is a named section of a program that performs a specific task. In this sense, a function is a type of procedure or routine. Some programming languages make a distinction between a *function*, which returns a value, and a *procedure*, which performs some operation but does not return a value.
Most programming languages come with a prewritten set of functions that are kept in a library. You can also write your own functions to perform specialized tasks.

Operators are symbols that tell the compiler to perform specific mathematical or logical manipulations. In this tutorial , we will try to cover the most commonly used operators in programming.
First, let's categorize them:
1. Arithmetic
2. Relational
3. Bitwise
4. Logical
5. Assignment
6. Increment
7. Miscellaneous

Arithmetic Operators:

Symbol	Operation	Usage	Explanation
+	addition	x+y	Adds values on either side of the operator
-	subtraction	x-y	Subtracts the right hand operand from the left hand operand
*	multiplication	x*y	Multiplies values on either side of the operator
/	division	x/y	Divides the left hand operand by the right hand operand
%	modulus	x%y	Divides the left hand operand by the right hand operand and returns remainder

Relational Operators: These operators are used for comparison. They return either **true** or **false** based on the comparison result. The operator '==' should not be confused with '='. The relational operators are as follows:

Symbol	Operation	Usage	Explanation
==	equal	x == y	Checks if the values of two operands are equal or not, if yes then condition becomes true.

Symbol	Operation	Usage	Explanation
!=	not equal	x != y	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	greater than	x > y	Checks if the value of the left operand is greater than the value of the right operand, if yes then condition becomes true
<	less than	x < y	Checks if the value of the left operand is less than the value of the right operand, if yes then condition becomes true.
>=	greater than or equal	x >= y	Checks if the value of the left operand is greater than or equal to the value of the right operand, if yes then condition becomes true.
<=	less than or equal	x <= y	Checks if the value of the left operand is less than or equal to the value of the right operand, if yes then condition becomes true.

Bitwise Operators: These operators are very useful and we have some tricks based on these operators. These operators convert the given integers into binary and then perform the required operation, and give back the result in decimal representation.

Symbol	Operation	Usage	Explanation
&	bitwise AND	x & y	Sets the bit to the result if it is set in both operands.
|	bitwise OR	x | y	Sets the bit to the result if it is set in either operand.
^	bitwise XOR	x ^ y	Sets the bit if it is set in one operand but not both
~	bitwise NOT	~x	Unary operator and has the effect of 'flipping' bits,i.e, flips 1 to 0 and 0 to 1.
<<	left shift	x << y	The left operand's value is moved left by the number of bits specified by the right operand. It is equivalent to multiplying x by 2^y
>>	right shift	x >> y	The left operand's value is moved right by the number of bits specified by the right operand.It is equivalent to dividing x by 2^y

Examples:
Assume x=42, y=27
x=00101010
y=00011011

x&y = 0000 1010= 10 (in decimal)
x|y = 0011 1011= 59
x ^ y = 0011 0001= 49
~x = 1101 0101
x<<2 = 1010 1000= 168. Notice, the bits are shifted 2 units to the left and the new bits are filled by 0s.
x>>2 = 0000 1010=10$$. Notice, the bits are shifted 2 units to the right and the new bits are filled by 0s.
For more information about how these operators work, see : Bit Manipulation

Logical Operators: These operators take boolean values as input and return boolean values as output.
Note: In C,C++ any non-zero number is treated as true and 0 as false but this doesn't hold for Java.

Symbol	Operation	Usage	Explanation
&&	**logical AND**	**x && y**	**Returns true if both x and y are true else returns false.**
||	**logical OR**	**x || y**	**Returns false if neither x nor y is true else returns true**
!	**logical NOT**	**! x**	**Unary operator. Returns true if x is false else returns false.**

Assignment Operators:

Symbol	Operation	Usage	Equivalence	Explanation
=	**assignment**	**x = y**		**Assigns value from the right side operand(s) to the left side operand.**
+=	**add and assignment**	**x += y**	x = x+y	**Adds the right side operand to the left side operand and assigns the result to the left side operand.**
-=	**subtract and assignment**	**x -= y**	x= x-y	**Subtracts the right side operand from the left side operand and assigns the**

Symbol	Operation	Usage	Equivalence	Explanation
				result to the left side operand.
***=**	**multiply and assignment**	**x *= y**	x= x*y	**Multiplies the right side operand with the left side operand and assigns the result to the left side operand.**
/=	**divide and assignment**	**x /= y**	x= x/y	**Divides the left side operand with the right side operand and assigns the result to the left side operand.**
%=	**modulus and assignment**	**x%=y**	x= x%y	**Takes modulus using the two operands and assigns the result to the left side operand.**
<<=	**left shift and assignment**	**x<<=y**	x= x<< y	**Shifts the value of x by y bits towards the left and stores the result back in x.**
>>=	**right shift and assignment**	**x>>=y**	x= x>>y	**Shifts the value of x by y bits towards the right and stores the result back in x.**
&=	**bitwise AND and assignment**	**x&=y**	x= x&y	**Does x&y and stores result back in x.**
|=	**bitwise OR and assignment**	**x|=y**	x= x|y	**Does x|y and stores result back in x**
^=	**bitwise XOR and assignment**	**x^=y**	x= x^y	**Does x^y and stores result back in x.**

Increment/Decrement Operators: These are **unary** operators. Unary operators are the operators which require only one operand.

Symbol	Operation	Usage	Explanation
++	**Postincrement**	**x++**	**Increment x by 1 after using its value**

Symbol	Operation	Usage	Explanation
--	Postdecrement	x--	Decrement x by 1 after using its value
++	Preincrement	++x	Increment x by 1 before using its value
--	Predecrement	--x	Decrement x by 1 before using its value

Examples:
Let x=10
then, after **y=x++**; y=10 and x=11, this is because x is assigned to y before its increment.
but if we had written **y=++x**; y=11 and x=11, because x is assigned to y after its increment.
Same holds for decrement operators.

Miscellaneous Operators:
Conditional Operator: It is similar to **if-else**:

x = (condition) ? a : b
If condition is true,then a is assigned to x else b is assigned to x. It is a ternary operator because it uses the condition, a and b i.e. three operands (the condition is also treated as a boolean operand).

Operator Precedence and Associativity:
Precedence Rules: The precedence rules specify which operator is evaluated first when two operators with different precedence are adjacent in an expression.
For example: x=a+++b
This expression can be seen as postfix increment on a and addition with b or prefix increment on b and addtion to a. Such issues are resolved by using precedence rules.
Associativity Rules: The associativity rules specify which operator is evaluated first when two operators with the same precedence are adjacent in an expression.
For example: a∗b/c
Operator Precedence: The following table describes the precedence order of the operators mentioned above. Here, the operators with the highest precedence appear at the top and those with the lowest at the bottom. In any given expression, the operators with higher precedence will be evaluated first.
LR= Left to Right
RL=Right to Left

Category	Associativity	Operator
Postfix	LR	++ --
Unary	RL	+ - ! ~ ++ --
Multiplicative	LR	* / %
Additive	LR	+ -

Category	Associativity	Operator
Shift	**LR**	**<< >>**
Relational	**LR**	**< <= > >=**
Equality	**LR**	**== !=**
Bitwise AND	**LR**	**&**
Bitwise XOR	**LR**	**^**
Bitwise OR	**LR**	**|**
Logical AND	**LR**	**&&**
Logical OR	**LR**	**||**
Conditional	**RL**	**?:**
Assignment	**RL**	**= += -= *= /= %= >>= <<= &= ^= |=**

Recursion and recursive rules

Recursion:

- Functions can call other functions.
- Functions can even call themselves. This is called *recursion*.
- If a function is called recursively, it is required that for some condition, which will eventually be true, that the function not call itself again, but return to its calling function.

Rules of recursion:

1. *Base cases:* You must always have some base or trivial case, which can be solved without recursion.
2. *Making progress:* For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward the base case.
3. *Design rule:* Assume that all the recursive calls work. Use proof by induction.
4. *Compound Interest Rule:* Never duplicate work by solving the same instance of a problem in separate recursive calls. If possible, use dynamic programming.

Lists

In computer science, a **list** or **sequence** is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a tuple or finite sequence; the (potentially) infinite analog of a list is a stream.[1]:§3.5 Lists are a basic example of containers, as they contain other values. If the same value occurs multiple times, each occurrence is considered a distinct item.

The name **list** is also used for several concrete data structures that can be used to implement abstract lists, especially linked lists and arrays. In some contexts, such as in Lisp programming, the term list may refer specifically to a linked list rather than an array. In class-based programming, lists are usually provided as instances of subclasses of a generic "list" class, and traversed via separate iterators.

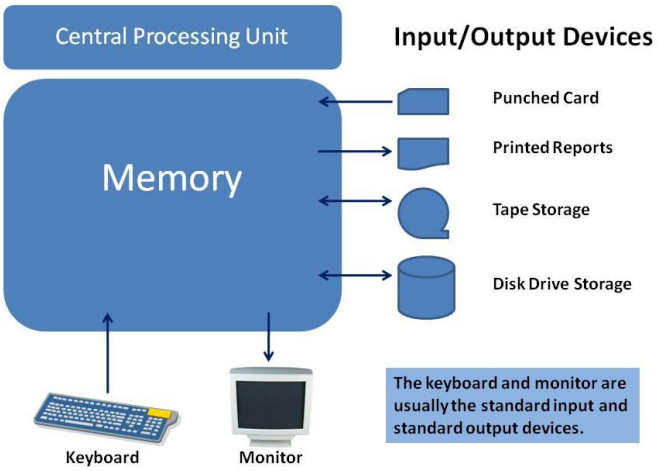
Many programming languages provide support for **list data types**, and have special syntax and semantics for lists and list operations. A list can often be constructed by writing the items in sequence, separated by commas, semicolons, and/or spaces, within a pair of delimiters such as parentheses '()', brackets '[]', braces '{}', or angle brackets '<>'. Some languages may allow list types to be indexed or sliced like array types, in which case the data type is more accurately described as an array.

In type theory and functional programming, abstract lists are usually defined inductively by two operations: *nil* that yields the empty list, and *cons*, which adds an item at the beginning of a list.

Input and Output

Input and output, or I/O is the communication between an information processing system, such as a computer, and the outside world, possibly a human or another information processing system. Inputs are the signals or data received by the system and outputs are the signals or data sent from it.

Every task we have the computer do happens inside the central processing unit (CPU) and the associated memory. Once our program is loaded into memory and the operating system directs the CPU to start executing our programming statements the computer looks like this:



CPU – Memory – Input/Output Devices

Our program now loaded into memory has basically two areas:

- Machine instructions – our instructions for what we want done
- Data storage – our variables that we using in our program

Often our program contains instructions to interact with the input/output devices. We need to move data into (read) and/or out of (write) the memory data area. A **device** is a piece of equipment that is electronically connected to the memory so that data can be transferred between the memory and the device. Historically this was done with punched cards and printouts. Tape drives were used for electronic storage. With time we migrated to using disk drives for storage with keyboards and monitors (with monitor output called soft copy) replacing punch cards and printouts (called hard copy).

Most computer operating systems and by extension programming languages have identified the keyboard as the **standard input device** and the monitor as the **standard output device**. Often the keyboard and monitor are treated as the default device when no other specific device is indicated.

Program Control

Program Control Instructions are the machine code that are used by machine or in assembly language by user to command the processor act accordingly. These instructions are of various types. These are used in assembly language by user also. But in level language, user code is translated into machine code and thus instructions are passed to instruct the processor do the task.

Types of Program Control Instructions:
There are different types of Program Control Instructions:

1. Compare Instruction:
Compare instruction is specifically provided, which is similar t a subtract instruction except the result is not stored anywhere, but flags are set according to the result.
Example:
CMP R1, R2 ;

2. Unconditional Branch Instruction:
It causes an unconditional change of execution sequence to a new location.
Example:
JUMP L2
Mov R3, R1 goto L2

3. Conditional Branch Instruction:
A conditional branch instruction is used to examine the values stored in the condition code register to determine whether the specific condition exists and to branch if it does.
Example:
Assembly Code : BE R1, R2, L1
Compiler allocates R1 for x and R2 for y
High Level Code: if (x==y) goto L1;

4. Subroutines:
A subroutine is a program fragment that lives in user space, performs a well-defined task. It is invoked by another user program and returns control to the calling program when finished.
Example:
CALL and RET

5. Halting Instructions:

- **NOP Instruction –** NOP is no operation. It cause no change in the processor state other than an advancement of the program counter. It can be used to synchronize timing.
- **HALT –** It brings the processor to an orderly halt, remaining in an idle state until restarted by interrupt, trace, reset or external action.

6. Interrupt Instructions:

Interrupt is a mechanism by which an I/O or an instruction can suspend the normal execution of processor and get itself serviced.

- **RESET –** It reset the processor. This may include any or all setting registers to an initial value or setting program counter to standard starting location.
- **TRAP –** It is non-maskable edge and level triggered interrupt. TRAP has the highest priority and vectored interrupt.
- **INTR –** It is level triggered and maskable interrupt. It has the lowest priority. It can be disabled by resetting the processor.

Logic Program design

A logic model is a tool used to design and build the evaluation of programs. It uses a simple visual to represent the relationship between the challenge or problem, the resources available, the activities and the goals of the program. Logic models demonstrate the causal relationship between what you put into a relationship and what you hope to get out of it.