**CSC 430 Lecture Note**

**By**

**Professor Oyelami M. Olufemi**

**CIT/CSC 430: ORGANIZATION OF PROGRAMMING LANGUAGES 3 Credits**
Language definition structure. Data types and structures, Review of basic data types, including lists and trees, control structure and dataflow, Run-time consideration, interpretative languages, lexical analysis and parsing.
45h(T); C. Prerequisite: CIT/CSC 304.

## Objectives of the Course

The objectives of this course are the following:

- To provide the learners with the parameters and tools needed for evaluating existing and future programming languages and programming language constructs;
- To provide an understanding of design decisions made in previous programming languages;
- To demonstrate the abstract and the theoretical notions of programming languages to help expand the student's understanding of programming concepts and programming language constructs; and
- The student should have gained a larger understanding of computer programming and computer science at the end of the course.

## Diversity of Languages

Over a thousand different programming languages have been designed by research groups, international committees and computer companies. Most of these programming languages have never been used outside the group which designed them while others once popular have been replaced by newer languages.

## Reasons for Studying Organization of Programming Languages

1. **Increased Capacity to Express Ideas**
    - Those with a limited group of natural languages are limited in the complexity of their thought, particularly in depth of abstraction.
    - It is difficult for people to conceptualize structures they cannot describe verbally or in writing.
    - Programmers are similarly constrained. The language in which they develop software places limits on the kinds of control structures, data structures and abstractions they can use.
    - Awareness of a wider variety of programming language features can reduce such limitations.
2. **Improved Background for Choosing Appropriate Languages**
    - Many professional programmers have had little formal education in Computer Science.
    - Others are trained in the distant past.

- They tend to use languages they are familiar with for a new project even if not suitable.
- If they were familiar with other languages and their features, they would be in a better position to make informed language choices.

3. **Increased Ability to Learn new Languages**
    - Computer programming is a young discipline with design methodologies, software development tools and programming languages still in continuous evolution.
    - Learning a new programming language can be lengthy and difficult.
    - Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes easier to see how concepts are incorporated into the design of the language being learned.
4. **Better Understanding of Implementation Significance**
    - An understanding of the implementation issues of the concepts of programming languages leads to the ability to use a language more intelligently as it was designed.
    - For example, a programmer who knows nothing about how recursion is implemented often does not know that a recursive algorithm can be far slower than an equivalent iterative algorithm.
5. **Increased Ability to Design new Languages**
6. **Overall Advancement of Computing**
    - Many believed ALGOL 60 would have displaced FORTRAN in the sixties.

- This is because of its elegance and much better control statements among others.
- But programmers and software development managers did not clearly understand its conceptual design.
- If those who choose languages are better informed, perhaps better languages would more quickly squeeze out poorer ones.

## Language Categories

Programming languages are often categorized into the following:

-

## Programming Domains

The following are areas of computer applications and their associated languages:

1. **Scientific Applications**
    - Have simple data structures but large numbers of floating-point arithmetic computations.
    - Common data structures are:
        i. Arrays and matrices
    - Common control structures are:
        i. Loops and selections
    - Examples: FORTRAN and ALGOL
2. **Business Applications**
    - Language for business application software

- Example: COBOL

3. **Artificial Intelligence**
    - Characterized by the use of symbolic rather than numeric computations
    - Examples: Lisp, Prolog
4. **Systems Programming**
    - Systems software consists of operating system and programming support tools.
        i. The languages must have fast execution and features that allow software interfaces to external devices
    - Examples: PL/1, PL/S, BLISS, Extended ALGOL, BCPL, Coral 66, Jovial, Java and XPL.
    - The most widely used are: C and C++
5. **Special-Purpose languages**
    - They are domain-specific languages
    - Examples:
        i. String Manipulation: COMIT, SNOBOL, SNOBOL 4
        ii. List Processing Languages: IPL-V, Lisp
        iii. Simulation Languages
            - ❖ Used for simulation of system e.g simulation of traffic system to predict how a proposed model will perform as traffic densities increase and show where bottlenecks could occur.
            - ❖ Examples: GPSS (General-Purpose Simulation System) and Simula 67

iv. Scripting Languages

- ❖ Intended for communicating with/ plugging together useful components in other programming languages like Java, C++ and HTML.
- ❖ They are interpreted instead of being compiled
- ❖ Examples: awk, Perl, Python, TCl, JavaScript, Php and ASP.

## Language Evaluation Criteria

The criteria are:

A. **Readability:** The ease with which programs can be read and understood.
B. **Writability:** A measure of how easily a language can be used to create programs for a chosen problem.
C. **Reliability:** A program is reliable if it performs to its specifications under all conditions.
**D. Cost:** Financial and computing resources.

## Characteristics Contributing to a Programming Language Readability

**i. Overall Simplicity**

- **Size of Basic Types**
  - A language that has large number of basic components is more difficult to learn than one with a small number of basic types

- **Feature Multiplicity:** Having more than one way to accomplish a particular task might be complicating for some programmers e.g

  **count = count +1, count + = 1, count ++ and ++count** in C++ and C all have the same meaning when used as standalone expressions although the last two have slightly different meanings.

- **Overloading**
  - Overloading is a useful feature, but can reduce program readability.

**ii. Orthogonality**

- Orthogonality in programming languages means that features work independently without interfering with each other.
- When a language is orthogonal, you can combine different features in a consistent way, making the language easier to learn and use.
- This means that a relatively small set of primitive constructs can be combined in a relatively small number to build control and data structures.
- It also means that every possible combination of primitives is legal and meaningful.
- The idea is that an orthogonal language will be simple.
- Extreme use of orthogonality can lead to unnecessary complexity e.g when conditional statements are used at the left side of assignment statements as in ALGOL 68.

**Example: Orthogonality in Variables and Data Types**

Imagine a language where you can declare variables independently of their data type. If you can use integers, floats, or strings in the same way without worrying about special rules, that's an example of orthogonality.

**Orthogonal example (Python)**

```
x = 10    # Integer
y = 10.5  # Float
z = "Hello"  # String
```

**Non-orthogonal example (C)**

```
int x = 10;
float y = 10.5;
char* z = "Hello";
```

**iii. Control Statements**

- The availability of control structures aid the readability of a program.
- Indiscriminate use of **'goto'** reduces program readability.

**iv. Data Types and Structures**

- Availability of facilities for defining data types and data structures aids readability.
- A language which uses numeric type for Boolean type is less readable.

**v. Syntax Considerations**

- **Identifier Forms:** Restricting identifiers to very short lengths reduces readability
- **Special Words:** The use of special words as variable names reduces readability.

**vi. Forms and Meaning**

- The appearance of a statement should at least partially indicate its purpose to aid readability.

## Factors Affecting Writability

i. Simplicity and orthogonality
ii. Support for abstraction
iii. Expressivity
    - Means a language has a relatively convenient way of specifying computation instead of being cumbersome. E.g **count++** instead of **count = count + 1**

## Factors Affecting Reliability

**i. Type Checking**

- Compile-time type errors checking is better and is less expensive to make the required repairs.

**ii. Exception Handling**

- A language that offers facility to intercept run-time errors and take corrective measures and then continue greatly aids reliability.

**iii. Aliasing**

- Aliasing means having two or more distinct referring methods or names for the same memory cell.
- Aliasing is a dangerous feature in a programming language.

**iv. Readability and Writability**

- The easier a program is to write, the more likely is to be correct, which eventually affects reliability.

- Readability affects reliability in both the writing and maintenance phases of the life cycle.

## Factors Affecting Cost

The ultimate cost of a programming language is a function of the following characteristics:

i. Cost of training programmers
ii. Cost of writing programs in the language
iii. Cost of compiling the program in the language
iv. Cost of language implementation system. Java provides a free compiler/interpreter and was rapidly accepted.
v. Cost of maintaining programs
vi. **Language Design:** A language that requires many run-time type checks will prohibit fast code execution.

## Influences on Language Design

The two major influences on language design are:

1. Computer architecture
    - The imperative languages were designed around von Neuman architecture which was prevalent that time.
    - Data and instructions are stored in the same memory and have to be piped to the CPU for execution.
    - Results of operations have to be piped back to the memory represented by the left hand side of assignment operation.

- Variables of imperative languages model the memory cells.
- Assignments statements are based on piping operations.

2. Programming design methodologies
    - The process-oriented and data-oriented program designs all brought about languages to support them.

## Language Design Tradeoff

- The task of choosing constructs and features when designing a programming language involves compromises and tradeoffs because some of the criteria conflict e.g reliability and cost of execution.

### Defining the Syntax and Semantics of Languages

- The task of providing a concise, yet understandable description of a programming language is difficult but essential to the language's success.
- Language description must take care of the diversity of people who must understand it.
- Most new programming languages are subjected to a period of scrutiny by potential users before their design are completed.
- Language study is divided into
    - Syntax and
    - Semantics
    - **Syntax:** The form of a language's expressions, statements and program units.
    - **Semantics:** The meaning of the expressions, statements and program

units

**Example:** while (<expr>)

{

<statements>

}

- The above is the syntax of a **while** loop in C++.
- The meaning is that the statement should be executed as long as the expression is true.

## The General Problem of Describing Syntax

- Languages, whether natural (like English) or artificial (like Java) are sets of strings of characters from some alphabet.
- The strings of a language are called s**tatement or sentences.**
- The syntax rules of a language specify which strings of characters from the language's alphabet are in the language.
- Formal descriptions of the syntax of programming languages, for simplicity's sake, however, often do not include the descriptions of the lowest-level syntactic units called **lexemes.**
- This is normally given by a lexical specification different from syntactic description.
- The lexeme of a programming language include its
  - Identifiers
  - Literals
  - Operators
  - Special words

- The semantics should follow directly from syntax in a well-designed language. That is, the form of a statement should suggest strongly the meaning of the statement.
- **Token:** The token of a language is the category of its lexemes.

**Example:** Consider the C++ statement: ***index = 2 * count + 17***

| **Lexemes** | **Tokens** |
|---|---|
| index | Identifier |
| = | Equal_sign |
| 2 | Int_literal |
| * | Mult_op |
| count | Identifier |
| + | Plus-op |
| 17 | Int_literal |
| ; | Semicolon |

- Generally, a language can be defined formally in two ways;
  - Recognition
  - Generation

**Recognition**

- A recognition device that is capable of reading strings of characters from the alphabet of the language is constructed.
- The device will indicate whether an input string belongs to the language or not.
- Because most useful languages are, for all practical purposes, infinite, this seems to be a lengthy and ineffective process.
- Syntax analyzer or parser of a compiler is a recognizer.

**Generation**

- A generator is a device that can be used to generate the sentences of a language.
- It can be thought of as a button that, when pressed, produces a sentence of the language.
- Because the particular sentence produced by it when pressed is unpredictable, it seems to be a device with limited usefulness as a language descriptor.
- Some generators can, however, read and understand sentences of a language.

## Methods of Describing the Syntax of a Language

- The most common method of describing syntax is called **context-free grammar or Backus-Naur Form/Backus-Normal Form (BNF).**
- BNF is a **metalanguage**.
- A **metalanguage** is a language used to describe other languages.
- BNF is a generative device for defining languages.
- The sentences of the language are generated through a sequence of application of rules, beginning with a special **nonterminal** of the grammar called the **start symbol.**

**Example**

<digit>::=0|1|2|3|4|5|6|7|8|9

<letter>::=a|b|c|d|… x|y|z

<identifier>::=<letter>|<identifier><digit>|<identifier><letter>

- A **nonterminal** is written within angle brackets **<** and **>.**
- The symbol **::=** means **'is defined as'**
- The symbol | means **'or'**
- **'ch1'** is an identifier that can be derived from <identifier> as follows:

*<identifier>*

*<identifier><digit>*

*<identifier><letter><digit>*

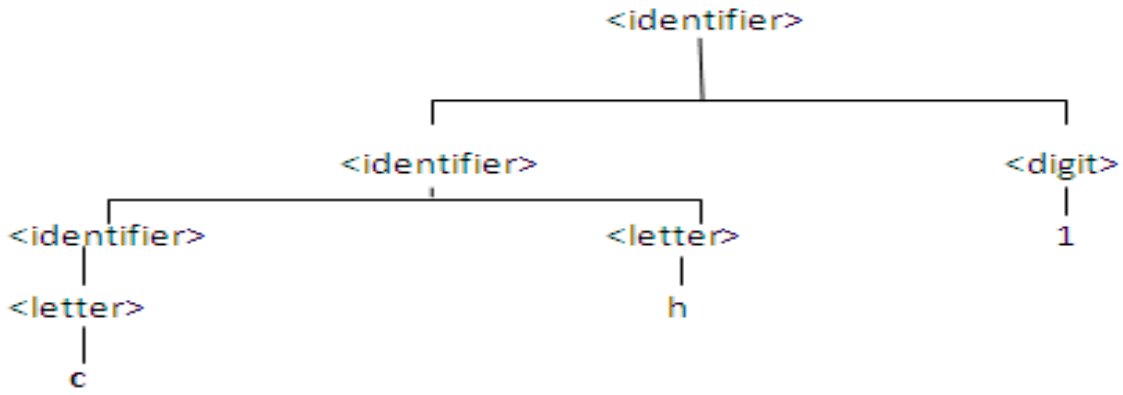
*<letter><letter><digit>*

*c<letter><digit>*

*ch<digit>*

*ch1*

- At each stage, the leftmost **nonterminal** has been replaced by the right part of its defining production rules.
- The sequence of **terminals** and **nonterminals** produced at each stage is a derivation is called a **sentential form.**
- The final sentential form with no **nonterminals** is called a **sentence.**
- The sentence of a derivation is best shown by a **derivation tree.**
- The tree for **'ch1'** is shown below:




- A complete programming language is defined by starting with a nonterminal such as **<program>**, called the **start symbol**.
- All possible programs can be defined from the start symbol by replacing the left parts of production rules by one of their corresponding right parts.
- Trees can be created in a **top-down** approach or **bottom-up** approach.
- In top-down approach, the required sentence is derived from start symbol as in the case of **'ch1'**
- In bottom-up approach, the required sentence is the starting point, which is reduced to the start symbol by replacing right parts by their corresponding left parts.
- Both approaches are used in **syntax analysis** phase of compilers.

**Recursion in Grammar**

- A grammar defining a programming language can be **left-recursive** or **right-recursive**.

**Example**

<identifier>::=<identifier><letter>

- The above is left-recursive because the nonterminal being defined, that is, <identifier>, is the leftmost symbol in the right part.

## Context-Free BNF

- A BNF grammar is context-free if the left part always consists of a single nonterminal.
- This means that the left part can always be replaced by one of its alternative right parts, irrespective of the context in which the left part appears.

## Ambiguity

- A grammar that generates a sentence for which there are two or more distinct derivation trees is said to be ambiguous.

**Example of Ambiguity**

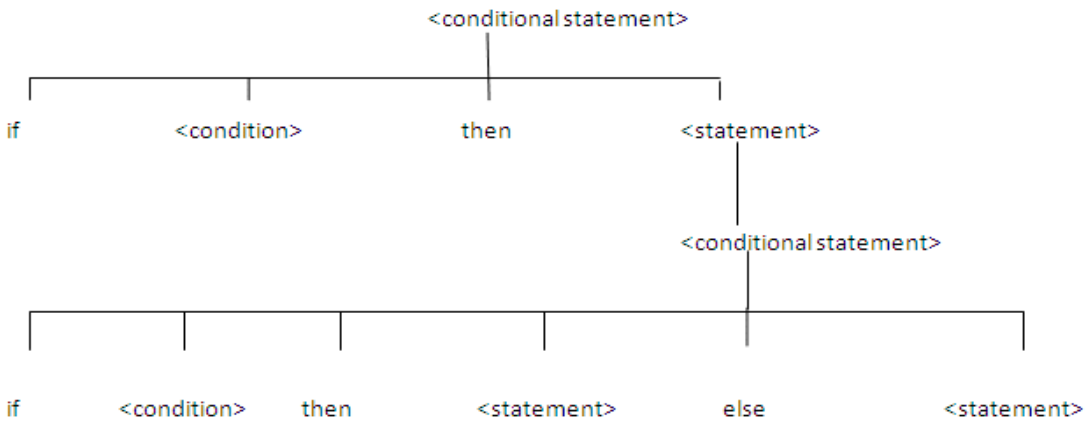
- Consider the definition:

<statement>::=<conditional statement>|begin<statement>end

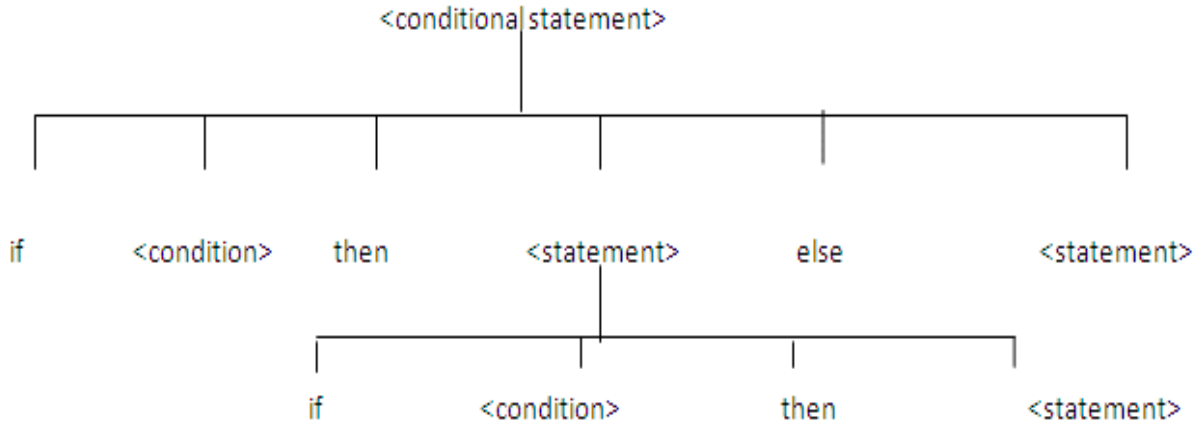
<conditional statement>::=if <condition> then <statement>|if <condition> then <statement>else <statement>

- Two different derivation trees exist for the conditional statement:
  ***if<condition> then if <condition> then <statement> else <statement>***
- The problem is, knowing to which **then** the **else** belongs.
- The derivation trees are shown in **a** and **b** below:

(a)




(b)




- **Note:** A grammar is always ambiguous if it is both left- and right-recursive with respect to the same nonterminal.

## Extended BNF (EBNF)

- Several variants of BNF notations have been used in language definitions.

An example is EBNF

- It sometimes uses **'->'** in place of **'::='**
- It does not use angle brackets
- It encloses terminals within quotes

**Example**

<exp>::=<exp> + <term> + <term>  will be written as  exp -> exp '+' term

## Describing the Semantics of a Programming Language

- Unlike the progress made in the formal definition of the syntax of a programming language, semantics definition is problematic.
- BNF was used for the definition of Algol 60, but the inability of Algol 60 committee to come up with a solution to the semantic problem resulted in the use of English language for the semantic explanation.
- Unfortunately, the use of English prose is ambiguous because two readers can interpret a sentence differently.

## Formal Approaches to Describing Semantics

1. Operational Semantics
2. Axiomatic Semantics
3. Denotational Semantics

**Operational Semantics**

- The idea is to describe the meaning of a program by executing its statements on machine, whether real or simulated.
- The changes that occur in the machine's state when it executes a given statement define the meaning of that statement.

**Example 1**

- Let the state of a machine be the values of all its registers and memory locations including condition codes and status registers.
- If one simply records the state of the computer, executes the instruction for which meaning is sought, and then examines the machine's new state, the semantics of that instruction are clear.

**Example 2 Step-by-Step Execution**

**Given:**

```
if x > 0:
   y = 10
else:
   y = 20
```

- Check x > 0.
  - If true, execute y = 10.
  - If false, execute y = 20.
- Store the result in y.

**Axiomatic Semantics**

- It is based on symbolic logic and developed in conjunction with a method to prove the correctness of programs.
- Uses rules of inference to deduce the effect of executing a construct.

- Axiomatic semantics defines the meaning of a program using logical rules. It describes what must be true before and after executing a piece of code.
- The meaning of a statement, or group of statements **S** is described in terms of:
    i. The condition **P** (the pre-condition) that is assumed or asserted to be true before **S** is executed, and
    ii. The condition **Q** (the post-condition) which can be deduced to be true after execution of **S**.
- This is usually written as:

  **{P} S {Q}**

**Example 1**

- Consider the assignment statement **x:= x + 1**

  If the condition **x ≥ 0** is true before execution of the statement, then **x > 0** will be true after execution of the statement.
- This is written as:

  **{x >= 0} x:=x + 1 {x > 0}**

**Example 2**

**Given:**

```
if x > 0:
   y = 10
else:
   y = 20
```

We express its meaning using logical rules:

- **Precondition:** We know x is some number.
- **Postconditions:**
  - If x > 0, then y = 10.
  - If x ≤ 0, then y = 20.

This lets us prove that y will always be either 10 or 20 after execution.

**Denotational Semantics**

- It is the most rigorous widely known method for describing the semantics of a program.
- It is based on recursive function theory.
- The fundamental concept of denotational semantics is to define for each language entity, both a mathematical object and a function that maps instances of the entity onto instances of the mathematical object.

Example 1: Describing the Meaning of Binary Numbers

Consider the grammar rules:

**<bin-num>:=0|1|<bin-num> 0|<bin-num> 1**

- The objects in this case are simple decimal numbers.
- In this example, the meaning of a binary number will be the meaning of its decimal equivalent.
- Let the domain of semantic values of the objects be **N**, the set of non-negative decimal integer values.
- It is these objects that we want to associate with binary numbers
- Let the semantic function be $M_{bin}$

- The function $M_{bin}$ is defined as follows:

$M_{bin}$ ('0') = 0

$M_{bin}$ ('1') = 1

$M_{bin}$ (<bin-num> '0') = 2 * $M_{bin}$ (<bin-num>)

$M_{bin}$ (<bin-num> '1') = 2 * $M_{bin}$ (<bin-num>) + 1

## Example 2

### Given:

```
if x > 0:
   y = 10
else:
   y = 20
```

We can define the meaning using a mathematical function:

$$f(x) = \begin{cases} 10, & \text{if } x > 0 \\ 20, & \text{otherwise} \end{cases}$$

## Runtime Considerations

- Programming languages are designed with different runtime considerations that impact their efficiency, reliability, and usability.

- Understanding these considerations is crucial for selecting the right language for a given application.

- Various runtime concerns are:

**Execution Models:** Different programming languages use distinct execution models that influence runtime behaviour. The primary execution models include:

- **Interpreted Execution**: Code is executed line-by-line by an interpreter, e.g., Python, JavaScript.
- **Compiled Execution**: Code is translated into machine code before execution, e.g., C, C++.
- **Hybrid Execution**: A mix of compilation and interpretation, e.g., Java (bytecode executed by JVM), .NET languages (executed by CLR).

Each model affects runtime performance, portability, and flexibility.

**Memory Management:** Efficient memory management is essential for optimizing runtime performance. Programming languages employ different strategies:

- **Manual Memory Management**: Requires explicit allocation and deallocation (e.g., C, C++ with malloc/free/delete).
- **Automatic Garbage Collection**: The system automatically reclaims unused memory (e.g., Java, Python, Go).
- **Reference Counting**: Tracks object references and deallocates when no references remain (e.g., Swift, Objective-C).
- **Region-Based Memory Management**: Allocates memory in blocks and releases them together (e.g., Rust with ownership model).

**Runtime Performance Considerations** Several factors impact the runtime efficiency of programming languages:

- **Execution Speed**: Compiled languages typically perform faster than interpreted ones.
- **Memory Usage**: Garbage collection can introduce overhead but helps prevent memory leaks.
- **Concurrency and Parallelism**: Support for multithreading and asynchronous execution affects performance.
- **Optimization Techniques**: Just-In-Time (JIT) compilation, loop unrolling, and inlining improve runtime efficiency.

**Type Systems and Runtime Behaviour** The choice between static and dynamic typing influences runtime performance:

- **Statically Typed Languages** (e.g., Java, C) perform type checking at compile-time, reducing runtime errors.

- **Dynamically Typed Languages** (e.g., Python, JavaScript) check types at runtime, allowing flexibility but adding overhead.

**Exception Handling and Error Management** Error handling mechanisms impact the robustness of a runtime system:

- **Checked Exceptions**: Enforced at compile-time (e.g., Java).
- **Unchecked Exceptions**: Occur at runtime without mandatory handling (e.g., Python, C++).
- **Structured Exception Handling**: Used in systems programming for reliability (e.g., Windows SEH, POSIX signals).

Understanding runtime considerations is crucial for selecting and optimizing programming languages for various applications.

**Types, Values and Declarations**

- The type of a data item determines its allowed values together with the set of operations that can be used to manipulate these values.
- Data items have a **value** and a **type** and may be held in what are called **variables**.
- Variables have a name, various **attributes** and refer to an area of computer store.
- There is a distinction between the following:
    - The name of a variable : **Identifier**
    - Where the variable is stored: **Reference or Address**
    - The **value** stored.

1. **Identifier**

**Types of Characters Used**

- An identifier typically is a combination of letters and digits, with the first character being a letter.

- An exception is Perl where an identifier has prefix to show whether it is scalar ($) and array @ or a hash(%) variable.

**Spaces**

- Spaces are not normally allowed within identifiers, but some languages like Ada, C, C++, etc allow the use of underscore.
- Although Java allows the use of underscore, the convention is to use internal capitals instead e.g**leftLink** instead of **left_link**.

**Length**

- Early languages often restricted the length of identifiers to six or eight characters, but most modern languages have no limit.

**Case Sensitivity**

- In C, C++ and Java, case matters.
- In Pascal, Fortran 90 and Ada, case does not matter.

## 2. Declaration and Binding

**Binding:** The time when different language features are associated with, or **bound** to one another.

- Binding can take place at:
    - Compile time;
    - Load time; and
    - Run time.
- Early (compile-time) binding leads to efficient execution while late (run-time) binding leads to more flexibility.

**Name-Declaration Binding/Scope**

- The connection between the use of a name in a statement and its declaration is called **name-declaration binding/scope.**

- Java, C, C++ and Ada allow us to determine scope by looking at the program text alone i.e at compile time. This is called **Static Scope**.
- Consider the Java class declaration:

```
class Example
{
private double x, y;
public voidop1()
{
int y, z;
y=34;
z=27.4;
}//

public void op2()
{
x=3.768;
y=x;
}//op2

}//Example
```

- **y** and **z** declared inside **op1** are local variables. They are not visible outside op1
- **x** and **y** declared under **Example** are global variables. They are visible in **op1** and **op2**.
- The use of **y** in **op1** and **op2** refer to local variables. Ada, however, allows y declared under Example to be accessed inside **op1** and **op2** by the name **Example.y**.
- A language that employs static binding is said to be **statically typed**.
- When there are no loopholes in the type checking, the language is said to be strongly typed e.g Ada, Algol, Java and Python.

- In C++ you have:

```
#include <iostream>
class Example {
public:
   static int y;  // Static class variable (declaration)
};

// Definition of the static variable (required outside the class)
int Example::y = 10;

int main() {
   std::cout << Example::y << std::endl;  // Accessing static variable
   return 0;
}
```

**Advantages of Static Typing**

Static typing leads to programs which are:

- **Reliable:** Because errors are detected at compile-time.
- **Understandable:** Because the connection, or binding between the use of an identifier and its declaration can be determined from the program text.
- **Efficient:** Because type checks do not have to be made at run time.

**Dynamic Scope:** The binding between the use of an identifier and its declaration depends on the order of execution, and so is delayed until run time.

**Declaration-Reference Binding/Scope**

- During program execution, a program variable is allocated memory location in which its value is stored.
- The store location has a reference (i.e address) through which they can be accessed.

- When a variable is allocated storage, we say that the **declaration** of the variable is bound to a **reference.**

**How Variables are Allocated Memory Locations**

1. **On Block Entry**
   - This is when the procedure or method is called. When control is returned from the method call, the memory is de-allocated.
   - Data members of class are allocated when the object is created and de-allocated when the object expires.
2. **Static Variables**

- Binding the declaration of a local variable to a different store location (i.e reference) each time a procedure or method is entered has the consequence that a local variable does not retain the value it had when the method was last executed.
- To handle this problem, Algol 60, C, C++ and Delphi allow the declaration of **static local variable.**
- In object-oriented languages, there is no need for static variables as information that needs to be held from one method call to the other can be declared at the class level called **class variable**.
- For example, only one instance of **counter** variable will be created in the following:

```
class Example
{
private static int counter=0;
…
}
```

**Reference-Value Binding**

- The binding of a variable to its value involves three bindings:
  i. The binding of the variable's name to its declaration (name-declaration binding) .

ii. The binding of its declaration to a store location (declaration-reference binding).
iii. The binding of the store location to a value (reference-value binding).

**Dereferencing:** The process of finding the value, given the reference is called dereferencing.

- In the statement below, the **y** on the right-hand side is dereferenced to find its value so that 1 can be added to it.

  y=y+1
- A reference is sometimes called an **L-value** and the value of a variable an **R-value**.

**Constants**

- Constants are data items which do not change in a program.
- A constant has a name and a value, but the constant identifier cannot appear on the left-hand side of an assignment statement.
- The example below shows how to define constants in C++ and Java.
  const int size =20; // C++
  final int size =29; //Java

## 3. Type Definitions

- The most important attribute of a variable is its type.
- The characteristics of a type are its allowed values together with the set of operations that can be used to manipulate these values.

**Kinds of Types**

- The three main kinds of types are:
  i. Scalar type

ii. Structured type and
iii. Reference type

**Scalar Types**

- Numeric Data Types: Integer, floating point and fixed point and complex.
- Logical Type: Called Boolean
- Character Type

Scalar types can be split into two categories:

i. **Discrete/Ordinal Type**: Where each value (except the minimum and the maximum values) has a successor and a predecessor.
ii. **Others**: Like floating-point types for which this is not the case.

**Built-in Types**: These are types that are immediately available to the programmer.

**Specifying Type Information**

- In most programming languages, all variables must be declared before they can be used.
- Variables can be declared to be either one of the built-in types or of a user-defined type.

**4. Numeric Data Types**

- The arithmetic operators (+,-,*,/) are defined for numeric data types.
- However, the effect of these operators depends on the context of use.

**Example**

2 + 3 ---Integer addition

2.5 + 3.5----Floating-point addition

- When the effect of an operator depends on the type of its operands, the operator is said to be **overloaded.**
- C, C++ and Java use a single overloaded division operator.
    - E.g 7/2 gives 3----Integer division
    - 7.0/2.0 gives 3.5----Floating-point division
- Other operators available for numeric types are:
    - Exponentiation
    - Relational operator e.g <, <=, >, >=
    - Equality e.g =, ==, !=, <>, .EQ., and .NE.

**Note:** floating-point numbers are always represented approximately unlike integers that are represented exactly. Therefore, using equality operator on floating-point numbers do not always give the expected result.

**Integer**

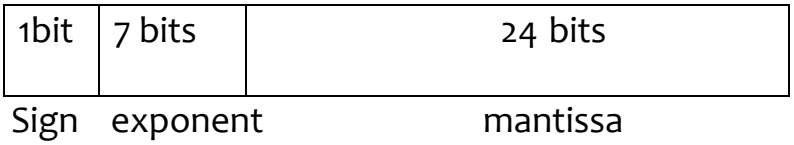
The range of integers is dependent on the machine hardware and the representation (no. of bits) used.

**How Integers can be Used**

- In normal mathematical sense
- As counters in a loop or array index
- Some languages have variants of Integer e.g C and C++ have types **short** and **long.**
- Java has 4 integer types:
    - byte 8 bits
    - short 16 bits
    - int 32 bits
    - long 64 bits

**Floating-Point Numbers**

- Most common type for mathematical calculations
- The range and precision is determined by implementation
- A floating-point number can be represented by 32 bits as follows:




- As the exponent increases, the range increases and the precision decreases
- Floating-point literals can be written with or without an exponent e.g 3.75, 2.5E+7, 0.17E-2, are valid in C, C++ and Java.

**5. Logical Types**

- Languages also support logical types
- In some languages, the name given to logical types is **Boolean.**
- In C, a false logical expression returns **0**, although any non-zero value is treated as true.
- C++(bool) and Java (boolean) have Boolean type.

**6. Character Types**

- Most high-level languages include a built-in character type together with character operations.
- The main use of characters is as the component of strings.
- Character values are usually enclosed by single quotes as in ‘a’, ‘A’, ‘;’, ‘ ‘, ‘3’

**Operations on Character**

- Input , output, assignment, relational and equality
- Java stores **char** values as 16 bit Unicode characters.
- In C and C++, a **char** is held in one byte and can be treated like an integer

- The inclusion of strings of characters in C, C++ and Pascal is accomplished by a data structure such as an array of characters.
- In Java and C++, there is a **String** library class.

### 7. Enumeration Type

- Enumeration types are user-defined types.
- In C and C++, an enumeration type for the days of the week could be defined as
  ***enum Days {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};***

  Operations on Enumeration Type
  - Relational
  - Equality
- In C and C++, enumeration types cannot be input and output and enumeration literals must be identifiers.

### 8. Reference and Pointer Variables

- Reference and pointer variables have an address as their value.
- C and C++ provide pointer arithmetic.
- Pointers can be assigned values and can also have a special value e.g**null** in C, C++ and Java.
- In most languages, the type being pointed to must be given as part of the declaration e.g in C and C++, we can have ***int * ponter;***
- Pointers can be used to allocate memory dynamically e.g
  - pointer=(int*****) malloc(sizeof(int)); --- in C
  - pointer=new int; ---------- in C++

**Garbage**

- The unused memory locations allocated dynamically is called garbage.
- Pascal and C++ provide a way to release the unused memory e.g

  ***int * pointer= new int;***
  ***delete pointer;***
- Java's garbage collector handles this automatically.

**Structured Data**

- A data structure is composed from simpler items.
- The items from which it is composed are called components which may be simple data items or other data structures.
- Two basic data structures in procedural languages are:
  - Array
  - Record
- In object-oriented languages, records are displaced with classes.

**Features of Array**

i. **Component Type**: All elements are homogeneous
ii. **Access to Components**: Through their positions
iii. **Efficiency**: Efficient access at run time, but not as much as simple variable due to computable index.
**iv. Use:**
   a. Used in conjunction with loops.
   b. In solution of scientific and engineering problems
   c. If the position of the required component has to be computed during run time, then arrays are an appropriate data structure.

**Attributes of an Array**

i. Type of its elements
ii. Number of components
iii. Type of subscripts

**Example:** C++

***int array[4];*** //An array of integer containing four elements and accessible via the subscript 0 to 3.

- In Java, an array is an object and is created using the **new** operator e.g

double [] arrow=new double[16];

In Python, a list can be created like the following:

numbers = [17, 123]
empty = []

- To create two-dimensional arrays in C, C++ and Java, we write
  - double a[10][16]; // C or C++
  - double [][]a=new double[10][16]; // Java
- To access the element of array **a** at row **i**, column **j**, we write **a[i][j]**
- To create a nested list in Python, we write
  - a = [[1, 2, 3], [4, 5, 6]]

**Accessing Arrays in C and C++**

Given: **double array[16];**

- Elements of an array can be accessed very efficiently in C and C++ using pointers.
- If b is declared as below

  ***double* b;***
- then the assignment ***b= &array[0];*** will assign the address of **array[0]** to **b**.
- This can also be written as **b=array;**

- The array element arrow[i] can be referred to using pointer arithmetic as ***(array+i)** or ***(b+i).**
- As **array** is constant, its value cannot be changed, but **b** is a variable and so it is possible to step through the elements of the array **array** by incrementing the value of the double pointer **b**.
- Hence instead of
  ***double total=0.0;***
  ***for (int i=0;i<6;i++)***
  ***total +=arow[i];***

It is possible to write:

***double total=0.0;***
***b*= array;**
***for(int i=0;i<16;i++) total +=*b++;***

which is more efficient.

**Accessing Lists in Python**

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
for cheese in cheeses:
        print (cheese)

numbers = [17, 123]
for i in range(len(numbers)):
        numbers[i] = numbers[i] * 2


matrix = [[1, 2, 3, 4], [5, 6], [7, 8, 9]]
for i in range(len(matrix)):
   for j in range(len(matrix[i])):
      print(matrix[i][j], end=' ')
   print()
```

**Name-Size Binding of Array**

1. This is the binding of the name of an array variable to its required amount of storage.
2. Three ways:
    1. Static Arrays (Compile-time binding)
        - The size is fixed when the array is declared and cannot change during run time.

**Advantages:**

- Simple to compile
- Fast at run time

**Disadvantages:**

- Inflexible
- Array must be declared maximum size, which can lead to storage wastage.

    2. Semi-Dynamic Arrays (Binding on block entry)
        - Size is not determined until block entry e.g Ada
    3. Dynamic Arrays ( Binding during statement execution)
        - Two types of dynamic arrays:
            ***i.*** In C++ and Java space can be allocated to an array using the ***new*** keyword e.g ***double * array=new double [16];***
            ***ii.*** The alternative is to have **extensible array** that allows new array elements at any time e.g SNOBOL 4, Icon and Perl.

**Array Parameter**

- As arrays are objects, we always pass a reference to an object in Java
- Array parameters in C++ are also passed by reference, but as array attributes are not available, the size of the array must be passed as a second parameter.

- To ensure that the array is not modified as part of the call, we can pass the array by reference constant as below:.
  ***double sum(const double a[], int length);***
- Alternatively, as array parameters are treated as pointers, we can write:
  ***double sum(const double* a, int length);***

**Name and Structural Equivalence**

- When a function or procedure is called, the types of the actual and formal parameters must match.
- Two approaches are employed:
  1. **Structural Equivalence:** This means that the actual and formal parameters have the same structure.
  2. **Name Equivalence:** This assumes that two variables have the same type if they have the same name.

**Operations on Complete Arrays or Slices of Arrays**

- An **array slice** is a substructure of the array. If **A** is a matrix, the following are slices: -first row, last column, first column, last row, etc.

1. **Array Aggregates:** Used for initializing an array e.g C, C++ and Java allow initialization of their arrays as below:
   int list[]={4,5,7,8,3};
   - The compiler sets the size of the array.
   - When the size of the array is given and the aggregate has insufficient values, the remaining values are initialized to zero.
   - In C++, aggregates can only be used to initialize an array within a declaration.
   - In Python, a list object can also be initialized like:

   numbers = [17, 123]

- Some languages allow operations on slices of array e.g Pascal, Ada and Python.

2. **Assignment**: E.g Ada
3. **Catenation**: In Ada, this is defined between two single-dimensioned arrays and between a single-dimensioned array and a scalar.
4. **Addition, Assignment, Relational, Logical Operations, matrix multiplication, matrix transpose and vector dot product**: Provided by Fortran 95.
5. APL provides four basic arithmetic operations for vectors (single-dimensioned arrays) and matrices, as well as scalar operands.
    - It also includes a collection of unary operators for vectors and matrices some of which are as follows (where **V** is a vector and **M** is a matrix):
        i. $\phi$ V reverses the elements of V
        ii. $\phi M$ reverses the columns of M
        iii. $\theta M$ reverses the row of M
    - It also includes special operators that take other operators as operands.

6. Python provides the following operations on a list:

| list Methods |
|---|
| `count`<br>Returns the number of times a given element appears in the list. Does not modify the list. |
| `insert`<br>Inserts a new element before the element at a given index. Increases the length of the list by one. Modifies the list. |
| `append`<br>Adds a new element to the end of the list. Modifies the list. |
| `index`<br>Returns the lowest index of a given element within the list. Produces an error if the element does not appear in the list. Does not modify the list. |
| `remove`<br>Removes the first occurrence (lowest index) of a given element from the list. Produces an error if the element is not found. Modifies the list if the item to remove is in the list. |
| `reverse`<br>Physically reverses the elements in the list. The list is modified. |
| `sort`<br>Sorts the elements of the list in ascending order. The list is modified. |

**Associative Arrays**

- An associative array is an array in which the index (called the key) can be a value of any type instead of computable scalar type of ordinary array.
- The indices of non-associative arrays never get stored, but the user-defined keys must be stored in the structure for associative arrays. Each element of an associative array is a pair of entities, a **key** and a **value**.

**Example:** Perl

- In Perl, associative arrays are often called **hashes**.
- Every hash variable must begin with a percent (%).
- Hashes can be set to literal values with the assignment statement, as in %salaries=("John"=>75000, "Mary"=>60000, "Tope"=80000);

**Records and Classes**

- A record is a structured type that is composed of heterogeneous elements of data.
- Its definition specifies its name and the types of the various fields of the record.
- A record groups logically related data as one structured variable.
- The individual items in a record are accessed by name (using dot notations).
- In C and C++, a record is called a ***struct*** and the components are called members.
- Type **Date** can be defined as :

```
struct Date
{
int day;
int month;
int year;
};
```

- To declare a variable of Date in C++, you write

***Date adate;***

- Traditionally, records contain fields. However, an extension is to allow the fields also to be operations.
- A C struct has only data members, while a C++ struct may also have function members.
- In C++ struct, data members are public but class data members are private by default.

**Variant Records**

- A variant record has a fixed part, with fields as in normal record followed by a variant part, in which there are alternative fields.
- Specific actions are taken based on the contents of the variant part of the record e.g if a record has a variant part that consists of whether a person is an undergraduate or postgraduate. A statement can be written to print advisor name is an undergraduate, otherwise, print the course and supervisor if a postgraduate.

**Features of Record**

1. **Component Type:** Heterogeneous
2. **Access to Components**: By the name of the component field of the record e.gjoe.age
3. **Efficiency:** As the field selectors are fixed a compile time, the components of a record may be accessed as efficiently as simple variables.
4. **Use:** They are used to group together, data items which are components of a single item in the problem domain.

**Strings**

- A string over an alphabet (of a language) is a sequence of one or more characters of the alphabet obtained through concatenation.

**Operations on String**

i. String comparison

ii. Selection of substring
iii. Searching for substring
iv. Moving of strings
v. Replacement of substrings with a string
vi. Appending one string to the end of the other

- Strings can be implemented as:
    i. Fixed length
    ii. Variable length

- **Fixed-length Strings**

o The length of strings are fixed so that all strings have the same length

o All strings are allowed the same amount of memory

o A string of lesser length must be supported with blank character to make up the length

o A string is implemented as a linear array of characters

o The size of the array is the same for all the strings

Below is how the strings "good", "for" and "you" are represented using an array of size 8

| g | o | o | d | | | | |
|---|---|---|---|---|---|---|---|

| F | o | r | | | | | |
|---|---|---|---|---|---|---|---|

| y | o | u | | | | | |
|---|---|---|---|---|---|---|---|

**Disadvantages**

- storage wastage
- time wastage for inserting or reading blank spaces that make up the string

**Advantages**

- modification, copying or extraction of a string is easily undertaken as all strings are of the same length

**Variable-length Strings**

- In this scheme, the string length may vary and so the need to accommodate varied sizes of memory store to strings
- The methods involved are:
    - Boundary marker technique
    - String descriptor technique and
    - Linked structure technique

**Boundary marker technique**

- A string is always terminated by a non-printable character
- The character is considered as a boundary marker to mark the end of the string

Strings "good", "for" and "you" are stored as below:

| g | o | o | d | # | | | |
|---|---|---|---|---|---|---|---|

| F | o | r | # | | | | |
|---|---|---|---|---|---|---|---|

| y | o | u | # | | | | |
|---|---|---|---|---|---|---|---|

using different arrays and

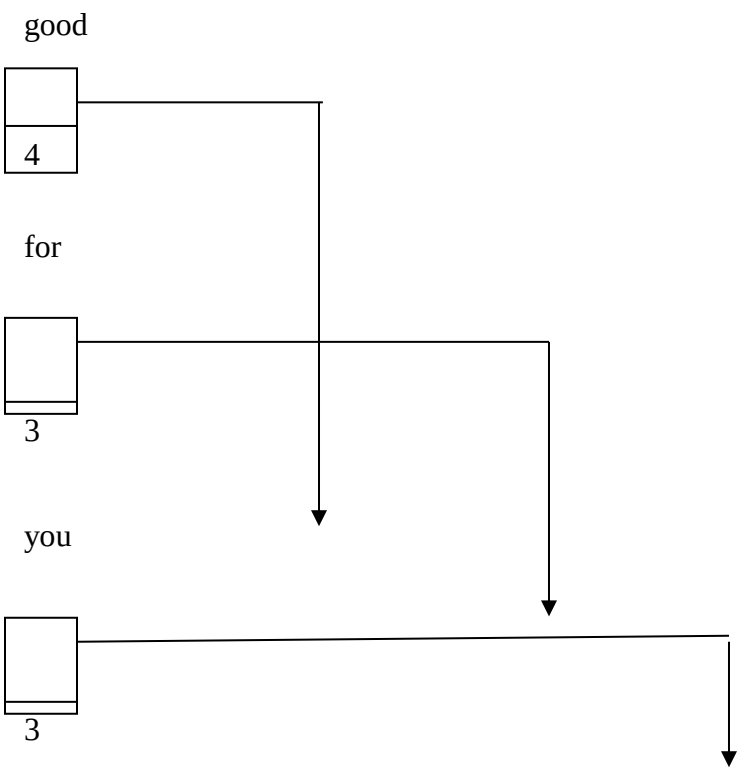
| g | o | o | d | # | f | o | r | # | y | o | u | # | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

using a single array

- The problem with this scheme is that the size of the string cannot be easily changed as much processing is involved.

**String Descriptor Technique**

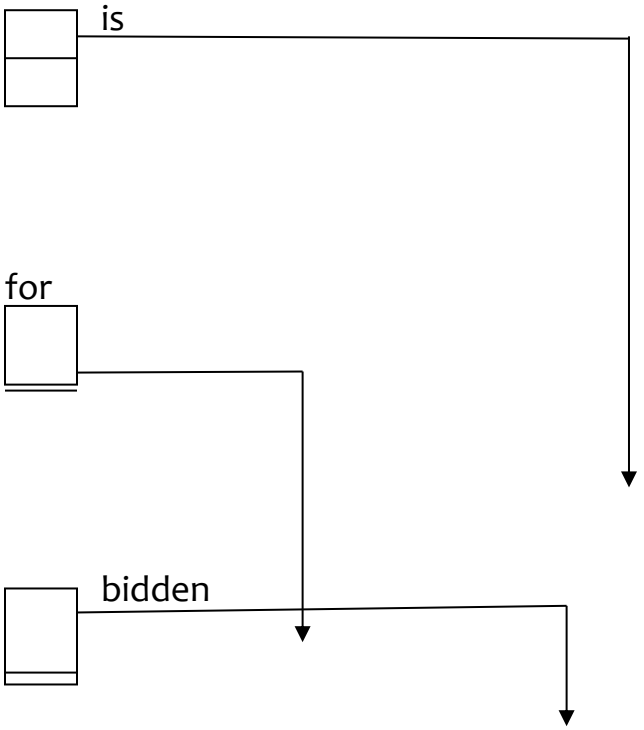
- The start address of the string as well as the length are kept
- Each string can be described by a record containing two fields:
  - **A pointer:** that points to the start of the string in memory
  - **A field of integer type:** that contains the length of the string




| g | o | o | d | f | o | r | y | o | u | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

- The difficulty in accommodating a string whose size may change is also the problem with the scheme

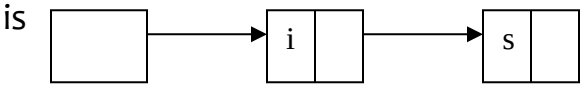
- **Linked-Storage Technique**

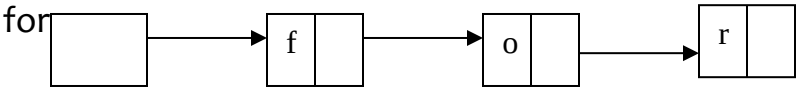



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| f | r | o | s | e | b | i | n | d | i | d | | | | | |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 0 | 2 | 0 | 8 | 10 | 4 | 0 | 5 | 1 | 9 | | | | | |

- An array of record is used.
- Each record has two fields, one of which stores a character of the string and the other containing the number of the array component containing the next character in the string.
- Dynamic storage can also be used. This consists of a sequence of records linked together by making each record to contain a pointer to the record containing the next character to the one it contains in the string.
- Shrinking and expansion of string is easily accommodated, its storage demand is the problem



- C represents strings as arrays of characters, but variable up to a maximum defined length.
- A C-style strings, which consists of an array of characters terminated by the null character '\0', and which have properties over and above those of an ordinary array of characters, as well as a whole library of functions for dealing with strings represented in this form. Its header file is **'cstring'**.
- In C++, there are two types of strings, C-style strings, and C++-style strings.
- A C++ style string is a **'class'** data type. The objects of C++ style string are instances of the C++ 'string' class.
- There is a library of C++ string functions as well, available by including the 'string' header file.
- In Java, string is an object. Many string operations in Java make it appear that **String** is a primitive type like **int**, **double** or **char**e.g

  *String st ="Hello";* instead of *String st = new String("Hello");*

- In C# and Java, the data type String is treated as reference type. Instance of Strings are treated as (immutable) objects in both languages, but support for string literals provides a specialized means of constructing them. C# also allows verbatim strings for quotation without escape sequences, which also allow newlines.
- In Python, string is an object with the following operations:

| str Methods |
| --- |
| `upper`<br>Returns a copy of the original string with all the characters converted to uppercase |
| `lower`<br>Returns a copy of the original string with all the characters converted to lower case |
| `rjust`<br>Returns a string right justified within a field padded with a specified character which defaults to a space |
| `ljust`<br>Returns a string left justified within a field padded with a specified character which defaults to a space |
| `center`<br>Returns a copy of the string centered within a string of a given width and optional fill characters; fill characters default to spaces |
| `strip`<br>Returns a copy of the given string with the leading and trailing whitespace removed; if provided an optional string, the `strip` function strips leading and trailing characters found in the parameter string |
| `startswith`<br>Determines if the string is a prefix of the string |
| `endswith`<br>Determines if the string is a suffix of the string |
| `count`<br>Determines the number times the string parameter is found as a substring; the count includes only non-overlapping occurrences |
| `find`<br>Returns the lowest index where the string parameter is found as a substring; returns −1 if the parameter is not a substring |
| `format`<br>Embeds formatted values in a string using `1`, `2`, etc. position parameters (see Listing 11.4 (`stripandcount.py`) for an example) parameter is found as a substring; returns −1 if the parameter is not a substring |

## Sets

- A set is an unordered collection of distinct elements of the same type.
- Very few programming languages include set with the exception of Pascal
- Ada, Java, C and C++ allow operations on bit strings which can be manipulated like sets.
- In C++

```
#include <iostream>
#include <set>
int main() {
std::set<int> mySet; // Declare a set
// Insert elements
```

mySet.insert(40);

- Python also has sets which can be created using either curly brackets ({}) or the set() constructor. For example:
  s1 = {1, 2, 3}
  s2 = set([1, 2, 3, 4])

**Files**

- Files are large collections of data that are kept on secondary storage devices.
- Most modern languages like C, C++, Java, C# and Python allow for the creation of files.

**Class**

- A class corresponds to a type.
- It defines the properties of an object
- An object is an instance of a class.
- A language that supports data abstraction, encapsulation and information hiding, is often described as object-based.
- To define a class in C++ for example, do:

```
class Student
{
private:
short Age;
stringMatriculationNumber;
public:
voidsetParticulars()
};
```

**A Class with Class Members**

```
#include <iostream>

class Outer {
public:
   class Inner {  // Nested class
   public:
      void display() {
         std::cout << "Inside Inner class" << std::endl;
      }
   };

   void createInner() {
      Inner in;
      in.display();
   }
};

int main() {
   Outer::Inner obj;  // Accessing nested class
   obj.display();

   Outer outer;
   outer.createInner();

   return 0;
}

#include <iostream>

class Engine {
public:
   void start() {
      std::cout << "Engine started!" << std::endl;
   }
};

class Car {
```

```
private:
   Engine engine;  // Engine class as a member
public:
   void startCar() {
      engine.start();
   }
};

int main() {
   Car myCar;
   myCar.startCar();  // Calls Engine's start method
   return 0;
}
```

**Abstraction**

- An abstraction is a view or representation of an entity that includes only the most significant attributes.
- It allows a programmer to collect instances of entities into groups in which their common attributes are not considered (abstracted away). Only the distinguishing ones are considered.
- It is a weapon against the complexity of programming; it simplifies the programming process.

**Types of Abstraction**

1. Process abstraction
2. Data abstraction

**Process Abstraction**

- All subprograms are process abstractions because they provide a way for a program to specify that some process is to be done, without providing the details of how it is to be done (at least in the calling program).
- For example, the call **sortList(list, size)** is an abstraction because the details of the implementation of the subprogram is not specified.

**Data Abstraction**

- An abstract data type is an enclosure that includes only the data representation of one specific data type and the subprograms that provide the operations for that type.

- Programs units that use an abstract data type declare variables of that type called **objects.**

**Examples of Abstract Data Types**

1. **Floating Point (A language-defined type)**
   - Because a floating-point type provides a way of creating variables of floating-point data and also provides a set of arithmetic operations for manipulating objects of the type, it is an example of an ADT.

2. **User-Defined Abstract Data Type**

   - A user-defined type should also provide the same characteristics provided by language-defined types:
     i. A type definition that allows program units to declare variables of the type.
     ii. A set of operations for manipulating objects of the type.
   - In the contexts of user-defined types, an abstract data type is a data type that satisfies the following:
     i. The declarations of the type and the protocols of the operations on objects of the type.
   - The representation of objects of the type is hidden from the program units that use the type, so the only direct operations possible on those objects are those provided in the type's definition.

**Design Issues for Abstract Data Types**

- A facility for defining ADTs in a language must provide a syntactic unit that encloses the type definition and subprograms definitions of the abstraction operations.
- It must be possible to make the type name and subprogram headers visible to clients (programs that use ADTs) of the abstraction.
- Few, if any, general built-in operations should be provided for objects of ADTs, other than those provided with the type definition.

- Some operations required by ADTs even though not universal, must be provided by the designer of the type e,g iterators, accessors, constructors and destructors.
- Should the language support parameterized ADTs? A parameterized ADT allows for storing elements of any scalar type.

**Abstract Data Type in C++**

- C++ was created by adding features to C.
- Because one of the primary components of object-oriented programming is abstract data types, C++ obviously must support them.

**Encapsulation**

- The data defined in a C++ class are called **data members**.
- The functions (methods) defined in a class are called **member functions**.
- All of the instances of a class share a single set of member functions.
- Each instance gets its own set of class's data members.
- Class instances can either be stack-dynamic (references directly with value variables) or heap-dynamic (referenced through pointers).
- Stack-dynamic instances of classes are always created by the elaboration of an object declaration.
- The lifetime of such a class instance ends when the end of the scope of its declaration is reached.
- Classes can have heap-dynamic data members, so that even though a class instance is stack-dynamic, it can include data members that are heap-dynamic.
- C++ provides the **new** and **delete** operators to manage the heap.
- A member function of a class can be defined in two distinct ways:
    i. **Inline Function:** The complete definition can appear in the class. This means that its code is placed in the caller's code rather requiring the usual call and return linkage process.
    ii. If only the header of a class member function appears in the class definition, its complete definition appears outside the class and is separately compiled.

- A good practice is to have small member functions **inlined,** because linkage time will be saved at a cost of only a small amount of space.

**Information Hiding**

- A C++ class can contain both hidden and visible entities (hidden from or visible to clients of the class)
- Entities for hiding are placed in a **private** clause.
- Visible ones are written in a **public** clause.

- When **protected** clause is used, it means that the members of a derived class can access the protected member whereas, it cannot access **private** members.
- Constructors are used to initialize the data members of newly created objects.
- They are called when an object of the class type is created.
- They have the same name as the class whose objects they initialize.
- A destructor can also be included in a class and is called when the lifetime of an instance of the class ends.
- All heap-dynamic objects live until explicitly de-allocated with **delete** operator.
- The name of a destructor is the class's name, preceded by a tilde (~).
- Constructors and destructors do not have **return** statements.
- They can be explicitly called.

**Abstract Data Types in Java**

- Java support for ADT is similar to that of C++, but there are, a few important differences:
    - All user-defined types in Java are classes (Java does not include structs).
    - All objects are allocated from the heap and accessed through reference variables.

- Subprograms (methods) in Java can be only in classes; Java ADTs are both declared and defined in a single syntactic unit.
- Definitions hidden from clients by making them private.
- Rather than having private and public classes in its class definitions, in Java, access modifiers can be attached to a method and variable definitions.

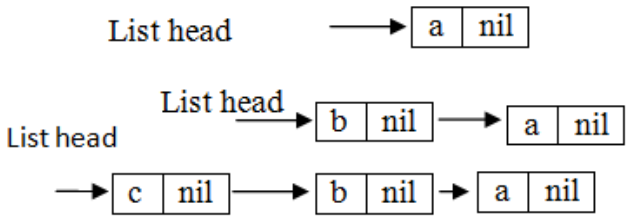
**Dynamic Data Structures**

- Involves the use of records and classes in conjunction with pointer variables.
- Components of a record or a class can be specified to be a pointer type that allows data structures in which individual records are linked to others.

**List**

- A list is a sequence of zero o more objects of some given type that may allow the deletion of its objects and the insertion of an object and so may vary in length.
- A list can be implemented using array or as a linked list.

Below are the stages




The code below implements the above linked list and prints out the contents.

```
#include <iostream>
using namespace std;
```

```
class Node
{
private:
char data;// to contain each character
Node* next;// to point to the nest node
public:
Node(char c, Node*p)
{
data=c;
next=p;
}// constructor
friend class List;// the class that makes use of class Node

};//Node
class List
{
private:
Node* head;// a pointer to store the memory location of the node

public:
List()
{
head=NULL;
}// constructor

void add (char c);
void inList();
};//List

void List::add(char c)
{
head=new Node(c, head);// allocate memory for a new node and store in head
}// add

void List::inList()
{
   Node*temp=head;
```

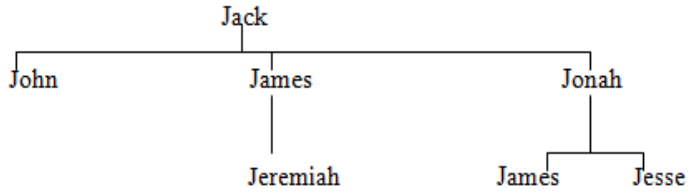
```
while(temp!=NULL)
   {
cout<<temp->data<<"\t\n";
temp=temp->next;
    }
}//inList

int main(void)
{
 List aList;
aList.add('a');
aList.add('b');
aList.add('c');
aList.inList();
system("pause");
return 0;
}
```

**Operations on ADT List**

i. Initialize(L): Makes the List empty.
ii. Empty(L): Returns true if the list is empty or false otherwise.
iii. Full(L): Returns true if the list is full or false otherwise (for bounded list)
iv. Insert (b,i,L): Inserts objects b at position number i on the list L
v. Delete(b,i,L): Deletes the object at position number I on the list L if it exists.
vi. Locate(i,L): returns the position of object b on list L if it exist
vii. Next(p,L): returns a pointer to the list item next to the one pointed to by p or null otherwise.
viii. Previous(p,L): returns a pointer to the list item previous to the one pointed to by p or null otherwise.
ix. Last(L): returns the pointer to the last item on the list or null otherwise.
x. Prinlist(L): prints the objects on list L in their order on the list.

## Trees

- A tree is a data structure that consists of a node called its root together with zero or more subtrees each of which is a tree.




- The root of the tree is at the top
- Its subtrees are drawn below.
- The direct descendant nodes of a tree are called it children nodes.
- These concepts can be extended to represent parent, grant child, and grandparent relationships.

## Levels of a Tree

- The tree root is at level one, its children are at level 2; their children at level 3 and so on.

## Degree of a Tree

- The degree of a node is the number of children it has.
- The degree of a leaf is 0.
- The degree of a tree is the maximum of the degree of its nodes. (i.e. the degree of the node with the highest degree).

## Binary Tree

- A binary tree is a finite (possibly empty) collection of elements.
- When the binary tree is not empty, it has a root element and the remaining elements (if any) are partitioned into two binary trees, which are called the left and right subtrees of t.

**Differences Between An Ordinary Tree and a Binary Tree**

- Each element in a binary tree has exactly (one or both of these subtrees may be empty). Each element in a tree can have any number of subtrees.
- The subtrees of each element in a binary tree are ordered, that is we distinguish between the left and the right subtrees. The subtrees in a tree are unordered.

**Operations on ADT Binary Tree**

**Operations:**

i. isEmpty() : return true if empty, return false otherwise root(); return the root element
ii. makeTree(root, left, right) : create a binary tree with root as the root element, left(right) as the left subtree.
iii. removeleftsubtree(): remove the left subtree and return it;
iv. removerightsubtree(): remove the right subtree and return
v. Preorder : preorder traversal of binary tree
vii. Inorder :Inorder traversal of binary tree
viii. Postorder :Postorder traversal of binary tree
ix. Levelorder: Level-order traversal of binary tree

**Expressions**

- An expression yields a value while a statement is a command to carry out some operation that normally alters the state (the state can informally be referred to locations in the memory and their associated values).
- Expressions are composed of one or more operands (variables or constants) whose values may be combined by operators.

- Precedence rules are normally applied when expressions are evaluated.
- Java has the following precedence rules:
    i. Evaluate any expressions enclosed within brackets, starting with the innermost brackets.
    ii. Apply the operator precedence below within brackets:

| | |
|---|---|
| ! | highest |
| *,/,% | |
| +,- | |
| <, >,<=, >= | |
| ==, != | |
| && | |
| \|\| | lowest |
| = | |

    iii. If the operators in **ii** have the same precedence, apply them in order from left to right.
- Fortran, Ada, C, C++, C# and Python have similar rules.
- Operators in expressions could be binary (dyadic) or unary (monadic).

**Boolean Expressions**

- In a Boolean expression, the value returned must either be **true** or **false.**
- They often involve relational operators, as in
    - a + b> o
    - x <= y

- A Boolean expression can also contain variables of type Boolean and Boolean operators.
- The common Boolean operators are **not**, **and** and **or** in Pascal and Ada and !, && and || in Java, C and C++.
- Ada, Java, C and C++ also include exclusive or (xor in Ada and ^ in Java, C and C++).
- Algol 60 has two extra Boolean operators for **implies** and **equivalence.**

**Precedence of Boolean Operators**

- The usual convention is to give **not** the highest precedence (above *) with **and** and **or** below the relational operators.
- Given if (i<0) or (a[i]>a[i+1]) then
  **A** **B**
- Short circuiting:
  - Given **A and B**, where **A** is false, **B** is not evaluated.
  - Given **A or B** where **A** is true, B is not evaluated.

**Mixed-Mode Expressions**

- A mixed-mode expression contains operands of different types.
- Languages such as Ada and Modula-2 forbid mixed-mode expressions.
- Pascal and Java allow **'sensible'** combinations such as adding integer to a floating-point value. The result will be floating-point.
- C and C++ allow **'unusual'** combinations like adding integer to character. The result can, however, lead to unexpected results.

**Statements**

- Statements are commands which perform actions and change the state.
- Typical statements are:
  - Assignment statement
  - Conditional statement

- Iterative statement
- Procedure and method calls

**Assignment Statement**

- The general form of the assignment statement is

  **el = er**

where **el** is the expression on the left-hand side that gives a reference as its result.

**er** is the expression on the right-hand side that gives a value as its result.

- Assignment statement is basically reference-value pairs.
- In C, C++ and Java, an assignment such as **el = er** is an expression and so can be used within other expressions.

  Many languages allow **el** to be only:
  - variable name;
  - an indexed variable;
  - a dereferenced pointer variable; or
  - the field of a record.
- For assignment compatibility between **el** and **er**, some languages say if the two are the same, then no problem, but if the two are not, then it is a compile-time error.
- Pascal and Java make use of what is called **widening**. If **el** has a floating-point type and **er** has an integer type, then the assignment is valid. If it is the other way round, they are not allowed.
- Algol 60 uses rounding while Fortran, C and C++ use truncation.

**Assignment Operators**

- Consider **a = a + expression;**
- C, C++, Java, Algol 68 and Modula-2 provide short-hand forms as:
  - a +=expression
  - others are -=,*=,and /=

- C, C++ and Java have:
  - ++ increment operator
  - -- decrement operator
- Consider
  - a =1; b = ++a // set **a** to 1, increment the value of **a** by 1 and then assign its value to **b**
  - a=1; b=a++; // sets **a** to 1, then assign **a** to **b** before incrementing **a.**
- Languages like Algol 68, C, C++ and Java allow multiple assignment statement like below:
  - a = b= c= expression
  - This kind of expression is executed from right to left.

**Compound Statements**

- Compound statements are essential when constructing structured control statements.
- Pascal uses begin and end brackets e.g

  ***begin***

  ***s1;s2;***

  ***…***

  ***end***
- Ada and Fortran 90 use explicit terminators for conditional and iterative statements.
- Consider the if statement below in Ada;

  ***if*** *A > B* ***then***

  *sum:= sum + A;*

  *Account :=Account + 1;*

  ***else***

  *sum ;+ sum + B;*

  *Bcount :=Bcount + 1;*

***end if;***

**Sequencing and Control**

- A **control structure** is a control statement and the collection of statements whose execution it controls.

**Selection**

- A **selection statement** provides the means of choosing between two or more execution paths in a program.

**Types:**

1. Two-way
2. N-way

**Two-Way Selection Statements**

General form:

***If control_expression***
***then clause***
***else clause***

**Control Expression**

- Control expressions are specified in parentheses if the **then** reserved word is not used as in C-based languages.
- When **then** is used, there is no need for parentheses.
- Arithmetic expressions can be used as control expressions as in C.
- In languages like Ada, Java, C++ and C#, Boolean expressions are used. C++ can also use arithmetic expressions.

**Clause Form**

- In most contemporary languages, the **then** and else clauses appear as either single or compound statements.

**Nested Ifs and Dangling Else Problem**

- Consider the statement below in Algol:
  ***if C1 then if C2 then S1 else S2***
- It is not clear whether the single else is associated with the first or the second if… then.
- Therefore, two interpretations are possible:
  **if** C1 **then begin if** C2 **then** S1 **end else** S2
  or
  **if** C1 **then begin if** C2 **then** S1 **else** S2 **end**

- Different languages provide different solutions for this ambiguity:
  - Algol 60 forbids conditional statement after a **then** and so extra **begin … end** brackets must be used.
  - Pascal, C, C++ and Java make the interpretation that the else is associated with the innermost **if**; when the other interpretation is required, a compound statement must be used.
- In cases when we have nested **ifs**, it is usually better for nested if statement to follow the **else.**

**Multiple Selection Constructs**

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

**Example**

- The general form of C multiple construct, **switch**, which is also part of C++ and Java is as below:

***switch(expression)***
***{***
***case constant_expression_1: statement 1;***
***…***

***case constant-expression_n; statement n;***
***default: statement_n+1;***
***}***

**Iterative Statements**

- Iterative statements allow for the execution of statements repeatedly until a certain condition is fulfilled.

**While Statement**

- This has the form of
  ***while (C ) S  //C, C++ and Java***

where **C** is a conditional expression and **S** is a statement.
**S** is normally a compound statement that must make the condition **C** false if we are not to have an infinite loop.

- It is possible to exit from a C, C++ and Java loop using a **break** statement.

- A variant of the **while** loop allows the statement to be executed at least once by placing the test at the end as in:

  **do statement-sequence while ( C) ;**
- Pascal form loops until the condition is true i.e **repeat statement-sequence until C**

**for Statement**

- This is used for iteration that is performed a fixed number of times.
- Pascal has two forms of the **for** statement:
  ***for*** *cv := low* ***to*** *high* ***do*** *S*
  ***for*** *cv := high* ***downto*** *low* ***do*** *S*

where **cv** is a control variable**, low** and **high** are discrete expressions and **S** is a statement.

- C, C++ and Java have a very complex **for** statement with the general form:
  ***for (e1;e2;e3) S***

where each of the expressions **e1**, **e2**, **e3** are optional.

- The form is equivalent to:

***e1;***
***while (e2)***
***{***
***S;***
***e3;***
***}***

**Scope of Control Variable for a '*for*' Loop**

- Ada states that control variable only has scope within the **for** statement, but not available outside.
- C++ and Java allow control variable to be declared within the **for** loop as in:
  ***for (int j=o;j<n;j++) S***
- The **j** variable can also be declared within the block where the **for** loop appears.
- Pascal, Ada, C and C++ allow enumeration types to be used as control variable as below:
  ***enum Month=(Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);***
  ***for (j=Jan; j<=Dec; j++)***

**goto Statement**

- There is a general consensus against the use of **goto** statements.
- The use of **break** as a controlled exit from a loop and having exception handling facilities preclude the use of **goto.**

**Exception Handling**

- Exceptions are unusual happenings in a program which could be as a result of errors like:
  - Division by zero;
  - Erroneous user input;
  - Out-of-bound array subscripts; and
  - Unexpected encounter of end of file.

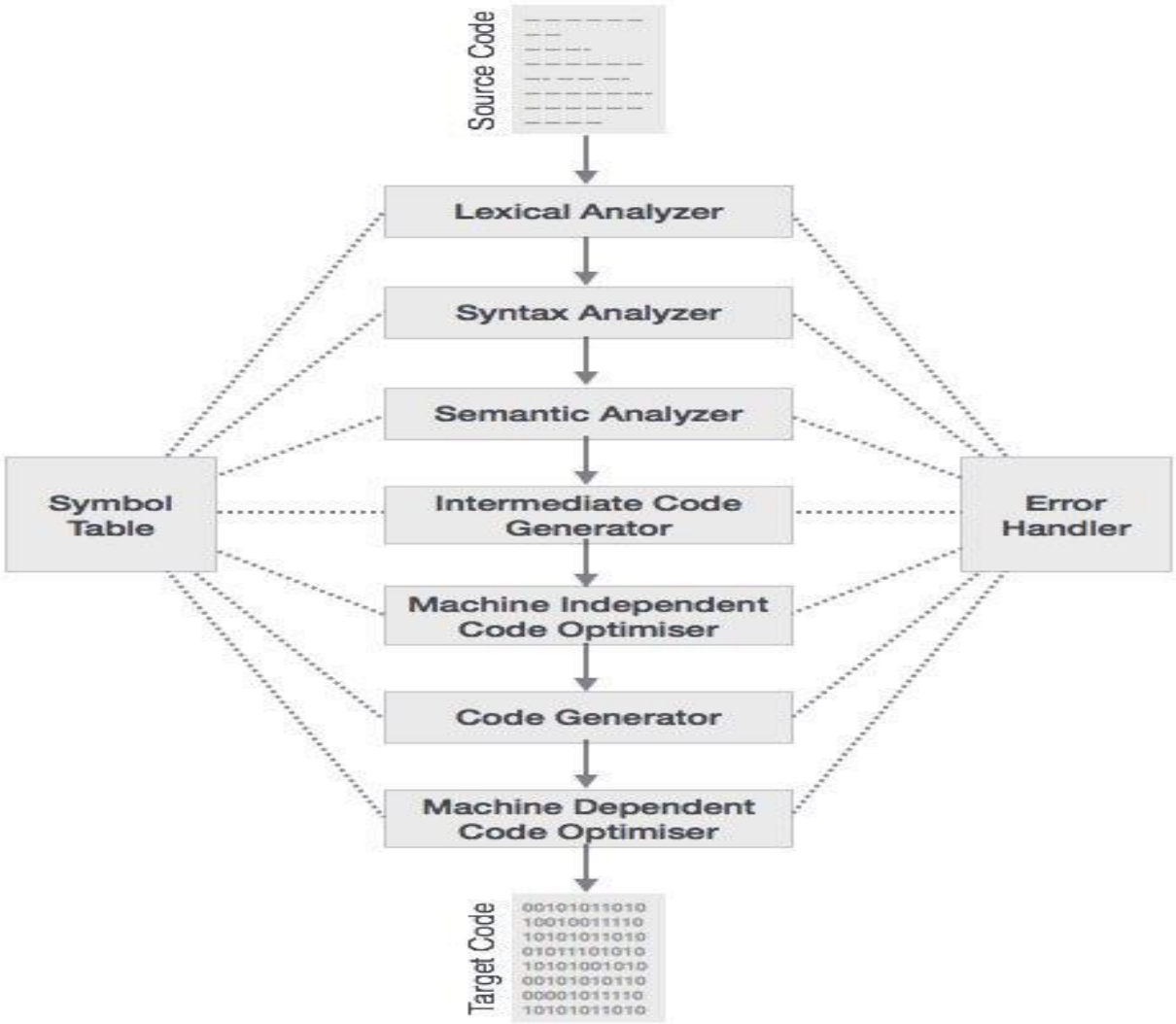
- Many programming languages leave the programmer to handle exceptions.
- PL/1, Ada, Java, Delphi, C++ and Python provide facilities to handle exceptions.
- When an exceptional event occurs, an exception is said to be **raised** (Ada and Delphi) or **thrown** (C++ and Java).
- Normal program execution is interrupted and control is transferred to the **exception handler**.
- A program that handles exceptional events that might occur is called a **fault-tolerant program.**
- In Java, a statement where an exception may occur should be enclosed inside a **try** block:

```
try
{
...
if ( a[i] < 0)...
...
}catch (ArrayIndexOutofBoundException e) { ...}
```

- If the subscript goes out of range, an exception is thrown and execution of the **try** block is terminated.
- The exception is then caught by a **catch** block where remedial action can take place.
- A method that throws an exception that is not caught and handled within it in Java must show such in its declaration e.g
  ***public void getfile( ) throws IOException{...}***
- The above states that function **getFile** throws an exception which is not caught and handled in the method.
- The syntax of exception handling in C++ and Java are similar, but the facilities provided by C++ are more complex.

## PHASES OF COMPILER

· The compilation process is a sequence of various phases. **Each phase takes input from its previous stage**, has its own representation of source program, and **feeds its output to the next phase of the compiler**




**Lexical Analysis:**

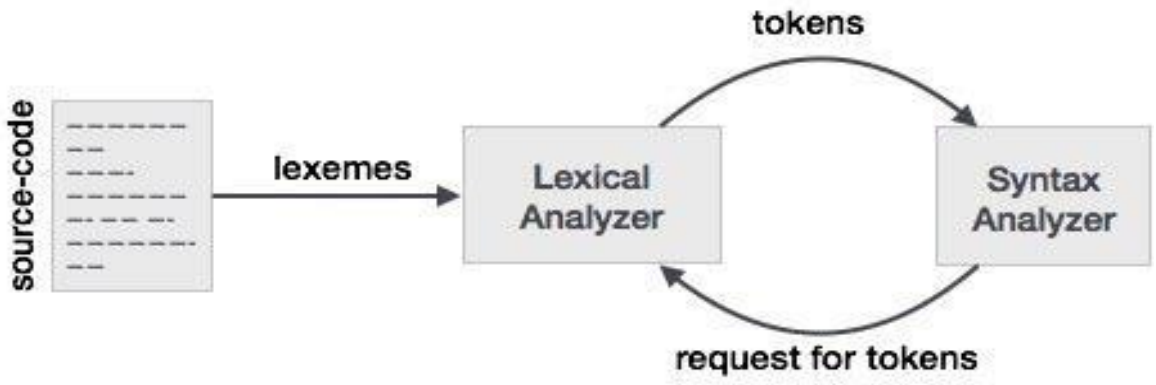
- The first phase of scanner works as a text scanner.
- This phase **scans the source code as a stream of characters** and **converts it into** meaningful **lexemes**, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.

**Syntax Analysis**

- This phase **takes the list of tokens** produced by the lexical analysis and **arranges these in a tree-structure** (called the **syntax tree**) that reflects the structure of the program.
- This phase is often called **parsing**.

## LEXICAL ANALYSIS

- First phase of a compiler.
- Takes the modified source code from language preprocessors, written in the form of sentences and **breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code**.
- It **reads character streams from the source code**, **checks for legal tokens**, and **passes the data to the syntax analyzer** when it demands.
- It **generates errors or any invalid token**.




### Token and Lexeme

**Token:** Token is a sequence of characters that can be treated as a single logical entity.

- Typical tokens are:

  1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

**Lexeme:** A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

**Example:** Consider the C++ statement: ***index = 2 * count + 17***

| **Lexemes** | **Tokens** |
|---|---|
| index | Identifier |
| = | Equal_sign |
| 2 | Int_literal |

| | |
|---|---|
| * | Mult_op |
| count | Identifier |
| + | Plus-op |
| 17 | Int_literal |
| ; | Semicolon |

- In C++ the variable declaration line: **int *value* = 100;** contains the tokens:
  int (*keyword*), *value* (*identifier*), = (operator), 100 (*constant*) and ; (*symbol*).
- In programming language, **keywords**, **constants**, **identifiers**, **strings**, **numbers**, **operators**, and **punctuations symbols** can be considered as tokens.
- The predefined rules for every lexeme to be identified as a valid token are defined by grammar rules, by means of a pattern/specification.
- **Lexers can be constructed by hand, but are normally constructed by lexer generators**, which transform human-readable patterns/specifications of tokens and white-space into efficient programs.
- For lexical analysis, specifications are traditionally written using **regular expressions**: an algebraic notation for describing sets of strings.
- The generated lexers are in a class of extremely simple programs called **finite automata**.

## SPECIFICATIONS OF TOKENS

- Let us understand how the language theory undertakes the following terms:

### Alphabets

- Any finite set of symbols {0,1} is a set of binary alphabets.
- {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets.
- {a-z, A-Z} is a set of English language alphabets.

### Strings

- **Any finite sequence of alphabets** is called a string.
- **Length of the string** is the **total number of occurrence of alphabets**, e.g., the length of the string tutor is 5 and is denoted by |tutor| = 5.
- A **string having no alphabets**, i.e. a string of zero length is known as an **empty string** and is **denoted** by **ε (epsilon)**.

**Special Symbols**

· A typical high-level language contains the following symbols:

| **Arithmetic Symbols** | **Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)** |
|---|---|
| Punctuation | Comma(,), Semicolon(;), Dot(.), Arrow(->) |
| Assignment | = |
| Special Assignment | +=, /=, *=, -= |
| Comparison | ==, !=, <, <=, >, >= |
| Preprocessor | # |
| Location Specifier | & |
| Logical | &, &&, \|, \|\|, ! |
| Shift Operator | >>, >>>, <<, <<< |

**SYNTAX ANALYSIS**

- **Syntax analysis** or **parsing** is the second phase of a compiler.
- A lexical analyzer can identify tokens with the help of regular expressions and pattern rules.
- But a **lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions.**
- Regular expressions cannot check balancing tokens, such as parenthesis.
- Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

- **To specify the syntax of a language, we use Backus-Naur Form(BNF) or Context Free Grammar(CFG).**

- CFG is a superset of Regular Grammar, as depicted below:


- It implies that every Regular Grammar is also context-free.
- CFG is a helpful tool in describing the syntax of programming languages.

**Context-Free Grammar**

- A context-free grammar is synonymous with a BNF grammar.

A context-free grammar has four components:

1. **The set of *terminal symbols* (Σ).**
- They are the basic symbols from which strings are formed.
- They are said to be terminal, because they cannot be substituted by any other symbols.
- The substitution process stops with terminal symbols.

2. **The set of *non-terminal symbols (V)*.**
- They denote syntactic classes and can be substituted.
- The non-terminals define sets of strings that help define the language generated by the grammar

**3. The set of *syntactic equations* (also called *productions*) (P).**
- These define the possible substitutions of nonterminal symbols.
- The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings.
- Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production

**4. The *start symbol (S)*.**
- It is a nonterminal symbol designated as the start symbol (S); from where the production begins.

- The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

**Example**

- We take the problem of palindrome language, which cannot be described by means of Regular Expression. That is, L = {w | w = $w^R$} is not a regular language. But it can be described by means of CFG, as illustrated below:

G = ( V, Σ, P, S )
Where:
V = { Q, Z, N }
Σ = { 0, 1 }
P = { Q → Z | Q → N | Q → Σ | Z → 0Q0 | N → 1Q1 }
S = { Q }

- This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc.

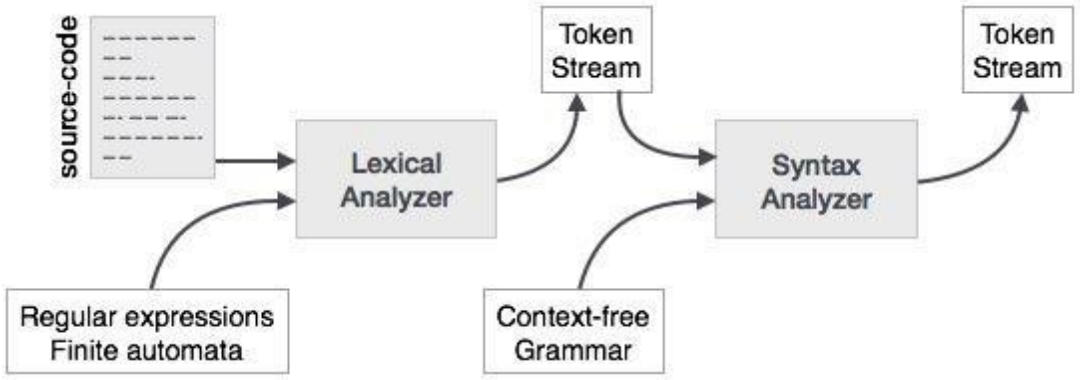
To show how 10101 is generated:

Q → N
1Q1 N → 1Q1
1Z1 Q → Z
10Q01 Z → 0Q0
10 Σ 101 Q → Σ

10101 Σ = { 0, 1 }

**Syntax Analyzers**

- A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams.
- The **parser analyzes the source code (token stream) against the production rules to detect any errors in the code**. The output of this phase is a **parse tree**.




- The parser accomplishes two tasks, i.e., parsing the code, looking for errors, and generating a parse tree as the output of the phase.
- Parsers are expected to parse the whole code even if some errors exist in the program.

**Derivation**

- **A derivation is basically a sequence of production rules, in order to get the input string**.
- During parsing, we take two decisions for some sentential form of input:

i. Deciding the non-terminal which is to be replaced.

ii. Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options:

i. Left-most derivation
ii. Right-most derivation

**Left-most Derivation**

- In the leftmost derivation, the input is scanned and replaced with the production rule from left to right. So in leftmost derivation, we read the input string from left to right.
- If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation.
- The sentential form derived by the left-most derivation is called the left-sentential form.

**Right-most Derivation**

- If we scan and replace the input with production rules, from right to left, it is known as right-most derivation.
- The sentential form derived from the right-most derivation is called the right-sentential form.

**Parse Tree**

- A parse tree is a graphical depiction of a derivation.
- It is convenient to see how strings are derived from the start symbol.
- The start symbol of the derivation becomes the root of the parse tree.

In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.

- A parse tree depicts **associativity** and **precedence** of operators.

- The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes

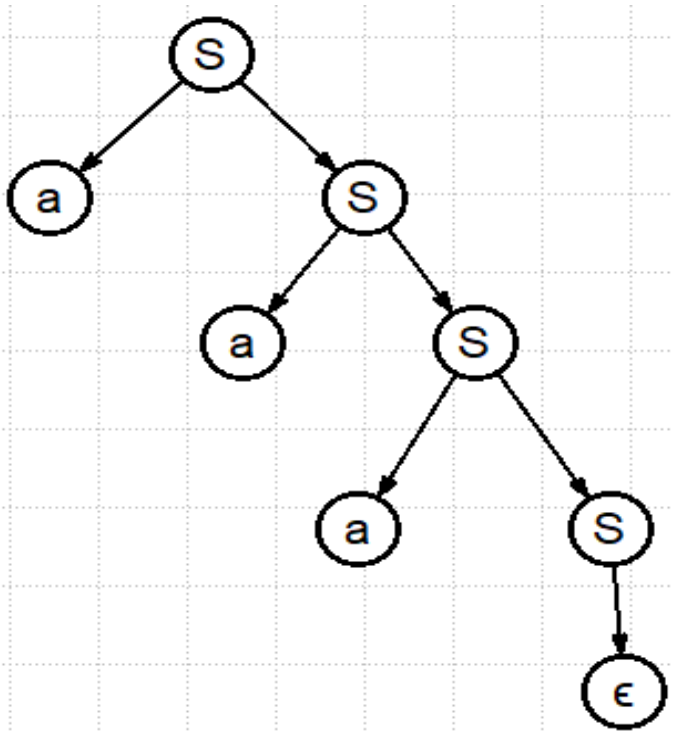
**Example: Leftmost Derivation**

- The process of deriving the input string by expanding the leftmost non-terminal

S -> aS|ϵ
To generate "aaa"

S -> aS
-> aaS (Using S -> aS)
-> aaaS (Using S -> aS)
-> aaaϵ (Using S -> ϵ)
-> aaa

**Leftmost Tree**




**Rightmost Derivation**

- The process of deriving the input string by expanding the right non-terminal

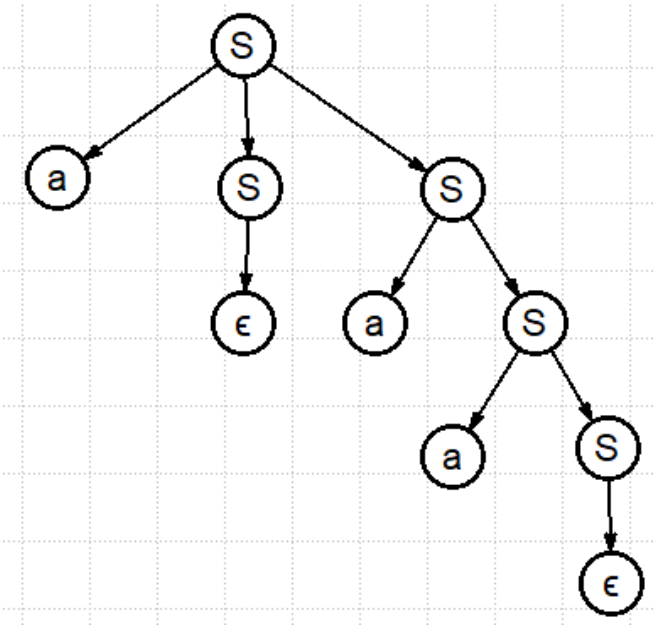
**Example**

S -> aSS|aS|ϵ

To generate “aaa”

S -> aSS
-> aSaS (Using S -> aS)
-> aSaaS (Using S -> aS)
-> aSaaϵ (Using S -> ϵ)
-> aaa (Using S -> ϵ)

**Rightmost Tree**




## References


1. Robert W. Sebesta (2012). Concepts of Programming Languages. Addison-Wesley. Tenth Edition
2. Comparative Programming Languages by Leslie B. Wilson and Robert G. Clark Addison-Wesley (Third Edition).
3. Programming Language Pragmatics by Michael L. Scott (Second Edition).
4. ChatGPT