

## Software

A set of instructions to perform specific tasks is called a program, and the collection of one or many programs for a specific purpose is termed as computer software or, simply, software. These instructions can be on internal command or an external input received from devices such as a mouse or keyboard.

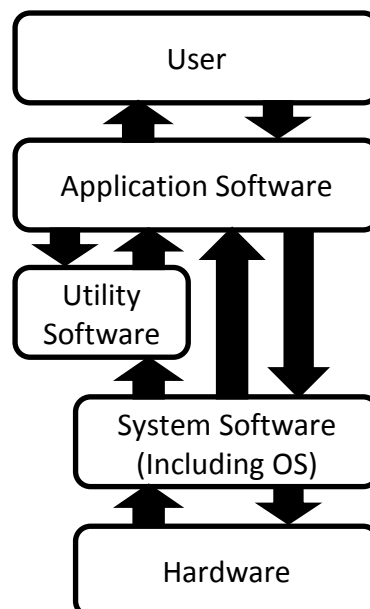
### *Characteristics:*

The software characteristics include performance, portability, and functionality. Developing any software requires the understanding of the following software quality factors:

- **Operational characteristics:** These include characteristics such as correctness, usability/learnability, integrity, reliability, efficiency, security, and safety.
- **Transitional characteristics:** These include interoperability, reusability, and portability.
- **Revision characteristics:** These are characteristics related to 'interior quality' of software such as efficiency, documentation, and structure. Various revision characteristics of software are maintainability, flexibility, extensibility, scalability, testability, and modularity.

## Software Hierarchy

Broadly, computer software includes various computer programs, system libraries and their associated documentation. Based on the nature of the task and goal, computer software can be classified into application software, utility software, and system software.



- **Application software:** Application software is designed to perform special functions other than the basic operations carried out by a computer. All such softwares are application-specific and cannot be directly understood by the underlying hardware of the computer. Application software is concerned with the solution of some problems; it uses a computer as

a tool and enables the end user to perform specific and productive tasks. There are different types of application software based on the range of tasks performed by the computer.

- **Utility software:** This software is designed for users to assist in maintenance and monitoring activities. These include anti-virus software, firewalls, and disk utilities. They help maintain and protect system but do not directly interface with the hardware.
- **System software:** System software can be viewed as software that logically binds components of a computer to work as a single unit and provides the infrastructure over which programs can operate. It is responsible for controlling computer hardware and other resources, and allows the application software to interact with computers to perform their tasks. System software includes operating system, device drivers, language translators, etc. Some specific system software are assemblers, linkers, loaders, macro processors, text editors, compilers, operating system, debugging system, source code control system, etc.

## System Programming

System programming is characterized by the fact that it is aimed at producing system software that provides services to the computer hardware or specialized system services. Many a time, system programming directly deals with the peripheral devices with a focus on input, process (storage), and output.

The **essential characteristics** of system programming are as follows:

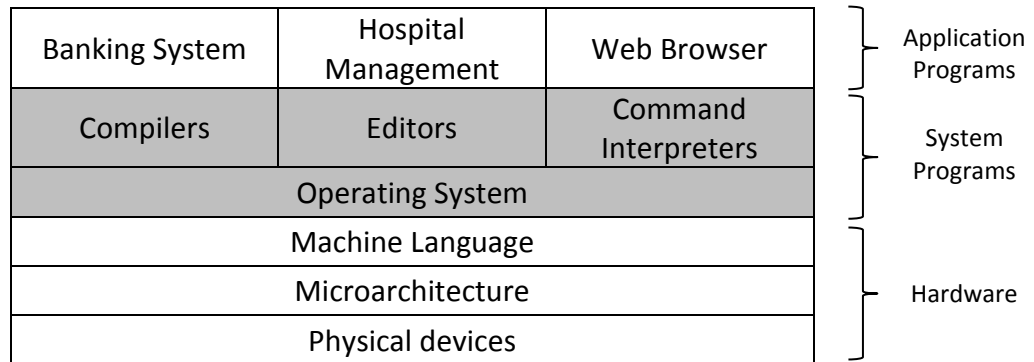
- Programmers are expected to know the hardware and internal behavior of the computer system on which the program will run. System programmers explore these known hardware properties and write software for specific hardware using efficient algorithms.
- Uses a low level programming language or some programming dialect.
- Requires little runtime overheads and can execute in a resource-constrained environment.
- These are very efficient programs with a small or no runtime library requirements.
- Has access to systems resources, including memory
- Can be written in assembly language

The following are the **limiting factors** of system programming:

- Many times, system programs cannot be run in debugging mode.
- Limited programming facility is available, which requires high skills for the system programmer.
- Less powerful runtime library (if available at all), with less error-checking capabilities.

## Machine Structure

A generic computer system comprises hardware components, collection of system programs, and a set of application programs.



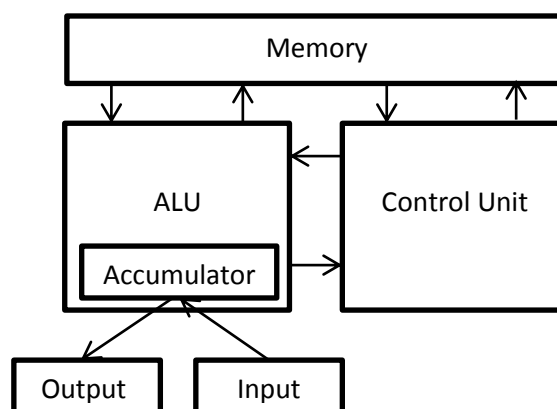
## Types of Computer Architecture

### 1) Von Neumann Architecture

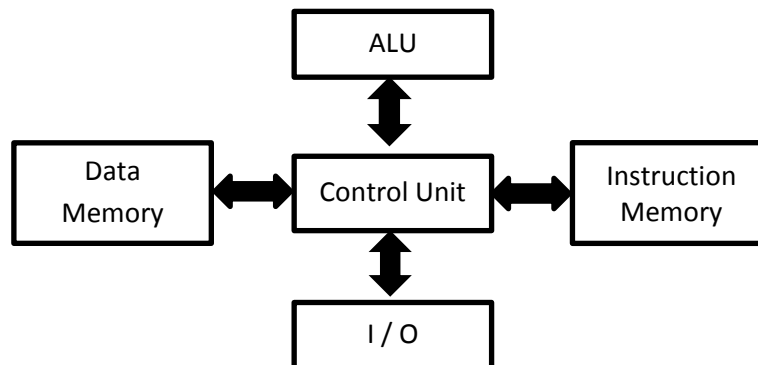
The important parts of this architecture include the following:

- Processing unit: It contains an Arithmetic Logic Unit (ALU) and a set of working registers for the processor.
- Control unit: It encompasses control mechanism to carry out functions and includes instruction register and program counter.
- Memory: It stores data and instructions used for applications and system processing, address mapping for external mass storage, and I/O mechanisms.

The traditional Von Neumann architecture describes stored-program computer, which does not allow fetching of instructions and data operations to occur at the same time. The reason is that the architecture uses a commonly shared bus to access both. This has led to limiting the performance of the system. The structure of a typical Von Neumann machine is shown in Figure:



### 2) Harvard Computer Architecture



The characteristics of Harvard Computer architecture are as follows:

- The Harvard architecture is stored-program computer system, which has separate sets of addresses and data buses to read and write data to the memory and also for fetching instructions.
- Basically, the Harvard architecture has physically distinct storage and signal paths access to data and instructions, respectively.
- In the Harvard architecture, two memories need not share characteristics.
- The structure and width of the word, timing characteristics, mechanism of implementation, and structure of addresses can vary, while program instructions reside in read-only memory, the program data often needs read-write memory.
- The requirements of instruction memory can be larger due to the fact that some systems have much more instruction memory than data memory. Therefore, the width of instruction addresses happens to be wider than that of data addresses.

## ***Von Neumann Architecture v/s Harvard Computer Architecture***

<b>Von Neumann Architecture</b>	<b>Harvard Computer Architecture</b>
Does not allow both fetching of instructions and data operations to occur at the same time.	CPU can both read program instruction and access data from the memory simultaneously.
Uses a common shared bus to access both instructions and data.	Instruction fetches and data access need separate pathways.
Shared bus usage for instruction and data memory results into performance bottleneck.	Faster for a given circuit complexity.
Has limited transfer rate between the CPU and memory.	Uses separate address spaces for code and data.

## **Interfaces**

An interface is defined as a border or an entry point across which distinct components of a digital computing system interchange data and information. There are three types of interfaces - software, hardware, and user interfaces.

### **Types of Interface**

#### ***1) Software Interface***

- Software interface comprises a set of statements, predefined functions, user options, and other methods of conveying instructions and data obtained from a program or

language for programmers.

- Access to resources including CPU, memory and storage, etc., is facilitated by software interfaces for the underlying computer system.
- While programming, the interface between software components makes use of program and language facilities such as constants, various data types, libraries and procedures, specifications for exception, and method handling.
- Operating system provides the interface that allows access to the system resources from applications. This interface is called Application Programming Interface (API). These APIs contain the collection of functions, definitions for type, and constants, and also include some variable definitions. While developing software applications, the APIs can be used to access and implement functionalities.

## 2) **Hardware Interface**

- Hardware interfaces are primarily designed to exchange data and information among various hardware components of the system, including internal and external devices.
- This type of interface is seen between buses, across storage devices and other I/O and peripherals devices.
- A hardware interface provides access to electrical, mechanical, and logical signals and implements signaling protocols for reading and sequencing them.
- These hardware interfaces may be designed to support either parallel or serial data transfer or both. Hardware interfaces with parallel implementations allow more than one connection to carry data simultaneously, while serial allows data to be sent one bit at a time.
- One of the popular standard interfaces is Small Computer System Interface (SCSI) that defines the standards for physically connecting and communicating data between peripherals and computers.

## 3) **User Interface**

User interface allows interaction between a computer and a user by providing various modalities of interaction including graphics, sound, position, movement, etc. These interfaces facilitate transfer of data between the user and the computing system. User interface is very important for all systems that require user inputs.

## **Address Space**

The amount of space allocated for all possible addresses for data and other computational entities is called address space. The address space is governed by the architecture and managed by the operating system. The computational entities such as a device, file, server, or networked computer all are addressed within space.

There are two types of address space namely,

### 1. **Physical Address Space**

Physical address space is the collection of all physical addresses produced by a computer program and provided by the hardware. Every machine has its own physical address space with its valid address range between 0 and some maximum limits supported by the

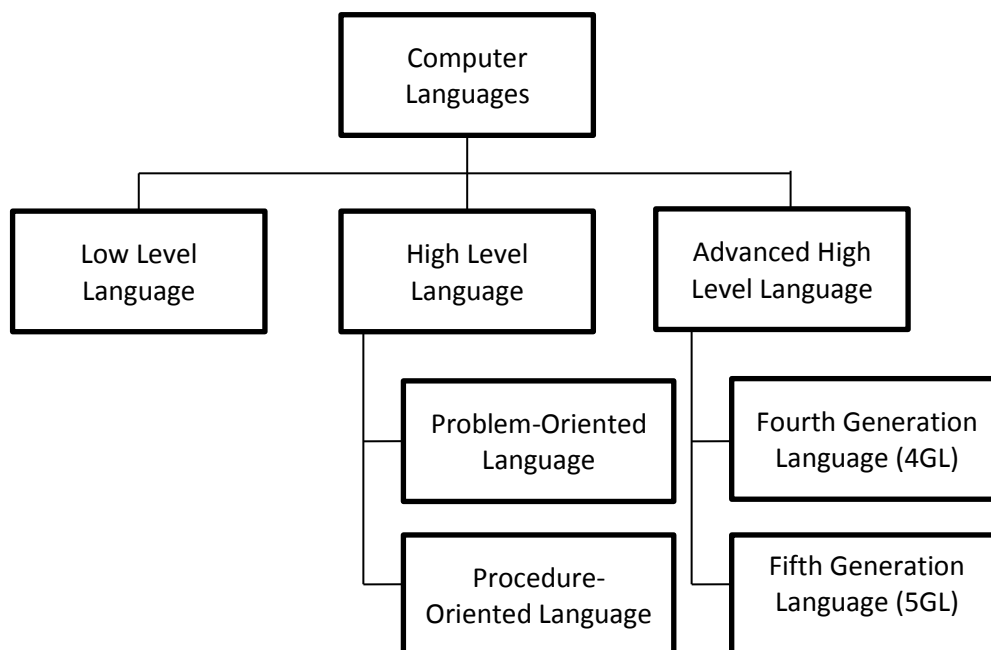
machine.

## 2. Logical Address Space

Logical address space is generated by the CPU or provided by the OS kernel. It is also sometimes called virtual address space. In the virtual address space, there is one address space per process, which may or may not start at zero and extend to the highest address.

## Computer Languages

Computer needs language to communicate across its components and devices and carry out instructions. A computer acts on a specific sequence of instructions written by a programmer for a specific job. This sequence of instructions is known as a program.



## Classification of Computer Languages

### Low Level Languages

Characteristically, low-level languages represent languages that can be directly understood or are very close to machines. They can be further classified into machine language and assembly language.

#### 1) Machine Language

Machine language is truly a computer-dependent programming language, which is written by using binary codes. It is also known as machine code. Machine code encompasses a function (Opcode) and an operand (Address) part. Since a computer understands machine codes, programs written by using machine language can be executed immediately without the requirement of any language translators. The disadvantages of machine language are as follows:

- Programs based on machine language are difficult to understand as well as develop.
- A pure machine-oriented language is difficult to remember and recall: All the instructions and data to the computer are fed in numerical form.
- Knowledge of computer internal architecture and codes is must for programming.

- Writing code using machine language is time consuming, cumbersome, and complicated.
- Debugging of programs is difficult.

## 2) **Assembly Language**

It is a kind of low-level programming language, which uses symbolic codes or mnemonics as instruction. Some examples of mnemonics include ADD, SUB, LDA, and STA that stand for addition, subtraction, load accumulator, and store accumulator, respectively. The processing of an assembly language program is done by using a language translator called assembler that translates assembly language code into machine code. Assembly language program needs to be translated into equivalent machine language code (binary code) before execution. The **advantages** of assembly language are as follows:

- Due to use of symbolic codes (mnemonics), an assembly program can be written faster.
- It makes the programmer free from the burden of remembering the operation codes and addresses of memory location.
- It is easier to debug.

The **disadvantages** of assembly language are as follows:

- As it is a machine-oriented language, it requires familiarity with machine architecture and understanding of available instruction set.
- Execution in an assembly language program is comparatively time consuming compared to machine language. The reason is that a separate language translator program is needed to translate assembly program into binary machine code.

## High Level Languages

- High level languages (HLL) were developed to overcome large time consumption and cost in developing machine and assembly languages.
- HLLs are much closer to English-like language.
- A separate language translator is required to translate HLL computer programs into machine readable object code.
- Some of the distinguished features of HLL include interactive, support of a variety of data types, a rich set of operators, flexible control structures, readability, modularity, file handling, and memory management.

The advantages of high level languages are as follows:

- It is machine-independent.
- It can be used both as problem- and procedure-oriented.
- It follows simple English-like structure for program coding.
- It does not necessitate extensive knowledge of computer architectures.
- Writing code using HLL consumes less time.
- Debugging and program maintenance are easier in HLL.

The disadvantage of high level languages is as follows:

- It requires language translators for converting instructions from high level language into low level object code.

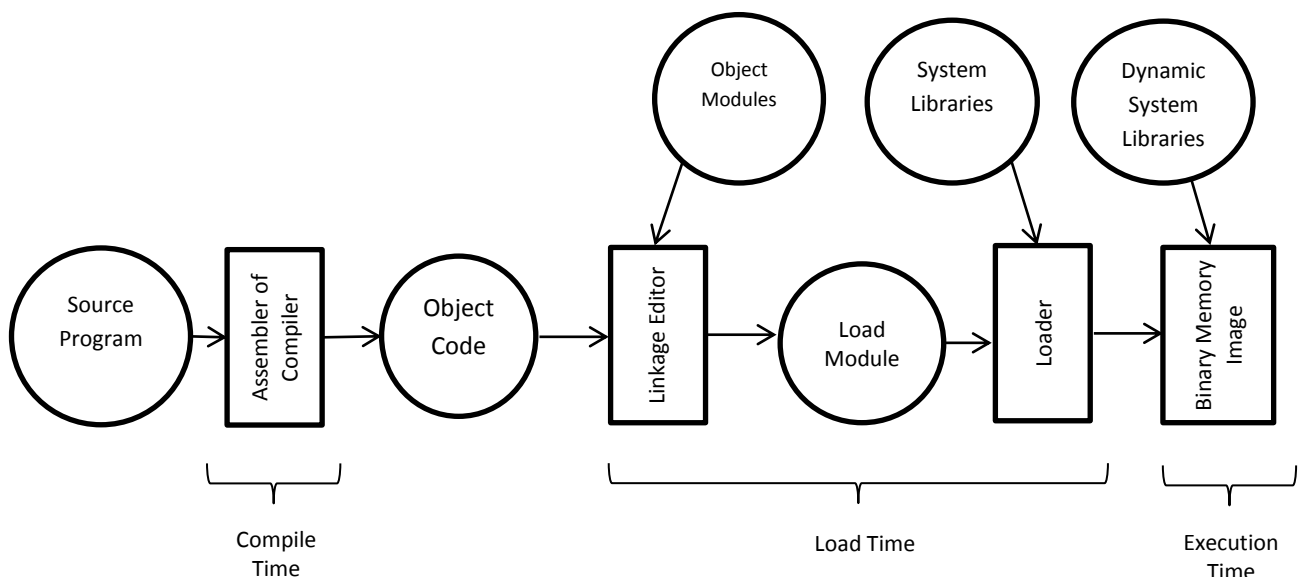
## **Life Cycle of a Source Program**

The life cycle of a source program defines the program behavior and extends through execution

stage, which exhibits the behavior specified in the program.

Every source program goes through a life cycle of several stages.

- **Edit time:** It is the phase where editing of the program code takes place and is also known as design time. At this stage, the code is in its raw form and may not be in a consistent state.
- **Compile time:** At the compile time stage, the source code after editing is passed to a translator that translates it into machine code. One such translator is a compiler. This stage checks the program for inconsistencies and errors and produces an executable file.
- **Distribution time:** It is the stage that sends or distributes the program from the entity creating it to an entity invoking it. Mostly executable files are distributed.
- **Installation time:** Typically, a program goes through the installation process, which makes it ready for execution within the system. The installation can also optionally generate calls to other stages of a program's life cycle.
- **Link time:** At this stage, the specific implementation of the interface is linked and associated to the program invoking it. System libraries are linked by using the lookup of the name and the interface of the library needed during compile time or throughout the installation time, or invoked with the start or even during the execution process.
- **Load time:** This stage actively takes the executable image from its stored repositories and places them into active memory to initiate the execution. Load time activities are influenced by the underlying operating system.
- **Run time:** This is the final stage of the life cycle in which the programmed behavior of the source program is demonstrated.



## System Software Development

Software development process follows the Software Development Life Cycle (SDLC), which has each step doing a specific activity till the final software is built. The system software development process also follows all the stages of SDLC, which are as follows:

- **Preliminary investigation:** It determines what problems need to be fixed by the system



software being developed and what would be the better way of solving those problems.

- **System analysis:** It investigates the problem on a large scale and gathers all the information. It identifies the execution environment and interfaces required by the software to be built.
- **System design:** This is concerned with designing the blueprint of system software that specifies how the system software looks like and how it will perform.
- **System tool acquisition:** It decides and works around the software tools to develop the functionalities of the system software.
- **Implementation:** It builds the software using the software tools with all the functionality, interfaces, and support for the execution. This may be very specific as the system software adheres to the architecture. Operating system support is sought for allocations and other related matters.
- **System maintenance:** Once the system software is ready, it is installed and used. The maintenance includes timely updating software what is already installed.

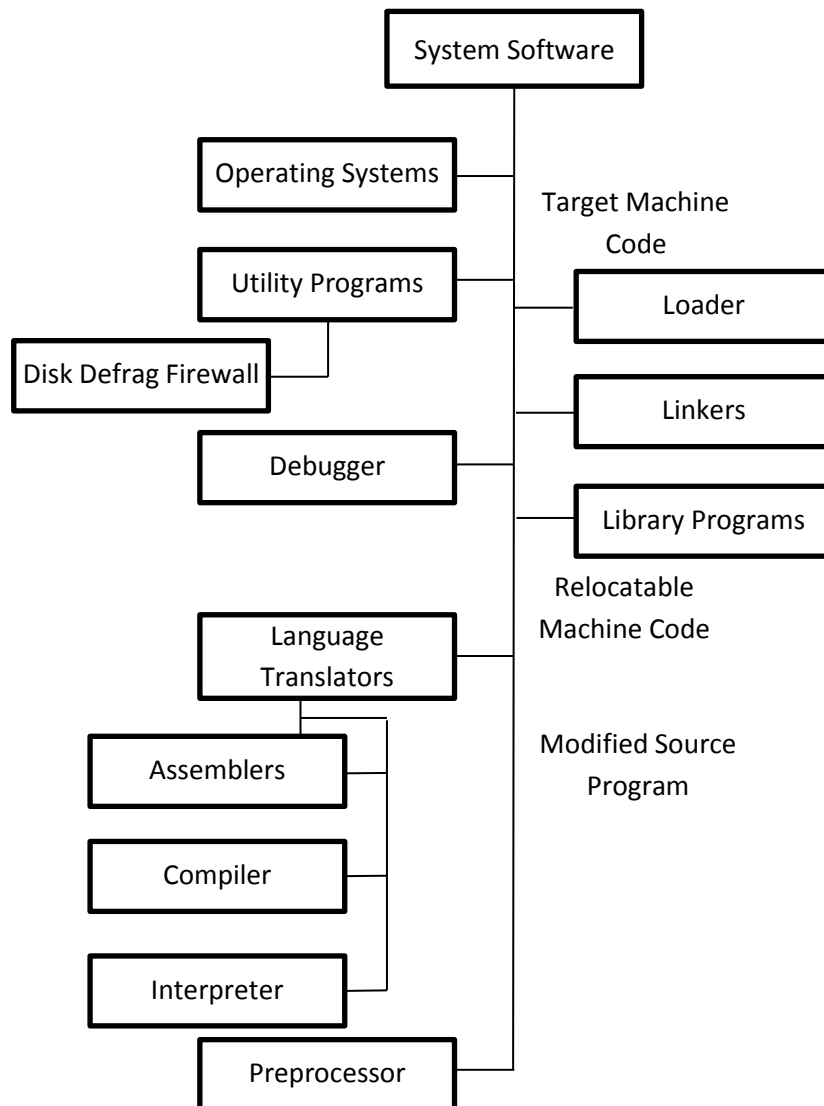
## Recent trends in Software Development

Latest trends in program development from the coding perspective include the following:

- Use of preprocessors against full software stack
- JavaScript MV\* frameworks rather than Java Script files
- CSS frameworks against generic cascading style sheets
- SVG with JavaScript on Canvas in competition with Flash
- Gaming frameworks against native game development
- Single-page Web apps against websites
- Mobile Web apps against native apps
- Android against iOS
- Moving towards GPU from CPU
- Renting against buying (Cloud Services)
- Web interfaces but not IDEs
- Agile development

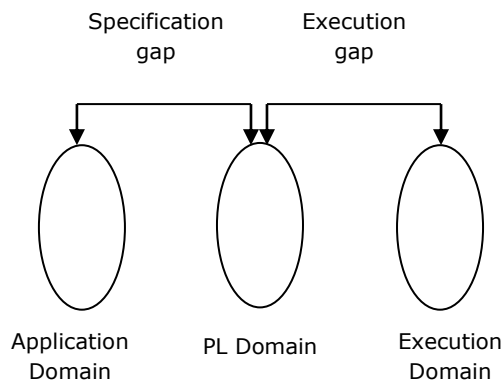
## Levels of System Software

Figure describes various levels of system software used with modern computer systems. A source program submitted by the programmer is processed during its life cycle.



### Explain following terms.

- **Semantic:** It represents the rules of the meaning of the domain.
- **Semantic gap:** It represents the difference between the semantic of two domains.
- **Application domain:** The designer expresses the ideas in terms related to application domain of the software.
- **Execution domain:** To implement the ideas of designer, their description has to be interpreted in terms related to the execution domain of computer system.
- **Specification gap:** The gap between application and PL domain is called specification and design gap or simply specification gap. Specification gap is the semantic gap between two specifications of the same task.
- **Execution gap:** The gap between the semantic of programs written in different programming language.



- **Language processor:** Language processor is software which bridges a specification or execution gap.
- **Language translator:** Language translator bridges an execution gap to the machine language of a computer system.
- **Detranslator:** It bridges the same execution gap as language translator, but in the reverse direction.
- **Preprocessor:** It is a language processor which bridges an execution gap but is not a language translator.
- **Language migrator:** It bridges the specification gap between two programming languages.
- **Interpreter:** An interpreter is a language processor which bridges an execution gap without generating a machine language program.
- **Source language:** The program which forms the input to a language processor is a source program. The language in which the source program is written is known source language.
- **Target language:** The output of a language processor is known as the target program. The language, to which the target program belongs to, is called target language.
- **Problem oriented language:** Programming language features directly model the aspects of the application domain, which leads to very small specification gap. Such a programming language can only be used for specific application; hence they are called problem oriented languages.

- **Procedure oriented language:** Procedure oriented language provides general purpose facilities required in most application domains. Such a language is independent of specific application domains and results in a large specification gap which has to be bridged by an application designer.
- **Forward Reference:** A forward reference of a program entity is a reference to the entity in some statement of the program that occurs before the statement containing the definition or declaration of the entity.
- **Language processor pass:** A Language processor pass is the processing of every statement in a source program, or in its equivalent representation, to perform a language processing function (a set of language processing functions).
- **Intermediate representation (IR):** An intermediate representation is a representation of a source program which reflects the effect of some, but not all analysis and synthesis functions performed during language processing.

An intermediate representation should have the following three properties:

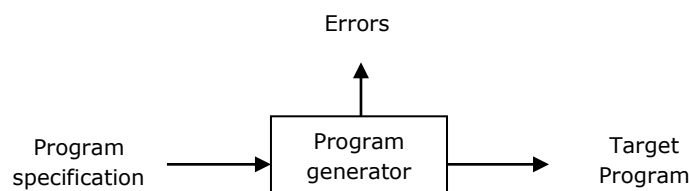
1. **Ease of use:** It should be easy to construct the intermediate representation and analyze it.
2. **Processing efficiency:** Efficient algorithms should be available for accessing the data structures used in the intermediate representation.
3. **Memory efficiency:** The intermediate representation should be compact so that it does not occupy much memory.

### Language processing activity

There are mainly two types of language processing activity which bridges the semantic gap between source language and target language.

#### 1. Program generation activities

- A program generation activity aims an automatic generation of a program.
- Program generator is software, which aspects source program and generates a program in target language.
- Program generator introduces a new domain between the application and programming language domain is called program generator domain.



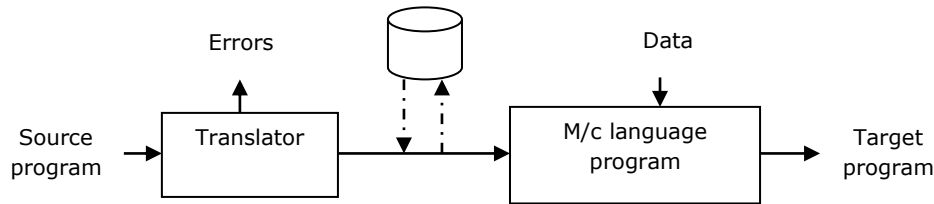
#### 2. Program Execution

Two popular models for program execution are translation and interpretation.

##### Translation

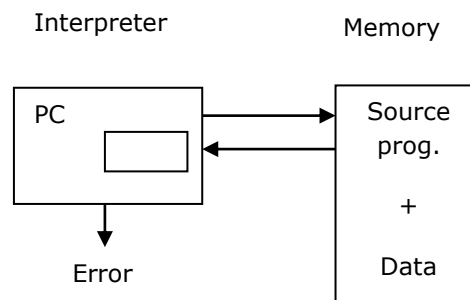
- ✓ The program translation model bridges the execution gap by translating a program written in PL, called source program, into an equivalent program in machine or assembly

language of the computer system, called target program.



### Interpretation

- The interpreter reads the source program and stores it in its memory.
- The CPU uses the program counter (PC) to note the address of the next instruction to be executed.
- The statement would be subjected to the interpretation cycle, which could consist the following steps:
  1. Fetch the instruction
  2. Analyze the statement and determine its meaning, the computation to be performed and its operand.
  3. Execute the meaning of the statement.

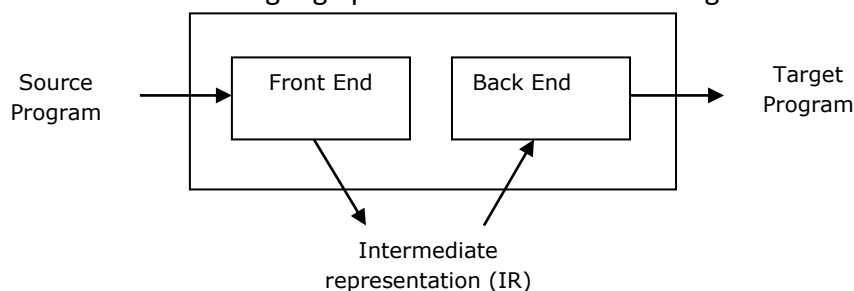


## Phases and passes of compiler (Toy Compiler)

**Language processor pass:** A language processor pass is the processing of every statement in a source program, to perform language processing function.

- Pass I: Perform analysis of the source program and note deduced information.
- Pass II: Perform synthesis of target program.

The classic two-pass schematic of language processor is shown in the figure.



- The first pass performs analysis of the source program and reflects its results in the intermediate representation.
- The second pass reads and analyzes the intermediate representation to perform synthesis

of the target program.

### Phases of Language processor (Toy compiler)

#### 1. Lexical Analysis (Scanning)

- Lexical analysis identifies the lexical unit in a source statement. Then it classifies the units into different lexical classes. E.g. id's, constants, keyword etc...And enters then into different tables.
- The most important table is symbol table which contains information concerning all identifiers used in the SP.
- The symbol table is built during lexical analysis.
- Lexical analysis builds a descriptor, called a token. We represent token as 

Code #no
----------

 where Code can be Id or Op for identifier or operator respectively and no indicates the entry for the identifier or operator in symbol or operator table.
- Consider following code

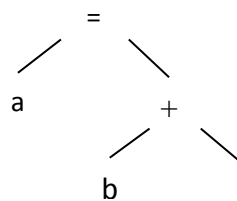
```
i: integer;
a, b: real;
a = b + i;
```

- The statement a=b+i is represented as a string of token

a	=	b	+	i
Id#1	Op#1	Id#2	Op#2	Id#3

#### 2. Syntax analysis (parsing)

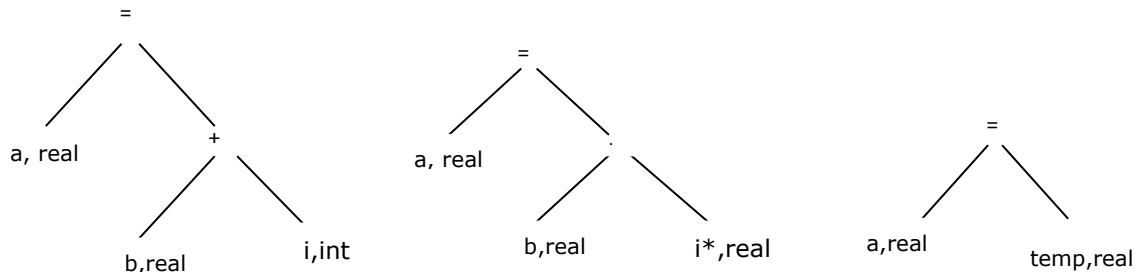
- Syntax analysis processes the string of token to determine its grammatical structure and builds an intermediate code that represents the structure.
- The tree structure is used to represent the intermediate code.
- Consider the statement a = b + i can be represented in tree form as



#### 3. Semantic Analysis

- Semantic analysis determines the meaning of a statement by applying the semantic rules to the structure of the statement.
- While processing a declaration statement, it adds information concerning the type, length and dimensionality of a symbol to the symbol table.
- While processing an imperative statement, it determines the sequence of actions that would have to be performed for implementing the meaning of the statement and represents them in the intermediate code.
- Considering the tree structure for the statement a = b + i

- If node is operand, then type of operand is added in the description field of operand.
- While evaluating the expression the type of b is real and i is int so type of i is converted to real i\*.



- The analysis ends when the tree has been completely processed.

### Intermediate representation

- IR contains intermediate code and table.
- Symbol table

	symbol	Type	length	address
1	i	int		
2	a	real		
3	b	real		
4	i*	real		
5	temp	real		

- Intermediate code
  1. Convert(id1#1) to real, giving (id#4)
  2. Add(id#4) to (id#3), giving (id#5)
  3. Store (id#5) in (id#2)

### Memory allocation

- The memory requirement of an identifier is computed from its type, length and dimensionality and memory is allocated to it.
- The address of the memory area is entered in the symbol table

	Symbol	Type	length	address
1	i	int		2000
2	a	real		2001
3	b	real		2002

### Code generation

- The synthesis phase may decide to hold the value of i\* and temp in machine registers and may generate the assembly code
 

```

CONV_R  AREG, I
ADD_R   AREG, B
      
```

MOVEM AREG, A

### Symbol tables

- An identifier used in the source program is called a symbol. Thus, names of variables, functions and procedures are symbols.
- A language processor uses the symbol table to maintain the information about attributes of symbols used in a source program.
- It performs the following four kinds of operations on the symbol table:
  1. Add a symbol and its attributes: Make a new entry in the symbol table.
  2. Locate a symbol's entry: Find a symbol's entry in the symbol table.
  3. Delete a symbol's entry: Remove the symbol's information from the table.
  4. Access a symbol's entry: Access the entry and set, modify or copy its attribute information.
- The symbol table consists of a set entries organized in memory.
- Two kinds of data structures can be used for organizing its entries:
  - **Linear data structure:** Entries in the symbol table occupy adjoining areas of memory. This property is used to facilitate search.
  - **Nonlinear data structure:** Entries in the symbol table do not occupy contiguous areas of memory. The entries are searched and accessed using pointers.

### Symbol table entry formats

- Each entry in the symbol table is comprised of fields that accommodate the attributes of one symbol. The symbol field of fields stores the symbol to which entry pertains.
- The symbol field is key field which forms the basis for a search in the table.
- The following entry formats can be used for accommodating the attributes:
  - **Fixed length entries:** Each entry in the symbol table has fields for all attributes specified in the programming language.
  - **Variable-length entries:** the entry occupied by a symbol has fields only for the attributes specified for symbols of its class.
  - **Hybrid entries:** A hybrid entry has fixed-length part and a variable-length part.

### Search Data Structures

Search data structures (Search structure) is used to create and organize various tables of information and mainly used during the analysis of the program.

### Features of Search data structures

The important features of search data structures include the following:

- An entry in search data structure is essentially a set of fields referred to as a record.
- Every entry in search structure contains two parts: fixed and variable. The value in fixed part determines the information to be stored in the variable part of the entry.



### Operations on Search Structures

Search structures are characterized by following operations:

- Insert Operation: To add the entry of a newly found symbol during language processing.
- Search Operation: To enable and support search and locate activity for the entry of symbol.
- Delete Operation: To delete the entry of a symbol especially when identified by language processor as redundant declarations.

### Sequential Search Organization

- In sequential search organization, during the search for a symbol, probability of all active entries being accessed in the table is same.
- For an unsuccessful search, the symbol can be entered using an 'add' operation into the table.

### Binary Search Organization

- Tables using binary search organization have their entries assumed to satisfy an ordering relation.
- It should be considered that for table containing 'f' occupied entries, the probability of successful search is  $\log_2 f$  and unsuccessful search is  $\log_2 f$ .
- The binary search organization requires that entry number of a symbol table should not change after 'add' operation. This may become limiting factor for addition and deletion during language processing.

### Hash Table Organization

- A hash table, also known as a hash map is a data structure that has the ability to map keys to the values using a hash function.
- Hash table organization is an efficient m implementing associative arrays and symbol tables that outperform other data structures with its capability of performing 'm' accesses on 'n' names.
- It has the following two parts:
  - A hash table that contains a fixed array of 'm' pointers to storage table entries.
  - Storage table entries organized into separate linked lists called buckets.
- Hash function is used for the mapping of a key value and the slot where that value belongs to the hash table.
- The hash function takes any key value from the collection and computes an integer value from it in the range of slot names, between 0 and m - 1.

### Linked List and Tree Structure Organizations

#### Linear List

- Linear list organization is the simplest and easiest way to implement the symbol tables.
- It can be constructed using single array or equivalently several arrays that store names and their associated information.
- During insertion of a new name, we must scan the list to ensure whether it is a new entry or not.
- If an entry is found during the scan, it may update the associated information but no new entries are made.

- If the symbol table has 'n' names, the insertion of new name will take effort proportional to 'n' and to insert 'n' names with 'm' information, the total effort is 'cn(n+m)', where 'c' is a constant representing the time necessary for a few machine operations.
- The advantage of using list is that it takes minimum possible space. On the other hand, it may suffer for performance for larger values of 'n' and 'm'.

### **Self-Organizing List**

- Searching in symbol table takes most of the time during symbol table management process.
- The pointer field called 'LINK' is added to each record, and the search is controlled by the order indicated by the 'LINK'.
- A pointer called 'FIRST' can be used to designate the position of the first record on the linked list, and each 'LINK' field indicates the next record on the list.
- Self-organizing list is advantageous over simple list implementation in the sense that frequently referenced name variables will likely to be at the top of the list.
- If the access is random, the self-organizing list will cost time and space.

### **Search Trees**

- Symbol tables can also be organized as binary tree organization with two pointer fields, namely, 'LEFT' and 'RIGHT' in each record that points to the left and right subtrees respectively.
- The left subtree of the record contains only records with names less than the current records name. The right subtree of the node will contain only records with name variables greater than the current name.
- The advantage of using search tree organization is that it proves efficient in searching operations, which are the most performed operations over the symbol tables.
- A binary search tree gives both performance compared to list organization at some difficulty in implementation.

## **Allocation Data Structures**

Allocation strategy is an important factor in efficient utilization of memory for objects, defining their scope and lives using either static, stack, or heap allocations.

### **Stack Allocation**

- Stack is a linear data structure that satisfies last-in, first-out (LIFO) policy for its allocation and deallocation.
- This makes only last element of the stack accessible at any time.
- Implementing stack data structure requires use of Stack Base (SB) that points to first entry of stack, and a Top of Stack (TOS) pointer to point to last entry allocated to stack.

### **Stack Allocation for Activation Records**

- The stack allocation is based on the principles of control stack in which entries are Activation Records (ARs) of some procedure.
- ARs are pushed and popped on each call (activations) and return (the end of procedure), respectively.
- On each activation request, the memory is allocated on the TOS and pointer is incremented

by the size of allocation.

- On execution of return from the procedure, AR is deallocated and TOS is decremented by the same size.
- Figure shows the AR on stack during procedure call.

PARAMETERS & RETURN VAL	Caller's Activation Record
Control Link	
Temporaries & local data	
PARAMETERS & RETURN VAL	Callee's Activation Record
Control Link	
Temporaries & local data	

### Heap Allocation

- Heaps are a kind of non-linear data structure that permits allocation and deallocation of entities in any (random) order as needed.
- Heap data structure returns a pointer to allocated and deallocated area in heap for an allocation request.
- Hence, an allocated entity must maintain a pointer to the memory area allocated to it.
- Space allocation strategy using heaps is optimized for performance by maintaining list of free areas and implementing policies such as first-fit and best-fit for new object allocation.

### Elements of Assembly Language Programming

An assembly language provides the following three basic facilities that simplify programming:

1. **Mnemonic operation codes:** The mnemonic operation codes for machine instructions (also called mnemonic opcodes) are easier to remember and use than numeric operation codes. Their use also enables the assembler to detect use of invalid operation codes in a program.
2. **Symbolic operands:** A programmer can associate symbolic names with data or instructions and use these symbolic names as operands in assembly statements. This facility frees the programmer from having to think of numeric addresses in a program. We use the term symbolic name only in formal contexts; elsewhere we simply say name.
3. **Data declarations:** Data can be declared in a variety of notations, including the decimal notation. It avoids the need to manually specify constants in representations that a computer can understand, for example, specify -5 as  $(11111011)_2$  in the two's complement representation.

### Statement format

An assembly language statement has the following format:

[Label] <Opcode> <operand specification>[,<operand specification>..]

where the notation [..] indicates that the enclosed specification is optional. If a label is specified in a statement, it is associated as a symbolic name with the memory word generated for the statement. If more than one memory word is generated for a statement, the label would be associated with the first of these memory words.

<operand specification> has the following syntax:

<symbolic name> [ $\pm$  <displacement> ] [( <index register> )]

Thus, some possible operand forms are as follows:

- The operand AREA refers to the memory word with which the name AREA is associated.
- The operand AREA+5 refers to the memory word that is 5 words away from the word with the name AREA. Here '5' is the displacement or offset from AREA.
- The operand AREA(4) implies indexing the operand AREA with index register 4—that is, the operand address is obtained by adding the contents of index register 4 to the address of AREA.
- The operand AREA+5 (4) is a combination of the previous two specifications.

### Types of Assembly Statements

#### 1. Imperative statement

- An imperative statement indicates an action to be performed during the execution of the assembled statement.
- Each imperative statement typically translates into one machine instruction.
- These are executable statements.
- Some example of imperative statement are given below  
MOVER BREG,X  
STOP

READ X  
PRINT Y  
ADD AREG,Z

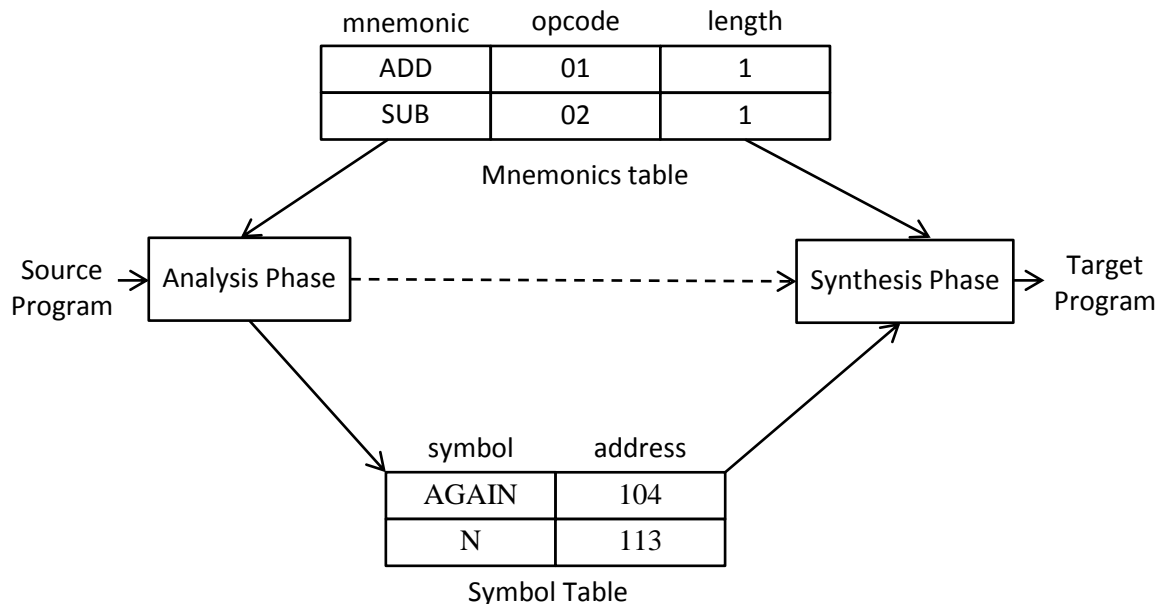
### 2. Declaration statement

- Declaration statements are for reserving memory for variables.
- The syntax of declaration statement is as follow:  
[Label] DS <constant>  
[Label] DC '<value>'  
DS: stands for Declare storage, DC: stands for Declare constant.
- The DS statement reserves area of memory and associates name with them.  
A DS 10  
Above statement reserves 10 word of memory for variable A.
- The DC statement constructs memory words containing constants.  
ONE DC '1'  
Above statement associates the name ONE with a memory word containing the value '1'
- Any assembly program can use constant in two ways- as immediate operands, and as literals.
- Many machine support immediate operands in machine instruction. Ex: ADD AREG, 5
- But hypothetical machine does not support immediate operands as a part of the machine instruction. It can still handle literals.
- A literal is an operand with the syntax='<value>'. EX: ADD AREG,='5'
- It differs from constant because its location cannot be specified in assembly program.

### 3. Assembler Directive

- Assembler directives instruct the assembler to perform certain action during the assembly program.
  - a. **START**
    - This directive indicates that first word of machine should be placed in the memory word with address <constant>.
    - START <Constant>
    - Ex: START 500
    - First word of the target program is stored from memory location 500 onwards.
  - b. **END**
    - This directive indicates end of the source program.
    - The operand indicates address of the instruction where the execution of program should begin.
    - By default it is first instruction of the program.
    - END <operand 2>
    - Execution control should transfer to label given in operand field.

**Assembler Design Criteria OR Design of Assembler OR Assembly scheme OR Analysis and synthesis phases of an assembler by clearly stating their tasks.**



**Figure 3.1: Design of Assembler**

## Analysis Phase

- The primary function performed by the analysis phase is the building of the symbol table.
- For this purpose it must determine address of the symbolic name.
- It is possible to determine some address directly, however others must be inferred. And this function is called memory allocation.
- To implement memory allocation a data structure called location counter (LC) is used, it is initialized to the constant specified in the START statement.
- We refer the processing involved in maintaining the location counter as LC processing.
- **Tasks of Analysis phase**
  1. Isolate the label, mnemonics opcode, and operand fields of a constant.
  2. If a label is present, enter the pair (symbol, <LC content>) in a new entry of symbol table.
  3. Check validity of mnemonics opcode.
  4. Perform LC processing.

## Synthesis Phase

- Consider the assembly statement,  

```
MOVER BREG, ONE
```
- We must have following information to synthesize the machine instruction corresponding to this statement:

1. Address of name ONE
  2. Machine operation code corresponding to mnemonics MOVER.
- The first item of information depends on the source program; hence it must be available by analysis phase.
  - The second item of information does not depend on the source program; it depends on the assembly language.
  - Based on above discussion, we consider the use of two data structure during synthesis phase:
    1. Symbol table:  
Each entry in symbol table has two primary field- name and address. This table is built by analysis phase
    2. Mnemonics table:  
An entry in mnemonics table has two primary field- mnemonics and opcode.
  - **Tasks of Synthesis phase**
    1. Obtain machine opcode through look up in the mnemonics table.
    2. Obtain address of memory operand from the symbol table.
    3. Synthesize a machine instruction.

**Types of Assembler OR Pass structure of assembler. OR Explain single pass and two pass assembler. OR Write difference between one pass and two pass assembler.**

### **Two pass translation**

- Two pass translations consist of pass I and pass II.
- LC processing is performed in the first pass and symbols defined in the program are entered into the symbol table, hence first pass performs analysis of the source program.
- So, two pass translation of assembly lang. program can handle forward reference easily.
- The second pass synthesizes the target form using the address information found in the symbol table.
- First pass constructs an intermediate representation of the source program and that will be used by second pass.
- IR consists of two main components: data structure + IC (intermediate code)

### **Single pass translation**

- A one pass assembler requires 1 scan of the source program to generate machine code.
- The process of forward references is talked using a process called back patching.
- The operand field of an instruction containing forward references is left blank initially.
- A table of instruction containing forward references is maintained separately called table of incomplete instruction (TII).
- This table can be used to fill-up the addresses in incomplete instruction.
- The address of the forward referenced symbols is put in the blank field with the help of back patching list.

**Assembling of forward references in a single pass assembler OR**

### How forward references can be solved using back-patching? Explain with example.

The assembler implements the backpatching technique as follows:

- It builds a table of incomplete instructions (TII) to record information about instructions whose operand fields were left blank.
- Each entry in this table contains a pair of the form (instruction address, symbol) to indicate that the address of symbol should be put in the operand field of the instruction with the address instruction address.
- By the time the END statement is processed, the symbol table would contain the addresses of all symbols defined in the source program and TII would contain information describing all forward references.
- The assembler can now process each entry in TII to complete the concerned instruction.
- Alternatively, entries in TII can be processed on the fly during normal processing.
- In this approach, all forward references to a symbol  $\text{symb}_i$  would be processed when the statement that defines symbol  $\text{symb}_i$  is encountered.
- The instruction corresponding to the statement  
MOVER BREG, ONE  
contains a forward reference to ONE.
- Hence the assembler leaves the second operand field blank in the instruction that is assembled to reside in location 101 of memory, and makes an entry (101, ONE) in the table of incomplete instructions (TII).
- While processing the statement  
ONE DC '1'  
address of ONE, which is 115, is entered in the symbol table.
- After the END statement is processed, the entry (101, ONE) would be processed by obtaining the address of ONE from the symbol table and inserting it in the second operand field of the instruction with assembled address 101.

### Advanced Assembler Directives

#### 1. ORIGIN

- The syntax of this directive is

*ORIGIN <address specification>*

where *<address specification>* is an *<operand specification>* or *<constant>*.

- This directive instructs the assembler to put the address given by *<address specification>* in the location counter.
- The ORIGIN statement is useful when the target program does not consist of a single contiguous area of memory.
- The ability to use an *<operand specification>* in the ORIGIN statement provides the ability to change the address in the location counter in a relative rather than absolute manner.

#### 2. EQU

- The EQU directive has the syntax

*<symbol> EQU <address specification>*



where *<address specification>* is either a *<constant>* or *<symbolic name> ± <displacement>*.

- The EQU statement simply associates the name *<symbol>* with the address specified by *<address specification>*. However, the address in the location counter is not affected.

### 3. LTORG

- The LTORG directive, which stands for 'origin for literals', allows a programmer to specify where literals should be placed.
- The assembler uses the following scheme for placement of literals: When the use of a literal is seen in a statement, the assembler enters it into a literal pool unless a matching literal already exists in the pool.
- At every LTORG statement, as also at the END statement, the assembler allocates memory to the literals of the literal pool and clears the literal pool.
- This way, a literal pool would contain all literals used in the program since the start of the program or since the previous LTORG statement.
- Thus, all references to literals are forward references by definition.
- If a program does not use an LTORG statement, the assembler would enter all literals used in the program into a single pool and allocate memory to them when it encounters the END statement.

Consider the following assembly program to understand ORIGIN, EQU and LTORG

1		START	200		
2		MOVER	AREG, ='5'	200)	+04 1 211
3		MOVEM	AREG, A	201)	+05 1 217
4	LOOP	MOVER	AREG, A	202)	+04 1 217
5		MOVER	CREG, B	203)	+04 3 218
6		ADD	CREG, ='1'	204)	+01 3 212
7		...			
12		BC	ANY, NEXT	210)	+07 6 214
13		LTORG			
			= '5'	211)	+00 0 005
			= '1'	212)	+00 0 001
14		...			
15	NEXT	SUB	AREG, ='1'	214)	+02 1 219
16		BC	LT, BACK	215)	+07 1 202
17	LAST	STOP		216)	+00 0 000
18		ORIGIN	LOOP + 2		
19		MULT	CREG, B	204)	+03 3 218
20		ORIGIN	LAST + 1		

```

21  A      DS      1      217)
22  BACK   EQU     LOOP
23  B      DS      1      218)
24                END
25                = '1'    219)  +00 0 001

```

### ORIGIN

Statement number 18 of the above program viz. ORIGIN LOOP + 2 puts the address 204 in the location counter because symbol LOOP is associated with the address 202. The next statement MULT CREG, B is therefore given the address 204.

### EQU

On encountering the statement BACK EQU LOOP, the assembler associates the symbol BACK with the address of LOOP i.e. with 202.

### LTORG

In assembly program, the literals = '5' and = '1' are added to the literal pool in Statements 2 and 6, respectively. The first LORG statement (Statement 13) allocates the addresses 211 and 212 to the values '5' and '1'. A new literal pool is now started. The value T is put into this pool in Statement 15. This value is allocated the address 219 while processing the END statement. The literal = '1' used in Statement 15 therefore refers to location 219 of the second pool of literals rather than location 212 of the first pool.

## Design of Two-pass Assembler

**Data structures of assembler pass I** OR **Explain the role of mnemonic opcode table, symbol table, literal table, and pool table in assembling process of assembly language program.** OR **Describe following data structures: OPTAB, SYMTAB, LITAB & POOLTAB.**

### OPTAB

- A table of mnemonics opcode and related information
- OPTAB contains the field mnemonics opcodes, class and mnemonics info.
- The class field indicates whether the opcode belongs to an imperative statement (IS), a declaration statement (DS), or an assembler directive (AD).
- If an imperative, the mnemonics info field contains the pair (machine code, instruction length), else it contains the id of a routine to handle the declaration or directive statement.

Mnemonics opcode	Class	Mnemonics info
MOVER	IS	(04,1)
DS	DL	R#7

START	AD	R#11
-------	----	------

### SYMTAB

- A SYMTAB entry contains the symbol name, field address and length.
- Some address can be determining directly, e.g. the address of the first instruction in the program, however other must be inferred.
- To find address of other we must fix the addresses of all program elements preceding it. This function is called memory allocation.

Symbol	Address	Length
LOOP	202	1
NEXT	214	1
LAST	216	1
A	217	1
BACK	202	1
B	218	1

### LITTAB

- A table of literals used in the program.
- A LITTAB entry contains the field literal and address.
- The first pass uses LITTAB to collect all literals used in a program.

### POOLTAB

- Awareness of different literal pools is maintained using the auxiliary table POOLTAB.
- This table contains the literal number of the starting literal of each literal pool.
- At any stage, the current literal pool is the last pool in the LITTAB.
- On encountering an LTORG statement (or the END statement), literals in the current pool are allocated addresses starting with the current value in LC and LC is appropriately incremented.

	literal	Address
1	= '5'	
2	= '1'	
3	= '1'	

LITTAB

Literal no
#1
#3

POOLTAB

### Algorithm for Pass I

- 1) loc\_cntr=0(default value)  
pooltab\_ptr=1; POOLTAB[1]=1;

`littab_ptr=1;`

2) While next statement is not END statement

a) If a label is present then

`this_label=symbol in label field`

`Enter (this_label, loc_cntr) in SYMTAB`

b) If an LTORG statement then

(i) Process literals LITAB to allocate memory and put the address field.update  
`loc_cntr` accordingly

(ii) `pooltab_ptr= pooltab_ptr+1;`

(iii) `POOLTAB[ pooltab_ptr]= littab_ptr`

c) If a START or ORIGIN statement then

`loc_cntr=value specified in operand field;`

d) If an EQU statement then

(i) `this_address=value specified in <address spec>;`

(ii) Correct the symtab entry for `this_label` to `(this_label, this_address);`

e) If a declaration

(i) `Code= code of the declaration statement`

(ii) `Size= size of memory area required by DC/DS`

(iii) `loc_cntr=loc_cntr+size;`

(iv) Generate IC '(DL,code)'..

f) If an imperative statement then

(i) `Code= machine opcode from OPTAB`

(ii) `loc_cntr=loc_cntr+instruction length from OPTAB;`

(iii) if operand is a literal then

`this_literal=literal in operand field;`

`LITAB[littab_ptr]=this_literal;`

`littab_ptr= littab_ptr +1;`

else

`this_entry= SYMTAB entry number of operand`

`generate IC '(IS, code)(S, this_entry)';`

3) (processing END statement)

a) Perform step2(b)

b) Generate IC '(AD,02)'

c) Go to pass II

### Intermediate code forms:

- Intermediate code consist of a set of IC units, each unit consisting of the following three fields
  - Address
  - Representation of mnemonics opcode
  - Representation of operands

### Mnemonics field

- The mnemonics field contains a pair of the form (statement class, code)
- Where statement class can be one of IS, DL, and AD standing for imperative statement, declaration statement and assembler directive respectively.
- For imperative statement, code is the instruction opcode in the machine language.
- For declarations and assembler directives, code is an ordinal number within the class.
- Thus, (AD, 01) stands for assembler directive number 1 which is the directive START.
- Codes for various declaration statements and assembler directives.

Declaration statement	
DC	01
DS	02

Assembler directive			
START	01	EQU	04
END	02	LTORG	05
ORIGIN	03		

- The information in the mnemonics field is assumed to have the same representation in all the variants.

### Intermediate code for Imperative statement

#### Variant I

- First operand is represented by a single digit number which is a code for a register or the condition code

Register	Code
AREG	01
BREG	02
CREG	03
DREG	04

Condition	Code
LT	01
LE	02
EQ	03
GT	04
GE	05
ANY	06

- The second operand, which is a memory operand, is represented by a pair of the form (operand class, code)
- Where operand class is one of the C, S and L standing for constant, symbol and literal.
- For a constant, the code field contains the internal representation of the constant itself. Ex: the operand descriptor for the statement START 200 is (C,200).
- For a symbol or literal, the code field contains the ordinal number of the operand's entry in SYMTAB or LITAB.

#### Variant II

- This variant differs from variant I of the intermediate code because in variant II symbols, condition codes and CPU register are not processed.
- So, IC unit will not generate for that during pass I.

		Variant I	Variant II
	START 200	(AD,01) (C, 200)	(AD,01) (C, 200)
	READ A	(IS, 09) (S, 01)	(IS, 09) A
LOOP	MOVER AREG, A	(IS, 04) (1)(S, 01)	(IS, 04) AREG, A
	.	.	.
	.	.	.
	SUB AREG, =1'	(IS, 02) (1)(L, 01)	(IS, 02) AREG,(L, 01)
	BC GT, LOOP	(IS, 07) (4)(S, 02)	(IS, 07) GT, LOOP
	STOP	(IS, 00)	(IS, 00)
A	DS 1	(DL, 02) (C,1)	(DL, 02) (C,1)
	LTORG	(AD, 05)	(AD, 05)
	.....		

### Comparison of the variants

Variant I	Variant II
IS, DL and AD all statements contain processed form.	DL and AD statements contain processed form while for IS statements, operand field is processed only to identify literal references.
Extra work in pass I	Extra work in pass II
Simplifies tasks in pass II	Simplifies tasks in pass I
Occupies more memory than pass II	Memory utilization of two passes gets better balanced.

### Algorithm for Pass - II

It has been assumed that the target code is to be assembled in the area named code\_area.

1. Code\_area\_adress= address of code\_areas;  
Pooltab\_ptr=1;  
Loc\_cntr=0;
2. While next statement is not an END statement
  - a) Clear machine\_code\_buffer;
  - b) If an LORG statement
    - i) Process literals in LITTAB and assemble the literals in machine\_code\_buffer.
    - ii) Size= size of memory area required for literals
    - iii) Pooltab\_ptr=pooltab\_ptr +1;

- c) If a START or ORIGIN statement
    - i) Loc\_cntr=value specified in operand field;
    - ii) Size=0;
  - d) If a declaration statement
    - i) If a DC statement then assemble the constant in machine\_code\_buffer;
    - ii) Size= size of memory area required by DC/DS;
  - e) If an imperative statement
    - i) Get operand address from SYMTAB or LITTAB
    - ii) Assemble instruction in machine\_code\_buffer;
    - iii) Size=size of instruction;
  - f) If size≠ 0 then
    - i) Move contents of machine\_code\_buffer to the address code\_area\_address+loc\_cntr;
    - ii) Loc\_cntr=loc\_cntr+size;
3. Processing end statement
- a) Perform steps 2(b) and 2(f)
  - b) Write code\_area into output file.

### Error reporting of assembler

#### Error reporting in pass I

- Listing an error in first pass has the advantage that source program need not be preserved till pass II.
- But, listing produced in pass I can only reports certain errors not all.
- From the below program, error is detected at statement 9 and 21.
- Statement 9 gives invalid opcode error because MVER does not matchwith any mnemonics in OPTAB.
- Statement 21 gives duplicate defination error because entry of A is already exist in symbol table.
- Undefined symbol B at statement 10 is harder to detect during pass I, this error can be detected only after completing pass I.

Sr.no	Statements	Address
1	START 200	
2	MOVER AREG,A	200
3	.	.
	.	.
9	MVER BREG, A	207
	**ERROR* Invalid opcode	
10	<b>ADD BREG, B</b>	208
14	A DS 1	209
.	.	.
.	.	.
21	A DC '5'	227

	**ERROR** duplicate definition of symbol A	
	.	
35	END	
	**ERROR** undefined symbol B in statement 10	

### Error reporting in pass II

- During pass II data structure like SYMTAB is available.
- Error indication at statement 10 is also easy because symbol table is searched for an entry B. if match is not found, error is reported.

## Single Pass Assembler for Intel x86

### Design

- The algorithm for the Intel 8088 assembler is given at the end of this section.
- LC processing in this algorithm differs from LC processing in the first pass of a two-pass assembler in one significant respect.
- In Intel 8088, the unit for memory allocation is a byte; however, certain entities require their first byte to be aligned on specific boundaries in the address space.
- While processing declarations and imperative statements, the assembler first aligns the address contained in the LC on the appropriate boundary. We call this action LC alignment.
- Allocation of memory for a statement is performed after LC alignment.
- The data structures of the assembler are as follows:

#### 1) Mnemonics Table (MOT)

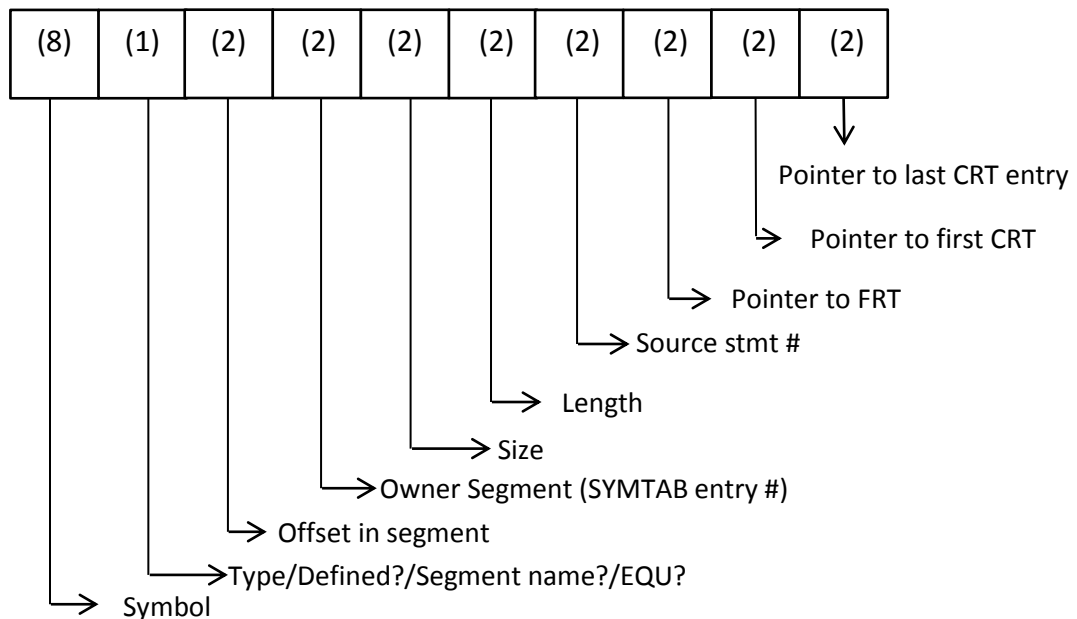
The mnemonics table (MOT) is hash organized and contains the following fields: mnemonic opcode, machine opcode, alignment/format info and routine id. The routine id field of an entry specifies the routine which handles that opcode. Alignment/format info is specific to a given routine.

Mnemonic opcode (6)	Machine opcode (2)	Alignment/format info (1)	Routine id (4)
JNE	75H	OOH	R2

#### 2) Symbol Table (Symtab)

The symbol table (SYMTAB) is hash-organized and contains information about symbols defined and used in the source program. The contents of some important fields are as follows: The owner segment field indicates the id of the segment in which the symbol is defined. It contains the SYMTAB entry # of the segment name. For a non-EQU symbol the type field indicates the alignment information. For an EQU symbol, the type field indicates whether the symbol is to be given a numeric value or a textual value, and the value itself is accommodated in the owner segment and offset fields of the entry.





### 3) Segment Register Table (SRTAB)

The segment register table (SRTAB) contains four entries, one for each segment register. Each entry shows the SYMTAB entry # of the segment whose address is contained in the segment register. SRTAB\_ARRAY is an array of SRTABs.

Segment Register (1)	SYMTAB entry # (2)
00 (ES)	23
.	
.	

### 4) Forward Reference Table (FRT)

Information concerning forward references to a symbol is organized as a linked list. Thus, the forward reference table (FRT) contains a set of linked lists, one for each forward referenced symbol.

Pointer (2)	SRTAB # (1)	Instruction address (2)	Usage Code (1)	Source stmt # (2)

### 5) Cross Reference Table (CRT)

A cross reference directory is a report produced by the assembler which lists all references to a symbol sorted in the ascending order by statement numbers. The assembler uses the cross reference table (CRT) to collect the relevant information.

Pointer (2)	Source stmt # (2)

## Algorithm of the Single-Pass Assembler

Important data structures used by the Single-Pass Assembler:

SYMTAB, SRTAB\_ARRAY, CRT, FRT and ERRTAB

LC	:	Location Counter
code_area	:	Area for assembling the target program
code_area_address	:	Contains address of code_area
srtab_no	:	Number of the current SRTAB
stmt_no	:	Number of the current statement
SYMTAB_segment_entry	:	SYMTAB entry # of current segment
machine_code_buffer	:	Area of constructing code for one statement

### Algorithm (Single pass assembler for Intel 8088)

- 1) *code\_area\_address* := address of *code\_area*;  
*srtab\_no* := 1;  
LC := 0;  
stmt\_no := 1;  
SYMTAB\_segment\_entry := 0;  
Clear ERRTAB, SRTAB\_ARRAY.
- 2) While the next statement is not an END statement
  - a) Clear machine\_code\_buffer.
  - b) If a symbol is present in the label field then  
this\_label := symbol in the label field;
  - c) If an EQU statement
    - i) this\_address := value of <address specification>;
    - ii) Make an entry for this\_label in SYMTAB with  
offset := this\_addr;  
Defined := 'yes'  
owner\_segment := owner\_segment in SYMTAB entry of the symbol in the operand field.  
source\_stmt\_# := stmt\_no;
    - iii) Enter stmt\_no in the CRT list of the label in the operand field.
    - iv) Process forward references to this\_label;
    - v) Size := 0;
  - d) If an ASSUME statement
    - i) Copy the SRTAB in SRTAB\_ARRAY[srtab\_no] into SRTAB\_ARRAY [srtab\_no+1]
    - ii) srtab\_no := srtab\_no+1;
    - iii) For each specification in the ASSUME statement
      - (a) this\_register := register mentioned in the specification.
      - (b) this\_segment := entry number of SYMTAB entry of the segment appearing in the specification.
      - (c) Make the entry (this\_register, this\_segment) in SRTAB\_ARRAY[srtab\_no]. (It overwrites an existing entry for this\_register.)
      - (d) size := 0;
  - e) If a SEGMENT statement

- i) Make an entry for this\_label in SYMTAB and note the entry number.
- ii) Set segment name? := true;
- iii) SYMTAB\_segment\_entry := entry no. in SYMTAB;
- iv) LC := 0;
- v) size := 0;
- f) If an ENDS statement then  
SYMTAB\_segment\_entry := 0;
- g) If a declaration statement
  - i) Align LC according to the specification in the operand field.
  - ii) Assemble the constant(s), if any, in the machine\_code\_buffer.
  - iii) size := size of memory area required;
- h) If an imperative statement
  - i) If the operand is a symbol symb then enter stmt\_no in CRT list of symb.
  - ii) If the operand symbol is already defined then  
Check its alignment and addressability.  
Generate the address specification (segment register, offset) for the symbol using its SYMTAB entry and SRTAB\_ARRAY[srtab\_no].  
else  
Make an entry for symb in SYMTAB with defined := 'no';  
Make the entry (srtab\_no, LC, usage code, stmt\_no) in FRT of symb.
  - iii) Assemble instruction in machine\_code\_buffer.
  - iv) size := size of the instruction;
- i) If size ≠ 0 then
  - i) If label is present then  
Make an entry for this\_label in SYMTAB with  
owner\_segment := SYMTAB\_segment\_entry;  
Defined := 'yes';  
offset := LC;  
source\_stmt\_# := stmt\_no;
  - ii) Move contents of machine\_code\_buffer to the address code\_area\_address + <LC>;
  - iii) LC := LC + size;
  - iv) Process forward references to the symbol. Check for alignment and addressability errors. Enter errors in the ERRTAB.
  - v) List the statement along with errors pertaining to it found in the ERRTAB.
  - vi) Clear ERRTAB.
- 3) (Processing of END statement)
  - a) Report undefined symbols from the SYMTAB.
  - b) Produce cross reference listing.
  - c) Write code\_area into the output file.

### Example 1: Assembly Program to compute N! & equivalent machine language program

					Opcode (2 digit)	Register operand (1 digit)	Memory operand (3 digit)
1		START	101				
2		READ	N	101)	09	0	113
3		MOVER	BREG, ONE	102)	04	2	115
4		MOVEM	BREG, TERM	103)	05	2	116
5	AGAIN	MULT	BREG, TERM	104)	03	2	116
6		MOVER	CREG, TERM	105)	04	3	116
7		ADD	CREG, ONE	106)	01	3	115
12		MOVEM	CREG, TERM	107)	05	3	116
13		COMP	CREG, N	108)	06	3	113
14		BC	LE, AGAIN	109)	07	2	104
15		MOVEM	BREG, RESULT	110)	05	2	114
16		PRINT	RESULT	111)	10	0	114
17		STOP		112)	00	0	000
18	N	DS	1	113)			
19	RESULT	DS	1	114)			
20	ONE	DC	'1'	115)	00	0	001
21	TERM	DS	1	116)			
22		END					

### Output of Pass-I

- Data Structure of Pass-I of an assembler

OPTAB		
Mnemonic OPCODE	Class	Mnemonic info
START	AD	R#11
READ	IS	(09,1)
MOVER	IS	(04,1)
MOVEM	IS	(05,1)
ADD	IS	(01,1)
BC	IS	(07,1)
DC	DL	R#5
SUB	IS	(02,1)
STOP	IS	(00,1)
COMP	IS	(06,1)

DS	DL	R#7
PRINT	IS	(10,1)
END	AD	
MULT	IS	(03,1)

SYMTAB		
Symbol	Address	Length
AGAIN	104	1
N	113	1
RESULT	114	1
ONE	115	1
TERM	116	1

**LITTAB** and **POOLTAB** are empty as there are no Literals defined in the the program.

- Intermediate Representation**

				Variant I		Variant II	
1		START	101	(AD, 01)	(C, 101)	(AD, 01)	(C, 101)
2		READ	N	(IS, 09)	(S, 02)	(IS, 09)	N
3		MOVER	BREG, ONE	(IS, 04)	(2)(S, 04)	(IS, 04)	(2) ONE
4		MOVEM	BREG, TERM	(IS, 05)	(2)(S, 05)	(IS, 05)	(2) TERM
5	AGAIN	MULT	BREG, TERM	(IS, 03)	(2)(S, 05)	(IS, 03)	(2) TERM
6		MOVER	CREG, TERM	(IS, 04)	(3)(S, 05)	(IS, 04)	(3) TERM
7		ADD	CREG, ONE	(IS, 01)	(3)(S, 04)	(IS, 01)	(3) ONE
12		MOVEM	CREG, TERM	(IS, 05)	(3)(S, 05)	(IS, 05)	(3) TERM
13		COMP	CREG, N	(IS, 06)	(3)(S, 02)	(IS, 06)	(3) N
14		BC	LE, AGAIN	(IS, 07)	(2)(S, 01)	(IS, 07)	(2) AGAIN
15		MOVEM	BREG, RESULT	(IS, 05)	(2)(S, 03)	(IS, 05)	(2) RESULT
16		PRINT	RESULT	(IS, 10)	(S, 03)	(IS, 10)	RESULT
17		STOP		(IS, 00)		(IS, 00)	
18	N	DS	1	(DL, 02)	(C, 1)	(DL, 02)	(C, 1)
19	RESULT	DS	1	(DL, 02)	(C, 1)	(DL, 02)	(C, 1)
20	ONE	DC	'1'	(DL, 01)	(C, 1)	(DL, 01)	(C, 1)
21	TERM	DS	1	(DL, 02)	(C, 1)	(DL, 02)	(C, 1)
22		END		(AD, 02)		(AD, 02)	

### Macro

- Formally, macro instructions (often called macro) are single-line abbreviations for groups of instructions.
- For every occurrence of this one-line macro instruction within a program, the instruction must be replaced by the entire block.
- The **advantages** of using macro are as follows:
  - Simplify and reduce the amount of repetitive coding.
  - Reduce the possibility of errors caused by repetitive coding.
  - Make an assembly program more readable.

### Macro Processors

- A preprocessor can be any program that processes its input data to produce output, which is used as an input to another program.
- The outputs of the macro processors are assembly programs that become inputs to the assembler.
- The macro processor may exist independently and be called during the assembling process or be a part of the assembler implementation itself.

### Difference between Macro and Subroutine

Macro	Subroutine
Macro name in the mnemonic field leads to expansion only.	Subroutine name in a call statement in the program leads to execution.
Macros are completely handled by the assembler during assembly time.	Subroutines are completely handled by the hardware at runtime.
Macro definition and macro expansion are executed by the assembler. So, the assembler has to know all the features, options, and exceptions associated with them.	Hardware executes the subroutine call instruction. So, it has to know how to save the return address and how to branch to the subroutine.
The hardware knows nothing about macros.	The assembler knows nothing about subroutines.
The macro processor generates a new copy of the macro and places it in the program.	The subroutine call instruction is assembled in the usual way and treated by the assembler as any other instruction.
Macro processing increases the size of the resulting code but results in faster execution of program for expanded programs.	Use of subroutines does not result into bulk object codes but has substantial overheads of control transfer during execution.

### Macro Definition and Call

- It has been aforementioned that a macro consists of a name, a set of formal parameters, and a body of codes.
- A macro can be defined by enclosing a set of statements between a macro header and a macro end statement.
- The formal structure of a macro includes the following features:
  - **Macro prototype statement:** Specifies the name of the macro and name and type of formal parameters.
  - **Model statements:** Specify the statements in the body of the macro from which assembly language statements are to be generated during expansion.
  - **Macro preprocessor statement:** Specifies the statement used for performing auxiliary function during macro expansion

- A macro prototype statement can be written as follows:

`<name_of_macro> [<formal parameter spec> [...]]`

where `<formal parameter spec> [...]` defines the parameter name and its kind, which are of the following form:

`&<name_of_parameter> [<parameter_type>]`

- A macro can be called by writing the name of the macro in the mnemonic field of the assembly language. The syntax of a typical macro call can be of the following form:

`<name_of_macro> [<actual_parameter_spec> [...]]`

- The MACRO directive in the mnemonic field specifies the start of the macro definition and it should compulsorily have the macro name in the label field.
- The MEND directive specifies the end of the macro definition.
- The statements between MACRO and MEND directives define the body (model statements) of the macro and can appear in the expanded code.
- Eg.

#### Macro Definition

```
MACRO
INCR    &MEM_VAL, &INC_VAL, &REG
MOVER   &REG      &MEM_VAL
ADD     &REG      &INC_VAL
MOVEM   &REG      &MEM_VAL
MEND
```

#### Macro Call

INCR A, B

### Macro Expansion

A macro call in a program leads to macro expansion. To expand a macro, the name of the macro is

placed in the operation field, and no special directives are necessary. During macro expansion, the macro name statement in the program is replaced by the sequence of assembly statements. Let us consider the following example:

```
START 100
A      DS      1
B      DS      1
INCR   A, B, AREG
PRINT  A
STOP
END
```

The preceding example uses a statement that calls the macro. The assembly code sequence INCR A, B, AREG is an example of the macro call, with A and B being the actual parameters of the macro. While passing over the assembly program, the assembler recognizes INCR as the name of the macro, expands the macro, and places a copy of the macro definition (along with the parameter substitutions). The expanded code for the code is as below.

```
START 100
A      DS  1
B      DS  1
+ MOVER REG A
+ ADD   REG B
+ MOVEM REG A
PRINT  A
STOP
END
```

The statements marked with a '+' sign in the preceding label field denote the expanded code and differentiate them from the original statements of the program.

### Types of formal parameters

#### Positional parameters

- For positional formal parameters, the specification <parameter kind> of syntax rule is simply omitted. Thus, a positional formal parameter is written as &<parameter name>, e.g., &SAMPLE where SAMPLE is the name of a parameter.
- In a call on a macro using positional parameters (see syntax rule (4.2)), the <actual parameter specification> is an ordinary string.
- The value of a positional formal parameter XYZ is determined by the rule of positional association as follows:
  - Find the ordinal position of XYZ in the list of formal parameters in the macro prototype statement.
  - Find the actual parameter specification that occupies the same ordinal position in the list of actual parameters in the macro call statement. If it is an ordinary string ABC, the value of formal parameter XYZ would be ABC.



### Keyword parameters

- For keyword parameters, the specification <parameter kind> is the string '=' in syntax rule. The <actual parameter specification> is written as <formal parameter name> = <ordinary string>. The value of a formal parameter is determined by the rule of keyword association as follows:
  - Find the actual parameter specification which has the form XYZ= <ordinary string>.
  - If the <ordinary string> in the specification is some string ABC, the value of formal parameter XYZ would be ABC.

### Specifying default values of parameters

- If a parameter has the same value in most calls on a macro, this value can be specified as its default value in the macro definition itself.
- If a macro call does not explicitly specify the value of the parameter, the preprocessor uses its default value; otherwise, it uses the value specified in the macro call. This way, a programmer would have to specify a value of the parameter only when it differs from its default value specified in the macro definition.
- Default values of keyword parameters can be specified by extending the syntax of formal parameter specification as follows:  
& < parameter name > [< parameter kind > [< default value >]]

### Macros with mixed parameter lists

- A macro definition may use both positional and keyword parameters. In such a case, all positional parameters must precede all keyword parameters in a macro call.
- For example, in the macro call  
SUMUP A,B,G=20,H=X
- A and B are positional parameters while G and H are keyword parameters. Correspondence between actual and formal parameters is established by applying the rules governing positional and keyword parameters separately.

## Advanced macro facilities

### Alteration of flow of control during expansion

#### Expansion time statement: OR (Explain expansion time statements –AIF and AGO for macro programming)

##### AIF

- An AIF statement has the syntax:
  - AIF (<expression>) <sequencing symbol>where <expression> is a relational expression involving ordinary strings, formal parameters and their attributes, and expansion time variables.
- If the relational expression evaluates to true, expansion time control is transferred to the statement containing <sequencing symbol> in its label field.

##### AGO

- An AGO statement has the syntax:
  - AGO <sequencing symbol>

- It unconditionally transfers expansion time control to the statement containing <sequencing symbol> in its label field.

### **Expansion time loops or (Explain expansion time loop)**

- It is often necessary to generate many similar statements during the expansion of a macro.
- This can be achieved by writing similar model statements in the macro.
- Expansion time loops can be written using expansion time variables (EV's) and expansion time control transfer statements AIF and AGO.

#### **Example**

```

MACRO
CLEAR      &X, &N
LCL        &M
&M         SET      0
MOVER      AREG, ='0'
.MORE      MOVEM    AREG, &X+&M
&M         SET      &M + 1
AIF        (&M      NE      N)
           .MORE
MEND

```

- The LCL statement declares M to be a local EV.
- At the start of expansion of the call, M is initialized to zero.
- The expansion of model statement MOVEM, AREG, &X+&M thus leads to generation of the statement MOVEM AREG, B.
- The value of M is incremented by 1 and the model statement MOVEM.. is expanded repeatedly until its value equals the value of N.

### **Expansion time variable or (Explain expansion time variable with example)**

- Expansion time variables (EV's) are variables which can only be used during the expansion of macro calls.
- A local EV is created for use only during a particular macro call.
- A global EV exists across all macro calls situated in a program and can be used in any macro which has a declaration for it.
- Local and global EV's are created through declaration statements with the following syntax:
  - LCL <EV specification> [, <EV specification> .. ]
  - GBL <EV specification> [, <EV specification> .. ]
<EV specification> has the syntax &<EV name>, where <EV name> is an ordinary string.
- Values of EV's can be manipulated through the preprocessor statement SET.
- A SET statement is written as:
  - < EV specification > SET <SET-expression>
where < EV specification > appears in the label field and SET in mnemonic field.
- A SET statement assigns value of <SET-expression> to the EV specified in <EV specification>.

### Example

```

MACRO
CONSTANTS
LCL          &A
SET          1
DB           &A
SET          &A+1
DB           &A
MEND

```

- The local EV A is created.
- The first SET statement assigns the value '1' to it.
- The first DB statement thus declares a byte constant '1'.
- The second SET statement assigns the value '2' to A and the second DB statement declares a constant '2'.

### Attributes of formal parameter

- An attribute is written using the syntax  
`<attribute name> ' <formal parameter spec>`
- It represents information about the value of the formal parameter, i.e. about the corresponding actual parameter.
- The type, length and size attributes have the names T, L and S.

#### • Example

```

MACRO
DCL_CONST    &A
AIF          (L'&A EQ 1) .NEXT
--
.NEXT        --
--
MEND

```

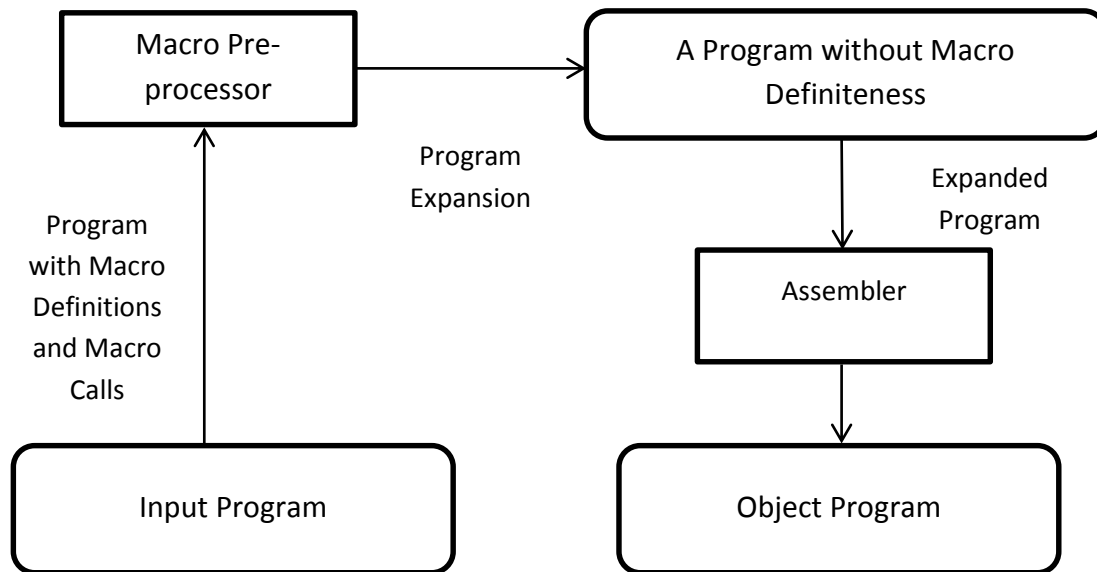
Here expansion time control is transferred to the statement having .NEXT field only if the actual parameter corresponding to the formal parameter length of ' 1'.

## Design of Macro Preprocessor

Macro preprocessors are vital for processing all programs that contain macro definitions and/or calls. Language translators such as assemblers and compilers cannot directly generate the target code from the programs containing definitions and calls for macros. Therefore, most language processing activities by assemblers and compilers preprocess these programs through macro processors. A macro preprocessor essentially accepts an assembly program with macro definitions and calls as its input and processes it into an equivalent expanded assembly program with no macro definitions and calls. The macro preprocessor output program is then passed over to an assembler

to generate the target object program.

The general design semantics of a macro preprocessor is shown as below



The design of a macro preprocessor is influenced by the provisions for performing the following tasks involved in macro expansion:

- **Recognize macro calls:** A table is maintained to store names of all macros defined in a program. Such a table is called Macro Name Table (MNT) in which an entry is made for every macro definition being processed. During processing program statements, a match is done to compare strings in the mnemonic field with entries in the MNT. A successful match in the MNT indicates that the statement is a macro call.
- **Determine the values of formal parameters:** A table called Actual Parameter Table (APT) holds the values of formal parameters during the expansion of a macro call. The entry into this table will be in pair of the form (<formal parameter name>, <value>). A table called Parameter Default Table (PDT) contains information about default parameters stored as pairs of the form (<formal parameter name>, <default value>) for each macro defined in the program. If the programmer does not specify value for any or some parameters, its corresponding default value is copied from PDT to APT.
- **Maintain the values of expansion time variables declared in a macro:** A table called Expansion time Variable Table (EVT) maintains information about expansion variables in the form (<EV name>, <value>). It is used when a preprocessor statement or a model statement during expansion refers to an EV.
- **Organize expansion time control flow:** A table called Macro Definition Table (MDT) is used to store the body of a macro. The flow of control determines when a model statement from the MDT is to be visited for expansion during macro expansion. MEC {Macro Expansion Counter} is defined and initialized to the first statement of the macro body in the MDT. MDT is updated following an expansion of a model statement by a macro preprocessor.
- **Determine the values of sequencing symbols:** A table called Sequencing Symbols Table

(SST) maintains information about sequencing symbols in pairs of the form

(<sequencing symbol name>, <MDT entry #>)

where <MDT entry #> denotes the index of the MDT entry containing the model statement with the sequencing symbol. Entries are made on encountering a statement with the sequencing symbol in their label field or on reading a reference prior to its definition.

- **Perform expansion of a model statement:** The expansion task has the following steps:
  - MEC points to the entry in the MDT table with the model statements.
  - APT and EVT provide the values of the formal parameters and EVs, respectively.
  - SST enables identifying the model statement and defining sequencing.

### Design of Macro Assembler

- A macro processor is functionally independent of the assembler, and the output of the macro processor will be a part of the input into the assembler.
- A macro processor, similar to any other assembler, scans and processes statements.
- Often, the use of a separate macro processor for handling macro instructions leads to less efficient program translation because many functions are duplicated by the assembler and macro processor.
- To overcome efficiency issues and avoid duplicate work by the assembler, the macro processor is generally implemented within pass 1 of the assembler.
- The integration of macro processor and assembler is often referred to as macro assembler. Such implementations will help in eliminating the overheads of creating intermediate files, thus improving the performance of integration by combining similar functions.
- The advantages of a macro assembler are as follows:
  - It ensures that many functions need not be implemented twice.
  - Results in fewer overheads because many functions are combined and do not need to create intermediate (temporary) files.
  - It offers more flexibility in programming and allows the use of all assembler features in combination with macros.
- The disadvantages of a macro assembler are as follows:
  - The resulting pass by combining macro processing and pass 1 of the assembler may be too large and sometimes suffer from core memory problems.
  - The combination of macro processor pass 0 and pass I may sometimes increase the complexity of program translation, which is not desired.

### Functions of Macro Processor

The design and operation of a macro processor greatly influence the activities performed by it. In general, a macro processor will perform the following tasks:

- Identifies macro definitions and calls in the program.
- Determines formal parameters and their values.
- Keeps track of the values of expansion time variables and sequencing symbols declared in a

macro.

- Handles expansion time control flow and performs expansion of model statements.

### Basic Tasks of Macro Processor

Macro processing involves the following two separate steps for handling macros in a program:

#### 1. Handling Macro Definition

- In general, a macro in a program can have only one definition, but it can be called (expanded) many times.
- For the purpose of handling macro definitions, the macro assembler maintains tables called Macro Name Table (MNT) and Macro Definition Table (MDT). MNT is used to maintain a list of macros names defined in the program, while MDT contains the actual definition statements (the body of macro).
- Macro definition handling starts with a MACRO directive in the statement. The assembler continually reads the definition from the source file and saves it in the MDT. During this, the assembler, in most cases, will just save the definition as it is in the MDT and not try to assemble it or execute it.
- On encountering the MACRO directive in the source assembly program, the assembler changes from the regular mode to a special macro definition mode, wherein it does the following activities:
  - Analyzes the available space in the MDT
  - Reads continually the statements and writes them to the MDT until the MEND directive is found.
- When a MEND directive is encountered in the source file, the assembler reverts to the normal mode. If the MEND directive is missing, the assembler will stay in the macro definition mode and continue to save program statements in the MDT until an obvious error occurs. This will happen in cases such as reading another MACRO directive or an END statement, where the assembler will generate an error (runaway definition) and abort the assembly process.

#### 2. Handling Macro Definition

- When an assembler encounters a source statement that is not an instruction or a directive, it becomes optimistic and does not flag error immediately. The assembler will rather search the MNT and MDT for a macro with name in the opcode file and, on locating a valid entry for it, change its mode of operation from normal mode to macro expansion mode. The succession of tasks performed in the macro expansion mode is as follows:
  - Reading back a source statement from the MDT entry.
  - Writing the statement on the new source listing file unless it is a pass-0 directive (such as a call or definition of another macro); in that case, it is executed immediately.
  - Repeating the previous two steps until the end of the macro is located in the MDT

### Design Issues of Macro Processor

The most highlighted key issues of a macro processing design are as follows:

- **Flexible data structures and databases:** They should maintain several data structures to keep track of locations, nesting structures, values of formal and positional parameters, and other important information concerning the source program.
- **Attributes of macro arguments:** Macro arguments used for expansion have attributes. These attributes include count, type, length, integer, scaling, and number attributes. Attributes can be used to make decisions each time a macro is expanded. Note that attributes are unknown at the time of macro definition and are known only when the macro is expanded.
- **Default arguments:** Many assemblers allow use of default arguments. This means when the actual argument that binds the formal argument is null in a certain expansion, the argument will be bound to default value specified in the definition.
- **Numeric values of arguments:** Although most macro processors treat arguments normally as strings. Some assemblers, like VAX assembler, optionally allow using the value, rather than the name of the argument.
- **Comments in macros:** Comments are printed with macro definition, but they might or might not be with each expansion. Some comments are meant only for definitions, while some are expected in the expanded code.

### Design Features of Macro Processor

The features of a macro processor design are as follows:

- **Associating macro parameters with their arguments:** All macro processors support associating macro parameters by position, name, and numeric position in the argument list.
- **Delimiting macro parameters:** Macro processors use specially defined characters such as delimiters or a scheme where parameters can be delimited in a general way. Characters like ';' and '.' are used in many macro processor implementations.
- **Directives related to arguments:** Modern macro processors support arguments that ease the task of writing sophisticated macros. The directive IF-ELSE-ENDIF helps decide whether an argument is blank or not, or whether identical or different arguments are used.
- **Automatic label generation:** Directives like IRP and PRINT provide facilities to work with labels. A pair of IRP directives defines a sequence of lines directing the assembler to repeatedly duplicate and assemble the sequence as many times as determined by a compound parameter. The PRINT directive suppresses listing of macro expansions or turns on such a listing.
- **Machine-independent features:** They include concatenation of macro parameter) generation of unique labels, conditional macro expansion, and keyword macro parameters.

### Macro Processor Design Options

### 1. Recursive Macro Expansion

For an efficient preprocessing and expansion, preprocessors use a language that supports recursion and allows the use of variables and data structure for recursive expansions. The support for handling global expansion variables together with local variables is highly desirable.

### 2. General-purpose Macro Processors

Modern macro processors are more generic and not restricted to any specific language. Facilities to call and process them are possible with macro languages. Although most macros are used in assembler language, they are used with higher-level languages as well and need to learn a different macro language for programming macros. Although desirable, yet such type of general macro facility is hard since each language follows its own way of doing tasks.

### 3. Macro Processing within Language Translators

It is essential that all macro definitions be processed, symbols be resolved, and calls be explained before a program is sent to language translators such as assemblers. Therefore, macro processing must be a preprocessing step for language assembling. Therefore, integrating macro processing within translation activity is highly desirable.

### 4. Line-by-Line Macro Processor

Design options that avoid making an extra pass over the source program is another interesting aspect to look into. Performing macro processing line by line enables sharing of data structures, utility functions and procedures, and supporting diagnostic messages. Line-by-line macro processor environment reads source program, processes macro definitions, expands macro call, and finally transfers output lines to translators like assemblers or compilers.

## Design of Two-pass Macro Preprocessor

### Pass 0 of Assembler

The activities of pass-0 macro processor is given in the following steps:

1. Read and examine the next source statement.
2. If MACRO statement, continue reading the source and copy the entire macro definition to the MDT. Go to Step 1.
3. If the statement is a pass-0 directive, execute it. Go to Step 1. (These directives are written to the new source file in a unique manner (different from normal directives). They are only needed for the listing in pass 2.
4. If the statement contains a macro name, it must perform expansion, that is, read model statements from the MDT corresponding to the call, substitute parameters, and write each statement to the new source file (or execute it if it is a pass-0 directive). Go to Step 1.
5. For any other statement, write the statement to the new source file. Go to Step 1.
6. If the current statement contains the END directive, stop (end of pass 0).



The assembler will be in one of the three modes:

- In the normal mode, the assembler will read statement lines from the source file and write them to the new source file. There is no translation or any change in the statements. In the macro definition mode, the assembler will continuously copy the source file to the MDT.
- In the macro expansion mode, the assembler will read statements from the MDT, substitute parameters, and write them to the new source file. Nested macros can be implemented using the Definition and Expansion (DE) mode.

### Pass 1 of Macro Processor - Processing Macro Definitions

1. Initialize MDTC and MNTC.
2. Read the next source statement of the program.
3. If the statement contains MACRO pseudo-op. go to Step 6.
4. Output the instruction of the program.
5. If the statement contains END pseudo-op, go to Pass 2, else go to Step 2.
6. Read the next source statement of the program.
7. Make an entry of the macro name and MTDC into MNT at location MNTC and increment the MNTC by 1.
8. Prepare the parameter (arguments) list array.
9. Enter macro name into the MDT and increment the MTDC by 1.
10. Read the next card and substitute index for the parameters (arguments).
11. Enter the statement into the MDT and increment the MDT by 1.
12. If MEND pseudo-op found, go to Step 2, else go to Step 10.

### Pass 2 of Macro Processor - Processing for Calls and Expansion of Macro

1. Read the next source statement copied by pass 1.
2. Search into the MNT for record and evaluate the operation code.
3. If the operation code has a macro name, go to Step 5.
4. Write the statement to the expanded source file.
5. If END pseudo-op found, pass the entire expanded code to the assembler for assembling and stop. Else go to Step 1.
6. Update the MDTP to the MDT index from the MNT entry.
7. Prepare the parameter (argument) list array.
8. Increment the MDTP by 1.
9. Read the statement from the current MDT and substitute actual parameters (arguments) from the macro call.
10. If the statement contains MEND pseudo-op, go to Step 1, else write the expanded source code and go to Step 8.

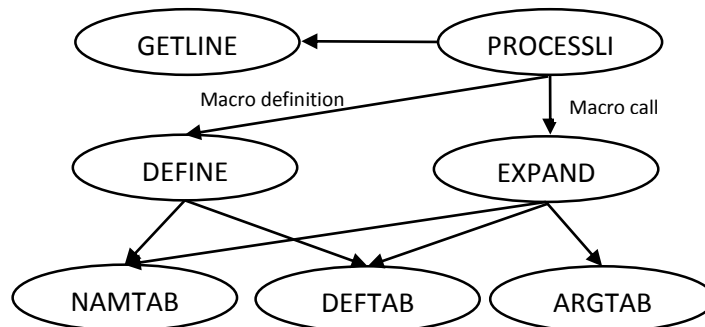
### One-pass Macro Processors

A one-pass macro processor is another design option available for macro processing. The restriction in working with one-pass macro processors is that they strictly require the definition of a macro to

appear always before any statements that invoke that macro in the program.

The important data structures required in a one-pass macro processor are:

- **DEFTAB (Definition Table):** It is a definition table that is used to store the macro definition including macro prototype and macro body. Comment lines are not included here, and references to the parameters use positional notation for efficiency in substituting arguments.
- **NAMTAB (Name Table):** This table is used for storing macros names. It serves as an index to DEFTAB and maintains pointers that point to the beginning and end of the macro definition in DEFTAB.
- **ARGTAB (Argument Table):** It maintains arguments according to their positions in the argument list. During expansion, the arguments from this table are substituted for the corresponding parameters in the macro body.
- One-pass Macro Processor scheme is presented as below.



### Introduction

- 1) **Translation time address:** Translation time address is used at the translation time. This address is assigned by translator
- 2) **Linked time address:** Link time address is used at the link time. This address is assigned by linker
- 3) **Load time address:** Load time address is used at the load time. This address is assigned by loader
- 4) **Translated origin:** Address of origin assumed by the translator
- 5) **Linked origin:** Address of origin assumed by the linker while producing a binary program
- 6) **Load origin:** Address of origin assumed by the loader while loading the program for execution.

### Relocation of Linking Concept

- Program relocation is the process of modifying the addresses used in the address sensitive instruction of a program such that the program can execute correctly from the designated area of memory.
- If linked origin  $\neq$  translated origin, relocation must be performed by the linker.
- If load origin  $\neq$  linked origin, relocation must be performed by the loader.
- Let AA be the set of absolute address - instruction or data addresses – used in the instruction of a program P.
- $AA \neq \phi$  implies that program P assumes its instructions and data to occupy memory words with specific addresses.
- Such a program – called an address sensitive program – contains one or more of the following:
  - An address sensitive instruction: an instruction which uses an address  $\alpha_i \in AA$ .
  - An address constant: a data word which contains an address  $\alpha_i \in AA$ .
- An address sensitive program P can execute correctly only if the start address of the memory area allocated to it is the same as its translated origin.
- To execute correctly from any other memory area, the address used in each address sensitive instruction of P must be 'corrected'.

#### Performing relocation

- Let the translated and linked origins of program P be  $t\_origin_p$  and  $l\_origin_p$ , respectively.
- Consider a symbol  $symbol$  in P.
- Let its translation time address be  $t_{symbol}$  and link time address be  $l_{symbol}$ .
- The relocation factor of P is defined as
- $Relocation\_factor_p = l\_origin_p - t\_origin_p \quad \dots(1)$
- Note that  $relocation\_factor_p$  can be positive, negative or zero.
- Consider a statement which uses  $symbol$  as an operand. The translator puts the address  $t_{symbol}$  in the instruction generated for it. Now,

- $T_{\text{symb}} = t_{\text{origin}_p} + d_{\text{symb}}$
- Where  $d_{\text{symb}_b}$  is the offset of symb in P. Hence
- $I_{\text{symb}} = I_{\text{origin}_p} + d_{\text{symb}}$
- Using (1),
- $I_{\text{symb}} = t_{\text{origin}_p} + \text{Relocation\_factor}_p + d_{\text{symb}}$
- $= t_{\text{origin}_p} + d_{\text{symb}} + \text{Relocation\_factor}_p$
- $= t_{\text{symb}} + \text{Relocation\_factor}_p \quad \dots(2)$
- Let  $IRP_p$  designate the set of instructions requiring relocation in program P. Following (2), relocation of program P can be performed by computing the relocation factor for P and adding it to the translation time address(es) in every instruction  $i \in IRP_p$ .

### Linking

- Consider an application program AP consisting of a set of program units  $SP = \{P_i\}$ .
- A program unit  $P_i$  interacts with another program unit  $P_j$  by using addresses of  $P_j$ 's instructions and data in its own instructions.
- To realize such interactions,  $P_j$  and  $P_i$  must contain public definitions and external references as defined in the following: (Explain public definition and external reference)
  - **Public definition:** a symbol  $\text{pub\_symb}$  defined in a program unit which may be referenced in other program units.
  - **External reference:** a reference to a symbol  $\text{ext\_symb}$  which is not defined in the program unit

## Design of a Linker

The design of linker is divided into two parts:

### 1) Relocation

- The linker uses an area of memory called the work area for constructing the binary program.
- It loads the machine language program found in the program component of an object module into the work area and relocates the address sensitive instructions in it by processing entries of the RELOCTAB.
- For each RELOCTAB entry, the linker determines the address of the word in the work area that contains the address sensitive instruction and relocates it.
- The details of the address computation would depend on whether the linker loads and relocates one object module at a time, or loads all object modules that are to be linked together into the work area before performing relocation.
- Algorithm: Program Relocation
  1.  $\text{program\_linked\_origin} := \langle \text{link origin} \rangle$  from the linker command;
  2. For each object module mentioned in the linker command
    - (a)  $t_{\text{origin}} := \text{translated origin of the object module};$   
 $\text{OMsize} := \text{size of the object module};$
    - (b)  $\text{relocation\_factor} := \text{program\_linked\_origin} - t_{\text{origin}};$
    - (c) Read the machine language program contained in the program component of the object module into the work-area.

- (d) Read RELOCTAB of the object module.
- (e) For each entry in RELOCTAB
  - i.  $\text{translated\_address} := \text{address found in the RELOCTAB entry};$
  - ii.  $\text{address\_in\_work\_area} := \text{address of work\_area} + \text{translated\_address} - \text{t\_origin};$
  - iii. Add relocation-factor to the operand address found in the word that has the address  $\text{address\_in\_work\_area}$ .
- (f)  $\text{Program\_linked\_origin} := \text{program\_linked\_origin} + \text{OM\_size};$

### 2) Linking

- An external reference to a symbol alpha can be resolved only if alpha is declared as a public definition in some object module.
- Using this observation as the basis, program linking can be performed as follows:
- The linker would process the linking tables (LINKTABs) of all object modules that are to be linked and copy the information about public definitions found in them into a table called the name table (NTAB).
- The external reference to alpha would be resolved simply by searching for alpha in this table, obtaining its linked address, and copying it into the word that contains the external reference.
- Accordingly, each entry of the NTAB would contain the following fields:  
**Symbol:** Symbolic name of an external reference or an object module.  
**Linked-address:** For a public definition, this field contains linked address of the symbol. For an object module, it contains the linked origin of the object module.
- Algorithm: Program Linking
  1.  $\text{program\_linked\_origin} := \text{<link origin> from the linker command}.$
  2. For each object module mentioned in the linker command
    - (a)  $\text{t\_origin} := \text{translated origin of the object module};$   
 $\text{OM\_size} := \text{size of the object module};$
    - (b)  $\text{relocation\_factor} := \text{program\_linked\_origin} - \text{t\_origin};$
    - (c) Read the machine language program contained in the program component of the object module into the work\_area.
    - (d) Read LINKTAB of the object module.
    - (e) Enter (object module name,  $\text{program\_linked\_origin}$ ) in NTAB.
    - (f) For each LINKTAB entry with type = PD  
 $\text{name} := \text{symbol field of the LINKTAB entry};$   
 $\text{linked\_address} := \text{translated\_address} + \text{relocation\_factor};$   
 Enter (name, linked\_address) in a new entry of the NTAB.
    - (g)  $\text{program\_linked\_origin} := \text{program\_linked\_origin} + \text{OM\_size};$
  3. For each object module mentioned in the linker command
    - (a)  $\text{t\_origin} := \text{translated origin of the object module};$   
 $\text{program\_linked\_origin} := \text{linked\_address from NTAB};$
    - (b) For each LINKTAB entry with type = EXT
      - i.  $\text{address\_in\_work\_area} := \text{address of work\_area} + \text{program\_linked\_origin} - \text{<link origin> in linker command} + \text{translated address} - \text{t\_origin};$

- ii. Search the symbol found in the symbol field of the LINKTAB entry in NTAB and note its linked address. Copy this address into the operand address field in the word that has the address `address_in_work_area`.

### Self-Relocating Programs

- A self relocating program is a program which can perform the relocation of its own address sensitive instructions.
- It contains the following two provisions for this purpose:
  - A table of information concerning the address sensitive instructions exists as a part of the program.
  - Code to perform the relocation of address sensitive instructions also exists as a part of the program. This is called the relocating logic.
- The start address of the relocating logic is specified as the execution start address of the program.
- Thus the relocating logic gains control when the program is loaded in memory for the execution.
- It uses the load address and the information concerning address sensitive instructions to perform its own relocation.
- Execution control is now transferred to the relocated program.
- A self –relocating program can execute in any area of the memory.
- This is very important in time sharing operating systems where the load address of a program is likely to be different for different executions.

### Linking in MSDOS

- We discuss the design of a linker for the Intel 8088/80x86 processors which resembles LINK of MS DOS in many respects.
- It may be noted that the object modules of MS DOS differ from the Intel specifications in some respects.

#### Object Module Format (Explain object module of the program)

- An Intel 8088 object module is a sequence of object records, each object record describing specific aspects of the programs in the object module.
- There are 14 types of object records containing the following five basic categories of information:
  - Binary image (i.e. code generated by a translator)
  - External references
  - Public definitions
  - Debugging information (e.g. line number in source program).
  - Miscellaneous information (e.g. comments in the source program).

- We only consider the object records corresponding to first three categories-a total of eight object record types.
- Each object record contains variable length information and may refer to the contents of previous object records.
- Each name in an object record is represented in the following format:

length( 1 byte)	name
-----------------	------

### THEADR, LNames and SEGDEF records

#### THEADR record

80H	length	T-module name	check-sum
-----	--------	---------------	-----------

- The module name in the THEADR record is typically derived by the translator from the source file name.
- This name is used by the linker to report errors.
- An assembly programmer can specify the module name in the NAME directive.

#### LNames record

96H	length	name-list	check-sum
-----	--------	-----------	-----------

- The LNames record lists the names for use by SEGDEF records.

#### SEGDEF record

98H	length	attributes (1-4)	segment length (2)	name index (1)	check-sum
-----	--------	---------------------	--------------------------	----------------------	-----------

- A SEGDEF record designates a segment name using an index into this list.
- The attributes field of a SEGDEF record indicates whether the segment is relocatable or absolute, whether (and in what manner) it can be combined with other segments, as also the alignment requirement of its base address (e.g. byte, word or paragraph, i.e. 16 byte, alignment).
- Stack segments with the same name are concatenated with each other, while common segments with the same name are overlapped with one another.
- The attribute field also contains the origin specification for an absolute segment.

#### EXTDEF and PUBDEF record

8CH	length	external list	reference	check-sum
-----	--------	------------------	-----------	-----------

90H	length	base (2-4)	name	offset	...	check-sum
-----	--------	---------------	------	--------	-----	-----------

- The EXTDEF record contains a list of external references used by the programs of this module.
- A FIXUPP record designates an external symbol name by using an index into this list.
- A PUBDEF record contains a list of public names declared in a segment of the object module.
- The base specification identifies the segment.

- Each (name, offset) pair in the record defines one public name, specifying the name of the symbol and its offset within the segment designated by the base specification.

### LEDATA records

A0H	length	segment index (1-2)	data offset (2)	data	check-sum
-----	--------	---------------------------	--------------------	------	-----------

- An LEDATA record contains the binary image of the code generated by the language translator.
- Segment index identifies the segment to which the code belongs, and offset specifies the location of the code within the segment.

### FIXUPP record

9CH	length	locat (1)	fix dat (1)	frame datum (1)	target datum (1)	target offset (2)	...	check- sum
-----	--------	--------------	-------------------	-----------------------	------------------------	-------------------------	-----	---------------

- A FIXUPP record contains information for one or more relocation and linking fixups to be performed.
- The locat field contains a numeric code called loc code to indicate the type of a fixup.
- The meanings of these codes are given in Table

Loc code	Meaning
0	Low order byte is to be fixed.
1	Offset is to be fixed.
2	Segment is to be fixed.
3	Pointer (i.e., segment: offset) is to be fixed.

- locat also contains the offset of the fixup location in the previous LEDATA record.
- The frame datum field, which refers to a SEGDEF record, identifies the segment to which the fixup location belongs.
- The target datum and target offset fields specify the relocation or linking information.
- Target datum contains a segment index or an external index, while target offset contains an offset from the name indicated in target datum.
- The fix dat field indicates the manner in which the target datum and target offset fields are to be interpreted.
- The numeric codes used for this purpose are given in below table.

Code	contents of target datum and offset fields
0	Segment index and displacement.
2	External index and target displacement.
4	Segment index (offset field is not used).
6	External index (offset field is not used).



### MODEND record

8AH	length	type (1)	start addr (5)	check-sum
-----	--------	-------------	-------------------	-----------

- The MODEND record signifies the end of the module, with the type field indicating whether it is the main program.
- This record also optionally indicates the execution start address.
- This has two components: (a) the segment, designated as an index into the list of segment names defined in SEGDEF record(s), and (b) an offset within the segment.

### Linking of Overlay Structured Programs

- An overlay is part of a program (or software package) which has the same load origin as some other part of the program.
- Overlay is used to reduce the main memory requirement of a program.

#### Overlay structured program

- We refer to a program containing overlays as an overlay structured program. Such a program consists of
  - A permanently resident portion, called the root.
  - A set of overlays.
- Execution of an overlay structured program proceeds as follows:
- To start with, the root is loaded in memory and given control for the purpose of execution.
- Other overlays are loaded as and when needed.
- Note that the loading of an overlay overwrites a previously loaded overlay with the same load origin.
- This reduces the memory requirement of a program.
- It also makes it possible to execute programs whose size exceeds the amount of memory which can be allocated to them.
- The overlay structure of a program is designed by identifying mutually exclusive modules that is, modules which do not call each other.
- Such modules do not need to reside simultaneously in memory.

#### Execution of an overlay structured program

- For linking and execution of an overlay structured program in MS DOS the linker produces a single executable file at the output, which contains two provisions to support overlays.
- First, an overlay manager module is included in the executable file.
- This module is responsible for loading the overlays when needed.
- Second, all calls that cross overlay boundaries are replaced by an interrupt producing instruction.
- To start with, the overlay manager receives control and loads the root.
- A procedure call which crosses overlay boundaries leads to an interrupt.

- This interrupt is processed by the overlay manager and the appropriate overlay is loaded into memory.
- When each overlay is structured into a separate binary program, as in IBM mainframe systems, a call which crosses overlay boundaries leads to an interrupt which is attended by the OS kernel.
- Control is now transferred to the OS loader to load the appropriate binary program.

### Dynamic Linking

#### Static Linking

- In static linking, the linker links all modules of a program before its execution begins; it produces a binary program that does not contain any unresolved external references.
- If statically linked programs use the same module from a library, each program will get a private copy of the module.
- If many programs that use the module are in execution at the same time, many copies of the module might be present in memory.

#### Dynamic Linking

- Dynamic linking is performed during execution of a binary program.
- The linker is invoked when an unresolved external reference and resumes execution of the program.
- This arrangement has several benefits concerning use, sharing and updating of library modules.
- If the module referenced by a program has already been linked to another program that is in execution, a copy of the module would exist in memory. The same copy of the module could be linked to his program as well, thus saving memory.
- To facilitate dynamic linking, each program is first processed by the static linker.
- The static linker links each external reference in the program to a dummy module whose

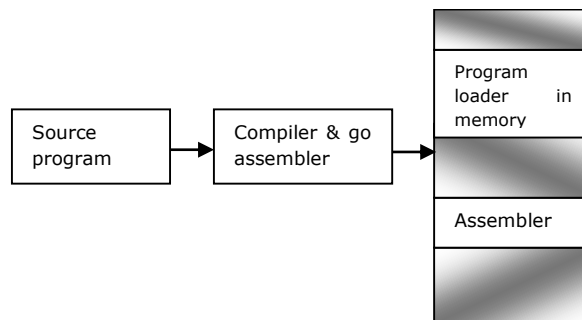
### Different Loading Schemes

#### Compile-and-Go Loaders

- Assembler is loaded in one part of memory and assembled program directly into their assigned memory location
- After the loading process is complete, the assembler transfers the control to the starting instruction of the loaded program.

#### Advantages

- The user need not be concerned with the separate steps of compilation, assembling, linking, loading, and executing.
- Execution speed is generally much superior to interpreted systems.
- They are simple and easier to implement.



### Disadvantages

- There is wastage in memory space due to the presence of the assembler.
- The code must be reprocessed every time it is run.

### General Loader Schemes

- The general loading scheme improves the compile/assemble-and-go scheme by allowing different source programs (or modules of the same program) to be translated separately into their respective object programs.
- The object code (modules) is stored in the secondary storage area; and then, they are loaded.
- The loader usually combines the object codes and executes them by loading them into the memory, including the space where the assembler had been in the assemble-and-go scheme.
- Rather than the entire assembler sitting in the memory, a small utility component called loader does the job.
- Note that the loader program is comparatively much smaller than the assembler, hence making more space available to the user for their programs.
- Advantages of the general loading scheme:
  - Saves memory and makes it available for the user program as loaders are smaller in size than assemblers. The loader replaces the assembler.
  - Reassembly of the program is no more needed for later execution of the program. The object file/deck is available and can be loaded and executed directly at the desired location.
  - This scheme allows use of subroutines in several different languages because the objectfiles processed by the loader utility will all be in machine language.
- Disadvantages of the general loading scheme:
  - The loader is more complicated and needs to manage multiple object files.
  - Secondary storage is required to store object files, and they cannot be directly placed into the memory by assemblers.

### Absolute Loaders

- An absolute loader loads a binary program in memory for execution.
- The binary program is stored in a file contains the following:
  - A Header record showing the load origin, length and load time execution start address of the program.

- A sequence of binary image records containing the program's code. Each binary image record contains a part of the program's code in the form of a sequence of bytes, the load address of the first byte of this code and a count of the number of bytes of code.
- The absolute loader notes the load origin and the length of the program mentioned in the header record.
- It then enters a loop that reads a binary image record and moves the code contained in it to the memory area starting on the address mentioned in the binary image record.
- At the end, it transfers control to the execution start address of the program.
- Advantages of the absolute loading scheme:
  - Simple to implement and efficient in execution.
  - Saves the memory (core) because the size of the loader is smaller than that of the assembler.
  - Allows use of multi-source programs written in different languages. In such cases, the given language assembler converts the source program into the language, and a common object file is then prepared by address resolution.
  - The loader is simpler and just obeys the instruction regarding where to place the object code in the main memory.
- Disadvantages of the absolute loading scheme:
  - The programmer must know and clearly specify to the translator (the assembler) the address in the memory for inner-linking and loading of the programs. Care should be taken so that the addresses do not overlap.
  - For programs with multiple subroutines, the programmer must remember the absolute address of each subroutine and use it explicitly in other subroutines to perform linking.
  - If the subroutine is modified, the program has to be assembled again from first to last.

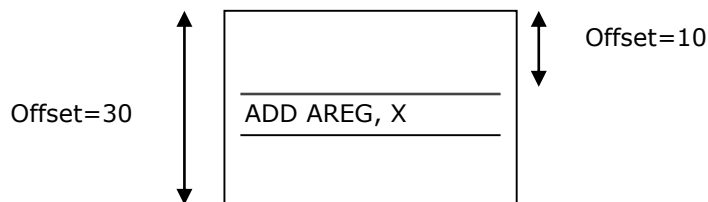
### Relocating Loaders

- A relocating loader loads a program in a designated area of memory, relocates it so that it can execute correctly in that area of memory and passes control to it for execution.
- The binary program is stored in a file contains the following:
  - A Header record showing the load origin, length and load time execution start address of the program.
  - A sequence of binary image records containing the program's code. Each binary image record contains a part of the program's code in the form of a sequence of bytes, the load address of the first byte of this code and a count of the number of bytes of code.
  - A table analogous to RELOCTAB table giving linked addresses of address sensitive instructions in the program.
- **Algorithm: Relocating Loader**
  1. Program\_load\_origin = load origin specified in the loader command
  2. program\_linked\_origin = linked origin specified in the header record
  3. relocation\_factor = program\_load\_origin – program\_linked\_origin
  4. For each binary image record
    - a. code\_linked\_address = linked address specified in the record

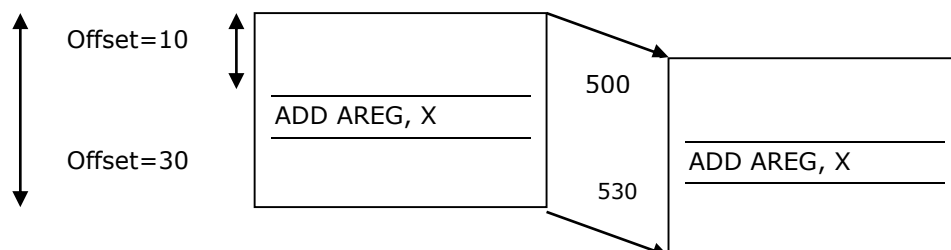
- b.  $\text{code\_load\_address} = \text{code\_linked\_address} + \text{relocation\_factor}$
  - c.  $\text{byte\_count} = \text{count of the number of bytes in the record}$
  - d. Move  $\text{byte\_count}$  bytes from the record to the memory area with start address  $\text{code\_load\_address}$
5. Read RELOCTAB of the program
6. For each entry in the RELOCTAB
  - a.  $\text{Instruction\_linked\_address} = \text{address specified in the RELOCTAB entry}$
  - b.  $\text{instruction\_load\_address} = \text{instruction\_linked\_address} + \text{relocation\_factor}$
  - c. Add  $\text{relocation\_factor}$  to the operand address used in the instruction that has the address  $\text{instruction\_load\_address}$

### Practical Relocating Loaders

- To avoid possible assembling of all subroutine when a single subroutine is changed and to perform task of allocation and linking for the programmer, the general class of relocating loader was introduced.
- Binary symbolic loader (BSS) is an example of relocating loader.
- The output of assembler using BSS loader is
  - Object program
  - Reference about other program to be accessed
  - Information about address sensitive entities.
- Let us consider a program segment as shown below



- In the above program the address of variable X in the instruction ADD AREG, X will be 30
- If this program is loaded from the memory location 500 for execution then the address of X in the instruction ADD AREG, X must become 530.



- Use of segment register makes a program address insensitive
- Actual address is given by content of segment register + address of operand in instruction
- So,  $500 + 30 = 530$  is actual address of variable X.

### Linking Loaders

- A modern program comprises several procedures or subroutines together with the main program module.
- The translator, in such cases as a compiler, will translate them all independently into distinct object modules usually stored in the secondary memory.
- Execution of the program in such cases is performed by linking together these independent object modules and loading them into the main memory.
- Linking of various object modules is done by the linker.
- Special system program called linking loader gathers various object modules, links them together to produce single executable binary program and loads them into the memory.
- This category of loaders leads to a popular class of loaders called direct-linking loaders.
- The loaders used in these situations are usually called linking loaders, which link the necessary library functions and symbolic references.
- Essentially, linking loaders accept and link together a set of object programs and a single file to load them into the core.
- Linking loaders additionally perform relocation and overcome disadvantages of other loading schemes.

### Relocating Linking Loaders

- Relocating linking loaders combines together the relocating capabilities of relocating loaders and the advanced linking features of linking loaders and presents a more robust loading scheme.
- This necessarily eliminates the need to use two separate programs for linking and loading respectively.
- These loaders can perform relocation and linking both.
- These types of loaders are especially useful in dynamic runtime environment, wherein the link and load origins are highly dependent upon the runtime situations.
- These loaders can work efficiently with support from the operating system and utilize the memory and other resources efficiently.

### Programming Language Grammars

- **Terminal symbol:** A symbol in the alphabet is known as a terminal symbol.
- **Alphabet:** The alphabet of a language  $L$  is the collection of graphic symbols such as letters and punctuation marks used in  $L$ . It is denoted by Greek symbol  $\Sigma$ . Eg.  $\Sigma = \{a, b, \dots, z, 0, 1, \dots, 9\}$
- **String:** A string is a finite sequence of symbols. It is denoted by Greek symbols  $\alpha, \beta$ , etc.
- **Nonterminal symbol:** A nonterminal symbol is the name of syntax category of a language, e.g., noun, verb, etc. A nonterminal symbol is written as a single capital letter, or as a name enclosed between  $\langle \dots \rangle$ , e.g.,  $A$  or  $\langle \text{Noun} \rangle$ . During grammatical analysis, a nonterminal symbol represents an instance of the category.
- **Production:** A production, also called a rewriting rule, is a rule of grammar. It has the form  

$$\text{A nonterminal symbol} \rightarrow \text{String of terminal and nonterminal symbols}$$
 where the notation ' $\rightarrow$ ' stands for 'is defined as'. Eg.  $\langle \text{Noun Phrase} \rangle \rightarrow \langle \text{Article} \rangle \langle \text{Noun} \rangle$
- **Grammar:** A grammar  $G$  of a language  $L_G$  is a quadruple  $(\Sigma, \text{SNT}, S, P)$  where
  - $\Sigma$  is the alphabet of  $L_G$ , i.e. the set of terminal symbols
  - $\text{SNT}$  is the set of nonterminal symbols
  - $S$  is the distinguished symbol (Start symbol)
  - $P$  is the set of productions

Eg.  $\langle \text{Noun Phrase} \rangle \rightarrow \langle \text{Article} \rangle \langle \text{Noun} \rangle$

$\langle \text{Article} \rangle \rightarrow a \mid \text{an} \mid \text{the}$

$\langle \text{Noun} \rangle \rightarrow \text{boy} \mid \text{apple}$

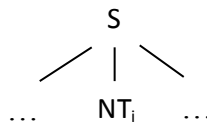
- **Derivation:** Let production  $P_1$  of grammar  $G$  be of the form  $P_1: A \rightarrow \alpha$  and let  $\beta$  be a string such that  $\beta \Rightarrow \gamma A \Theta$ , then replacement of  $A$  by  $\alpha$  in string  $\beta$  constitutes a derivation according to production  $P_1$ . It yields the string  $\gamma \alpha \Theta$ .

$$\begin{aligned} \langle \text{Noun Phrase} \rangle &\Rightarrow \langle \text{Article} \rangle \langle \text{Noun} \rangle \\ &\Rightarrow \text{the } \langle \text{Noun} \rangle \\ &\Rightarrow \text{the boy} \end{aligned}$$

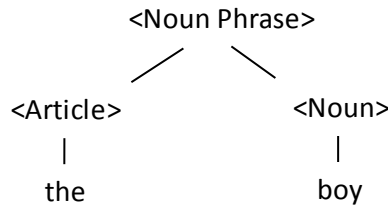
- **Reduction:** Let production  $P_1$  of grammar  $G$  be of the form  $P_1: A \rightarrow \alpha$  and let  $\sigma$  be a string such that  $\sigma \Rightarrow \gamma \alpha \Theta$ , then replacement of  $\alpha$  by  $A$  in string  $\sigma$  constitutes a reduction according to production  $P_1$ .

Step	String
0	the boy
1	$\langle \text{Article} \rangle \text{boy}$
2	$\langle \text{Article} \rangle \langle \text{Noun} \rangle$
3	$\langle \text{Noun Phrase} \rangle$

- **Parse trees:** The tree representation of the sequence of derivations that produces a string from the distinguished (start) symbol is termed as parse tree.



Eg.



## Classification of Grammar

### Type-0 grammars

These grammars are known as phrase structure grammars. Their productions are of the form

$$\alpha \rightarrow \beta$$

where both  $\alpha$  and  $\beta$  can be strings of terminal and nonterminal symbols. Such productions permit arbitrary substitution of strings during derivation or reduction, hence they are not relevant to specification of programming languages.

### Type-1 grammars

Productions of Type-1 grammars specify that derivation or reduction of strings can take place only in specific contexts. Hence these grammars are also known as context sensitive grammars. A production of a Type-1 grammar has the form

$$\alpha A \beta \rightarrow \alpha \pi \beta$$

Here, a string  $\pi$  can be replaced by 'A' (or vice versa) only when it is enclosed by the strings  $\alpha$  and  $\beta$  in a sentential form. These grammars are also not relevant for programming language specification since recognition of programming language constructs is not context sensitive in nature.

### Type-2 grammars

These grammars do not impose any context requirements on derivations or reductions. A typical Type-2 production is of the form

$$A \rightarrow \pi$$

which can be applied independent of its context. These grammars are therefore known as context free grammars (CFG). CFGs are ideally suited for programming language specification.

### Type-3 grammars

Type-3 grammars are characterized by productions of the form

$$A \rightarrow tB|t \text{ or } A \rightarrow Bt|t$$

Note that these productions also satisfy the requirements of type-2 grammars. The specific form of the RHS alternatives—namely a single nonterminal symbol or a string containing a single terminal symbol and a single nonterminal symbol, gives some practical advantages in scanning. However, the nature of the



productions restricts the expressive power of these grammars, e.g., nesting of constructs or matching of parentheses cannot be specified using such productions. Hence the use of Type-3 productions is restricted to the specification of lexical units, e.g., identifiers, constants, labels, etc. Type-3 grammars are also known as linear grammars or regular grammars.

### Operator grammars

Productions of an operator grammar do not contain two or more consecutive nonterminal symbols in any RHS alternative. Thus, nonterminal symbols occurring in an RHS string are separated by one or more terminal symbols.

Eg.

```
<exp> → <exp> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → <factor> ↑ <primary> | <primary>
<primary> → <id> | <constant> | (<exp>)
<id> → <letter> | <id> <letter> | <id> <digit>
<const> → [+|-] <digit> | <const> <digit>
<letter> → a|b|c|...|z
<digit> → 0|1|2|3|4|5|6|7|8|9
```

### Ambiguous Grammar

- A grammar is ambiguous if a string can be interpreted in two or more ways by using it.
- In natural languages, ambiguity may concern the meaning of a word, the syntax category of a word, or the syntactic structure of a construct. For example, a word can have multiple meanings or it can be both a noun and a verb and a sentence can have more than one syntactic.
- In a formal language grammar, ambiguity would arise if identical strings can occur on the RHS of two or more productions. For example, if a grammar has the

$$N_1 \rightarrow \alpha$$

$$N_2 \rightarrow \alpha$$

- The string  $\alpha$  can be derived from or reduced to either  $N_1$  or  $N_2$ .
- Ambiguity at the level of the syntactic structure of a string would mean that more than one parse tree can be built for the string.
- Eg.

```
<exp> → <id> | <exp> + <exp> | <exp> * <exp>
<id> → a | b | c
```

Two parse trees can be built for the source string  $a+b*c$  according to this grammar – one in which  $a+b$  is first reduced to  $\langle \text{exp} \rangle$  and another in which  $b*c$  is first reduced to  $\langle \text{exp} \rangle$ .

Eliminating ambiguity in the above grammar can be rewritten as

```
<exp> → <exp> + <term> | <term>
<term> → <term> * <id> | <id>
<id> → a | b | c
```

### Scanning

- **Finite state automaton (FSA):** A finite state automaton is a triple  $(S, \Sigma, T)$  where
  - $S$  is a finite set of states, one of which is the initial state  $s_{init}$ , and one or more of which are the final states
  - $\Sigma$  is the alphabet of source symbols
  - $T$  is a finite set of state transitions defining transitions out of states in  $S$  on encountering symbols in  $\Sigma$ .
- **Deterministic finite state automaton:** It is a finite state automaton none of whose states has two or more transitions for the same source symbol. The DFA has the property that it reaches a unique state for every source string input to it.
- **Regular Expression:** A regular expression is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations.
- **Give the regular expressions for the following:**
  1. 0 or 1  
 $0+1$
  2. 0 or 11 or 111  
 $0+11+111$
  3. Regular expression over  $\Sigma=\{a,b,c\}$  that represent all string of length 3.  
 $(a+b+c)(a+b+c)(a+b+c)$
  4. String having zero or more a.  
 $a^*$
  5. String having one or more a.  
 $a^+$
  6. All binary string.  
 $(0+1)^*$
  7. 0 or more occurrence of either a or b or both  
 $(a+b)^*$
  8. 1 or more occurrence of either a or b or both  
 $(a+b)^+$
  9. Binary no. end with 0  
 $(0+1)^*0$
  10. Binary no. end with 1  
 $(0+1)^*1$
  11. Binary no. starts and end with 1.  
 $1(0+1)^*1$
  12. String starts and ends with same character.  
 $0(0+1)^*0$       or       $a(a+b)^*a$   
 $1(0+1)^*1$        $b(a+b)^*b$
  13. All string of a and b starting with a  
 $a(a/b)^*$
  14. String of 0 and 1 end with 00.

- $(0+1)^*00$
15. String end with abb.  
 $(a+b)^*abb$
16. String start with 1 and end with 0.  
 $1(0+1)^*0$
17. All binary string with at least 3 characters and 3rd character should be zero.  
 $(0+1)(0+1)0(0+1)^*$
18. Language which consist of exactly two b's over the set  $\Sigma=\{a,b\}$   
 $a^*ba^*ba^*$
19.  $\Sigma=\{a,b\}$  such that 3rd character from right end of the string is always a.  
 $(a+b)^*a(a+b)(a+b)$
20. Any no. of a followed by any no. of b followed by any no. of c.  
 $a^*b^*c^*$
21. It should contain at least 3 one.  
 $(0+1)^*1(0+1)^*1(0+1)^*1(0+1)^*$
22. String should contain exactly Two 1's  
 $0^*10^*10^*$
23. Length should be at least be 1 and at most 3.  
 $(0+1) + (0+1) (0+1) + (0+1) (0+1) (0+1)$
24. No.of zero should be multiple of 3  
 $(1^*01^*01^*01^*)^*+1^*$
25.  $\Sigma=\{a,b,c\}$  where a are multiple of 3.  
 $((b+c)^*a (b+c)^*a (b+c)^*a (b+c)^*)^*$
26. Even no. of 0.  
 $(1^*01^*01^*)^*$
27. Odd no. of 1.  
 $0^*(10^*10^*)^*10^*$
28. String should have odd length.  
 $(0+1)((0+1)(0+1))^*$
29. String should have even length.  
 $((0+1)(0+1))^*$
30. String start with 0 and has odd length.  
 $0((0+1)(0+1))^*$
31. String start with 1 and has even length.  
 $1(0+1)((0+1)(0+1))^*$
32. Even no of 1  
 $(0^*10^*10^*)^*$
33. String of length 6 or less  
 $(0+1+^)^6$
34. String ending with 1 and not contain 00.  
 $(1+01)^+$
35. All string begins or ends with 00 or 11.  
 $(00+11)(0+1)^*+(0+1)^*(00+11)$
36. All string not contains the substring 00.  
 $(1+01)^* (^+0)$
37. Language of all string containing both 11 and 00 as substring.

$((0+1)^*00(0+1)^*11(0+1)^*) + ((0+1)^*11(0+1)^*00(0+1)^*)$   
38. Language of C identifier.  
 $(\_+L)(\_+L+D)^*$

## Regular Expression to DFA Conversion

$(a+b)^*abb$

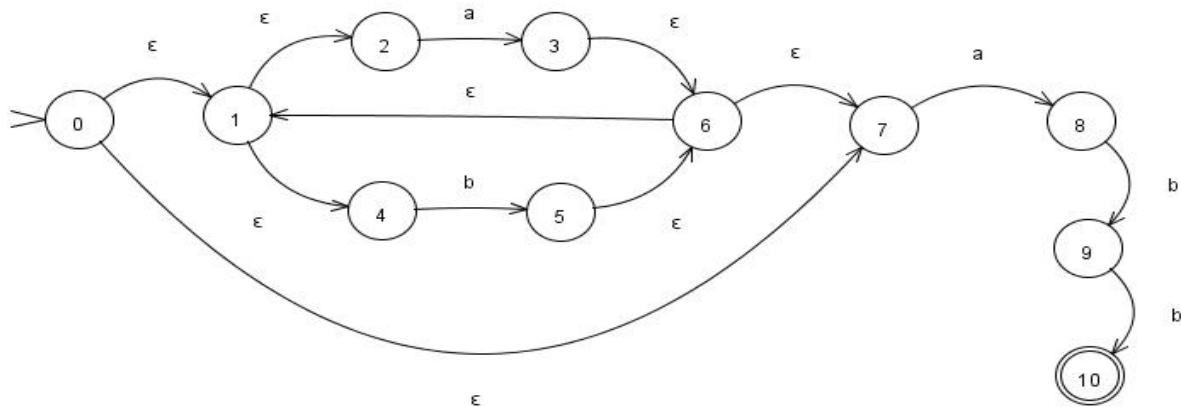


Figure 6.1: NFA for  $(a+b)^*abb$

- $\epsilon$  – closure (0) = {0,1,2,4,7} ---- Let A
- $\text{Move}(A,a) = \{3,8\}$   
 $\epsilon$  – closure ( $\text{Move}(A,a)$ ) = {1,2,3,4,6,7,8}---- Let B  
 $\text{Move}(A,b) = \{5\}$   
 $\epsilon$  – closure ( $\text{Move}(A,b)$ ) = {1,2,4,5,6,7}---- Let C
- $\text{Move}(B,a) = \{3,8\}$   
 $\epsilon$  – closure ( $\text{Move}(B,a)$ ) = {1,2,3,4,6,7,8}---- B  
 $\text{Move}(B,b) = \{5,9\}$   
 $\epsilon$  – closure ( $\text{Move}(B,b)$ ) = {1,2,4,5,6,7,9}---- Let D
- $\text{Move}(C,a) = \{3,8\}$   
 $\epsilon$  – closure ( $\text{Move}(C,a)$ ) = {1,2,3,4,6,7,8}---- B  
 $\text{Move}(C,b) = \{5\}$   
 $\epsilon$  – closure ( $\text{Move}(C,b)$ ) = {1,2,4,5,6,7}---- C
- $\text{Move}(D,a) = \{3,8\}$   
 $\epsilon$  – closure ( $\text{Move}(D,a)$ ) = {1,2,3,4,6,7,8}---- B  
 $\text{Move}(D,b) = \{5,10\}$   
 $\epsilon$  – closure ( $\text{Move}(D,b)$ ) = {1,2,4,5,6,7,10}---- Let E
- $\text{Move}(E,a) = \{3,8\}$   
 $\epsilon$  – closure ( $\text{Move}(E,a)$ ) = {1,2,3,4,6,7,8}---- B  
 $\text{Move}(E,b) = \{5\}$   
 $\epsilon$  – closure ( $\text{Move}(E,b)$ ) = {1,2,4,5,6,7}---- C

States	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Table 6.1: Transition Table

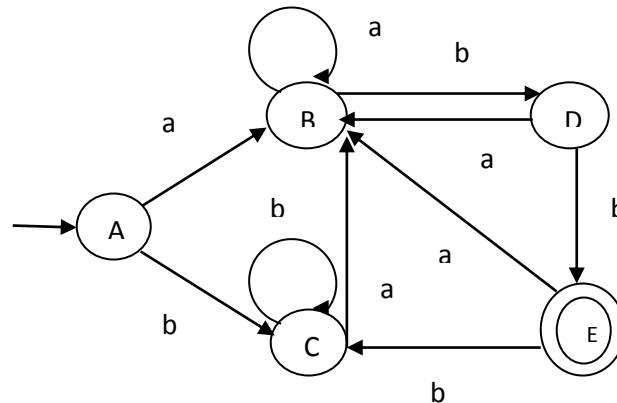


Figure 6.2: DFA for  $(a+b)^*abb$

**Note:** For more examples refer separate notes has been attached.

### Parsing

- This top-down parsing is non-recursive. LL(1) – the first L indicates input is scanned from left to right. The second L means it uses leftmost derivation for input string and 1 means it uses only input symbol to predict the parsing process.
- The data structure used by LL(1) parser are input buffer, stack and parsing table.
- The parser works as follows,
- The parsing program reads top of the stack and a current input symbol. With the help of these two symbols parsing action can be determined.
- The parser consult the LL(1) parsing table each time while taking the parsing actions hence this type of parsing method is also called table driven parsing method.
- The input is successfully parsed if the parser reaches the halting configuration. When the stack is empty and next token is \$ then it corresponds to successful parsing.
- Steps to construct LL(1) parser
  1. Remove left recursion / Perform left factoring.
  2. Compute FIRST and FOLLOW of nonterminals.
  3. Construct predictive parsing table.
  4. Parse the input string with the help of parsing table.

**Example:**

$E \rightarrow E+T/T$

$T \rightarrow T * F / F$

$F \rightarrow (E)/id$

**Step1:** Remove left recursion

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

**Step2:** Compute FIRST & FOLLOW

	FIRST	FOLLOW
E	{(,id}	{\$,)}
E'	{+, $\epsilon$ }	{\$,)}
T	{(,id}	{+,\$,)}
T'	{*, $\epsilon$ }	{+,\$,)}
F	{(,id}	{*,+,\$,)}

Table 6.2 First & Follow set

**Step3:** Predictive Parsing Table

	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Table 6.3: Predictive parsing table

**Step4:** Parse the string

Stack	Input	Action
\$E	id+id*id\$	
\$E'T	id+id*id\$	$E \rightarrow TE'$
\$E'T'F	id+id*id\$	$T \rightarrow FT'$
\$E'T'id	id+id*id\$	$F \rightarrow id$
\$E'T'	+id*id\$	
\$E'	+id*id\$	$T' \rightarrow \epsilon$
\$E'T+	+id*id\$	$E' \rightarrow +TE'$
\$E'T	id*id\$	
\$E'T'F	id*id\$	$T \rightarrow FT'$
\$E'T'id	id*id\$	$F \rightarrow id$
\$E'T'	*id\$	
\$E'T'F*	*id\$	$T' \rightarrow *FT'$
\$E'T'F	id\$	
\$E'T'id	id\$	$F \rightarrow id$
\$E'T'	\$	

\$ E'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

Table 6.4: Moves made by predictive parse

### Bottom Up Parsing

- **Handle:** A “handle” of a string is a substring of the string that matches the right side of a production, and whose reduction to the nonterminal of the production is one step along the reverse of rightmost derivation.
- **Handle pruning:** The process of discovering a handle and reducing it to appropriate Left hand side non terminal is known as handle pruning.

Right sentential form	Handle	Reducing production
id1+id2*id3	id1	$E \rightarrow id$
E+id2*id3	id2	$E \rightarrow id$
E+E*id3	id3	$E \rightarrow id$
E+E*E	E*E	$E \rightarrow E*E$
E+E	E+E	$E \rightarrow E+E$
E		

Table 6.5: Handle and Handle Pruning

### Shift Reduce Parsing

- The shift reduce parser performs following basic operations,
- **Shift:** Moving of the symbols from input buffer onto the stack, this action is called shift.
- **Reduce:** If handle appears on the top of the stack then reduction of it by appropriate rule is done. This action is called reduce action.
- **Accept:** If stack contains start symbol only and input buffer is empty at the same time then that action is called accept.
- **Error:** A situation in which parser cannot either shift or reduce the symbols, it cannot even perform accept action then it is called error action.

Example: Consider the following grammar,

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

Perform shift reduce parsing for string id + id \* id.

Stack	Input buffer	Action
\$	id+id*id\$	Shift
\$id	+id*id\$	Reduce $F \rightarrow id$
\$F	+id*id\$	Reduce $T \rightarrow F$
\$T	+id*id\$	Reduce $E \rightarrow T$
\$E	+id*id\$	Shift
\$E+	id*id\$	shift

\$E+ id	*id\$	Reduce F->id
\$E+F	*id\$	Reduce T->F
\$E+T	*id\$	Shift
\$E+T*	id\$	Shift
\$E+T*id	\$	Reduce F->id
\$E+T*F	\$	Reduce T->T*F
\$E+T	\$	Reduce E->E+T
\$E	\$	Accept

Table 6.6: Configuration of shift reduce parser on input id + id\*id

### Operator Precedence Parsing

- **Operator Grammar:** A Grammar in which there is no  $\epsilon$  in RHS of any production or no adjacent non terminals is called operator precedence grammar.
- In operator precedence parsing, we define three disjoint precedence relations  $<$ ,  $>$  and  $=$  between certain pair of terminals.

Relation	Meaning
$a < b$	a “yields precedence to” b
$a = b$	a “has the same precedence as” b
$a > b$	a “takes precedence over” b

Table 6.7: Precedence between terminal a & b

#### Leading:

Leading of a nonterminal is the first terminal or operator in production of that nonterminal.

#### Trailing:

Trailing of a nonterminal is the last terminal or operator in production of that nonterminal

#### Example:

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow id$

**Step-1:** Find leading and trailing of NT.

Leading

$(E) = \{+, *, id\}$

$(T) = \{*, id\}$

$(F) = \{id\}$

Trailing

$(E) = \{+, *, id\}$

$(T) = \{*, id\}$

$(F) = \{id\}$

**Step-2:** Establish Relation

1.  $a < b$

$Op \cdot NT \rightarrow Op < \text{Leading}(NT)$

$+T \quad + < \{*, id\}$

$*F \quad * < \{id\}$

2.  $a > b$



$NT \cdot Op \rightarrow \text{Trailing}(NT) \cdot > Op$

$E+ \quad \{+, *, id\} \cdot > +$

$T* \quad \{*, id\} \cdot > *$

3.  $\$ < \{+, *, id\}$

4.  $\{+, *, id\} \cdot > \$$

**Step-3:** Creation of table

	+	*	id	\$
+	'>	<'	<'	'>
*	'>	'>	<'	'>
id	'>	'>		'>
\$	<'	<'	<'	

**Table 6.8: Precedence table**

**Step-4:** Parsing of the string  $\langle id \rangle + \langle id \rangle * \langle id \rangle$  using precedence table.

We will follow following steps to parse the given string,

1. Scan the input string until first  $>$  is encountered.
2. Scan backward until  $<$  is encountered.
3. The handle is string between  $<$  and  $>$ .

$\$ < id > + < id > * < id > \$$	Handle id is obtained between $< \cdot >$ Reduce this by $F \rightarrow id$
$\$ F + < id > * < id > \$$	Handle id is obtained between $< \cdot >$ Reduce this by $F \rightarrow id$
$\$ F + F * < id > \$$	Handle id is obtained between $< \cdot >$ Reduce this by $F \rightarrow id$
$\$ F + F * F \$$	Perform appropriate reductions of all nonterminals.
$\$ E + T * F \$$	Remove all nonterminal
$\$ + * \$$	Place relation between operators
$\$ < + < * > \$$	The $*$ operator is surrounded by $< \cdot >$ . This indicates $*$ becomes handle we have to reduce $T * F$ .
$\$ < + > \$$	$+$ becomes handle. Hence reduce $E + T$ .
$\$ \$$	Parsing Done

**Table 6.9: Moves for parsing  $\langle id \rangle + \langle id \rangle * \langle id \rangle$**

### Operator Precedence Parsing Algorithm using Stack

**Data Structure:**

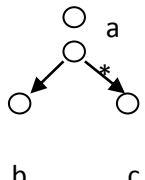
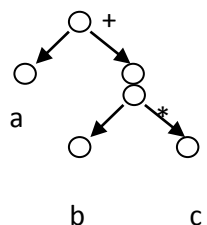
Stack: each stack entry is a record with two fields, *operator* and *operand\_pointer*

Node: a node is a record with three fields *symbol*, *left\_pointer*, and *right\_pointer*.

**Functions:** *Newnode (operator, l\_operand\_pointer, r\_operand\_pointer)* creates a node with appropriate

pointer fields and returns a pointer to the node.

1. TOS:= SB-1; SSM=0;
2. Push '|-' on the stack.
3. ssm=ssm+1;
4. x:=newnode(source symbol, null, null)  
TOS.operand\_pointer:=x;  
Go to step 3;
5. While TOS operator .> current operator,  
x:=newnode(TOS operator, TOSM.operand\_pointer, TOS.operand\_pointer)  
pop an entry of the stack;  
TOS.operand\_pointer:=x;
6. If TOS operator <. current operator, then  
Push the current operator on the stack.  
Go to step 3;
7. While TOS operator .= current operator, then  
if TOS operator = |-- then exit successfully  
if TOS operator ='(', then  
temp:=TOS.operand\_pointer;  
pop an entry off the stack  
TOS.operand\_pointer:=temp;  
Go to step 3;
8. If no precedence define between TOS operator and current operator the report error and exit unsuccessfully.

	Current Operator	Stack		AST		
(a)	' + '	SB,TOS	<table><tr><td>  -</td><td></td></tr></table> →	-		a ○
-						
(b)	' * '	SB	<table><tr><td>  -</td><td></td></tr></table> →	-		○
		-				
TOS	<table><tr><td>+</td><td></td></tr></table> →	+		○ b		
+						
(c)	' -  '	SB	<table><tr><td>  -</td><td></td></tr></table> →	-		○
		-				
		TOS	<table><tr><td>+</td><td></td></tr></table> →	+		○ b
+						
TOS	<table><tr><td>*</td><td></td></tr></table> →	*		○ c		
*						
(d)	' -  '	SB	<table><tr><td>  -</td><td></td></tr></table> →	-		
		-				
TOS	<table><tr><td>+</td><td></td></tr></table> →	+				
+						
(e)	' -  '	SB,TOS	<table><tr><td>  -</td><td></td></tr></table> →	-		
-						

### Language Processor Development Tools

The two widely used language processor development tools are the lexical analyzer generator LEX and the parser generator YACC. The input to these tools are specifications of the lexical and syntactic constructs of a programming language L, and the semantic actions that should be performed on recognizing the constructs. Figure shows a schematic for developing the analysis phase of a compiler for language L by using LEX and YACC.

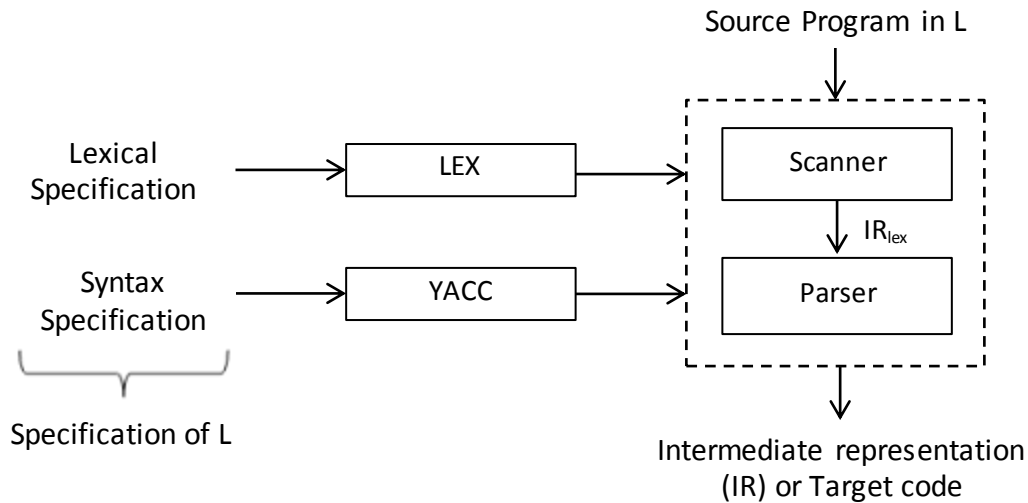


Figure 6.3: Using LEX and YACC

### LEX

- The input to LEX consists of two components.
- The first component is a specification of strings that represents the lexical units in L.
- This specification is in the form of regular expressions.
- The second component is a specification of semantic actions that are aimed at building the intermediate representation.
- The intermediated representation produced by a scanner would consist of a set of tables of lexical units and a sequence of tokens for the lexical units occurring in a source statement.
- The scanner generated by LEX would be invoked by a parser whenever the parser needs the next token.
- Accordingly, each semantic action would perform some table building actions and return a single token.

### YACC

- Each translation rule input to YACC has a string specification that resembles a production of a grammar—it has a nonterminal on the LHS and a few alternatives on the RHS.
- For simplicity, we will refer to a string specification as a production. YACC generates an LALR(1) parser for language L from the productions, which is a bottom-up parser.
- The parser would operate as follows: For a shift action, it would invoke the scanner to obtain the next token and continue the parse by using that token. While performing a reduce action in accordance with a production, it would perform the semantic action associated with that production.
- The semantic actions associated with productions achieve building of an intermediate representation or target code as follows: Every nonterminal symbol in the parser has an attribute. The semantic action associated with a production can access attributes of

nonterminal symbols used in that production—a symbol '\$n' in the semantic action, where n is an integer, designates the attribute of the n<sup>th</sup> nonterminal symbol in the RHS of the production and the symbol '\$\$' designates the attribute of the LHS nonterminal symbol of the production. The semantic action uses the values of these attributes for building the intermediate representation or target code. The attribute type can be declared in the specification input to YACC.

### Causes of Large Semantic Gap

- Two aspects of compilation are:
  - a) Generate code to implement meaning of a source program in the execution domain (target code generation)
  - b) Provide diagnostics for violations of PL semantics in a program (Error reporting)
- There are four issue involved in implementing these aspects (**Q. What are the issue in code generation in relation to compilation of expression? Explain each issue in brief. (June-13 GTU)**)
  1. Data types : semantics of a data type require a compiler to ensure that variable of a type are assigned or manipulated only through legal operation  
Compiler must generate type specific code to implement an operation.
  2. Data structures: to compile a reference to an element of a data structure, the compiler must develop a memory mapping to access the memory word allocated to the element.
  3. Scope rules: compiler performs operation called scope analysis and name resolution to determine the data item designated by the use of a name in the source program
  4. Control structure: control structure includes conditional transfer of control, conditional execution, iteration control and procedure calls. The compiler must ensure that source program does not violate the semantics of control structures.

### Binding and Binding Times

#### Binding

A binding is the association of an attribute of a program entity with a value.

The binding of an attribute may be performed at any convenient time subject to the condition that the value of the attribute should be known when the attribute is referenced. Binding time is the time at which a binding is actually performed.

The following binding times arise in compilers:

- 1) Language definition time of a programming language L, which is the time at which features of the language are specified.
- 2) Language implementation time of a programming language L, which is the time at which the design of a language translator for L is finalized.
- 3) Compilation time of a program P.
- 4) Execution init time of a procedure *proc*
- 5) Execution time of a procedure *proc*

#### Importance of binding times

- The binding time of an entity's attributes determines the manner in which a language processor can handle use of the entity in the program.
- A compiler can tailor the code generated to access an entity if a relevant binding was performed before or during compilation time.
- However, such tailoring is not possible if the binding is performed later than compilation time. So the compiler has to generate a general purpose code that would find information

about the relevant binding during its execution and use it to access the entity appropriately. It affects execution efficiency of the target program.

### Memory Allocation

- Three important tasks of memory allocation are:
  1. Determine the amount of memory required to represent the value of a data item.
  2. Use an appropriate memory allocation model to implement the lifetimes and scopes of data items.
  3. Determine appropriate memory mappings to access the values in a non scalar data item, e.g. values in an array.
- Memory allocation are mainly divides into two types:
  1. Static binding
  2. Dynamic binding

#### Static memory allocation

- In static memory allocation, memory is allocated to a variable before the execution of a program begins.
- Static memory allocation is typically performed during compilation.
- No memory allocation or deallocation actions are performed during the execution of a program. Thus, variables remain permanently allocated

#### Dynamic memory allocation

- In dynamic memory allocation, memory bindings are established and destroyed during the execution of a program
- Dynamic memory allocation has two flavors'-automatic allocation and program controlled allocation.
- In automatic dynamic allocation, memory is allocated to the variables declared in a program unit when the program unit is entered during execution and is deallocated when the program unit is exit. Thus the same memory area may be used for the variables of different program units
- In program controlled dynamic allocation, a program can allocate or deallocate memory at arbitrary points during its execution.
- It is obvious that in both automatic and program controlled allocation, address of the memory area allocated to a program unit cannot be determined at compilation time.

### Memory Allocation in block structured language

- The block is a sequence of statements containing the local data and declarations which are enclosed within the delimiters.

```
Ex:
A
{
    Statements
    ....
}
```

- The delimiters mark the beginning and the end of the block. There can be nested blocks for ex: block B2 can be completely defined within the block B1.
- A block structured language uses dynamic memory allocation.
- Finding the scope of the variable means checking the visibility within the block
- Following are the rules used to determine the scope of the variable:
  1. Variable X is accessed within the block B1 if it can be accessed by any statement situated in block B1.
  2. Variable X is accessed by any statement in block B2 and block B2 is situated in block B1.
- There are two types of variable situated in the block structured language
  1. Local variable
  2. Non local variable
- To understand local and non-local variable consider the following example

```

Procedure A
{
    int x,y,z
    Procedure B
    {
        Int a,b
    }
    Procedure C
    {
        Int m,n
    }
}
  
```

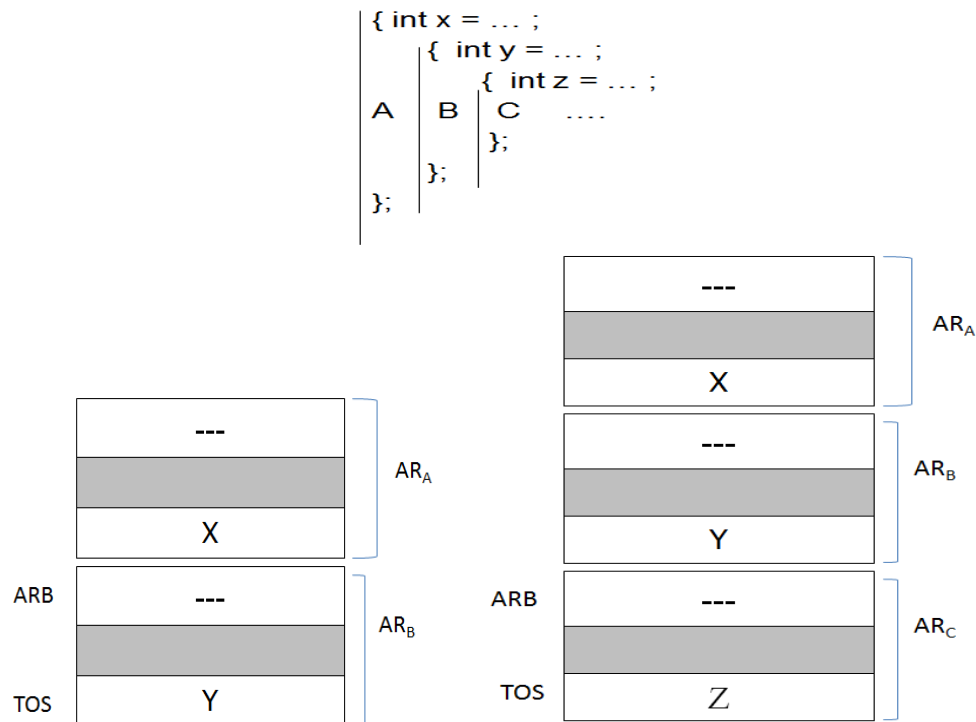
Procedure	Local variables	Non local variables
A	x,y,z	
B	a,b	x,y,z
C	m,n	x,y,z

- Variables x, y and z are local variables to procedure A but those are non-local to block B and C because these variable are not defined locally within the block B and C but are accessible within these blocks.
- Automatic dynamic allocation is implemented using the extended stack model.
- Each record in the stack has two reserved pointers instead of one.
- Each stack record accommodates the variable for one activation of a block, which we call an activation record (AR).

### Dynamic pointer

- The first reserved pointer in block's AR points to the activation record of its dynamic parent. This is called dynamic pointer and has the address 0 (ARB).
- The dynamic pointer is used for de-allocating an AR.
- Following example shows memory allocation for program given below.





### Static pointer

- Access to non local variable is implemented using the second reserved pointer in AR. This pointer which has the address 1 (ARB) is called the static pointer.

### Activation record

- The activation record is a block of memory used for managing information needed by a single execution of a procedure.

Return value
Actual parameter
Control link
Access link
Saved M/c status
Local variables
Temporaries

- Temporary values:** The temporary variables are needed during the evaluation of expressions. Such variables are stored in the temporary field of activation record.
- Local variables:** The local data is a data that is local to the execution procedure is stored in this field of activation record.
- Saved machine registers:** This field holds the information regarding the status of machine just before the procedure is called. This field contains the registers and program counter.

4. Control link: This field is optional. It points to the activation record of the calling procedure. This link is also called dynamic link.
5. Access link: This field is also optional. It refers to the non local data in other activation record. This field is also called static link field.
6. Actual parameters: This field holds the information about the actual parameters. These actual parameters are passed to the called procedure.
7. Return values: This field is used to store the result of a function call.

## Compilation of Expression

### Operand Descriptor

An operand descriptor has the following fields:

1. Attributes: Contains the subfields type, length and miscellaneous information
2. Addressability: Specifies where the operand is located, and how it can be accessed. It has two subfields
  - Addressability code: Takes the values 'M' (operand is in memory), and 'R' (operand is in register). Other addressability codes, e.g. address in register ('AR') and address in memory ('AM'), are also possible,
  - Address: Address of a CPU register or memory word.
  - Ex:  $a*b$

MOVER AREG, A

MULT AREG, B

Three operand descriptors are used during code generation. Assuming  $a, b$  to be integers occupying 1 memory word, these are:

Attribute	Addressability
(int, 1)	Address(a)
(int, 1)	Address(b)
(int, 1)	Address(AREG)

### Register descriptors

A register descriptor has two fields

1. Status: Contains the code free or occupied to indicate register status.
2. Operand descriptor #: If status = occupied, this field contains the descriptor for the operand contained in the register.
  - Register descriptors are stored in an array called Register\_descriptor. One register descriptor exists for each CPU register.
  - In above Example the register descriptor for AREG after generating code for  $a*b$  would be  

Occupied      #3
  - This indicates that register AREG contains the operand described by descriptor #3.

### Intermediate code for expression

There are two types of intermediate representation

1. Postfix notation
2. Three address code.

### 1) Postfix notation

- Postfix notation is a linearized representation of a syntax tree.
- it a list of nodes of the tree in which a node appears immediately after its children
- the postfix notation of  $x = -a * b + -a * b$  will be  
 $x \ a \ -b \ * \ a \ -b \ * \ + =$

### 2) Three address code

- In three address code form at the most three addresses are used to represent statement.  
 The general form of three address code representation is  $-a := b \ op \ c$
- Where a,b or c are the operands that can be names, constants.
- For the expression like  $a = b + c + d$  the three address code will be  
 $t1 = b + c$   
 $t2 = t1 + d$
- Here  $t1$  and  $t2$  are the temporary names generated by the compiler. There are most three addresses allowed. Hence, this representation is three-address code.
- There are three representations used for three code such as quadruples, triples and indirect triples.

### Quadruple representation

- The quadruple is a structure with at the most four fields such as op, arg1, arg2 and result.
- The op field is used to represent the internal code for operator, the arg1 and arg2 represent the two operands used and result field is used to store the result of an expression.
- Consider the input statement  $x := -a * b + -a * b$

		Op	Arg1	Arg2	result
$t1 = \text{uminus } a$	(0)	uminus	a		t1
$t2 := t1 * b$	(1)	*	t1	b	t2
$t3 = -a$	(2)	uminus	a		t3
$t4 := t3 * b$	(3)	*	t3	b	t4
$t5 := t2 + t4$	(4)	+	t2	t4	t5
$x = t5$	(5)	:=	t5		x

### Triples

- The triple representation the use of temporary variables is avoided by referring the pointers in the symbol table.

- the expression  $x := -a * b + -a * b$  the triple representation is as given below

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	X	(4)

### Indirect Triples

- The indirect triple representation the listing of triples is been done. And listing pointers are used instead of using statements.

Number	Op	Arg1	Arg2		Statement
(0)	uminus	a		(0)	(11)
(1)	*	(11)	b	(1)	(12)
(2)	uminus	a		(2)	(13)
(3)	*	(13)	b	(3)	(14)
(4)	+	(12)	(14)	(4)	(15)
(5)	:=	X	(15)	(5)	(16)

### Code Optimization

#### I. Compile Time Evaluation

- Compile time evaluation means shifting of computations from run time to compilation.
- There are two methods used to obtain the compile time evaluation.

##### 1. **Folding**

- In the folding technique the computation of constant is done at compile time instead of run time.

**example :**  $\text{length} = (22/7) * d$

- Here folding is implied by performing the computation of  $22/7$  at compile time

##### 2. **Constant propagation**

- In this technique the value of variable is replaced and computation of an expression is done at the compilation time.

**example :**  $\text{pi} = 3.14; r = 5;$

$\text{Area} = \text{pi} * r * r$

- Here at the compilation time the value of  $\text{pi}$  is replaced by 3.14 and  $r$  by 5 then computation of  $3.14 * 5 * 5$  is done during compilation.

### II. Common Sub Expression Elimination

- The common sub expression is an expression appearing repeatedly in the program which is computed previously.
- Then if the operands of this sub expression do not get changed at all then result of such sub expression is used instead of recomputing it each time

- Example:

```
t1 := 4 * i
t2 := a[t1]
t3 := 4 * j
t4 := 4 * i
t5 := n
t6 := b[t4] + t5
```

- The above code can be optimized using common sub expression elimination

```
t1 = 4 * i
t2 = a[t1]
t3 = 4 * j
t5 = n
t6 = b[t1] + t5
```

- The common sub expression  $t4 := 4 * i$  is eliminated as its computation is already in  $t1$  and value of  $i$  is not been changed from definition to use.  
}

### III. Loop invariant computation (Frequency reduction)

- Loop invariant optimization can be obtained by moving some amount of code outside the loop and placing it just before entering in the loop.
- This method is also called code motion.

- Example:

```
while(i <= max-1)
{
    sum = sum + a[i];
}
```

Can be optimized as a

```
N = max-1;
While(i <= N)
{
    sum = sum + a[i];
}
```

### IV. Strength Reduction

- Strength of certain operators is higher than others.

- For instance strength of \* is higher than +.
- In this technique the higher strength operators can be replaced by lower strength operators.

- Example:

```
for(i=1;i<=50;i++)  
{  
    count = i x 7;  
}
```

- Here we get the count values as 7, 14, 21 and so on up to less than 50.
- This code can be replaced by using strength reduction as follows

```
temp=7  
for(i=1;i<=50;i++)  
{  
    count = temp;  
    temp = temp+7;  
}
```

### V. Dead Code Elimination

- A variable is said to be **live** in a program if the value contained into is subsequently.
- On the other hand, the variable is said to be **dead** at a point in a program if the value contained into it is never been used. The code containing **such a variable** supposed to be a dead code. And an optimization can be performed by eliminating such a dead code.

- Example :

```
i=0;  
if(i==1)  
{  
    a=x+5;  
}
```

- **if** statement is a dead code as this condition will never get satisfied hence, statement can be eliminated and optimization can be done.

### Overview of Interpretation OR Write a note on Interpreter

An interpreter is system software that translates a given High-Level Language (HLL) program into a low-level one, but it differs from compilers. Interpretation is a real-time activity where an interpreter takes the program, one statement at a time, and translates each line before executing it.

### Comparison between Compilers and Interpreters

Compilers	Interpreters
Compilers are language processors that are based on the language translation-linking-loading model.	Interpreters are a class of language processors based on the interpretation model.
Generate a target output program as an output, which can be run independently from the source program written in Source Language.	Do not generate any output program; rather they evaluate the source program at each time for execution.
Program execution is separated from compilation and performed only after the entire output program is produced.	Program execution is a part of interpretation and performed on a statement by statement basis.
Target program executes independently and does not need the presence of compiler in the memory.	The interpreter exists in the memory during interpretation, i.e. it coexists with the source program to be interpreted.
Do not generate the output program for execution if there is any error in any of the source program statements.	Can evaluate and execute program statement until an error is found.
Need recompilation for generating a fresh output program in target language after each modification in the source program.	The interpreter is independent of program modification issues as it processes the source program each time during execution.
Compilers are suitable for production environment.	Interpreters are suited for program development environment.
Compilers are bound to a specific target machine and cannot be ported.	Can be made portable by carefully coding them in higher level language.

### Comparing the Performance of Compilers and Interpreters

- Comparative performance of a compiler and an interpreter can be realized by inspecting the average CPU time cost for different kinds of processing of a statement.
- Let  $t_i$ ,  $t_c$ , and  $t_e$  be the interpretation-time statement, compilation-time statement, and execution-time of compiled statement, respectively.
- It is assumed that  $t_c \approx t_i$  since both the compilers and interpreters involve lexical, syntax, and semantic analyses of the source statement. In addition, the code generation effort for a

statement performed by the compiler is of the same order of magnitude as the effort involved in the interpretation of the statement.

- If a 400-statement program is executed on a test data with only 80 statements being visited during the test run, the total CPU time in compilation followed by the execution of the program is  $400 * t_c + 80 * t_e$ , while the total CPU time in interpretation of the program is  $80 * t_i = 80 * t_c$ . This shows that the interpretation will be cheaper in such cases. However, if more than 400 statements are to be executed, compilation followed by execution will be cheaper, which means that using interpreter is advantageous up to the execution of 400 statements during the execution. This clearly indicates that from the point of view of the CPU time cost, interpreters are a better choice at least for the program development environment.

### Benefits of Interpretation

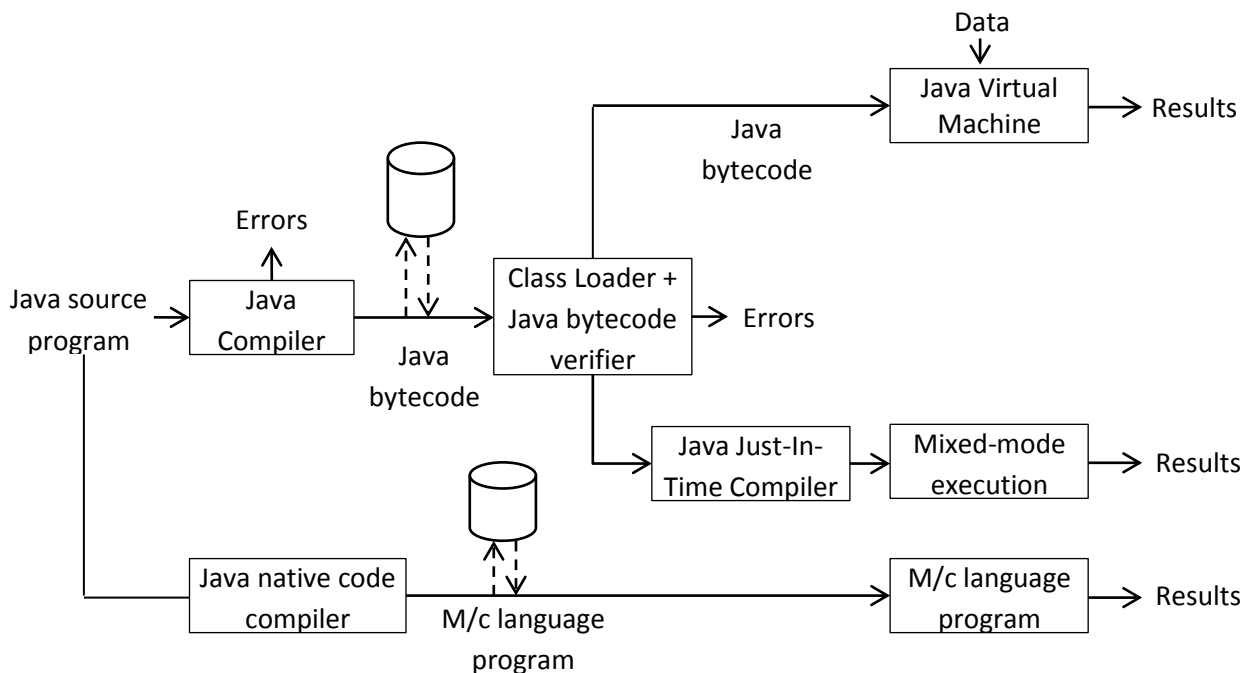
The distinguished benefits of interpretation are as follows:

- Executes the source code directly. It translates the source code into some efficient Intermediate Code (IC) and immediately executes it. The process of execution can be performed in a single stage without the need of a compilation stage.
- Handles certain language features that cannot be compiled.
- Ensures portability since it does not produce machine language program.
- Suited for development environment where a program is modified frequently. This means alteration of code can be performed dynamically.
- Suited for debugging of the code and facilitates interactive code development.

### Java Language Environment

- Java language environment has four key features:
  - The Java virtual machine (JVM), which provides portability of Java programs.
  - An impure interpretive scheme, whose flexibility is exploited to provide a capability for inclusion of program modules dynamically, i.e., during interpretation.
  - A Java bytecode verifier, which provides security by ensuring that dynamically loaded program modules do not interfere with the operation of the program and the operating system.
  - An optional Java just-in-time (JIT) compiler, which provides efficient execution.
- Figure 8.1 shows a schematic of the Java language environment. The Java compiler converts a source language program into the Java bytecode, which is a program in the machine language of the Java virtual machine.
- The Java virtual machine is implemented by a software layer on a computer, which is itself called the Java virtual machine for simplicity. This scheme provides portability as the Java bytecode can be 'executed' on any computer that implements the Java virtual machine.





**Figure 8.1: Java Language Environment**

- The Java virtual machine essentially interprets the bytecode form of a program. The Java compiler and the Java virtual machine thus implement the impure interpretation scheme.
- Use of an interpretive scheme allows certain elements of a program to be specified during interpretation. This feature is exploited to provide a capability for including program modules called Java class files during interpretation of a Java program.
- The class loader is invoked whenever a new class file is to be dynamically included in program. The class loader locates the desired class file and passes it to the Java bytecode verifier.
- The Java bytecode verifier checks whether
  - The program forges pointers, thereby potentially accessing invalid data or performing branches to invalid locations.
  - The program violates access restrictions, e.g., by accessing private data.
  - The program has type-mismatches whereby it may access data in an invalid manner.
  - The program may have stack overflows or underflows during execution.
- The Java language environment provides the two compilation schemes shown in the lower half of Figure 8.1. The Java Just-In-Time compiler compiles parts of the Java bytecode that are consuming a significant fraction of the execution time into the machine language of the computer to improve their execution efficiency. It is implemented using the scheme of dynamic compilation.
- After the just-in-time compiler has compiled some part of the program, some parts of the Java source program has been converted into the machine language while the remainder of the program still exists in the bytecode form. Hence the Java virtual machine uses a mixed-mode execution approach.
- The other compilation option uses the Java native code compiler shown in the lower part of

Figure 8.1. It simply compiles the complete Java program into the machine language of a computer. This scheme provides fast execution of the Java program; however, it cannot provide any of the benefits of interpretation or just-in-time compilation.

### Java Virtual Machine

- A Java compiler produces a binary file called a class file which contains the bytecode for a Java program. The Java virtual machine loads one or more class files and executes programs contained in them. To achieve it, the JVM requires the support of the class loader, which locates a required class file, and a bytecode verifier, which ensures that execution of the bytecode would not cause any breaches of security.
- The Java virtual machine is a stack machine. By contrast, a stack machine performs computations by using the values existing in the top few entries on a stack and leaving their results on the stack. This arrangement requires that a program should load the values on which it wishes to operate on the stack before performing operations on them and should take their results from the stack.
- The stack machine has the following three kinds of operations:
  - Push operation: This operation has one operand, which is the address of a memory location. The operation creates a new entry at the top of the stack and copies the value that is contained in the specified memory location into this entry.
  - Pop operation: This operation also has the address of a memory location as its operand. It performs the converse of the push operation—it copies the value contained in the entry that is at the top of the stack into the specified memory location and also deletes that entry from the stack.
  - n-ary operation: This operation operates on the values existing in the top n entries of the stack, deletes the top n entries from the stack, and leaves the result, if any, in the top entry of the stack. Thus, a unary operation operates only on the value contained in the top entry of the stack, a binary operation operates on values contained in the top two entries of the stack, etc.
- A stack machine can evaluate expressions very efficiently because partial results need not be stored in memory—they can be simply left on the stack.

### Types of Errors

#### Syntax Error

- Syntax errors occur due to the fact that the syntax of the programming language is not followed.
- The errors in token formation, missing operators, unbalanced parenthesis, etc., constitute syntax errors.
- These are generally programmer induced due to mistakes and negligence while writing a program.
- Syntax errors are detected early during the compilation process and restrict the compiler to

proceed for code generation.

- Let us see the syntax errors with Java language in the following examples.

**Example 1:** Missing punctuation-"semicolon"

```
int age = 50    // note here semicolon is missing
```

**Example 2:** Errors in expression syntax

```
x = ( 30 - 15;    // note the missing closing parenthesis ")"
```

### Semantic Error

- Semantic errors occur due to improper use of programming language statements.
- They include operands whose types are incompatible, undeclared variables, incompatible arguments to function or procedures, etc.
- Semantic errors are mentioned in the following examples.

**Example:** Type incompatibility between operands

```
int msg = "hello";    //note the types String and int are incompatible
```

### Logical Error

- Logical errors occur due to the fact that the software specification is not followed while writing the program. Although the program is successfully compiled and executed error free, the desired results are not obtained.
- Let us look into some logical errors with Java language.

**Example :** Errors in computation

```
public static int mul(int a, int b) {  
    return a + b ;  
}
```

// this method returns the incorrect value with respect to the specification that requires to multiply two integers

**Example :** Non-terminating loops

```
String str = br. readLine();  
while (str != null) {  
    System.out.println(str);  
} // the loop in the code did not terminate
```

- Logical errors may cause undesirable effect and program behaviors. Sometimes, these errors remain undetected unless the results are analyzed carefully.

### Debugging Procedures

- Whenever there is a gap between an expected output and an actual output of a program, the program needs to be debugged.
- An error in a program is called bug, and debugging means finding and removing the errors present in the program.
- Debugging involves executing the program in a controlled fashion.
- During debugging, the execution of a program can be monitored at every step.

- In the debug mode, activities such as starting the execution and stopping the execution are in the hands of the debugger.
- The debugger provides the facility to execute a program up to the specified instruction by inserting a breakpoint.
- It gives a chance to examine the values assigned to the variables present in the program at any instant and, if required, offers an opportunity to update the program.
- Types of debugging procedures:
  - **Debug Monitors:**

A debug monitor is a program that monitors the execution of a program and reports the state of a program during its execution. It may interfere in the execution process, depending upon the actions carried out by a debugger (person). In order to initiate the process of debugging, a programmer must compile the program with the debug option first. This option, along with other information, generates a table that stores the information about the variables used in a program and their addresses.
  - **Assertions:**

Assertions are mechanisms used by a debugger to catch the errors at a stage before the execution of a program. Sometimes, while programming, some assumptions are made about the data involved in computation. If these assumptions went wrong during the execution of the program, it may lead to erroneous results. For this, a programmer can make use of an assert statement. Assertions are the statements used in programs, which are always associated with Boolean conditions. If an assert() statement is evaluated to be true, nothing happens. But if it is realized that the statement is false, the execution program halts.

## Classification of Debuggers

### Static Debugging

- Static debugging focuses on semantic analysis.
- In a certain program, suppose there are two variables: var1 and var2. The type of var1 is an integer, and the type of var2 is a float. Now, the program assigns the value of var2 to var1; then, there is a possibility that it may not get correctly assigned to the variable due to truncation. This type of analysis falls under static debugging.
- Static debugging detects errors before the actual execution.
- Static code analysis may include detection of the following situations:
  - Dereferencing of variable before assigning a value to it
  - Truncation of value due to wrong assignment
  - Redclaration of variables
  - Presence of unreachable code

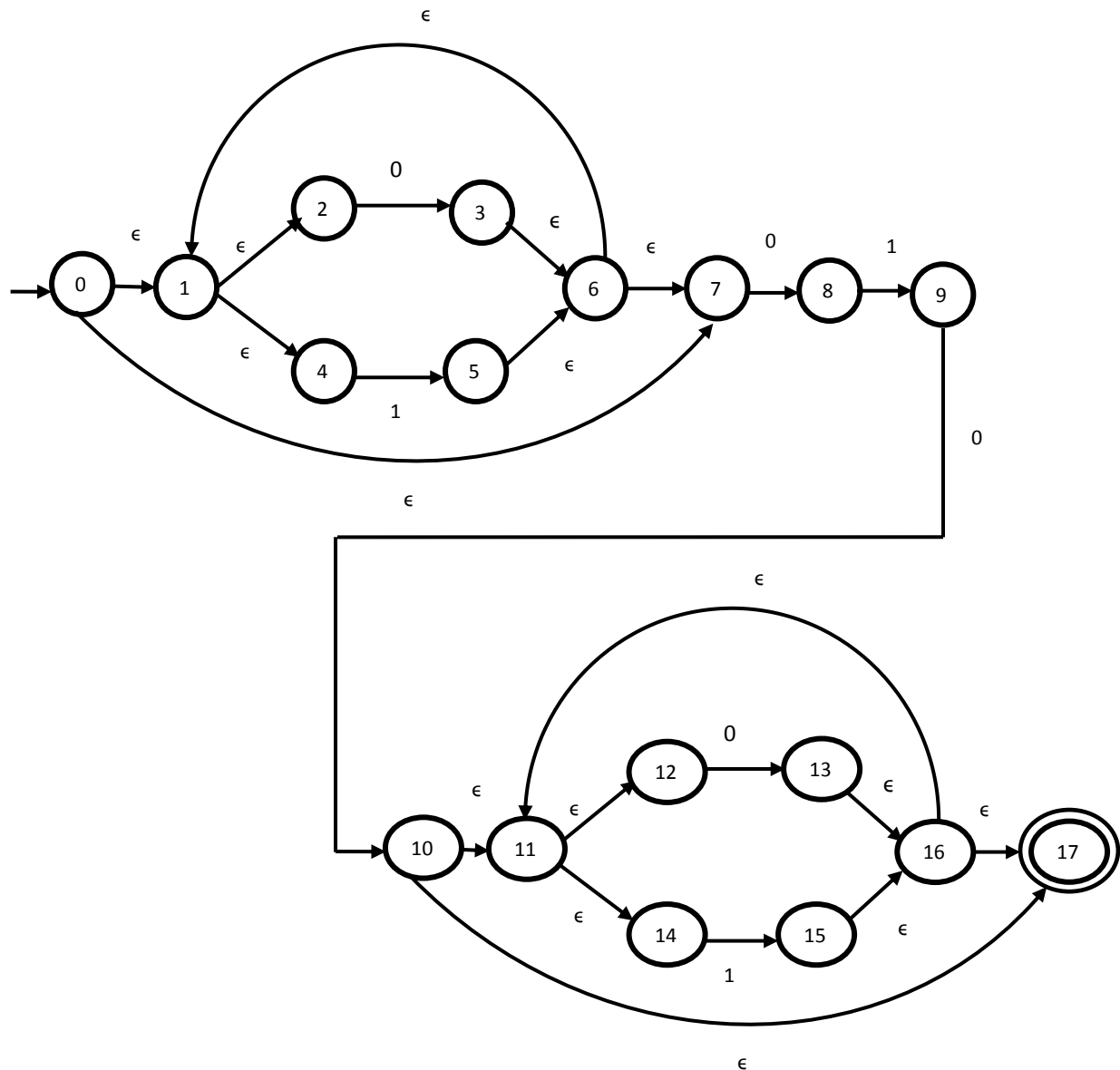
### Dynamic/Interactive Debugger

Dynamic analysis is carried out during program execution. An interactive debugging system provides programmers with facilities that aid in testing and debugging programs interactively.

A dynamic debugging system should provide the following facilities:

- **Execution sequencing:** It is nothing but observation and control of the flow of program execution. For example, the program may be halted after a fixed number of instructions are executed.
- **Breakpoints:** Breakpoints specify the position within a program till which the program gets executed without disturbance. Once the control reaches such a position, it allows the user to verify the contents of variables declared in the program.
- **Conditional expressions:** A debugger can include statements in a program to ensure that certain conditions are reached in the program. These statements, known as assertions, can be used to check whether some pre-condition or post-condition has been met in the program during execution.
- **Tracing:** Tracing monitors step by step the execution of all executable statements present in a program. The other name for this process is "step into". Another possible variation is "step over" debugging that can be executed at the level of procedure or function. This can be implemented by adding a breakpoint at the last executable statement in a program.
- **Traceback:** This gives a user the chance to traceback over the functions, and the traceback utility uses stack data structure. Traceback utility should show the path by which the current statement in the program was reached.
- **Program-display capabilities:** While debugging is in progress, the program being debugged must be made visible on the screen along with the line numbers.
- **Multilingual capability:** The debugging system must also consider the language in which the debugging is done. Generally, different programming languages involve different user environments and applications systems.
- **Optimization:** Sometimes, to make a program efficient, programmers may use an optimized code. Debugging of such statements can be tricky. However, to simplify the debugging process, a debugger may use an optimizing compiler that deals with the following issues:
  - Removing invariant expressions from a loop
  - Merging similar loops
  - Eliminating unnecessary statements
  - Removing branch instructions

## 1. $(0+1)^*010(0+1)^*$



- $\epsilon$ -closure (0) = {0, 1, 2, 4, 7} ---- A
- Move (A, 0) = {3, 8}  
 $\epsilon$ -closure (Move (A, 0)) = {3, 6, 7, 1, 2, 4, 8} ---- B  
 Move (A, 1) = {5}  
 $\epsilon$ -closure (Move (A, 1)) = {5, 6, 7, 1, 2, 4} ---- C
- Move (B, 0) = {3, 8}  
 $\epsilon$ -closure (Move (B, 0)) = {3, 6, 7, 1, 2, 4, 8} ---- B  
 Move (B, 1) = {5, 9}

$\epsilon$ - closure (Move (B, 1)) = {5, 6, 7, 1, 2, 4, 9} ---- D

- Move (C, 0) = {3, 8}

$\epsilon$ - closure (Move(C, 0)) = {3, 6, 7, 1, 2, 4, 8} ---- B

Move (C, 1) = {5}

$\epsilon$ - closure (Move(C, 1)) = {5, 6, 7, 1, 2, 4} ---- C

- Move (D, 0) = {3, 8, 10}

$\epsilon$ - closure (Move (D, 0)) = {3, 6, 7, 1, 2, 4, 8, 10, 11, 12, 14, 17} ---- E

Move (D, 1) = {5}

$\epsilon$ - closure (Move(D, 1)) = {1, 2, 4, 5, 6, 7} ---- C

- Move (E, 0) = {3, 8, 13}

$\epsilon$ - closure (Move (E, 0)) = {1, 2, 3, 4, 6, 7, 8, 13, 16, 17, 11, 12, 14} ---- F

Move (E, 1) = {5, 9, 15}

$\epsilon$ - closure (Move (E, 1)) = {1, 2, 4, 5, 6, 7, 9, 15, 16, 17, 11, 12, 14} ---- G

- Move (F, 0) = {3, 8, 13}

$\epsilon$ - closure (Move (F, 0)) = {1, 2, 3, 4, 6, 7, 8, 13, 16, 17, 11, 12, 14} ---- F

Move (F, 1) = {5, 9, 15}

$\epsilon$ - closure (Move (F, 1)) = {1, 2, 4, 5, 6, 7, 9, 11, 12, 14, 17, 15, 16} ---- G

- Move (G, 0) = {3, 10, 13}

$\epsilon$ - closure (Move (G, 0)) = {1, 2, 3, 4, 6, 7, 10, 13, 16, 17, 11, 12, 14} ---- H

Move (G, 1) = {5, 15}

$\epsilon$ - closure (Move (G, 1)) = {1, 2, 4, 5, 6, 7, 15, 16, 17, 11, 12, 14} ---- I

- Move (H, 0) = {3, 8, 13}

$\epsilon$ - closure (Move (H, 0)) = {1, 2, 3, 4, 6, 7, 8, 13, 16, 17, 11, 12, 14} ---- F

Move (H, 1) = {5, 15}

$\epsilon$ - closure (Move (H, 1)) = {1, 2, 4, 5, 6, 7, 15, 16, 17, 11, 12, 14} ---- I

- Move (I, 0) = {3, 8, 13}

$\epsilon$ - closure (Move (I, 0)) = {1, 2, 3, 4, 6, 7, 8, 13, 16, 17, 11, 12, 14} ---- F

Move (I, 1) = {5, 15}

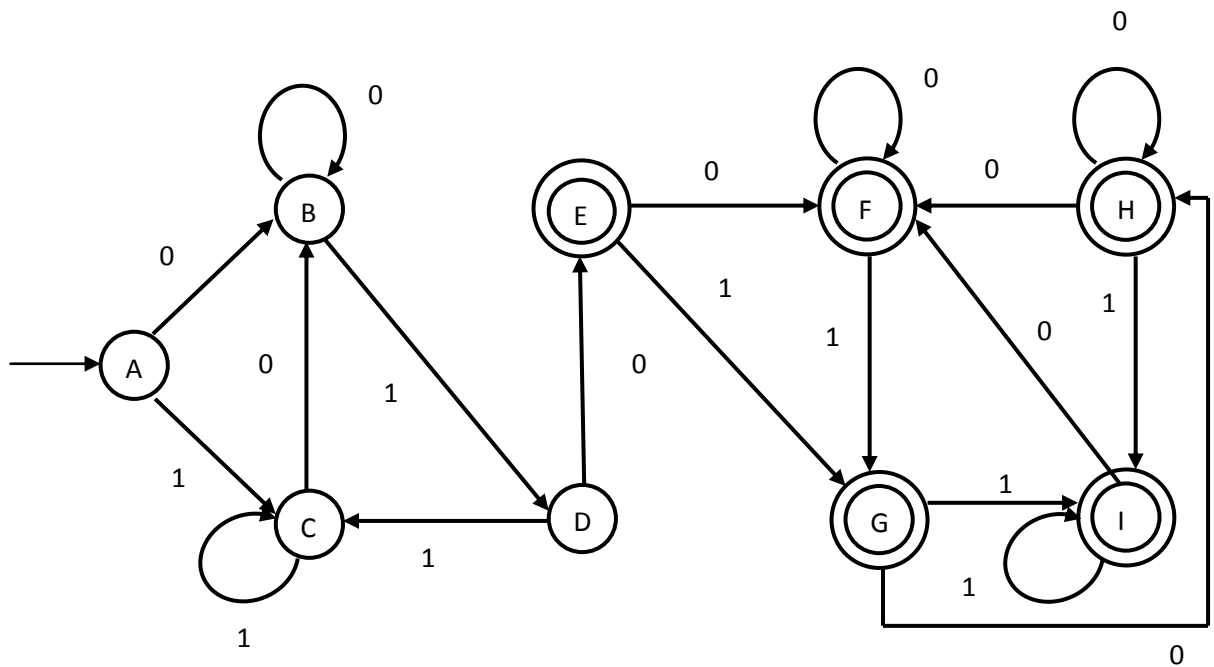
$\epsilon$ - closure (Move (I, 1)) = {1, 2, 4, 5, 6, 7, 15, 16, 17, 11, 12, 14} ---- I

**Transition Table:**

States	0	1
A	B	C
B	B	D
C	B	C

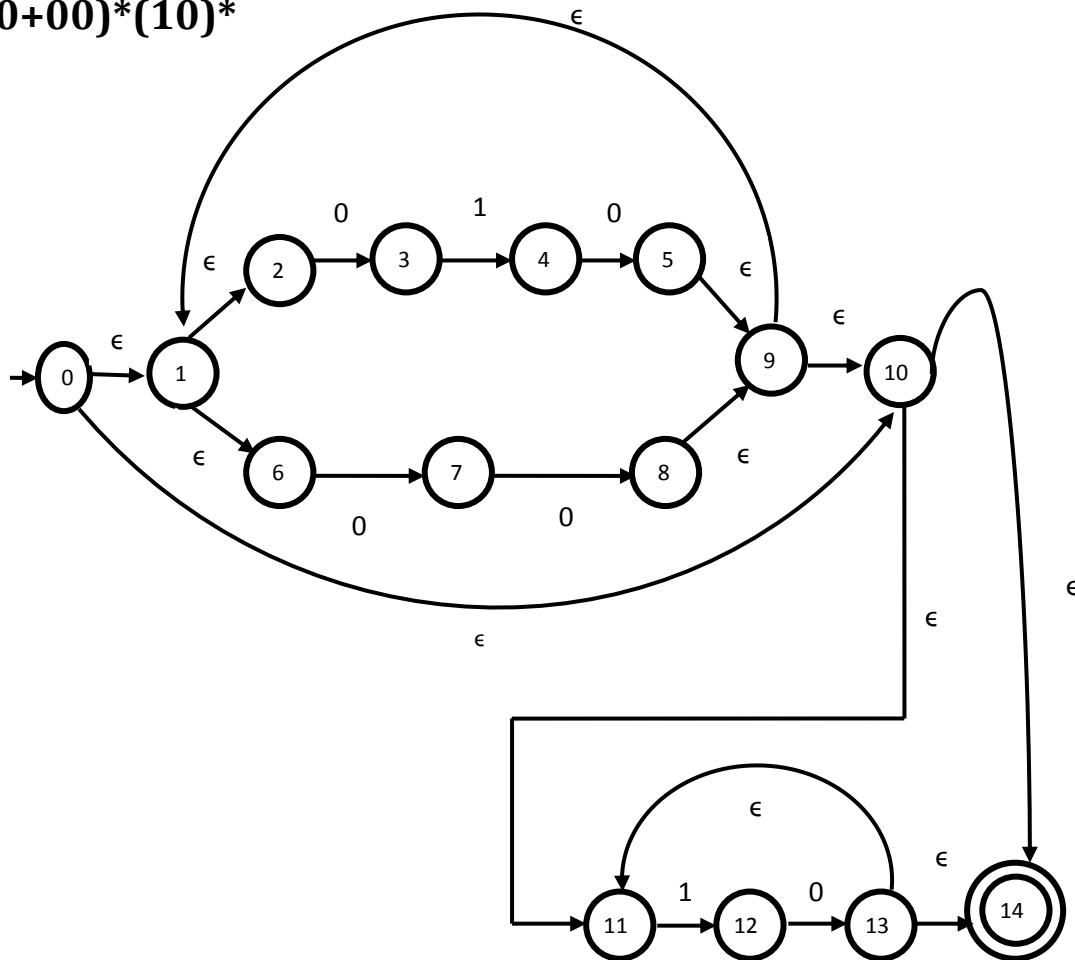
D	E	C
E	F	G
F	F	G
G	H	I
H	F	I
I	F	I

**DFA:**





## 2. $(010+00)^*(10)^*$



- $\epsilon$ -closure (0) = {0, 1, 2, 6, 10, 11, 14} ---- A
- Move (A, 0) = {3, 7}  
 $\epsilon$ -closure (Move (A, 0)) = {3, 7} ---- B  
 Move (A, 1) = {12}  
 $\epsilon$ -closure (Move (A, 1)) = {12} ---- C
- Move (B, 0) = {8}  
 $\epsilon$ -closure (Move (B, 0)) = {8, 9, 10, 11, 14, 1, 2, 6} ---- D  
 Move (B, 1) = {4}  
 $\epsilon$ -closure (Move (B, 1)) = {4} ---- E
- Move (C, 0) = {13}  
 $\epsilon$ -closure (Move (C, 0)) = {13, 11, 14} ---- F  
 Move (C, 1) =  $\phi$
- Move (D, 0) = {3, 7}  
 $\epsilon$ -closure (Move (D, 0)) = {3, 7} ---- B

Move (D, 1) = {12}

ε- closure (Move(D, 1)) = {12} ---- C

- Move (E, 0) = {5}

ε- closure (Move (E, 0)) = {5, 9, 1, 2, 6, 10, 11, 14} ---- G

Move (E, 1) =  $\phi$

- Move (F, 0) =  $\phi$

Move (F, 1) = {12}

ε- closure (Move(F, 1)) = {12} ---- C

- Move (G, 0) = {3, 7}

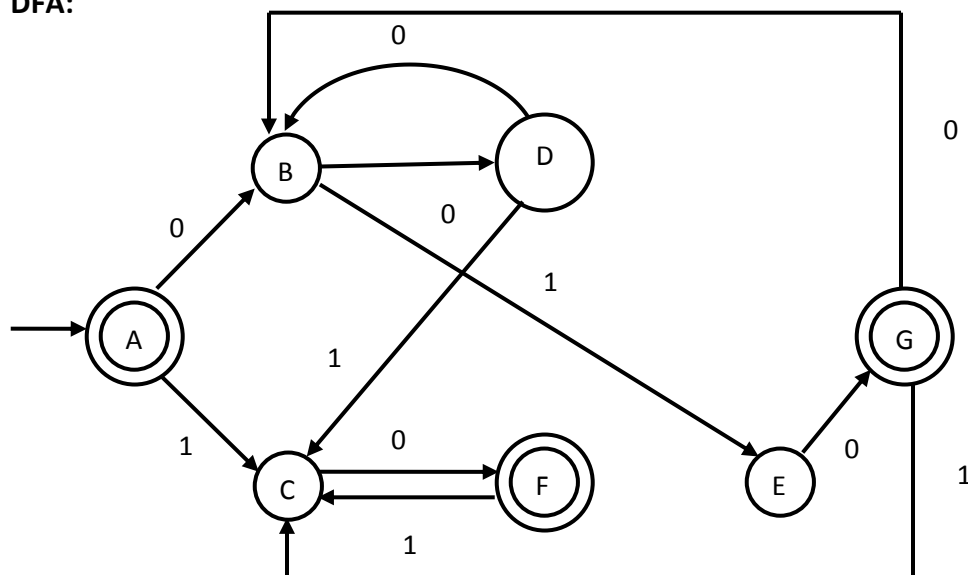
ε- closure (Move (G, 0)) = {3, 7} ---- B

Move (G, 1) = {12}

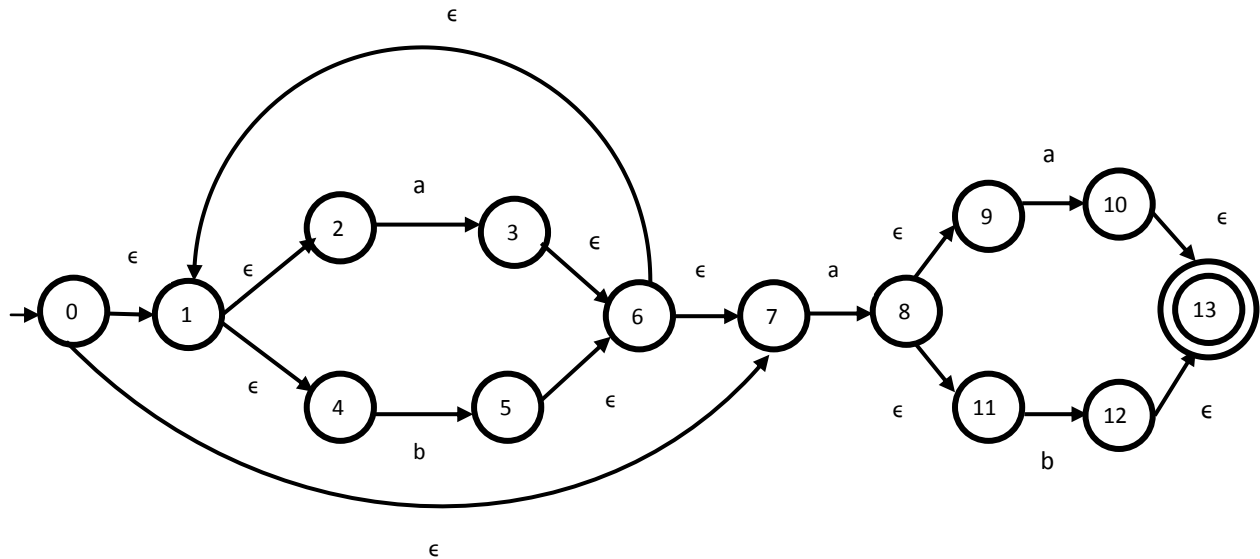
ε- closure (Move (G, 1)) = {12} ---- C

States	0	1
A	B	C
B	D	E
C	F	$\phi$
D	B	C
E	G	$\phi$
F	$\phi$	C
G	B	C

**DFA:**



## 3. $(a+b)^*a(a+b)$

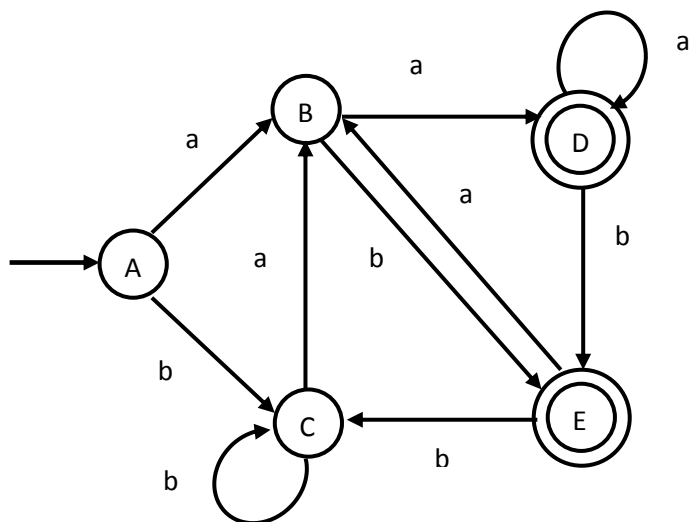


- $\epsilon$ - closure (0) = {0,1,2,4,7} ---- A
- Move (A, a) = {3,8}  
 $\epsilon$ - closure (Move (A, a)) = {3,6,7,1,2,4,8,9,11} ---- B  
 Move (A, b) = {5}  
 $\epsilon$ - closure (Move (A, b)) = {5,6,7,1,2,4} ---- C
- Move (B, a) = {3,8,10}  
 $\epsilon$ - closure (Move (B, a)) = {3,6,7,1,2,4,8,9,11,10,13} ---- D  
 Move (B, b) = {5,12}  
 $\epsilon$ - closure (Move (B, b)) = {5,6,7,1,2,4,12,13} ---- E
- Move (C, a) = {3,8}  
 $\epsilon$ - closure (Move (C, a)) = {3,6,7,1,2,4,8,9,11} ---- B  
 Move (C, b) = {5}  
 $\epsilon$ - closure (Move (C, b)) = {5,6,7,1,2,4} ---- C
- Move (D, a) = {3,8,10}  
 $\epsilon$ - closure (Move (D, a)) = {3,6,7,1,2,4,8,9,11,10,13} ---- D  
 Move (D, b) = {5, 12}  
 $\epsilon$ - closure (Move (D, b)) = {5,6,7,1,2,4,12,13} ---- E
- Move (E, a) = {3,8}  
 $\epsilon$ - closure (Move (E, 0)) = {3,6,7,1,2,4,8,9,11} ---- B  
 Move (E, b) = {5}  
 $\epsilon$ - closure (Move (E, 1)) = {5,6,7,1,2,4} ---- C

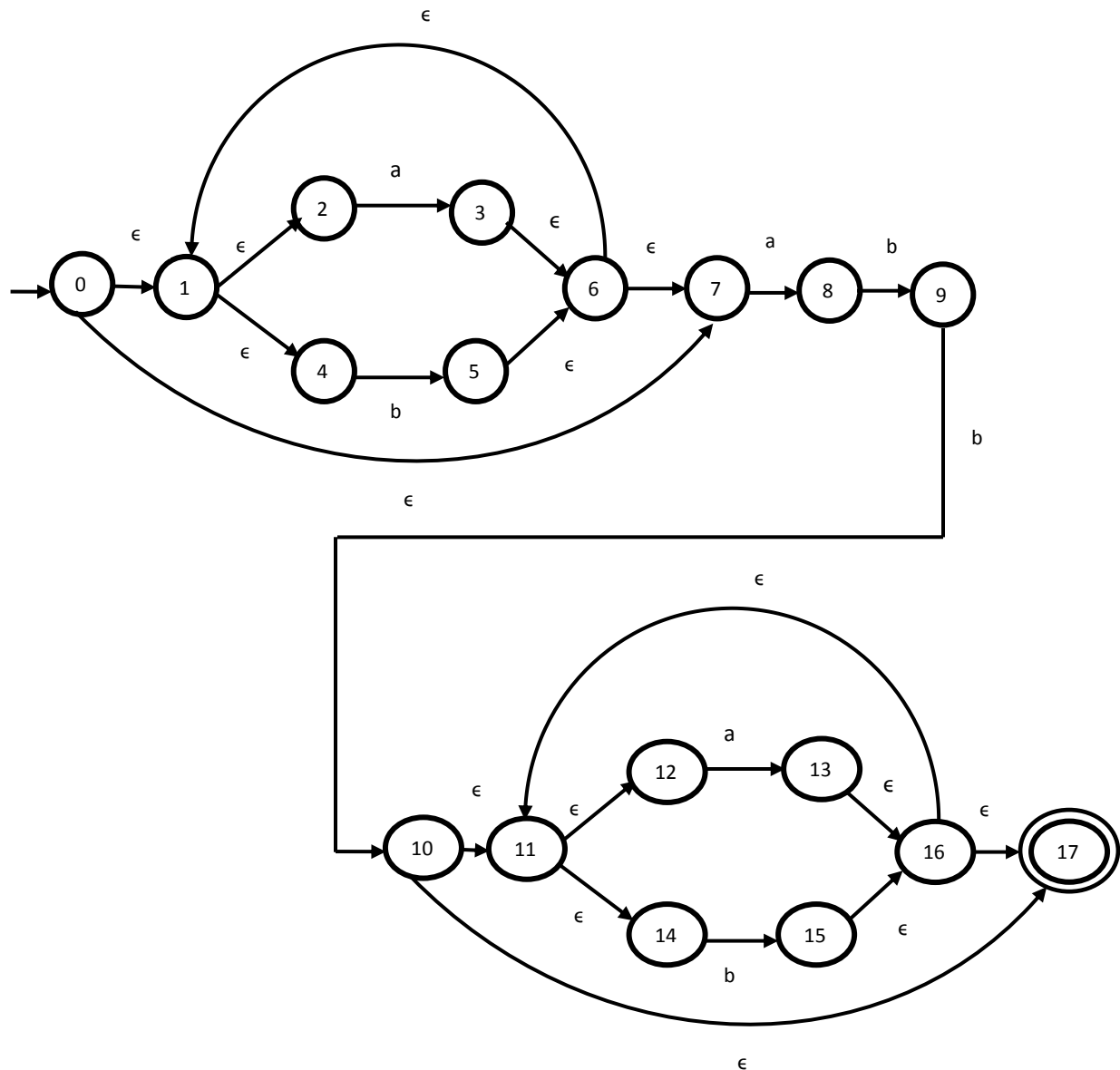
**Transition Table:**

States	a	b
A	B	C
B	D	E
C	B	C
D	D	E
E	B	C

**DFA:**



## 4. $(a+b)^*abb(a+b)^*$



- $\epsilon$ -closure (0) = {0, 1, 2, 4, 7} ---- A
- Move (A, a) = {3, 8}  
 $\epsilon$ -closure (Move (A, a)) = {3, 6, 1, 2, 4, 7, 8} ---- B  
 Move (A, b) = {5}  
 $\epsilon$ -closure (Move (A, b)) = {5, 6, 1, 2, 4, 7} ---- C
- Move (B, a) = {3, 8}  
 $\epsilon$ -closure (Move (B, a)) = {3, 6, 1, 2, 4, 7, 8} ---- B  
 Move (B, b) = {5, 9}

$\epsilon$ - closure (Move (B, b)) = {5, 6, 7, 1, 2, 4, 9} ---- D

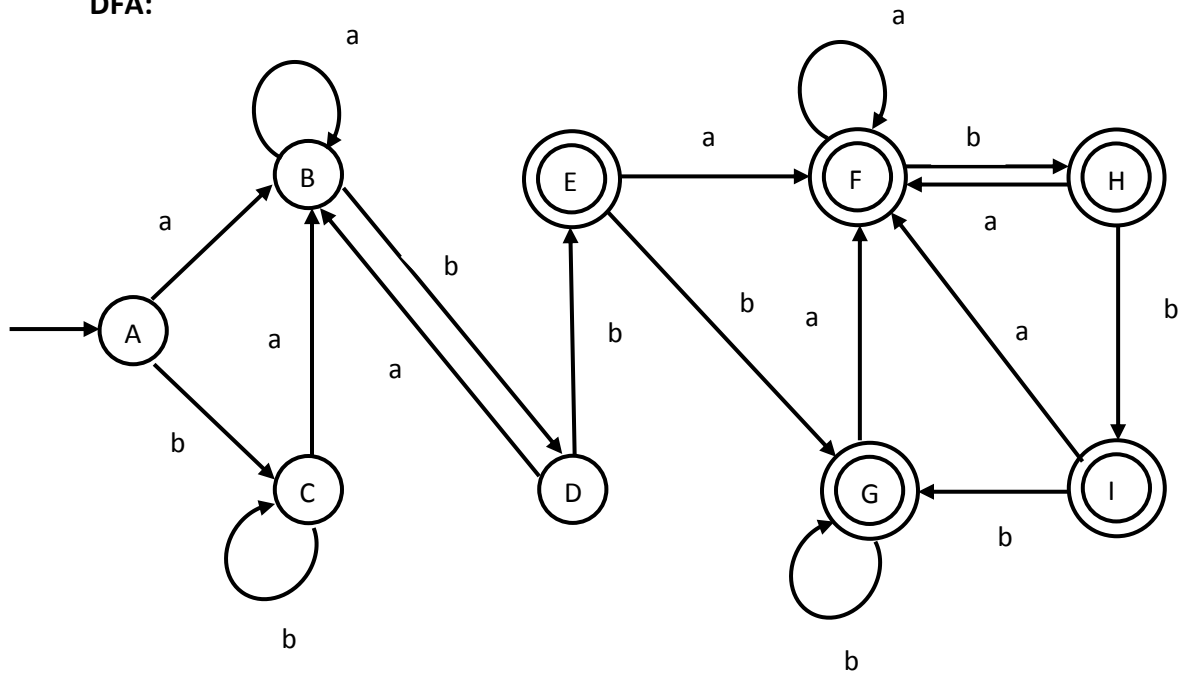
- Move (C, a) = {3, 8}  
 $\epsilon$ - closure (Move(C, a)) = {3, 6, 1, 2, 4, 7, 8} ---- B  
 Move (C, b) = {5}  
 $\epsilon$ - closure (Move(C, b)) = {5, 6, 1, 2, 4, 7} ---- C
- Move (D, a) = {3, 8}  
 $\epsilon$ - closure (Move (D, a)) = {3, 6, 1, 2, 4, 7, 8} ---- B  
 Move (D, b) = {5, 10}  
 $\epsilon$ - closure (Move(D, b)) = {5, 6, 7, 1, 2, 4, 10, 11, 12, 14, 17} ---- E
- Move (E, a) = {8, 3, 13}  
 $\epsilon$ - closure (Move (E, a)) = {8, 3, 6, 7, 1, 2, 4, 13, 16, 17, 11, 12, 14} ---- F  
 Move (E, b) = {5, 15}  
 $\epsilon$ - closure (Move (E, b)) = {5, 6, 7, 1, 2, 4, 15, 16, 17, 11, 12, 14} ---- G
- Move (F, a) = {8, 3, 13}  
 $\epsilon$ - closure (Move (F, a)) = {8, 3, 6, 7, 1, 2, 4, 13, 16, 17, 11, 12, 14} ---- F  
 Move (F, b) = {5, 9, 15}  
 $\epsilon$ - closure (Move (F, b)) = {9, 5, 6, 7, 1, 2, 4, 15, 16, 17, 11, 12, 14} ---- H
- Move (G, a) = {8, 3, 13}  
 $\epsilon$ - closure (Move (G, a)) = {8, 3, 6, 7, 1, 2, 4, 13, 16, 17, 11, 12, 14} ---- F  
 Move (G, b) = {5, 15}  
 $\epsilon$ - closure (Move (G, b)) = {5, 6, 7, 1, 2, 4, 15, 16, 17, 11, 12, 14} ---- G
- Move (H, a) = {8, 3, 13}  
 $\epsilon$ - closure (Move (H, a)) = {8, 3, 6, 7, 1, 2, 4, 13, 16, 17, 11, 12, 14} ---- F  
 Move (H, b) = {10, 5, 15}  
 $\epsilon$ - closure (Move (H, b)) = {10, 11, 12, 14, 17, 5, 6, 7, 1, 2, 4, 15, 16} ---- I
- Move (I, a) = {3, 8, 13}  
 $\epsilon$ - closure (Move (I, a)) = {8, 3, 6, 7, 1, 2, 4, 13, 16, 17, 11, 12, 14} ---- F  
 Move (I, b) = {5, 15}  
 $\epsilon$ - closure (Move (I, b)) = {5, 6, 7, 1, 2, 4, 15, 16, 17, 11, 12, 14} ---- G

**Transition Table:**

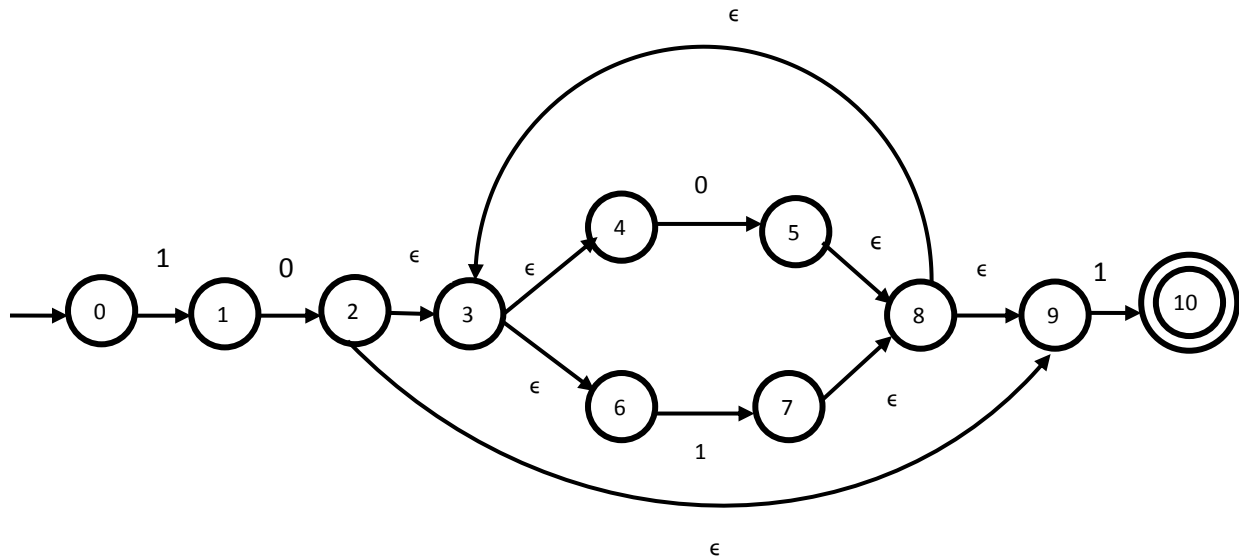
States	a	b
A	B	C
B	B	D
C	B	C

D	B	E
E	F	G
F	F	H
G	F	G
H	F	I
I	F	G

**DFA:**



## 5. $10(0+1)^*1$



- $\epsilon$ - closure (0) = {0} ---- A
- Move (A, 0) =  $\phi$   
 $\epsilon$ - closure (Move (A, 0)) =  $\phi$   
 Move (A, 1) = {1}  
 $\epsilon$ - closure (Move (A, 1)) = {1} ---- B
- Move (B, 0) = {2}  
 $\epsilon$ - closure (Move (B, 0)) = {2, 3, 4, 6, 9} ---- C  
 Move (B, 1) =  $\phi$   
 $\epsilon$ - closure (Move (B, 1)) =  $\phi$
- Move (C, 0) = {5}  
 $\epsilon$ - closure (Move (C, 0)) = {5, 8, 9, 3, 4, 6} ---- D  
 Move (C, 1) = {7, 10}  
 $\epsilon$ - closure (Move (C, 1)) = {3, 4, 6, 7, 8, 9, 10} ---- E
- Move (D, 0) = {5}  
 $\epsilon$ - closure (Move (D, 0)) = {5, 8, 9, 3, 4, 6} ---- D  
 Move (D, 1) = {7, 10}  
 $\epsilon$ - closure (Move (D, 1)) = {3, 4, 6, 7, 8, 9, 10} ---- E
- Move (E, 0) = {5}  
 $\epsilon$ - closure (Move (E, 0)) = {5, 8, 9, 3, 4, 6} ---- D  
 Move (E, 1) = {7, 10}  
 $\epsilon$ - closure (Move (E, 1)) = {3, 4, 6, 7, 8, 9, 10} ---- E



**Transition Table:**

States	0	1
A	$\phi$	B
B	C	$\phi$
C	D	E
D	D	E
E	D	E

**DFA:**

