1. **Software and software hierarchy.**
   - A set of instructions to perform specific tasks is called a Program.
   - The collection of one or many programs for a specific purpose is called Software.

**Software Hierarchy (Types of Software)**

- **OS** is an interface between Hardware and User.
- **Device drivers** are a set of instructions that introduce our PC to a hardware device. Examples are Drivers for keyboard, mouse, printer, speaker, microphone, webcam, etc...
- **System utilities** helps to maintain and protect system. Examples are Disk Compression, Backup Utilities, Disk Cleaners, Anti-Virus, Firewalls
- Main functions of **preprocessor** are:
  - File Inclusion
  - Macro expansions
- **Translator** converts source program to target program. Examples are compiler, interpreter, and assembler.
• **Linker** is a computer program that takes one or more object files generated by a compiler and combines them into one, executable program.
• **Loader** is the part of an operating system that is responsible for loading programs and libraries for execution.

2. **Levels of System Software.**

3. **Difference between system software and application software.**

<table>
<thead>
<tr>
<th>System Software</th>
<th>Application software</th>
</tr>
</thead>
<tbody>
<tr>
<td>The software which provides a platform for the user to interact with the hardware of a computer are known as system software</td>
<td>Those software which runs on an operating system serving specific purpose are called application software</td>
</tr>
<tr>
<td>System software are needed to run application software</td>
<td>Application software are not needed to run system software.</td>
</tr>
</tbody>
</table>
Run in background and act as a platform. | Run in foreground and interact with user.
---|---
Example: language processor, OS, disk drivers, etc... | Example: video player, text editors, browser, etc...

4. **System Programming**
System programming involves designing and writing computer programs that allow the computer hardware to interface with the programmer and the user, leading to the effective execution of application software on the computer system.

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.

5. **Classification of Computer Languages**
- To write a program, we require a computer language.
- Types of computer languages are:

```
Types of Languages
  Machine Language
  Assembly Language
  High level Language
```

**Machine Language**
- It is the lowest level of programming language.
- Machine languages are the only languages understood by computers.
- While easily understood by computers, machine languages are almost impossible for us to use because they consist entirely of numbers.
- It is machine dependent language.

**Assembly Language**
- Assembly language uses symbolic codes or mnemonics as instruction.
Examples of symbolic codes are:
- ADD (addition)
- SUB (subtraction)
- LDA (load accumulator)
- STA (store accumulator)

The processing of an assembly language program is done by using a language translator called assembler that translates assembly language code into machine code.

High Level Languages
- High level languages (HLL) are much closer to English-like language.
- High level languages (HLL) were developed to overcome large time consumption and cost in developing machine and assembly languages.
- A separate language translator is required to translate HLL computer programs into machine readable object code.

6. Life Cycle of 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.

7. **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.

8. **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.

**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 interfaces facilitate transfer of data between the user and the computing system.

9. Address Space
   - The amount of space allocated for all possible addresses for data and other computational entities is called address space.
   - Types of address space are:
     1. 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.
     2. 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 ranges between 0 and some maximum limits supported by the machine.

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

<table>
<thead>
<tr>
<th>Banking System</th>
<th>Hospital Management</th>
<th>Web Browser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compilers</td>
<td>Editors</td>
<td>Command Interpreters</td>
</tr>
<tr>
<td>Operating System</td>
<td>Machine Language</td>
<td>Microarchitecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Physical devices</td>
</tr>
</tbody>
</table>

**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 a 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.

11. Different views on the machine of a program

- Abstract model is a programmer view.
- Object code is a translator view of a program.
- Executable code is view of linker.
- Loader view is an executable program in the computer memory for execution.
- Machine code is hardware view.
1. Define 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.

   ![Diagram of Language Processors](image)

   - **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

2. **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.

3. Fundamental of language processor

Language Processing = Analysis of Source Program + Synthesis of Target Program
Lexical Analysis

- Lexical Analyzer divides the given source statement into the tokens.
  1. Categories of Tokens are:
     2. Identifier
     3. Keyword
     4. Operator
     5. Constant
     6. Special symbol
     7. Format of token is:
- Lexical analyzer also builds symbol table.
- Consider following code

\[
\begin{align*}
i & : \text{integer;} \\
a, b & : \text{real;} \\
a &= b + i;
\end{align*}
\]

The statement \(a=b+i\) is represented as a string of token

\[
\begin{array}{c|c|c|c|c}
\text{a} & \text{=} & \text{b} & + & \text{i} \\
\hline
\text{Id#1} & \text{Op#1} & \text{Id#2} & \text{Op#2} & \text{Id#3}
\end{array}
\]

Syntax Analysis

- Syntax Analysis is also called Parsing or Hierarchical Analysis.
- Syntax analysis processes the string of token to determine its grammatical structure and builds the tree structure.
- Consider the statement \(a = b + i\) can be represented in tree form as
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 and length of a symbol to the symbol table.
- While evaluating the expression the type of b is real and i is int so type of i is converted to real i*.

Intermediate code generation

Following two important properties of intermediate code are:

1. It should be easy to produce.
2. Easy to translate into target program.

<table>
<thead>
<tr>
<th>symbol</th>
<th>Type</th>
<th>length</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>i*</td>
<td>real</td>
<td></td>
</tr>
</tbody>
</table>
Intermediate code

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

Memory allocation

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>length</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i</td>
<td>int</td>
<td>2000</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>real</td>
<td>2001</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>real</td>
<td>2002</td>
</tr>
</tbody>
</table>

Code generation

Two key decisions involved in generating good quality target code are:

1. Which instructions should be used for each of the actions in the intermediate code?
2. Which CPU registers should be used for evaluating expressions?
   - CONV_R   AREG, I
   - ADD_R     AREG, B
   - MOVEM    AREG, A

4. Pass structure of language processor

- One complete scan of a source program is called pass.
- A pass includes reading an input file and writing to the output file.
- A language processor pass is the processing of every statement in a source program, or in its equivalent representation to perform language processing function.

Pass I

- Perform analysis of the source program and note relevant information.
Pass II

- Perform synthesis of the target program.

5. **Symbol table**

- A language processor uses the symbol table to maintain the information about attributes used in a source program.
- Language processor performs following 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 and returns a pointer to the symbol’s entry in the 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.
- Following data structures can be used for organizing symbol table’s entries:
  - **Linear data structure:** Entries in the symbol table occupy contiguous areas of memory.
  - **Nonlinear data structure:** Entries in the symbol table do not occupy contiguous areas of memory.

**Attributes of different classes of symbols:**

<table>
<thead>
<tr>
<th>Symbol Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>Type, length, number and bounds of dimensions</td>
</tr>
<tr>
<td>Procedure</td>
<td>Number of parameters, address of parameter list</td>
</tr>
<tr>
<td>Function</td>
<td>Number of parameters, address of parameter list, type of returned value, length of returned value</td>
</tr>
</tbody>
</table>
Entry formats of the symbol table:
1. Fixed length entries: Each entry in the symbol table has fields for all attributes specified in the programming language.
2. Variable-length entries: the entry occupied by a symbol has fields only for the attributes specified for symbols of its class.
3. Hybrid entries: Combination of previous two.

6. 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:
  o A hash table that contains a fixed array of 'm' pointers to storage table entries.
  o 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

7. 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 (UFO) 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.

<table>
<thead>
<tr>
<th>PARAMETERS &amp; RETURN VAL</th>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Link</td>
<td></td>
</tr>
<tr>
<td>Temporaries &amp; local data</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARAMETERS &amp; RETURN VAL</th>
<th>Callee’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Link</td>
<td></td>
</tr>
<tr>
<td>Temporaries &amp; local data</td>
<td></td>
</tr>
</tbody>
</table>

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
1. **Assembler**
   - Assembler is a language processor that converts assembly language program to machine language program.
   - An assembly language is machine dependent language.
   - It is low level programming language that is specific to certain computer or to the certain family of computers.

2. **Facilities of assembly language**
   - Assembly language provides following three basic facilities:
     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.
     3. **Data declarations**: Data can be declared in a variety of notations, including the decimal notation.

   **Example**: Consider following instruction,
   
   $\text{MOVER AREG,X}$
   
   - $\text{MOVER}$ is a mnemonic opcode for the operation to be performed.
   - $\text{AREG}$ is a register operand in a symbolic form.
   - $\text{X}$ is a memory operand in a symbolic form.
   - Let us consider another instruction for data declaration
   
   $\text{X DS 1}$
   
   - DS (Declare storage) reserves area of memory.
   - Name of variable is $\text{X}$
   - It reserves a memory area of 1 word and associates the name $\text{X}$ with it

3. **Assembly language statement format**
   - An assembly language statement has the following format:
     
     $[\text{Label}] <\text{Opcode}> <\text{operand specification}>[,<\text{operand specification}>..]$  

     Where the notation $[..]$ indicates that the enclosed specification is optional.

   $<\text{operand specification}>$ has the following syntax:

   $<\text{symbolic name}> [\pm <\text{displacement}> ] [(<\text{index register}>)]$
Thus, some possible examples of operand forms are as follows:

- **AREA:** It returns a memory word with which name AREA is associated.
- **AREA+5:** Refers a memory word 5 word away from the word with name AREA. ‘5’ is displacement.
- **AREA(4):** It implies indexing the operand AREA with index register 4. That is the operand address is obtained by adding the content of index register 4 to the address of AREA.
- **AREA+5(4):** It is a combination of previous two specifications.

### 4. Types of assembly language statements

Three types of assembly statements are:

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.

3. **Assembler Directive**
   - Assembler directives instruct the assembler to perform certain action during the assembly program.
   - Assembler directives are not converted into machine code rather they give instruction to
assembler itself.

START

• This directive indicates that first word of machine should be placed in the memory word with address <constant>.
• Syntax: START <Constant>
• Example: START 100
• First word of the target program is stored from memory location 100 onwards.

END

• Syntax: END [<operand specification>]
• This directive indicates end of the source program.
• The operand specification indicates address of the instruction from where the execution of program should begin.
• Execution control should transfer to label given in operand field.
• By default, execution begins with the first instruction of the assembly program.

5. Simple Assembly scheme OR

Assembler Design Criteria OR

Design of Assembler OR

Analysis and synthesis phases of an assembler by clearly stating their tasks.

Design specification for an assembler

• Following steps are used to develop a design specification for an assembler.
  1. Identify the information required to perform a task.
  2. Design a suitable data structure for recording the information.
  3. Determine the processing required for obtaining and maintaining the information.
  4. Determine the processing required for performing the task by using the recorded information.

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>opcode</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>SUB</td>
<td>02</td>
<td>1</td>
</tr>
</tbody>
</table>

Mnemionics table

Source Program \(\rightarrow\) Analysis Phase \(\rightarrow\) Synthesis Phase \(\rightarrow\) Target Program

symbol address

<table>
<thead>
<tr>
<th>AGAIN</th>
<th>104</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>113</td>
</tr>
</tbody>
</table>

Symbol Table

Fig: Design of an Assembler
Analysis phase
- The primary function performed by the analysis phase is to build the symbol table.
- Addresses of symbolic names are required to build a symbol table.
- It is possible to determine some address directly, however others must be inferred. This function is called memory allocation.
- Location Counter (LC) is used to implement a memory allocation.
- The processing involved in maintaining the location counter is called LC processing.

Tasks of analysis phase
1. Separate the content of label, mnemonics opcode, and operand fields of a statement.
2. If a label is present, enter the pair (symbol, <LC content>) in a new entry of the 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: It depends on the source program; hence it must be available by analysis phase. Entry of ONE will in symbol table.
  2. Machine operation code corresponding to mnemonics MOVER: it depends on the assembly language and it can be taken from Mnemonics table.

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. (conversion from assembly language to machine language)

6. **Pass structure of Assembler** OR
**Types of Assembler** OR
**Explain single pass and two pass assembler** OR
**Write difference between one pass and two pass assembler.**
- A complete scan of the program is called pass.
- Types of assembler are:
  1. Two pass assembler (Two pass translation)
  2. Single pass assembler (Single pass translation)
Two pass assembler (Two pass translation)

- The first pass performs analysis of the source program.
- The first pass performs Location Counter processing and records the addresses of symbols in the symbol table.
- It constructs intermediate representation of the source program.
- Intermediate representation consists of following two components:
  1. Intermediate code
  2. Data structures

Fig: Two Pass Assembler

One pass assembler (One pass translation)

- A one pass assembler requires one scan of the source program to generate machine code.
- LC processing, symbol table construction and target code generation proceed in single pass.
- The issue of forward references can be solved using a process called back patching.

7. Define forward references. How it can be solved using back-patching? Explain with example.
   - 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.
   - The operand field of an instruction containing forward references is left blank initially.
   - It builds a Table of Incomplete Instructions (TII) to record information about instructions whose operand fields were left blank.
• Each entry in TII contains a pair in the form of \((\text{instruction address, symbol})\)
• When END statement is processed, the symbol table would contain addresses of all symbols defined in the source program.
• TII would contain information describing all forward references.
• The assembler can now process each entry in TII to complete the concerned instruction.
• Example:

<table>
<thead>
<tr>
<th>Assembly Statements</th>
<th>Memory Location</th>
<th>Opcode</th>
<th>Register Operand</th>
<th>Memory Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>START 101</td>
<td>101</td>
<td>09</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>READ N</td>
<td>102</td>
<td>04</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MOVER BREG, ONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>N DC ‘1’</td>
<td>115</td>
<td>00</td>
<td>0 001</td>
</tr>
<tr>
<td>ONE DS 1</td>
<td>116</td>
<td></td>
<td></td>
<td>No code generation for DS</td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction address</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>N</td>
</tr>
<tr>
<td>102</td>
<td>ONE</td>
</tr>
</tbody>
</table>

8. **Advanced assembler directive**

• Advanced assembler directives are:
  1. ORIGIN
  2. EQU
  3. LTORG

1. **ORIGIN**

• Syntax:

  \[ \text{ORIGIN } <\text{address specification}> \]

• Where \(<\text{address specification}>\) is an \(<\text{operand specification}>\) or \(<\text{constant}>\).
• This directive indicates that LC should be set to the address given by \(<\text{address specification}>\).
• The ORIGIN statement is useful when the target program does not consist of consecutive memory words.
2. EQU

- 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.

<table>
<thead>
<tr>
<th>Assembly Statements</th>
<th>Memory Location</th>
<th>Opcode</th>
<th>Register Operand</th>
<th>Memory Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>START 100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>102)</td>
<td>04</td>
<td>1</td>
<td>117</td>
</tr>
<tr>
<td>MOVER AREG, A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>115)</td>
<td>07</td>
<td>1</td>
<td>102</td>
</tr>
<tr>
<td>LT, BACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACK EQU LOOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No LC counter is processed

3. LTORG

- The LTORG directive, stands for 'origin for literals'.
- LTORG 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 also at the END statement, the assembler allocates memory to the literals of the literal pool then clears the literal pool.

<table>
<thead>
<tr>
<th>Assembly Statements</th>
<th>Memory Location</th>
<th>Opcode</th>
<th>Register Operand</th>
<th>Memory Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>START 200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>200)</td>
<td>04</td>
<td>1</td>
<td>211</td>
</tr>
<tr>
<td>MOVER AREG, =’5’</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>201)</td>
<td>01</td>
<td>3</td>
<td>212</td>
</tr>
<tr>
<td>CREG, =’1’</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTORG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=’5’</td>
<td>211)</td>
<td>00</td>
<td>0</td>
<td>005</td>
</tr>
<tr>
<td>=’1’</td>
<td>212)</td>
<td>00</td>
<td>0</td>
<td>001</td>
</tr>
<tr>
<td>SUB</td>
<td>214)</td>
<td>02</td>
<td>1</td>
<td>219</td>
</tr>
<tr>
<td>AREG, =’1’</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>219)</td>
<td>00</td>
<td>0</td>
<td>001</td>
</tr>
</tbody>
</table>

9. Intermediate code forms

- The Intermediate code consists of a sequence of intermediate code units. (IC units)
Each unit consists of the following three fields:
1. Address
2. Representation of mnemonics opcode
3. Representation of operands

There are two variants of intermediate code:
1. Variant I
2. Variant II

**Intermediate code forms:**
- Intermediate code consist of a set of IC units, each unit consisting of the following three fields
  1. Address
  2. Representation of mnemonics opcode (same in both variant)
  3. Representation of operands (differ in variant I and variant II)

**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.
- Codes for various mnemonics, declaration statements and assembler directives is given in the below table:

**Codes for mnemonics:**

<table>
<thead>
<tr>
<th>Instruction opcode</th>
<th>Assembly mnemonics</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STOP</td>
<td>Stop execution</td>
</tr>
<tr>
<td>01</td>
<td>ADD</td>
<td>Perform addition</td>
</tr>
<tr>
<td>02</td>
<td>SUB</td>
<td>Perform subtraction</td>
</tr>
<tr>
<td>03</td>
<td>MULT</td>
<td>Perform multiplication</td>
</tr>
<tr>
<td>04</td>
<td>MOVER</td>
<td>Memory to Register move</td>
</tr>
<tr>
<td>05</td>
<td>MOVEM</td>
<td>Register to Memory move</td>
</tr>
<tr>
<td>06</td>
<td>COMP</td>
<td>Compare &amp; set condition</td>
</tr>
<tr>
<td>07</td>
<td>BC</td>
<td>Branch on condition</td>
</tr>
<tr>
<td>08</td>
<td>DIV</td>
<td>Perform division</td>
</tr>
<tr>
<td>09</td>
<td>READ</td>
<td>Read into Register</td>
</tr>
<tr>
<td>10</td>
<td>PRINT</td>
<td>Print content of Register</td>
</tr>
</tbody>
</table>
Codes for Condition:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LT</td>
<td>01</td>
</tr>
<tr>
<td>LE</td>
<td>02</td>
</tr>
<tr>
<td>EQ</td>
<td>03</td>
</tr>
<tr>
<td>GT</td>
<td>04</td>
</tr>
<tr>
<td>GE</td>
<td>05</td>
</tr>
<tr>
<td>ANY</td>
<td>06</td>
</tr>
</tbody>
</table>

Codes for Register:

<table>
<thead>
<tr>
<th>Register</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREG</td>
<td>01</td>
</tr>
<tr>
<td>BREG</td>
<td>02</td>
</tr>
<tr>
<td>CREG</td>
<td>03</td>
</tr>
<tr>
<td>DREG</td>
<td>04</td>
</tr>
</tbody>
</table>

Codes for Declaration statement and assembler directives:

<table>
<thead>
<tr>
<th>Declaration statement</th>
<th>Assembler directive</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>01</td>
</tr>
<tr>
<td>DS</td>
<td>02</td>
</tr>
<tr>
<td>START</td>
<td>01</td>
</tr>
<tr>
<td>EQU</td>
<td>04</td>
</tr>
<tr>
<td>END</td>
<td>02</td>
</tr>
<tr>
<td>LTORG</td>
<td>05</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>03</td>
</tr>
</tbody>
</table>

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.
- 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 LITTAB.

**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.

### Intermediate code in variant I & variant II

<table>
<thead>
<tr>
<th>Assembly Program</th>
<th>Variant I</th>
<th>Variant II</th>
</tr>
</thead>
<tbody>
<tr>
<td>START 200</td>
<td>(AD,01) (C, 200)</td>
<td>(AD,01) (C, 200)</td>
</tr>
<tr>
<td>READ A</td>
<td>(IS, 09) (S, 01)</td>
<td>(IS, 09) A</td>
</tr>
<tr>
<td>LOOP MOVER AREG, A</td>
<td>(IS, 04) (1)(S, 01)</td>
<td>(IS, 04) AREG, A</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>SUB AREG, =’1’</td>
<td>(IS, 02) (1)(L, 01)</td>
<td>(IS, 02) AREG,(L, 01)</td>
</tr>
<tr>
<td>BC GT, LOOP</td>
<td>(IS, 07) (4)(S, 02)</td>
<td>(IS, 07) GT, LOOP</td>
</tr>
<tr>
<td>STOP A</td>
<td>(IS, 00)</td>
<td>(IS, 00)</td>
</tr>
<tr>
<td>DS 1</td>
<td>(DL, 02) (C,1)</td>
<td>(DL, 02) (C,1)</td>
</tr>
<tr>
<td>LTORG</td>
<td>(AD, 05)</td>
<td>(AD, 05)</td>
</tr>
</tbody>
</table>

### Comparison of the variants

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

### 10. Data structure of Assembler - Pass I

Describe following data structures: OPTAB, SYMTAB, LITTAB & POOLTAB.

OR

Explain the role of mnemonic opcode table, symbol table, literal table, and pool table in assembling process of assembly language program.
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.

<table>
<thead>
<tr>
<th>Mnemonics opcode</th>
<th>Class</th>
<th>Mnemonics info</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVER</td>
<td>IS</td>
<td>(04,1)</td>
</tr>
<tr>
<td>DS</td>
<td>DL</td>
<td>R#7</td>
</tr>
<tr>
<td>START</td>
<td>AD</td>
<td>R#11</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>202</td>
<td>1</td>
</tr>
<tr>
<td>NEXT</td>
<td>214</td>
<td>1</td>
</tr>
<tr>
<td>LAST</td>
<td>216</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>217</td>
<td>1</td>
</tr>
<tr>
<td>BACK</td>
<td>202</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>218</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>literal</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>=’5’</td>
</tr>
<tr>
<td>2</td>
<td>=’1’</td>
</tr>
<tr>
<td>3</td>
<td>=’1’</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Literal no</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
</tr>
<tr>
<td>#3</td>
</tr>
</tbody>
</table>

### 11. Assembler pass I Algorithm

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 LITTAB 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;
LITTAB[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

12. Assembler pass II Algorithm

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

1. Code_area_address= address of code_areas;
Poohtable_ptr=1;
Loc_cntr=0;
2. While next statement is not an END statement
   a) Clear machine_code_buffer;
   b) If an LTORG 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.

13. 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 match with any mnemonics in OPTAB.
- Statement 21 gives duplicate definition 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.

<table>
<thead>
<tr>
<th>Sr.no</th>
<th>Statements</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>START 200</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MOVER AREG,A</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MVER BREG, A</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td><strong>ERROR</strong> Invalid opcode</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>ADD BREG, B</td>
<td>208</td>
</tr>
<tr>
<td>14</td>
<td>A DS 1</td>
<td>209</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>A DC ‘5’</td>
<td>227</td>
</tr>
<tr>
<td></td>
<td><strong>ERROR</strong> duplicate definition of symbol A</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>END</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>ERROR</strong> undefined symbol B in statement 10</td>
<td></td>
</tr>
</tbody>
</table>

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.
1. **Define macro and its advantages**
   - A macro is a named block of assembly language statements.
   - Macro allows a sequence of source language code to be defined.
   - Once defined, it can be called one or more times.
   - Example: Macro in 'C' Language.
     ```
     #define PI 3.1415926535
     ```
   - The advantages of macro are:
     1. Simplify and reduce the amount of repetitive coding.
     2. Reduce the possibility of errors caused by repetitive coding.
     3. Make an assembly program more readable.

2. **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.
   - Features of macro are:
     - A **Macro prototype statement**: Specifies the name of the macro and name and type of formal parameters.
     - A **Model statements**: Specifies the statements in the body of the macro from which assembly language statements are to be generated during expansion.
     - A **Macro preprocessor statement**: Specifies the statement used for performing auxiliary function during macro expansion.

   **Example: Macro definition and call**
   ```
   MACRO
   INCR &MEM_VAL, &INC_VAL, &REG
   MOVER &REG, &MEM_VAL
   ADD &REG, &INC_VAL
   MOVEM &REG, &MEM_VAL
   MEND
   ```

   - 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>`

   **Macro Call**: The syntax of a typical macro call can be of the following form:
   ```
   <name_of_macro> [<actual_parameter_spec> [...] ]
   ```

   **Example**: INCR A, B

3. **Macro expansion**
   - During macro expansion, the macro name statement in the program is replaced by the
sequence of assembly statements.

• Example:

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

```
START     100
A          DS  1
B          DS  1
+            MOVER    AREG    A
+            ADD       AREG    B
+            MOVEM    AREG    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.

4. **Macro expansion algorithm**

1. MEC: statement number of the first model statement following the prototype statement in the definition of the called macro;
2. While the statement pointed to by MEC is not a MEND statement
   a) If a model statement then
      i. Expand the statement through lexical substitution.
      ii. MEC = MEC + 1;
   b) Else
      MEC = Value specified in the preprocessor statement.
3. Exit from macro expansion.

5. **Types of formal parameter**

Two types of formal parameters are:

1. **Positional parameters**: Order cannot be changed in macro call.
   - Syntax: &<parameter name>
   - The value of a positional formal parameter XYZ is determined by the rule of positional association as follows:
     1. Find the ordinal position of &XYZ in the list of formal parameters in the macro prototype statement.
     2. Find the actual parameter specification that occupies the same ordinal position in the
list of actual parameters in the macro call statement.

Example:
Prototype statement: INCR &MEM_VAL, &INC_VAL, &REG
Macro call: INCR A, B, AREG

2. **Keyword parameters**: Order can be changed in macro call.
   Syntax: &<parameter name>=
   - 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.

   Example:
   Prototype statement: INCR &MEM_VAL=, &INC_VAL=, &REG=
   Macro call: INCR INCR_VAL=B, REG=AREG, MEM_VAL=A

**Macro 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.
Example: SUMUP A, B, G=20, H=X

6. **Advanced macro facilities**

![Advanced macro facilities](image)

**Altering flow of control during expansion**
Expansion time statements are:

1. **AIF (Give suitable example for macro by using conditional expansion)**
   - Syntax: AIF (<expression>) <sequencing symbol>
   - Where <expression> is a relational expression consists of:
     1. Ordinary strings.
     2. Formal parameters with their attributes.
     3. 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.
• Example:

<table>
<thead>
<tr>
<th>MACRO</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL_CONST &amp;A</td>
</tr>
<tr>
<td>AIF (L' &amp;A EQ 1).NEXT</td>
</tr>
<tr>
<td>--</td>
</tr>
<tr>
<td>--</td>
</tr>
<tr>
<td>--</td>
</tr>
<tr>
<td>MEND</td>
</tr>
</tbody>
</table>

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

3. **ANOP**
   - Syntax: <sequencing symbol> ANOP
   - It has the simply effect of defining the sequencing symbol.
   - No operation is carried out by an ANOP instruction. Instead, if a branch is taken to the ANOP instruction, the assembler processes the next sequential instruction.

<table>
<thead>
<tr>
<th>MACRO</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_CONST &amp;X, &amp;Y</td>
</tr>
<tr>
<td>AIF (T' &amp;X EQ B).BYTE</td>
</tr>
<tr>
<td>&amp;Y DW 25</td>
</tr>
<tr>
<td>AGO .OVER</td>
</tr>
<tr>
<td>.BYTE ANOP</td>
</tr>
<tr>
<td>&amp;Y DB 25</td>
</tr>
<tr>
<td>.OVER MEND</td>
</tr>
</tbody>
</table>

**Expansion time variables (Explain expansion time variable in macro)**

- Expansion time variables (EV's) are variables which can only be used during the expansion of macro calls.
- Types of expansion time variables:
  1. **Local expansion time variables**
     - Local expansion time variables can be used only within one macro and it does not retain
its value across the macro call.
- Syntax: LCL <EV specification> [, <EV specification> .. ]
- Example: LCL &A: local variable A is created.

2. **Global expansion time variables**
- Global expansion time variables can be used in every macro definition that has a declaration for it and it retains its value across the macro call.
- Syntax: GBL <EV specification> [, <EV specification> .. ]
- Example: GBL &A: Global variable A is created.

- <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.
- Syntax: <EV specification > SET <SET-expression>

### Attributes of parameter (Explain 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
```

- 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'.

7. **Expansion time loops**
- 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
<table>
<thead>
<tr>
<th>CLEAR</th>
<th>&amp;X, &amp;N</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCL</td>
<td>&amp;M</td>
</tr>
<tr>
<td>&amp;M</td>
<td>SET</td>
</tr>
</tbody>
</table>
```
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 loops (REPT statement)
- Syntax: REPT <expression>
- Example: Declare 10 constants with the values 1, 2... 10 using REPT statement.

Expansion time loops (IRP statement)
- Syntax: IRP <formal parameter>, <argument list>
- The formal parameter mentioned in the statement takes successive values from the argument list.
- For each value the statements between the IRP and ENDM statements are expanded once.
- A macro call CONSTS 4, 10 leads to declaration of 3 constants with the values 4, 7, and 10.
8. **Nested macro call**

- A model statement in a macro may constitute a call on another macro. Such a call is known as a nested macro call.
- The macro that contains the nested call is known as outer macro and the macro that is called in the nested call is known inner macro.
- The macro preprocessor performs expansion of nested macro calls using the last-in first-out (LIFO) rule.

<table>
<thead>
<tr>
<th>MACRO</th>
<th>MACRO</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPUTE &amp;FIRST, &amp;SECOND</td>
<td>INCR_D &amp;MEM VAL=, &amp;INCR VAL=, &amp;REG=BREG</td>
</tr>
<tr>
<td>MOVEM BREG, TMP</td>
<td>MOVER &amp;REG, &amp;MEM VAL</td>
</tr>
<tr>
<td>INCR_D &amp;FIRST, &amp;SECOND, &amp;REG=BREG</td>
<td>ADD &amp;REG, &amp;INC VAL</td>
</tr>
<tr>
<td>MOVEM BREG, TMP</td>
<td>MOVEM &amp;REG, &amp;MEM VAL</td>
</tr>
<tr>
<td>MEND</td>
<td>MEND</td>
</tr>
</tbody>
</table>

**Example:**

```plaintext
COMPUTE X, Y
+ MOVEM BREG TMP [1]
+ INCR_D X, Y
+ MOVER BREG, TMP [5]
```

```plaintext
+ MOVER BREG, X [2]
+ ADD BREG, Y [3]
+ MOVEM BREG, X [4]
```

9. **Design of a macro preprocessor**

- 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 pre-processors.
- A macro preprocessor essentially accepts an assembly program with macro definitions and calls as its input.
- It converts into an equivalent expanded assembly program with no macro definitions and calls.

Tasks involved in macro expansion

1. **Recognizing macro calls**
   - Macro Name Table (MNT) is used to store names of all macros defined in a program.
   - 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.

2. **Determining the values of formal parameters**
   - Preprocessor needs to know the names of formal parameters and default values of keyword parameters.
   - It obtains this information from macro definition and builds a table called Parameter Default Table (PDT) to store the pairs of the form (<formal parameter name>, <default value>) for each macro defined in the program.
   - Actual Parameter Table (APT) is used to holds the values of formal parameters during the expansion of a macro call in the form (<formal parameter name>, <value>).

3. **Maintaining the values of expansion time variables**
   - Expansion time Variable Table (EVT) is used to maintains information about expansion variables in the form (<EV name>, <value>).
   - The expansion time variable is needed when the preprocessor statement or the model statement under expansion refers to an Expansion time variable.

4. **Organizing expansion time control flow**
   - 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 after an expansion of a model statement by a macro preprocessor.

5. **Determine the values of sequencing symbols**
   - Sequencing Symbols Table (SST) is used to 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.

6. **Perform expansion of a model statement**
   - The expansion task has the following steps:
     1. MEC (Macro Expansion Counter) points to the MDT (Macro Definition Table) entry
that contains the model statement.
2. APT (Actual Parameter Table) and EVT (Expansion Time variable Table) provides the values of the formal parameters and EVs, respectively.
3. SST (Sequencing Symbol Table) enables identifying the model statement and defining sequencing symbol.

10. Data structures of the macro preprocessor

<table>
<thead>
<tr>
<th>Table</th>
<th>Fields in each entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro name table (MNT)</td>
<td>Macro name, Number of positional parameters (#PP), Number of keyword parameters (#KP), &quot;Number of expansion time variables (#EV), MDT pointer (MDTP), KPDTAB pointer (KPDP), SSTAB pointer (SSTP)</td>
</tr>
<tr>
<td>Parameter Name Table (PNTAB)</td>
<td>Parameter name</td>
</tr>
<tr>
<td>EV Name Table (EVNTAB)</td>
<td>Expansion time variable name</td>
</tr>
<tr>
<td>SS Name Table (SSNTAB)</td>
<td>Sequencing Symbol name</td>
</tr>
<tr>
<td>Keyword Parameter Default Table (KPDTAB)</td>
<td>Parameter name, default value</td>
</tr>
<tr>
<td>Macro Definition Table (MDT)</td>
<td>Label, Opcode, Operands</td>
</tr>
<tr>
<td>Expansion time Variable Table (EVTAB)</td>
<td>Value</td>
</tr>
<tr>
<td>Actual Parameter Table (APTAB)</td>
<td>Value</td>
</tr>
<tr>
<td>SS Table (SSTAB)</td>
<td>MDT entry #</td>
</tr>
</tbody>
</table>

11. Basic functions of macro processor

Two basic functions of macro processors (two pass macro processor) are:
1. Handling macro definition
2. Handling macro expansion

Handling Macro Definition

- On encountering the MACRO directive in the source assembly program, the assembler is being changed from the regular mode to a special macro definition mode, wherein it does the following activities:
  1. Analyzes the available space in the MDT.
  2. Reads continually the statements and writes them to the MDT until the MEND directive is found.
  3. When a MEND directive is found in the source file, the assembler reverts to the normal mode.
4. 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.

Handling macro expansion

- The tasks performed in the macro expansion mode are as follows:
  1. Reading a source statement from the MDT entry.
  2. Writing the statement on the new source listing.
  3. Repeating step 1 and 2 until the end of the macro is located in the MDT.

12. Algorithm: Macro Definition

1. \text{SSNTAB\_ptr} := 1; \text{PNTAB\_ptr} := 1;
2. Process the macro prototype statement and form the MNT entry
   (a) name := macro name; \#PP = 0; \#KP = 0;
   (b) For each positional parameter
      i. Enter parameter name in PNTAB[PNTAB\_ptr].
      ii. PNTAB\_ptr := PNTAB\_ptr + 1;
      iii. \#PP := \#PP + 1;
   (c) KPDTAB := KPDTAB\_ptr;
   (d) For each keyword parameter
      i. Entry parameter name and default value (if any), in KPDTAB [KPDTAB\_ptr].
      ii. Enter parameter name in PNTAB [PNTAB\_ptr].
      iii. KPDTAB\_ptr := KPDTAB\_ptr +1;
      iv. PNTAB\_ptr := PNTAB\_ptr + 1;
      v. \#KP := \#KP + 1;
   (e) MDTP := MDT\_ptr;
   (f) \#EV := 0;
   (g) SSTP := SSTAB\_ptr;
3. While not a MEND statement
   (a) If an LCL statement then
      i. Enter expansion time variable name in EVNTAB.
      ii. \#EV := \#EV + 1;
   (b) If a model statement then
      i. If label field contains a sequencing symbol then
         If symbol is present in SSNTAB then
            q := entry number in SSNTAB;
         else
            Enter symbol in SSNTAB[SSNTAB\_ptr].
            q := SSNTAB\_ptr;
            SSNTAB\_ptr := SSNTAB\_ptr + 1;
            SSTAB[SSTP + q – 1] := MDT\_ptr;
      ii. For a parameter, expansion variable, and sequencing symbol generate the specification (P, \#n), (E, \#n) and (S, \#n) respectively.
      iii. Record the IC in MDT [MDT\_ptr];
      iv. MDT\_ptr := MDT\_ptr + 1;
(c) If a preprocessor statement then
   i. If a SET statement
      Search each expansion time variable name used in the statement in
      EVNTAB and generate the specification (E, #n).
   ii. If an AIF or AGO statement then
      If sequencing symbol used in the statement is present in SSNTAB then
      \[ q := \text{entry number in SSNTAB}; \]
      else
      \[ q := \text{SSNTAB}_{\text{ptr}}; \]
      \[ \text{SSNTAB}_{\text{ptr}} := \text{SSNTAB}_{\text{ptr}} + 1; \]
      Replace the symbol by \((S, \text{SSTP} + q - 1)\).
   iii. Record the IC in MDT \([\text{MDT}_{\text{ptr}}]\).
   iv. \[ \text{MDT}_{\text{ptr}} := \text{MDT}_{\text{ptr}} + 1; \]

4. **(MEND statement)**
   a) Record the intermediate code of the statement in MDT \([\text{MDT}_{\text{ptr}}]\);
   b) \[ \text{MDT}_{\text{ptr}} = \text{MDT}_{\text{ptr}} + 1 \]
   c) If SSNTAB \(_{\text{ptr}} = 1\) (i.e. SSNTAB is empty) then \(\text{SSTP} := 0\);
   d) If \#KP = 0 then KPDTAB \(_{\text{KP}} = 0\)

13. **Algorithm: Macro Expansion**

1. Perform initialization for the expansion of a macro
   (a) \(\text{MEC} := \text{MDTP field of the MNT entry}\);
   (b) Create EVTAB with \#EV entries and set EVTAB\(_{\text{ptr}}\).
   (c) Create APTAB with \#PP+\#KP entries and set APTAB\(_{\text{ptr}}\).
   (d) Copy keyword parameter defaults from the entries KPDTAB \([\text{KPDTP}]\) ... \(\text{KPDTAB}[\text{KPDTP} + \#KP - 1]\) into APTAB\([\#PP+1]... APTAB[\#PP+\#KP].\)
   (e) Process Positional parameters in the actual parameter list copy them into APTAB \([1]... APTAB[\#PP].\)
   (f) For keyword parameters in the actual parameter list Search the keyword name in parameter name field of KPDTAB \([\text{KPDTP}]... \text{KPDTAB}[\text{KPDTP} + \#KP-1].\)

2. While statement pointed by \(\text{MEC}\) is not MEND statement
   a. If a model statement then
      i. Replace operands of the form \((P, \#n)\) and \((E, \#m)\) by values in APTAB \([n]\) and
         EVTAB \([m]\) respectively.
      ii. Output the generated statement.
      iii. \(\text{MEC} := \text{MEC} + 1;\)
   b. If a SET statement with the specification \((E, \#m)\) in the label field then
      i. Evaluate the expression in the operand field and set an appropriate Value
         in EVTAB \([m]\).
      ii. \(\text{MEC} := \text{MEC} + 1;\)
   c. If an AGO statement with \((S, \#s)\) in operand field then
MEC := SSTAB[SSTP+s -1];
    d. If an AIF statement with (S,#s) in operand field then
        If condition in the AIF statement is true then
            MEC := SSTAB [SSTP+s - 1];

3. Exit From macro expansion.

14. **Macro assembler and its design**
   - A macro processor is functionally independent of the assembler.
   - The output of the macro processor will be a part of the input into the assembler.
   - A macro processor scans and processes the 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.

**Advantages**
- It ensures that many functions need not be implemented twice. (Example: Analysis of a source statement to detect macro calls requires us to process the mnemonics field. A similar function is required in the first pass of the assembler.
- 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.

**Disadvantages**
- The resulting pass by combining macro processing and pass 1 of the assembler may be too large and sometimes suffer from core memory problems.
- Sometimes increase the complexity of program translation, which may not be desired.

15. **Pass structure of macro assembler**
   - To design pass structure of a macro assembler we identify those functions of the macro preprocessor and the conventional assembler that can be merged to advantage.
   - After merging, the functions can be structured into passer of the macro assembler.
   - It leads to the following pass structure.

**Pass I**
- Macro definition processing
- Entering of names and types of symbols in the SYMTAB

**Pass II**
- Macro expansion
- Memory allocation and LC processing
- Processing of literals
- Intermediate code generation
16. **Design issues of macro processor**

- **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.

- **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.

17. **Design features of macro processor**

- **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.

18. **Macro processor design options**
   - Recursive macro Expansion
   - General-purpose macro processors
   - Macro processing within language translators
   - Line-by-Line macro processor
1. **Schematic of program’s execution.**

![Diagram of program execution](image)

- Steps of program execution:
  1. Translation
  2. Linking
  3. Relocation
  4. Loading

2. **Explain Linker & task of linker.**

- The linker is a system program that combines the code of a target program with codes of other programs and library routines.
- Language translator builds an object module for a program which contains:
  
  `Target Code of program + Information about other programs and library routines needed to invoke during execution`

- The Linker performs following tasks:
  - Extracts the information from the object module
  - Locates the needed programs and routines
  - Combines them with the target code of the program
  - Produces an executable program in machine (binary) language

3. **Define following terms.**

- **Translation time address:** Translation time address is used at the translation time. This address is assigned by translator.
- **Linked time address:** Link time address is used at the link time. This address is assigned by linker.
- **Load time address:** Load time address is used at the load time. This address is assigned by loader.
- **Translated origin:** Address of origin assumed by the translator (address specified in ORIGIN or START or default).
- **Linked origin:** Address of origin assumed by the linker while producing a binary program.
- **Load origin:** Address of origin assumed by the loader while loading the program for execution.
4. **Relocation and 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 ≠ translated origin, relocation must be performed by the linker.
- If load origin ≠ 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 ≠ φ 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 αi ∈ AA.
  - An address constant: a data word which contains an address αi ∈ AA.

**Relocation (how relocation is performed explain with example)**

- Relocation_factorp = l_originp - t_originp .....(1)
- T_symb = t_originp + d_symb (dsym_b is the offset of symb in P)
- l_symb = t_originp + d_symb

Using (1)
- l_symb = Relocation_factorp + t_originp + d_symb
- = Relocation_factorp + t_symb .....(2)

**Example:**

![Relocation Example Diagram](image)

**Linking**

- Consider an application program AP consisting of a set of program units SP = {Pi}.
- A program unit Pi interacts with another program unit Pj by using addresses of Pj’s instructions and data in its own instructions.
- To realize such interactions, Pj and Pi must contain public definitions and external references.
as defined in the following:

- **Public definition**: a symbol pub_symb defined in a program unit which may be referenced in other program units.
- **External reference**: a reference to a symbol ext_symb which is not defined in the program unit.

5. **EXTRN AND ENTRY statements**

- An **ENTRY** statement in a program unit lists the public definitions of the program unit.
- An **EXTERN** statement lists the symbols to which external references are made in the program unit.
- **Example**:

```
ORIGIN 500
ENTRY TOTAL
EXTERN MAX, ALPHA
READ A

LOOP
...
...
MOVER AREG, ALPHA
BC ANY, MAX
...
...
BC LT, LOOP
STOP
A DS 1
TOTAL DS 1
END
```

6. **Object module**

- The object module consists of:
  1. Header contains translated origin, size and execution start address.
  2. Program contains the machine language program.
  3. Relocation table describes the instructions that require relocation. Each RELOCTAB entry contains a single field, translated address.
  4. Linking table contains information about the public definitions and external references. Each LINKTAB entry contains – symbol, type and translated address.
Example:

| ORIGIN   | 500 |
| ENTRY    | TOTAL, A |
| EXTERN   | MAX, ALPHA |
| READ     | A |
| LOOP     | 500, 09, 0, 540 |
| ...      | 501 |
| ...      | |
| MOV服务业 | AREG, ALPHA |
| BC       | ANY, MAX |
| ...      | |
| BC       | LT, LOOP |
| STOP     | |
| A DS 1   | |
| TOTAL DS 1 | |
| END      | |
| Assembly Program | |
| Translated Program | |

7. **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. program_linked_origin := <link origin> from the linker command;
  2. For each object module mentioned in the linker command
     (a) t_origin := translated origin of the object module;
        OMsize := size of the object module;
     (b) relocation_factor := program_linked_origin – t_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 address\_in\_work\_area.
(f) \( \text{Program\_linked\_origin} := \text{program\_linked\_origin} + \text{OM\_size}; \)

2) Linking (How external reference is resolved in 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. \( \text{program\_linked\_origin} := <\text{link origin}> \text{from the linker command}. \)
     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, program\_linked\_origin) in NTAB.
        (f) For each LINKTAB entry with type = PD
            name := 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) \( \text{t\_origin} := \text{translated origin of the object module}; \)
        \( \text{program\_linked\_origin} := \text{linked\_adress from NTAB}; \)
(b) For each LINKTAB entry with type = EXT
   i. address_in_work_area := address of work_area + program_linked_origin - <link origin> in linker command + translated address – 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.

8. **Self-Relocating Programs**
   - Types of Programs are:
     1. A non-relocatable program
     2. A relocatable program
     3. A Self-relocatable program
   - A self-relocating program is a program which can **perform its own relocation of address sensitive instructions**.
   - It contains following two provisions for this purpose:
     1. A table of information concerning the address sensitive instructions exists as a part of the program.
     2. 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.

9. **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.

10. Types of Linking

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 is encountered during 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 lined to its program as well.
• 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 only function is to call the dynamic linker and pass the name of the external symbol to it. Hence, the dynamic linker would be activated only when an external reference is identified during execution.

11. 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.

12. 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
  o Saves memory and makes it available for the user program as loaders are smaller in size than assemblers. The loader replaces the assembler.
  o 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.
  o This scheme allows use of subroutines in several different languages because the object files processed by the loader utility will all be in machine language.

• Disadvantages
  o The loader is more complicated and needs to manage multiple object files.
  o Secondary storage is required to store object files, and they cannot be directly placed into the memory by assemblers.

13. Absolute Loaders
• An absolute loader loads a binary program in memory for execution.
• The binary program is stored in a file contains the following:
  o A Header record showing the load origin, length and load time execution start address of the program.
  o 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
  o Simple to implement and efficient in execution.
  o Saves the memory (core) because the size of the loader is smaller than that of the assembler.
  o 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.
  o The loader is simpler and just obeys the instruction regarding where to place the object code in the main memory.

• Disadvantages
  o 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.
  o 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.
  o If the subroutine is modified, the program has to be assembled again from first to last.

14. 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:
  o A Header record showing the load origin, length and load time execution start address of the program.
  o 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.
  o 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. code_load_address = code_linked_address + relocation_factor
     c. byte_count = count of the number of bytes in the record
     d. Move byte_count bytes from the record to the memory area with start address code_load_address
  5. Read RELOCTAB of the program
  6. For each entry in the RELOCTAB
     a. Instruction_linked_address = address specified in the RELOCTAB entry
     b. instruction_load_address = instruction_linked_address + relocation_factor
     c. Add relocation_factor to the operand address used in the instruction that has the address instruction_load_address

15. 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
  1. Object program
  2. Reference about other program to be accessed
  3. Information about address sensitive entities.
Let us consider a program segment as shown below:

```
  ADD AREG, X
```

Offset=30

```
  ADD AREG, X
```

Offset=10

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.

```
  ADD AREG, X
```

Offset=10

```
  ADD AREG, X
```

Offset=30

```
  ADD AREG, X
```

500

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.

16. 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.

17. 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.
1. **Define following terms:**

1. **Graphical symbol:** 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 the Greek symbol $\Sigma$.  
   e.g. $\Sigma = \{a, b,...,z, 0, 1,...9\}$

2. **Formal language:** A formal language is a collection of valid sentences, where each sentence is a sequence of words, and each word is a sequence of graphic symbols acceptable in a language.

3. **Formal language grammar:** Set of rules that specify the construction of words and sentences is called formal language grammar.

4. **Programming language grammar:** A grammar $G$ of a language $L_G$ is a quadruple $(\Sigma, SNT, S, P)$ where
   
   $\Sigma$ is the alphabet of $L_G$, i.e. the set of terminal symbols  
   SNT is the set of nonterminal symbols  
   $S$ is the distinguished symbol (Start symbol)  
   $P$ is the set of productions  

   e.g. $<\text{Noun Phrase}> \rightarrow <\text{Article}><\text{Noun}>$  
   $<\text{Article}> \rightarrow a | an | the$  
   $<\text{Noun}> \rightarrow \text{boy} | \text{apple}$

**Terminal symbol:** A symbol in the alphabet is known as a terminal symbol.

**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 $<$, e.g., A or $<\text{Noun}>$.

**Production:** A production, also called a rewriting rule, is a rule of grammar. It has the form  
$A$ nonterminal symbol $\rightarrow$ String of terminal and nonterminal symbols  
Where the notation ‘$\rightarrow$’ stands for ‘is defined as’. Eg. $<\text{Noun Phrase}> \rightarrow <\text{Article}><\text{Noun}>$

**Start symbol:** First nonterminal symbol of the grammar is called start symbol.

2. **Classification of grammar**

**Type-0 grammar**

- This grammar is also known as phrase structure grammar.
- 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.
- Example: $S \rightarrow A\text{CaB}$  
  $Bc \rightarrow acB$  
  $CB \rightarrow DB$  
  $aD \rightarrow Db$

**Type-1 grammar**

- Their productions are of the form:
\[ \alpha A \beta \rightarrow \alpha \pi \beta \]

- Where \( A \) is non terminal and \( \alpha, \beta, \pi \) are strings of terminals and non-terminals.
- The strings \( \alpha \) and \( \beta \) may be empty, but \( \pi \) must be non-empty.
- 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.
- 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.
- These grammars are also not relevant for programming language specification since recognition of programming language constructs is not context sensitive in nature.
- Example: \( AB \rightarrow AbBc \)
  \[ A \rightarrow bcA \]
  \[ B \rightarrow b \]

**Type-2 grammar**

- This grammar is also known as Context Free Grammar (CFG).
- Their productions are of the form:

  \[ A \rightarrow \pi \]

- Where \( A \) is non terminal and \( \pi \) is string of terminals and non terminals.
- These grammars do not impose any context requirements on derivations or reductions which can be applied independent of its context.
- CFGs are ideally suited for programming language specification.
- Example: \( S \rightarrow Xa \)
  \[ X \rightarrow a \]
  \[ X \rightarrow aX \]
  \[ X \rightarrow abc \]

**Type-3 grammar**

- These grammar is also known as linear grammar or regular grammar.
- Their productions are of the form:

  \[ A \rightarrow tB \mid t \quad \text{or} \quad A \rightarrow Bt \mid t \]

- Where \( A, B \) are non terminals and \( t \) is terminal.
- The specific form of the RHS alternatives - namely a single terminal symbol or a string containing a single terminal and a single nonterminal.
- 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.
- Example: \( X \rightarrow a \mid aY \)
  \[ Y \rightarrow b \]
  \[ Z \rightarrow c \]
  \[ A \rightarrow dX \mid Zc \]
3. Derivation
   • Let production P₁ of grammar G be of the form P₁: A → α and let β be a string such that β = γAθ, then replacement of A by α in string β constitutes a derivation according to production P₁.

   \[
   \begin{align*}
   \text{<Noun Phrase>} & \rightarrow \text{<Article> <Noun>} \\
   & \rightarrow \text{the <Noun>} \\
   & \rightarrow \text{the boy}
   \end{align*}
   \]

   • Types of derivations are:
     1. Leftmost derivation
     2. Rightmost derivation

Leftmost Derivation
   • A derivation of a string W in a grammar G is a left most derivation if at every step the left most non terminal is replaced.
   • Grammar: \( S → S+S \mid S-S \mid S*S \mid S/S \mid a \)
   • Output string: \( a*a-a \)
     \[
     \begin{align*}
     & \rightarrow S \\
     & \rightarrow S-S \\
     & \rightarrow S*S-S \\
     & \rightarrow a*S-S \\
     & \rightarrow a*a-S \\
     & \rightarrow a*a-a
     \end{align*}
     \]

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

Rightmost Derivation
• A derivation of a string \( W \) in a grammar \( G \) is a right most derivation if at every step the right most non terminal is replaced.
• It is also called canonical derivation.
• Grammar: \( S \rightarrow S+S \mid S-S \mid S*S \mid S/S \mid a \)
• Output string: \( a*a-a \)

\[ 
S \\
\rightarrow S \\
\rightarrow S*S \\
\rightarrow S*S-S \\
\rightarrow S*S-a \\
\rightarrow S*a-a \\
\rightarrow a*a-a \\
\]

Figure: Parse tree

4. Ambiguous grammar
• In formal language grammar, ambiguity would arise if identical string can occur on the RHS of two or more productions.
• For example, if a grammar have the productions:
  \( N_1 \rightarrow \alpha \)
  \( N_2 \rightarrow \alpha \)
• The string \( \alpha \) can be derived from or reduced to either \( N_1 \) or \( N_2 \).
• Ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.
• Grammar: \( S \rightarrow S+S \mid S*S \mid (S) \mid a \)
• Output string: \( a+a*a \)

\[ 
S \\
\rightarrow S \\
\rightarrow S*S \\
\rightarrow S+S*S \\
\rightarrow a+S*S \\
\rightarrow a+a*S \\
\rightarrow a+a*a \\
\]

Figure: Parse tree for ambiguous grammar
• Here, Two leftmost derivation for string \( a+a*a \) is possible hence, above grammar is ambiguous.
Elimination of Ambiguous grammar
Grammar: \( S \rightarrow S+S \ | S*S \ | (S) \ | a \)
Equivalent unambiguous grammar is
\( S \rightarrow S + T \ | T \)
\( T \rightarrow T * F \ | F \)
\( F \rightarrow (S) \ | a \)
\[→S\]
\[→ S+T\]
\[→ T+T\]
\[→ F+T\]
\[→ a+T\]
\[→ a+T*F\]
\[→ a+F*F\]
\[→ a+a*F\]
\[→ a+a*a\]
• Here, two left most derivation is not possible for string a+a*a hence, grammar is unambiguous.

5. Difference between Parse tree & Abstract syntax tree
Parse tree: it shows the steps in the parsing of a source string according to a grammar, so it is useful for understanding the process of parsing.

Abstract syntax tree: it represents the structure of a source string in more economical manner.

6. Reduction
• Let production P1 of grammar G be of the form P1: \( A \rightarrow \alpha \) 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 P1.
7. **Current sentential form**
   - Current sentential form is any string derivable from start symbol.
   - **Grammar:** $S \rightarrow S+S \mid S-S \mid S*S \mid S/S \mid a$
   - **Output string:** $a+a*a$
     $$
     \begin{align*}
     \rightarrow & S \\
     \rightarrow & S*S \\
     \rightarrow & S+S*S \\
     \rightarrow & a+S*S \\
     \rightarrow & a+a*S \\
     \rightarrow & a+a*a \\
     \end{align*}
     $$
   $$

8. **Scanning**
   - Scanning considers an input program to be a string of characters and identifies lexical units.
   - Lexical units are operators, identifiers, keywords and constants.
   - A scanner is a program that identifies lexical units in a program, enters each lexical units into a relevant table and constructs a token to represent it.

9. **Regular expression**
   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. ends with 0
      $$(0|1)^*0$$
10. Binary no. starts and ends with 1.
   1(0|1)*1
11. String starts and ends with same character.
   0(0|1)*0 or a(a|b)*a
   1(0|1)*1 b(a|b)*b
12. All string of a and b starts with a
   a(a|b)*
13. String of 0 and 1 ends with 00.
   (0|1)*00
14. String ends with abb.
   (a|b)*abb
15. String starts with 1 and ends with 0.
   1(0|1)*0
16. All binary string with at least 3 characters and 3rd character should be zero.
   (0|1)(0|1)(0|1)*
17. Language which consist of exactly two b’s over the set Σ={a,b}
   a*ba*ba*
18. Σ ={a,b} such that 3rd character from right end of the string is always a.
   (a|b)*a(a|b)(a|b)
19. Any no. of a followed by any no. of b followed by any no. of c.
   a*b*c*
20. String should contain at least 3 one.
   0|1)*1(0|1)*1(0|1)*1(0|1)*
21. String should contain exactly two 1’s
   0*10*10*
22. Length should be at least be 1 and at most 3.
   (0|1) | (0|1)(0|1) | (0|1)(0|1)(0|1)
23. No.of zero should be multiple of 3
   (1*01*01*01*)*+1*
24. Σ ={a,b,c} where a are multiple of 3.
   ((b|c)*a(b|c)*a(b|c)*a(b|c))*
25. Even no. of 0.
   (1*01*01*)*
26. Odd no. of 1.
   0*(10*10*)*10*
27. String should have odd length.
   (0|1)((0|1)(0|1))*
28. String should have even length.
   ((0|1)(0|1))*
29. String start with 0 and has odd length.
   0((0|1)(0|1))*
30. String start with 1 and has even length.
   1((0|1)(0|1)(0|1))*
31. Even no of 1
   \((0^*10^*10^*)^*\)

32. String of length 6 or less
   \((0|1|\epsilon)^6\)

33. String ending with 1 and not contain 00.
   \((1|01)^*\)

34. All string begins or ends with 00 or 11.
   \((00|11)(0|1)^*  \ |  (0|1)^*(00|11)\)

35. All string not contains the substring 00.
   \((1|01)^* (\epsilon|0)\)

36. Language of all string containing both 11 and 00 as substring.
   \(((0|1)^*00(0|1)^*11(0|1)^*)  \ |  ((0|1)^*11(0|1)^*00(0|1)^*)\)

37. Language of C identifier.
   \(\_|L(L|D)^*\)

\[L \rightarrow A|B|......a|b...|z\]
\[D \rightarrow 0|1|......|9\]

10. Finite automata
    - 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 \(\Sigma\).
    - There are two types of finite automata;
      - Deterministic finite automata (DFA): In DFA for each state exactly one edge leaving out for each symbol.
      - Example: DFA to recognize integer strings.

<table>
<thead>
<tr>
<th>State</th>
<th>Next symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Int</td>
</tr>
<tr>
<td>Int</td>
<td>Int</td>
</tr>
</tbody>
</table>

- Nondeterministic finite automata (NFA): In NFA there are no restrictions on the edges leaving a state. There can be several with the same symbol as label and some edges can be labeled with \(\epsilon\).

11. Thompson’s rules
    1. For \(\epsilon\), construct the NFA
2. For \( \alpha \) in \( \Sigma \), construct the NFA

![NFA](image1)

**Figure: Thompson’s rule for a**

3. For regular expression \( s|t \)

![NFA](image2)

**Figure: Thompson’s rule for \( s|t \)**

4. For regular expression \( st \)

![NFA](image3)

**Figure: Thompson’s rule for \( st \)**

5. For regular expression \( s^* \)

![NFA](image4)

**Figure: Thompson’s rule for \( s^* \)**

12. NFA to DFA for \((a+b)^*abb\) using subset construction method.

![NFA](image5)

**Figure: NFA for \((a+b)^*abb\)**
• $\varepsilon$ – closure (0) = \{0,1,2,4,7\} ---- A
• Move(A,a) = \{3,8\}
  $\varepsilon$ – closure (Move(A,a)) = \{1,2,3,4,6,7,8\} ---- B
  Move(A,b) = \{5\}
  $\varepsilon$ – closure (Move(A,b)) = \{1,2,4,5,6,7\} ---- C
• Move(B,a) = \{3,8\}
  $\varepsilon$ – closure (Move(B,a)) = \{1,2,3,4,6,7,8\} ---- B
  Move(B,b) = \{5,9\}
  $\varepsilon$ – closure (Move(B,b)) = \{1,2,4,5,6,7,9\} ---- D
• Move(C,a) = \{3,8\}
  $\varepsilon$ – closure (Move(C,a)) = \{1,2,3,4,6,7,8\} ---- B
  Move(C,b) = \{5\}
  $\varepsilon$ – closure (Move(C,b)) = \{1,2,4,5,6,7\} ---- C
• Move(D,a) = \{3,8\}
  $\varepsilon$ – closure (Move(D,a)) = \{1,2,3,4,6,7,8\} ---- B
  Move(D,b) = \{5,10\}
  $\varepsilon$ – closure (Move(D,b)) = \{1,2,4,5,6,7,10\} ---- E
• Move(E,a) = \{3,8\}
  $\varepsilon$ – closure (Move(E,a)) = \{1,2,3,4,6,7,8\} ---- B
  Move(E,b) = \{5\}
  $\varepsilon$ – closure (Move(E,b)) = \{1,2,4,5,6,7\} ---- C

<table>
<thead>
<tr>
<th>States</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

Table: Transition table

Figure: DFA for (a+b)*abb
DFA Optimization

- Initial partition consists of two groups (E) accepting state and non-accepting states (ABCD).
- E is single state so, cannot be split further.
- For (ABCD), on input a each of these state has transition to B. but on input b, however A, B and C go to member of the group (ABCD), while D goes to E, a member of other group.
- Thus, (ABCD) split into two groups, (ABC) and (D). So, new groups are (ABC) (D) and (E).
- Apply same procedure again no splitting on input a, but (ABC) must be splitting into two group (AC) and (B), since on input b, A and C each have a transition to C, while B has transition to D. so, new groups (AC)(B)(D)(E).
- Now, no more splitting is possible.
- If we chose A as the representative for group (AC), we obtain optimized table shown below,

<table>
<thead>
<tr>
<th>States</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

Table: Optimized transition table

13. Parsing & Types of parsing

- Parsing is a technique that takes input string and produces output either as a parse tree if string is valid sentence of grammar, or an error message indicating that string is not valid.
- Types of parsing are:
  1. **Top down parsing**: Top down parser starts from the root and works up to the leaves.
  2. **Bottom up parsing**: Bottom up parser starts from leaves and works up to the root.

Classification of parsing
14. Naïve top down parsing algorithm
1. Current sentential form (CSF) = ‘S’
2. If CSF is a string of terminal symbols that matches with $\alpha$, Exit with success.
3. Let CSF be of the form $\beta A \pi$, such that $\beta$ is a string of terminals and $A$ is the leftmost non terminal in CSF. Make a derivation $A \rightarrow \beta_1 B \delta$ according to a production $A = \beta B \delta$ of $G$ such that $\beta_1$ is a string of terminals. This makes CSF = $\beta \beta_1 \pi \delta$
4. Go to step 2.

\[ S \]
\[ \beta \]
\[ A \]
\[ \pi \]
\[ CSF = \beta A \pi \]

\[ S \]
\[ \beta \]
\[ A \]
\[ \pi \]
\[ \beta_1 \]
\[ B \]
\[ \delta \]
\[ CSF = \beta \beta_1 A \pi \delta \]

Figure: Making the derivation $A \rightarrow \beta_1 B \delta$ in top down parsing

15. Backtracking
• In backtracking, expansion of nonterminal symbol we choose one alternative and if any mismatch occurs then we try another alternative.
• Grammar: $S \rightarrow cAd$
  Input string: cad
  $A \rightarrow ab | a$

\[ S \]
\[ c \]
\[ A \]
\[ d \]

\[ S \]
\[ c \]
\[ A \]
\[ d \]
\[ a \]
\[ b \]
\[ \text{Backtrack} \]

Figure: Backtracking

16. Left factoring and left recursion

**Left Factoring**
• For each non-terminal $A$ with two or more alternatives (production rules) with a common non empty prefix, let say $A \rightarrow \alpha \beta_1 | \ldots | \alpha \beta_n | \gamma_1 | \ldots | \gamma m$
  Converted it into
  $A \rightarrow \alpha A' | \gamma_1 | \ldots | \gamma m$
  $A' \rightarrow \beta_1 | \ldots | \beta n$
• Example:
  $A \rightarrow xByA | xByAzA | a$
  $B \rightarrow b$
Left factored, the grammar becomes

\[ A \rightarrow xByAA' | a \]
\[ A' \rightarrow zA | \epsilon \]
\[ B \rightarrow b \]

**Left Recursion**

- A grammar is left-recursive if we can find some non-terminal A which will eventually derive a sentential form with itself as the left-symbol.
- Immediate left recursion occurs in rules of the form
  \[ A \rightarrow A\alpha | \beta \]
- Where \( \alpha \) and \( \beta \) are sequences of non-terminals and terminals, and \( \beta \) doesn’t start with \( A \).
- Example: \[ E \rightarrow E+T | T \]
  After removing Left recursion, the grammar becomes
  \[ E \rightarrow TE' \]
  \[ E' \rightarrow +TE' | \epsilon \]

17. **Top down parsing without backtracking.**

- Elimination of backtracking in top down parsing have several advantages.
- Parsing would become more efficient and it would be possible to perform semantic action and precise error reporting during parsing.
- We use left factoring to ensure that the RHS alternatives will produce a unique terminal symbol in first position.
- Consider the grammar
  \[ E \rightarrow T+E | T \]
  \[ T \rightarrow V*T | V \]
  \[ V \rightarrow id \]
- Perform left factoring on given grammar
  \[ E \rightarrow TE' \]
  \[ E' \rightarrow +E | \epsilon \]
  \[ T \rightarrow VT' \]
  \[ T' \rightarrow *T | \epsilon \]
  \[ V \rightarrow id \]
- Now parsing of the string \(<id>+<id>*<id>\)

<table>
<thead>
<tr>
<th>Sr No.</th>
<th>Current Sentential Form</th>
<th>Symbol</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E</td>
<td>&lt;id&gt;</td>
<td>E→TE’</td>
</tr>
<tr>
<td>2</td>
<td>TE’</td>
<td>&lt;id&gt;</td>
<td>T→VT’</td>
</tr>
<tr>
<td>3</td>
<td>VT’E’</td>
<td>&lt;id&gt;</td>
<td>V→&lt;id&gt;</td>
</tr>
<tr>
<td>4</td>
<td>&lt;id&gt;T’E’</td>
<td>+</td>
<td>T’→E</td>
</tr>
<tr>
<td>5</td>
<td>&lt;id&gt;E’</td>
<td>+</td>
<td>E’→+E</td>
</tr>
<tr>
<td>6</td>
<td>&lt;id&gt;+E</td>
<td>&lt;id&gt;</td>
<td>E→TE’</td>
</tr>
<tr>
<td>7</td>
<td>&lt;id&gt;+TE’</td>
<td>&lt;id&gt;</td>
<td>T→VT’</td>
</tr>
<tr>
<td>8</td>
<td>&lt;id&gt;+VT’E’</td>
<td>&lt;id&gt;</td>
<td>V→&lt;id&gt;</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt;+&lt;id&gt;T’E’</td>
<td>*</td>
<td>T’→*T</td>
</tr>
</tbody>
</table>
18. **LL(1) 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.

![Figure: Predictive parser](image)

- **Steps to construct LL(1) parser**
  1. Remove left recursion / perform left factoring.
  2. Compute FIRST and FOLLOW of non-terminals
  3. Construct predictive parsing table.
  4. Parse the input string with the help of parsing table.

**Example:**

- E → E + T | T
- T → T * F | F
- F → (E) | id

**Step 1: Remove left recursion**

- E → TE’
- E’ → +TE’ | ε
- T → FT’

---

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>&lt;id&gt;+&lt;id&gt;*TE’</td>
<td>&lt;id&gt;</td>
<td>T→VT’</td>
</tr>
<tr>
<td>11</td>
<td>&lt;id&gt;+&lt;id&gt;*V T’E’</td>
<td>&lt;id&gt;</td>
<td>V→&lt;id&gt;</td>
</tr>
<tr>
<td>12</td>
<td>&lt;id&gt;+&lt;id&gt;*&lt;id&gt;T’E’</td>
<td>-</td>
<td>T’→ε</td>
</tr>
<tr>
<td>13</td>
<td>&lt;id&gt;+&lt;id&gt;*&lt;id&gt;E’</td>
<td>-</td>
<td>E’→ε</td>
</tr>
<tr>
<td>14</td>
<td>&lt;id&gt;+&lt;id&gt;*&lt;id&gt;</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table: Top down parsing without backtracking**
Step 2: Compute FIRST & FOLLOW

Find FIRST

First(E)
E → TE’
First(E) = First(T) = { (, id}

First(T)
T → FT’
First(T) = First(F) = { (, id}

First(F)
F → (E)
F → id
First(F) = { } 
First(F) = { id

First(E’) = { +, ε
E’ → + TE’
E’ → ε
First(E’) = { +
First(E’) = { ε

First(T’) = { *, ε
T’ → * FT’
T’ → ε
First(T’) = { *
First(T’) = { ε

Find FOLLOW

FOLLOW(E)
Follow of (E) = { $ } (Note: E is start symbol)
F → (E)
Follow (E) = { $, })

FOLLOW(E’)
E → TE’
Follow(E’) = follow(E)
E’ → + TE’
Follow(E’) = follow(E’)
FOLLOW(E’) = { $, }

FOLLOW(T)
E → TE’
Follow(T) = { +, $, }
E’ → + TE’
Follow(T) = { +, $, }
Follow(T) = { +, $, }

FOLLOW(T’)
T → FT’
Follow(T’) = follow(T)
T'→ *FT' follow(T')=follow(T')
FOLLOW(T')= \{+,$,\}\nFOLLOW(F)
T→ FT' follow(F)=\{*,+,$,\}\nT'→ *FT' follow(F)=\{*,+,$,\}\nFOLLOW(F)= \{*,+,$,\}\n
<table>
<thead>
<tr>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
</table>
| E \{\{,id\}\} \{\$,\}\n| E' \{+,\}\{\$,\}\n| T \{\{,id\}\} \{+,\}\{\$,\}\n| T' \{*,\}\{+,\}\{\$,\}\n| F \{\{,id\}\} \{*,+,\}\{\$,\}\n
Table: First & Follow set

Step 3: Predictive Parsing Table

<table>
<thead>
<tr>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E→TE'</td>
<td></td>
<td>E→TE'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>E'→+TE'</td>
<td>E'→\epsilon</td>
<td>E'→\epsilon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T→FT'</td>
<td></td>
<td>T→FT'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>T'→\epsilon</td>
<td>T'→*FT'</td>
<td>T'→\epsilon</td>
<td>T'→\epsilon</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F→id</td>
<td></td>
<td>F→(E)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table: Predictive parsing table

Step 4: Parse the string

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$</td>
<td>id+id*id$</td>
<td></td>
</tr>
<tr>
<td>TE'$</td>
<td>id+id*id$</td>
<td>E→TE'</td>
</tr>
<tr>
<td>FT'E'$</td>
<td>id+id*id$</td>
<td>T→FT'</td>
</tr>
<tr>
<td>idT'E'$</td>
<td>id+id*id$</td>
<td>F→id</td>
</tr>
<tr>
<td>T'E'$</td>
<td>+id*id$</td>
<td></td>
</tr>
<tr>
<td>E'$</td>
<td>+id*id$</td>
<td>T'→\epsilon</td>
</tr>
<tr>
<td>+TE'$</td>
<td>+id*id$</td>
<td>E'→+TE'</td>
</tr>
<tr>
<td>TE'$</td>
<td>id*id$</td>
<td></td>
</tr>
<tr>
<td>FT'E'$</td>
<td>id*id$</td>
<td>T→FT'</td>
</tr>
<tr>
<td>idT'E'$</td>
<td>id*id$</td>
<td>F→id</td>
</tr>
<tr>
<td>T'E'$</td>
<td>*id$</td>
<td></td>
</tr>
<tr>
<td>*FT'E'$</td>
<td>*id$</td>
<td>T'→*FT'</td>
</tr>
</tbody>
</table>
19. Recursive decent parsing

- “A top down parser that executes a set of recursive procedures to process the input without backtracking is called recursive-decent parser, and parsing is called recursive decent parsing”
- Ex:

  \[ E \rightarrow T \{+T\}^* \]
  \[ T \rightarrow V \{*V\}^* \]
  \[ V \rightarrow <id> \]

- A recursive descent parsing program consists of a set of procedures, one for each non terminal.
- Consider RHS of any production rule as definition of the procedure.
- Execution begins with the procedure for the start symbol which halts and announces success if its procedure body scans the entire input string.
- A recursive descent parser example for the given grammar is as follows:

```pascal
procedure proc_E : (tree_root);
/* this procedure constructs an syntax tree for ‘E’ and returns a pointer to its root*/
var
   a , b : pointer to a tree node;
begin
   proc_T(a);
   /* Returns a pointer to the root of tree for T */
   while (nextsymb = ‘+’) do
      match(‘+’);
      proc_T(b);
      a = treebuild(‘+’, a, b);
      /* Builds an syntax tree and returns pointer to its root */
      tree_root = a;
      return;
end proc_E;

procedure proc_T : (tree_root);
var
   a , b : pointer to a tree node;
```
begin
    proc_V(a);
    while (nextsymb = ‘*’) do
        match(‘*’);
        proc_V(b);
        a = treebuild(‘*’, a, b);
        tree_root = a;
        return;
end proc_T;

procedure proc_V(tree_root);
var
    a : pointer to a tree node;
begin
    if (nextsymb = <id>) then
        tree_root = treebuild (<id>, - , -);
    else print “ERROR”;
    return;
end proc_V;

• The procedure proc_E, proc_T and proc_V handle the parsing for E, T, and V respectively and build syntax trees for these NTs using the procedure tree build.
• Procedure ‘match’ increments SSM.
• Proc_E always calls proc_T to perform parsing and syntax tree building for T.
• if the next symbol is ‘+’ then another T is expected. Hence proc_T is called again.
• This process is repeated until the next symbol is not a ‘+’.
• At this point proc_E returns to the parser control routine.
• The control routine checks if an entire string is successfully parsed else indicate an error.

20. Naïve bottom up parsing (Shift reduce parsing)
1. SSM := 1; n := 0;
2. r := n;
3. Compare the string of r symbols to the left of SSM with all RHS alternatives in G which have length of r symbols.
4. If a match is found with a production A ::=α, then
   reduce the string of r symbols to Non terminal A;
   n := n − r + 1;
   Goto step 2;
5. r:=r-1;
   if (r > 0), then goto step 3;
6. If no more symbols exist to the right of SSM then
   If current string form = ‘S’ then
Exit with success;
else report error and exit with failure;

7. SSM := SSM + 1;
   n := n + 1;
   goto step 2;
   • The parser performs a reduction in step 4; we call it a reduce action.
   • The parser performs as many reduce action as possible at current position of SSM
   • When no reduction is possible at the current position, the SSM is incremented by one symbol
     in step 7, which is called shift action.
   • The parsing process thus consists of shift and reduce actions applied in a left to right manner.
   • Hence bottom up parsing is also known as LR parsing or shift reduce parsing.

Problem occurs during parsing:
1. It is insufficient because of the large number of comparisons.
2. It performs reductions in an arbitrary manner, so the reduction may conflict with operator

21. Define following terms:
1. Simple precedence: A grammar symbol a precedes symbol b, where each of a, b is a terminal
   or non-terminal of G, if in a sentential form \( ...ab... \), a should be reduced prior to b in a bottom
   up parsing.
2. Simple precedence grammar: A grammar G is a simple precedence grammar if for all terminal
   and non-terminal symbol a, b of G, a unique precedence relation exist for a, b.
3. Simple phrase: \( \alpha \) is a simple phrase of the sentential form \( ...\alpha\beta... \) if there exist a production of
   the grammar \( A = \alpha \) and \( \alpha \rightarrow A \) is a reduction in the sequence of reduction \( ...A\beta... \rightarrow ... \rightarrow S \).
4. Handle: A handle of a sentential form is the leftmost simple phrase in it.
5. Handle pruning: The process of discovering a handle and reducing it to appropriate LHS non-

22. Operator precedence parsing
Operator Grammar: A Grammar in which there is no \( \in \) in RHS of any production or no adjacent
non terminals is called operator grammar.
• Operator precedence parsing is based on bottom-up parsing techniques and uses a
precedence table to determine the next action.
• The table is easy to construct and is typically hand-coded.
• In operator precedence parsing we define following disjoint relations:

<table>
<thead>
<tr>
<th>Relation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a&lt;b</td>
<td>a “yields precedence to” b</td>
</tr>
<tr>
<td>a=b</td>
<td>a “has the same precedence as” b</td>
</tr>
<tr>
<td>a&gt;b</td>
<td>a “takes precedence over” b</td>
</tr>
</tbody>
</table>

Table: Relation between a and b

• Operator Precedence Matrix for the operators +, *, id, (, ) is given as follows:
• Now consider the grammar
  \[ E \rightarrow E + T \mid T \]
  \[ T \rightarrow T * F \mid F \]
  \[ F \rightarrow \text{id} \]

• String: \text{id} + \text{id} * \text{id}

• 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 \( > \):

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{L.H.S} & \text{R.H.S.} \\
\hline
+ & * & ( & ) & \text{id} & \text{id} \\
\hline
\hline
+ & > & < & < & \text{\textgreater} & \text{\textgreater} \\
* & > & > & < & \text{\textgreater} & \text{\textgreater} \\
( & < & < & < = & < & \text{Error} \\
) & > & > & \text{Error} & \text{\textgreater} & \text{Error} \\
\text{id} & > & > & \text{Error} & \text{\textgreater} & \text{Error} \\
\hline
\hline
\end{array}
\]

\text{Table: Operator precedence matrix}

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{L.H.S} & \text{R.H.S.} \\
\hline
\text{id} & + & \text{id} & * & \text{id} & > \\
\hline
\text{id} & > & \text{id} & > & \text{id} & > \\
\hline
\text{id} & \text{id} & \text{id} & \text{id} & > & \text{id} > \\
\hline
\text{id} & \text{id} & \text{id} & \text{id} & \text{id} & \text{id} \\
\hline
\hline
\end{array}
\]

\text{Table: String parsing}

23. **Operator precedence parsing algorithm**

Data Structures

\textit{Stack:} Each stack entry is a record with two fields, operator and operand_pointer.

\textit{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.

Input
An expression string enclosed between ‘|’ and ‘-‘

1. TOS:= SB-1; SSM=0;
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.

• Here, Consider parsing of the string
   | - <id>a + <id>b * <id>c -|
   according to grammar , where <id>a represents a.
• Figure below shows steps in its parsing.
• Figures (a)-(c) show the stack and the AST when current operator is ’+’, ’*’ and ’-|’ respectively.
• In Fig. (c), TOS operator > current operator.
• This leads to reduction of ’*’. Figure (d) shows the situation after the reduction.
• The new TOS operator, i.e. ’+’, > current operator.
• This leads to reduction of ’+’ as shown in Fig. (e).
24. **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.

**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.

**YACC**
- The input to YACC consists of a syntax specification and a specification of L.
- The syntax specification is a description of the grammar of L.
- The specification of L describes the semantic actions associated with each syntactic construct.

---

**Figure: Using LEX and YACC**

Lexical Specification \[\xrightarrow{\text{LEX}}\] Source Program in L

Syntax Specification \[\xrightarrow{\text{YACC}}\] IR\text{lex}

Specification of L

Intermediate representation (IR)

Scanner

Parser

---

<table>
<thead>
<tr>
<th>Current Operator</th>
<th>Stack</th>
<th>AST</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘+’</td>
<td>SB,TOS</td>
<td>a</td>
</tr>
<tr>
<td>‘*’</td>
<td>SB, TOS</td>
<td>a</td>
</tr>
<tr>
<td>‘-’</td>
<td>SB, TOS</td>
<td>a</td>
</tr>
<tr>
<td>‘*’</td>
<td>SB, TOS</td>
<td>a</td>
</tr>
<tr>
<td>‘-’</td>
<td>SB, TOS</td>
<td>a</td>
</tr>
</tbody>
</table>
• 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 nth 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.
1. **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.

2. **Binding and binding times**
   - Binding is the association of an attribute of a program entity with a value.
   - Binding time is the time at which a binding is actually performed.
   - When compiler processes the statement,
     ```
     int alpha;
     ```
     • In a program, it binds the type of variable alpha to int.
     • To facilitate memory allocation to alpha the size of int should be known at this time, so the size attribute of int should have been bound sometime before the compiler processes the statement int alpha.
   - The following binding times arise in compilers:
     • Language definition time of a programming language L, which is the time at which features of the language are specified. (Example: bind operator symbols to operations)
     • Language implementation time of a programming language L, which is the time at which the design of a language translator for L is finalized. (Example: bind floating point type to a representation)
     • Compilation time of a program P. (Example: bind a variable to a type in C or Java)
     • Execution init time of a procedure proc. (Example: bind a C or C++ static variable to a memory cell)
     • Execution time of a procedure proc. (Example: bind a non static local variable to a
memory cell)

- There are two types of binding:
- **Static Binding**: A static binding is a binding performed before the execution of a program begins.
- **Dynamic Binding**: A dynamic binding is a binding performed after the execution of a program has begun.

3. **Data structure used in compiling**
   - Two types of data structures used in a compiler are:
     - **Stack**
       - The stack is a linear data structure.
       - It follows LIFO (Last In First Out) rule.
       - Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, space is popped off the stack.
     - **Heap**
       - The heap data structure permits allocation and deallocation of entities in a random order.
       - Heap is nonlinear data structure.
       - When a program makes an allocation request, the heap manager allocates a memory area and returns its address.
       - The program is expected to save this address in a pointer and use it to access the allotted entity.
       - When a program makes a deallocation request, the heap manager deallocates a memory area.

4. **Memory allocation** *(State different storage allocation strategies. Explain static allocation and dynamic allocation in detail.)*
   - 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 (Explain 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.

**Scope rules**

- A block in a program is a function, a procedure, or simply a unit of code that may contain data declaration.
- Entities declared in a block must have unique names.
- These entities can be accessed only within the block.
- Thus, places where an entity can be accessed or visible is referred to the **scope of that entity**.
• 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
  }

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Local variables</th>
<th>Non local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>x,y,z</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>a,b</td>
<td>x,y,z</td>
</tr>
<tr>
<td>C</td>
<td>m,n</td>
<td>x,y,z</td>
</tr>
</tbody>
</table>

• 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.

<table>
<thead>
<tr>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual parameter</td>
</tr>
<tr>
<td>Control link</td>
</tr>
<tr>
<td>Access link</td>
</tr>
<tr>
<td>Saved M/c status</td>
</tr>
<tr>
<td>Local variables</td>
</tr>
<tr>
<td>Temporaries</td>
</tr>
</tbody>
</table>

1. Temporary values: The temporary variables are needed during the evaluation of expressions. Such variables are stored in the temporary field of activation record.
2. Local variables: The local data is a data that is local to the execution procedure is stored in this field of activation record.
3. 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.

5. Compilation of expression

- The major issues in code generation for expressions are as follows:
  1. Determination of an evaluation order for the operators in an expression.
  2. Selection of instruction to be used in target code.
  3. Use of registers.

A toy code generator for expressions

- A toy code generator has to track both the registers (for availability) and addresses (location of values) while generating the code.
- For both of them, the following two descriptors are used:
- The code generator uses the notation of an operand descriptor to maintain type, length and addressability information for each operand.
- It uses a register descriptor to maintain information about what operand or partial result would be contained in a CPU register during execution of the generated code.

**Operand Descriptor (Explain operand descriptor and register descriptor for a*b)**

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.
3. Ex: a*b

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Addressability</th>
</tr>
</thead>
<tbody>
<tr>
<td>(int, 1)</td>
<td>Address(a)</td>
</tr>
</tbody>
</table>

MOV 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:
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 (Define and explain different intermediate code representations. OR Explain triple, quadruple and indirect triples representation with example)
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 \( x := -a*b + -a*b \)

<table>
<thead>
<tr>
<th>Number</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>minus</td>
<td>a</td>
<td></td>
<td>t1</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>t1</td>
<td>b</td>
<td>t2</td>
</tr>
<tr>
<td>2</td>
<td>minus</td>
<td>a</td>
<td></td>
<td>t3</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>t3</td>
<td>b</td>
<td>t4</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>t2</td>
<td>t4</td>
<td>t5</td>
</tr>
<tr>
<td>5</td>
<td>:=</td>
<td>t5</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Number</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>minus</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>(0)</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>minus</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>(2)</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>(1)</td>
<td>(3)</td>
</tr>
<tr>
<td>5</td>
<td>:=</td>
<td>X</td>
<td>(4)</td>
</tr>
</tbody>
</table>

**Indirect Triples**
• The indirect triple representation the listing of triples is been done. And listing pointers are used instead of using statements.

<table>
<thead>
<tr>
<th>Number</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>minus</td>
<td>a</td>
<td></td>
<td>(0) (11)</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>(11)</td>
<td>b</td>
<td>(1) (12)</td>
</tr>
<tr>
<td>2</td>
<td>minus</td>
<td>a</td>
<td></td>
<td>(2) (13)</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>(13)</td>
<td>b</td>
<td>(3) (14)</td>
</tr>
</tbody>
</table>
6. **Compilation of control structure**

**Parameter passing mechanisms**
- Types of parameter passing methods are:
  
  **Call by value**
  - This is the simplest method of parameter passing.
  - The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function.
  - The operations on formal parameters do not change the values of a parameter.

  **Call by result**
  - It is similar to call by value with one difference.
  - At the time of return from the called function, the values of formal parameters are copied back into corresponding actual parameter.

  **Call by reference**
  - This method is also called as call by address or call by location.
  - The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter.
  - Inside the function, the address is used to access the actual argument used in the call.
  - It means the changes made to the parameter affect the passed argument.

  **Call by name**
  - This is less popular method of parameter passing.
  - Procedure is treated like macro.
  - The procedure body is substituted for call in caller with actual parameters substituted for formals.
  - The local names of called procedure and names of calling procedure are distinct.

7. **Code optimization**

1. **Common sub expressions elimination**
   - Compile time evaluation means shifting of computations from run time to compile time.
   - There are two methods used to obtain the compile time evaluation.
     
     **Folding**
     - In the folding technique the computation of constant is done at compile time instead of run time.
     - Example: length = (22/7)*d
     - Here folding is implied by performing the computation of 22/7 at compile time.

     **Constant propagation**
     - In this technique the value of variable is replaced and computation of an expression is done at compilation time.
     - Example:
pi = 3.14; r = 5;
Area = pi * r * r;

• Here at the compilation time the value of pi is replaced by 3.14 and r by 5 then computation of 3.14 * 5 * 5 is done during compilation.

2. Common sub expressions elimination

• The common sub expression is an expression appearing repeatedly in the program which is computed previously.

• If the operands of this sub expression do not get changed at all then result of such sub expression is used instead of re-computing it each time.

Example:
\[
\begin{align*}
  t1 & := 4 * i \\
  t2 & := a[t1] \\
  t3 & := 4 * j \\
  t4 & := 4 * i \\
  t5 & := n \\
  t6 & := b[t4]+t5
\end{align*}
\]

• 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.

3. Variable propagation

• Variable propagation means use of one variable instead of another.

Example:
\[
\begin{align*}
  x & = \pi; \\
  area & = x * r * r;
\end{align*}
\]

• The optimization using variable propagation can be done as follows, \( area = \pi * r * r; \)

• Here the variable \( x \) is eliminated. Here the necessary condition is that a variable must be assigned to another variable or some constant.

4. Code movement

• There are two basic goals of code movement:

  I. To reduce the size of the code.

  II. To reduce the frequency of execution of code.

Example:
\[
\begin{align*}
  for(i=0;i<=10;i++) & \quad temp = y*5 \\
  \{ & \quad for(i=0;i<=10;i++) \\
  x & = y*5; \\
  k & = (y*5)+50; \\
  \} & \quad \{ x = temp; \\
  \} & \quad k = (temp) + 50;
\end{align*}
\]

Loop invariant computation

• 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];
}

While(i<=N) {
    sum=sum+a[i];
}

5. **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:
     ```c
     for(i=1;i<=50;i++)
     {
         count = i*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:
     ```c
     temp=7;
     for(i=1;i<=50;i++)
     {
         count = temp;
         temp = temp+7;
     }
     ```

6. **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:
     ```c
     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.

8. **Control Flow Analysis**
   - Control flow analysis analyses a program to collect information concerning its structure.
   - The control flow concept of interest are:
     - **Predecessors & Successors**
Unit 7 – Compilers

- Basic block B1 is a predecessor of B2.
- Basic block B2 is a successor of B1.

**Paths, Ancestors & Descendants**

- A path is a sequence of edges such that the destination node of one edge is the source node of the following edge.
- If path exists from B1 to B2, B1 is an Ancestors of B2 and B2 is Descendants of B1.

**Dominateds**

- In a flow graph, a node d dominates n if every path to node n from initial node goes through d only. This can be denoted as ‘d dom n’.
- Every initial node dominates all the remaining nodes in the flow graph.
- Every node dominates itself.
- Node 1 is initial node and it dominates every node as it is initial node.
- Node 2 dominates 3, 4 and 5.

9. **Data flow analysis**

**Available expression**

- An expression x+y is available at a program point w if and only if along all paths are reaching to w.
  
  I. The expression x+y is said to be available at its evaluation point.
  II. The expression x+y is said to be available if no definition of any operand of the expression (here either x or y) follows its last evaluation along the path. In other word, if neither of the two operands get modified before their use.
• Expression $4 \times i$ is the available expression for $B_2$, $B_3$ and $B_4$ because this expression has not been changed by any of the block before appearing in $B_4$.

**Live variable**

• A variable $\text{var}$ is said to be live at a program point $\text{pi}$ in a basic block $\text{bi}$ if the value contained in $\text{var}$ at $\text{pi}$ is likely to be used during subsequent execution of the program.

• Otherwise the variable is said to be dead at the point.
   - **Definition:** Interpreter is a system software that reads a program written in source language and translates it into an equivalent program in target language line by line.
   - **Types of interpreters:**
     1. **Pure interpreter:** The source program is maintained in the source form throughout its interpretation.
     2. **Impure interpreter:** Perform some preliminary processing of the source program to reduce the analysis overhead during interpretation. Pre-processor converts program to an IR (Intermediate representation) which is used during interpretation.

![](image1.png) Fig: Pure interpreter

![](image2.png) Fig: Impure interpreter

**Advantages of interpreter**
- Meaning of source statement is implemented using interpretation routine which results to simplified implementation.
- Avoid generation of machine language instruction.
- Helps make interpretation portable.
- Interpreter itself is coded in high level language.

**Dis-advantage of interpreter**
- Expensive in terms of CPU time.

**Components of interpreter**
1. Symbol Table: Holds information concerning entities present in program.
2. Data Store: Contains values of data items declared.
3. Data Manipulation Routines: A set containing a routine for every legal data manipulation actions in the source language.

2. Types of errors
   Error can be classified into mainly two categories,
   1. Compile time error
2. Runtime error

![Diagram of Types of Error]

**Lexical Error**
This type of errors can be detected during lexical analysis phase. Typical lexical phase errors are,
1. Spelling errors. Hence get incorrect tokens.
2. Exceeding length of identifier or numeric constants.
3. Appearance of illegal characters.
Example:
```
fi ( )
{
}
```
- In above code 'fi' cannot be recognized as a misspelling of keyword if rather lexical analyzer will understand that it is an identifier and will return it as valid identifier. Thus misspelling causes errors in token formation.

**Syntax error**
These types of error appear during syntax analysis phase of compiler.
Typical errors are:
1. Errors in structure.
2. Missing operators.
3. Unbalanced parenthesis.
- The parser demands for tokens from lexical analyzer and if the tokens do not satisfy the grammatical rules of programming language then the syntactical errors get raised.

**Semantic error**
This type of error detected during semantic analysis phase.
Typical errors are:
1. Incompatible types of operands
2. Undeclared variable
3. Not matching of actual argument with formal argument
3. Debugging procedure
   • 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.
   • 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 procedure:
1. Assertion
   • 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.
   • \textit{If expression evaluates to TRUE, assert () does nothing.}
   • \textit{If expression evaluates to FALSE, the execution program halts.}
   ```
   #include <assert.h>
   #include <stdio.h>
   int main()
   {
      int a;
      printf("Enter an integer value: ");
      scanf("%d", &a);
      assert(a >= 10);
      printf("Integer entered is %d\n", a);
   }
   ```

2. 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.

4. Classification of Debuggers
   There are two types of debuggers
   1. Static Debugging
      • Static debugging focuses on semantic analysis.
      • Static debugging detects errors before the actual execution.
Unit 8 – Interpreters & Debuggers

- Static code analysis may include detection of the following situations:
  1. Truncation of value due to wrong assignment
  2. Redeclaration of variables
  3. Presence of unreachable code

2. 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:
  1. Execution sequencing: It is nothing but observation and control of the flow of program execution.
  2. 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.
  3. Conditional expressions: A debugger can include statements in a program to ensure that certain conditions are reached in the program.
  4. Tracing: Tracing monitors step by step the execution of all executable statements present in a program.
  5. 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.
  6. Program-display capabilities: While debugging is in progress, the program being debugged must be made visible on the screen along with the line numbers.
  7. Multilingual capability: The debugging system must also consider the language in which the debugging is done.
  8. Optimization: A debugger may use an optimizing compiler that deals with the following issues:
     1. Removing invariant expressions from a loop
     2. Merging similar loops
     3. Eliminating unnecessary statements
     4. Removing branch instructions

5. Java language environment
- Java language environment has four key features:
  1. The Java virtual machine (JVM), which provides portability of Java programs.
  2. An impure interpretive scheme, whose flexibility is exploited to provide a capability for inclusion of program modules dynamically, i.e., during interpretation.
  3. 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.
  4. An optional Java just-in-time (JIT) compiler, which provides efficient execution.
- Figure below 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.

- The Java virtual machine essentially interprets the byte code 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 above Figure. 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 above. 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.

6. **Java virtual machine**

- A Java compiler produces a binary file called a class file which contains the byte code 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 byte code verifier, which ensures that execution of the byte code 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.