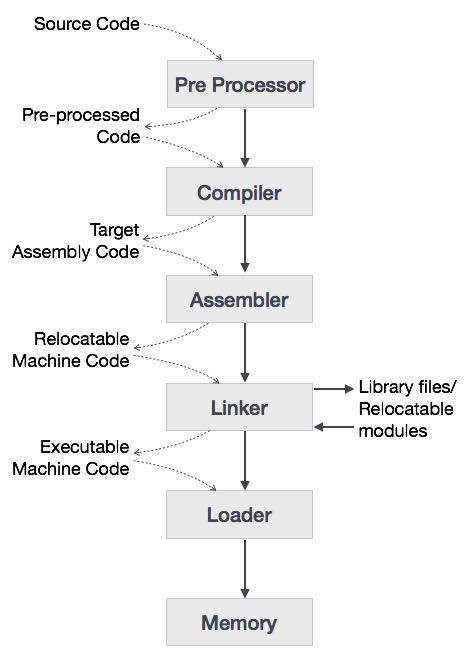
|  |  |  |
| --- | --- | --- |
| **BASIS FOR COMPARISON** | **SYSTEM SOFTWARE** | **APPLICATION SOFTWARE** |
| Basic | System Software manages system resources and provides a platform for application software to run. | Application Software, when run, perform specific tasks, they are designed for. |
| type | specific | General |
| Classification | Package Program  Customized Program | Time sharing  Resource sharing  Client server |
| Task level | System software runs in the background and the users typically do not interact with it. | The application software run in the foreground, and the users interact with it frequently for all their computing needs. |
| Language | System Software is written in a low-level language, i.e. assembly language. | Application Software is written in a high-level language like Java, C++, .net, VB, etc. |
| Run/Execution | System Software starts running when the system is turned on, and runs till the system is shut down./ Throughput | Application Software runs as and when the user requests./According to requirement |
| control | System software can function independent of the application software. | The application software depends on the system software and cannot run without it. |
| ratio | There are much fewer system software as compared to application software. | There are many more application software as compared to system software. |
| Workgroup support | Network operating systems | Electronic mail, group scheduling, shared work |
| Requirement | A system is unable to run without system software. | Application software is even not required to run the system; it is user specific. |
| Examples | Operating system.  Complier, debuggers  Windows OS, BIOS, device firmware, Mac OS X, Linux etc. | Microsoft Office, Photoshop, Animation Software, etc.  Windows Media Player, Adobe Photoshop, World of Warcraft (game), iTunes, MySQL etc. |
|  |

Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by a compatible software. Hardware understands instructions in the form of electronic charge, which is the counterpart of binary language in software programming. Binary language has only two alphabets, 0 and 1. To instruct, the hardware codes must be written in binary format, which is simply a series of 1s and 0s. It would be a difficult and cumbersome task for computer programmers to write such codes, which is why we have compilers to write such codes.

**Language Processing System**

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.



The high-level language is converted into binary language in various phases. A **compiler** is a program that converts high-level language to assembly language. Similarly, an **assembler** is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using C compiler, is executed on a host machine.

* User writes a program in C language (high-level language).
* The C compiler, compiles the program and translates it to assembly program (low-level language).
* An assembler then translates the assembly program into machine code (object).
* A linker tool is used to link all the parts of the program together for execution (executable machine code).
* A loader loads all of them into memory and then the program is executed.

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

**Preprocessor**

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

**Interpreter**

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

**Assembler**

An assembler translates assembly language programs into machine code.The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

**Linker**

Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

**Loader**

Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

**Cross-compiler**

A compiler that runs on platform (A) and is capable of generating executable code for platform (B) is called a cross-compiler.

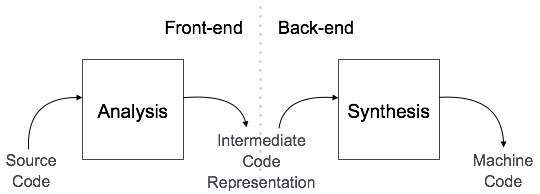
Source-to-source Compiler

A compiler that takes the source code of one programming language and translates it into the source code of another programming language is called a source-to-source compiler.

A compiler can broadly be divided into two phases based on the way they compile.

**Analysis Phase**

Known as the front-end of the compiler, the **analysis** phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.



**Synthesis Phase**

Known as the back-end of the compiler, the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.

A compiler can have many phases and passes.

* **Pass** : A pass refers to the traversal of a compiler through the entire program.
* **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

**PHASES OF COMPLIER**

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



**Lexical Analysis**

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

**Syntax Analysis**

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

**Semantic Analysis**

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

**Intermediate Code Generation**

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

**Code Optimization**

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

**Code Generation**

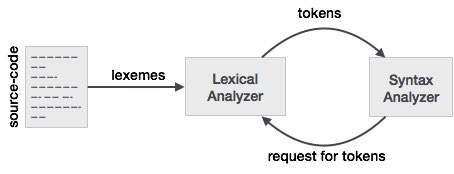
In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

**Symbol Table**

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



**Tokens**

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

int value = 100;

contains the tokens:

int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).

**Specifications of Tokens**

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

Alphabets

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

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by |tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

**Special Symbols**

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

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

**Language**

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

**For example:**

int intvalue;

While scanning both lexemes till ‘int’, the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

Operations

The various operations on languages are:

* Union of two languages L and M is written as

L U M = {s | s is in L or s is in M}

* Concatenation of two languages L and M is written as

LM = {st | s is in L and t is in M}

* The Kleene Closure of a language L is written as

L\* = Zero or more occurrence of language L.

**Notations**

If r and s are regular expressions denoting the languages L(r) and L(s), then

* **Union** : (r)|(s) is a regular expression denoting L(r) U L(s)
* **Concatenation** : (r)(s) is a regular expression denoting L(r)L(s)
* **Kleene closure** : (r)\* is a regular expression denoting (L(r))\*
* (r) is a regular expression denoting L(r)

**Precedence and Associatively**

* \*, concatenation (.), and | (pipe sign) are left associative
* \* has the highest precedence
* Concatenation (.) has the second highest precedence.
* | (pipe sign) has the lowest precedence of all.

Representing valid tokens of a language in regular expression

If x is a regular expression, then:

* x\* means zero or more occurrence of x.

i.e., it can generate { e, x, xx, xxx, xxxx, … }

* x+ means one or more occurrence of x.

i.e., it can generate { x, xx, xxx, xxxx … } or x.x\*

* x? means at most one occurrence of x

i.e., it can generate either {x} or {e}.

[a-z] is all lower-case alphabets of English language.

[A-Z] is all upper-case alphabets of English language.

[0-9] is all natural digits used in mathematics.

Representing occurrence of symbols using regular expressions

letter = [a – z] or [A – Z]

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]

sign = [ + | - ]

Representing language tokens using regular expressions

Decimal = (sign)?(digit)+

Identifier = (letter)(letter | digit)\*

he only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

**Finite automata**

**Finite automata** is a state machine that takes a string of symbols as input and changes its state accordingly. Finite automata is a recognizer for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand. The mathematical model of finite automata consists of:

* Finite set of states (Q)
* Finite set of input symbols (Σ)
* One Start state (q0)
* Set of final states (qf)
* Transition function (δ)

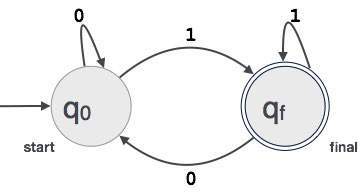
The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), Q × Σ ➔ Q

Finite Automata Construction

Let L(r) be a regular language recognized by some finite automata (FA).

* **States** : States of FA are represented by circles. State names are written inside circles.
* **Start state** : The state from where the automata starts, is known as the start state. Start state has an arrow pointed towards it.
* **Intermediate states** : All intermediate states have at least two arrows; one pointing to and another pointing out from them.
* **Final state** : If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles. It may have any odd number of arrows pointing to it and even number of arrows pointing out from it. The number of odd arrows are one greater than even, i.e. **odd = even+1**.
* **Transition** : The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows points to the destination state. If automata stays on the same state, an arrow pointing from a state to itself is drawn.

**Example** : We assume FA accepts any three digit binary value ending in digit 1. FA = {Q(q0, qf), Σ(0,1), q0, qf, δ}



Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

|  |  |
| --- | --- |
| We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.  CFG, on the other hand, is a superset of Regular Grammar, as depicted below | Relation of CFG and Regular Grammar |

It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

## Context-Free Grammar

In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.

A context-free grammar has four components:

* A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
* A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.
* A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.
* One of the non-terminals is designated as the start symbol (S); from where the production begins.

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

### Example

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

G = ( V, Σ, P, S )

Where:

V = { Q, Z, N }

Σ = { 0, 1 }

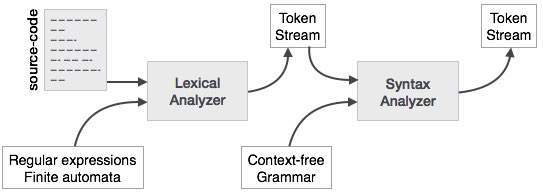
P = { Q → Z | Q → N | Q → ℇ | Z → 0Q0 | N → 1Q1 }

S = { Q }

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

## Syntax Analyzers

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



This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies, which we will learn later in this chapter.

## Derivation

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

* Deciding the non-terminal which is to be replaced.
* Deciding the production rule, by which, the non-terminal will be replaced.

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

### Left-most Derivation

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

### Right-most Derivation

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

**Example**

Production rules:

E → E + E

E → E \* E

E → id

Input string: id + id \* id

The left-most derivation is:

E → E \* E

E → E + E \* E

E → id + E \* E

E → id + id \* E

E → id + id \* id

Notice that the left-most side non-terminal is always processed first.

The right-most derivation is:

E → E + E

E → E + E \* E

E → E + E \* id

E → E + id \* id

E → id + id \* id

## Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

We take the left-most derivation of a + b \* c

The left-most derivation is:

E → E \* E

E → E + E \* E

E → id + E \* E

E → id + id \* E

E → id + id \* id

Step 1:

|  |  |
| --- | --- |
| E → E \* E | Parse Tree Construction |

Step 2:

|  |  |
| --- | --- |
| E → E + E \* E | Parse Tree Construction |

Step 3:

|  |  |
| --- | --- |
| E → id + E \* E | Parse Tree Construction |

Step 4:

|  |  |
| --- | --- |
| E → id + id \* E | Parse Tree Construction |

Step 5:

|  |  |
| --- | --- |
| E → id + id \* id | Parse Tree Construction |

In a parse tree:

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

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

## Ambiguity

A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

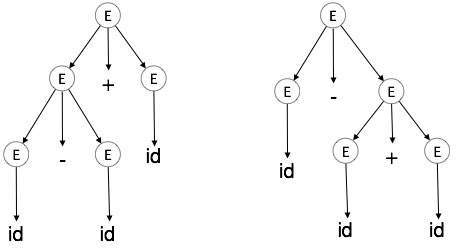
**Example**

E → E + E

E → E – E

E → id

For the string id + id – id, the above grammar generates two parse trees:



The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

## Associatively

If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

**Example**

Operations such as Addition, Multiplication, Subtraction, and Division are left associative. If the expression contains:

id op id op id

it will be evaluated as:

(id op id) op id

For example, (id + id) + id

Operations like Exponentiation are right associative, i.e., the order of evaluation in the same expression will be:

id op (id op id)

For example, id ^ (id ^ id)

## Precedence

If two different operators share a common operand, the precedence of operators decides which will take the operand. That is, 2+3\*4 can have two different parse trees, one corresponding to (2+3)\*4 and another corresponding to 2+(3\*4). By setting precedence among operators, this problem can be easily removed. As in the previous example, mathematically \* (multiplication) has precedence over + (addition), so the expression 2+3\*4 will always be interpreted as:

2 + (3 \* 4)

These methods decrease the chances of ambiguity in a language or its grammar.

## Left Recursion

A grammar becomes left-recursive if it has any non-terminal ‘A’ whose derivation contains ‘A’ itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

**Example:**

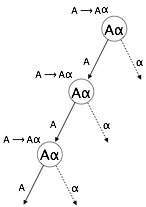
(1) A => Aα | β

(2) S => Aα | β

A => Sd

(1) is an example of immediate left recursion, where A is any non-terminal symbol and α represents a string of non-terminals.

(2) is an example of indirect-left recursion.



A top-down parser will first parse the A, which in-turn will yield a string consisting of A itself and the parser may go into a loop forever.

### Removal of Left Recursion

One way to remove left recursion is to use the following technique:

The production

A => Aα | β

is converted into following productions

A => βA'

A'=> αA' | ε

This does not impact the strings derived from the grammar, but it removes immediate left recursion.

Second method is to use the following algorithm, which should eliminate all direct and indirect left recursions.

START

Arrange non-terminals in some order like A1, A2, A3,…, An

for each i from 1 to n

{

for each j from 1 to i-1

{

replace each production of form Ai ⟹Aj𝜸

with Ai ⟹ δ1𝜸 | δ2𝜸 | δ3𝜸 |…| 𝜸

where Aj ⟹ δ1 | δ2|…| δn are current Aj productions

}

}

eliminate immediate left-recursion

END

**Example**

The production set

S => Aα | β

A => Sd

after applying the above algorithm, should become

S => Aα | β

A => Aαd | βd

and then, remove immediate left recursion using the first technique.

A => βdA'

A' => αdA' | ε

Now none of the production has either direct or indirect left recursion.

## Left Factoring

If more than one grammar production rules has a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand.

**Example**

If a top-down parser encounters a production like

A ⟹ αβ | α𝜸 | …

Then it cannot determine which production to follow to parse the string as both productions are starting from the same terminal (or non-terminal). To remove this confusion, we use a technique called left factoring.

Left factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions.

**Example**

The above productions can be written as

A => αA'

A'=> β | 𝜸 | …

Now the parser has only one production per prefix which makes it easier to take decisions.

## First and Follow Sets

An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing T[A, t] = α with some production rule.

### First Set

This set is created to know what terminal symbol is derived in the first position by a non-terminal. For example,

α → t β

That is α derives t (terminal) in the very first position. So, t ∈ FIRST(α).

#### ALGORITHM FOR CALCULATING FIRST SET

Look at the definition of FIRST(α) set:

* if α is a terminal, then FIRST(α) = { α }.
* if α is a non-terminal and α → ℇ is a production, then FIRST(α) = { ℇ }.
* if α is a non-terminal and α → 𝜸1 𝜸2 𝜸3 … 𝜸n and any FIRST(𝜸) contains t then t is in FIRST(α).

First set can be seen as:

https://www.tutorialspoint.com/compiler_design/images/first_formula.jpg

### Follow Set

Likewise, we calculate what terminal symbol immediately follows a non-terminal α in production rules. We do not consider what the non-terminal can generate but instead, we see what would be the next terminal symbol that follows the productions of a non-terminal.

#### ALGORITHM FOR CALCULATING FOLLOW SET:

* if α is a start symbol, then FOLLOW() = $
* if α is a non-terminal and has a production α → AB, then FIRST(B) is in FOLLOW(A) except ℇ.
* if α is a non-terminal and has a production α → AB, where B ℇ, then FOLLOW(A) is in FOLLOW(α).

Follow set can be seen as: FOLLOW(α) = { t | S \*αt\*}

## Limitations of Syntax Analyzers

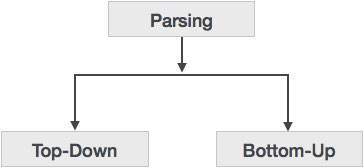
Syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers. Lexical analyzers are responsible for the validity of a token supplied by the syntax analyzer. Syntax analyzers have the following drawbacks -

* it cannot determine if a token is valid,
* it cannot determine if a token is declared before it is being used,
* it cannot determine if a token is initialized before it is being used,
* it cannot determine if an operation performed on a token type is valid or not.

These tasks are accomplished by the semantic analyzer, which we shall study in Semantic Analysis.’

# Types of Parsing

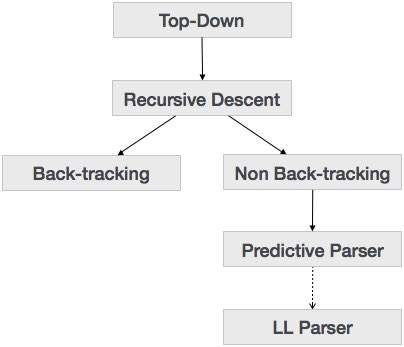
Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types : top-down parsing and bottom-up parsing.



Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

We have learnt in the last chapter that the top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. The types of top-down parsing are depicted below:



Recursive Descent Parsing

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Back-tracking

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

S → rXd | rZd

X → oa | ea

Z → ai

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. ‘e’). The parser tries to expand non-terminal ‘X’ and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

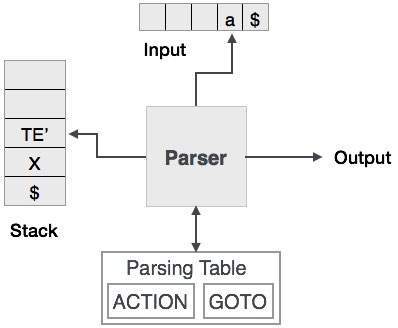
Now the parser matches all the input letters in an ordered manner. The string is accepted.

|  |  |  |  |
| --- | --- | --- | --- |
| Back Tracking | Back Tracking | Back Tracking | Back Tracking |

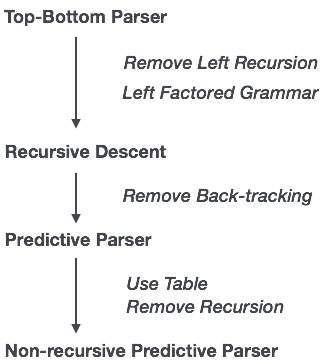
Predictive Parser

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol **$**to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

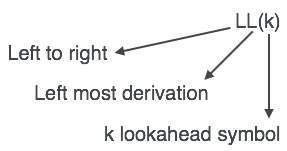


In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

LL Parser

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).



LL Parsing Algorithm

We may stick to deterministic LL(1) for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not LL(1), then usually, it is not LL(k), for any given k.

Given below is an algorithm for LL(1) Parsing:

Input:

string ω

parsing table M for grammar G

Output:

If ω is in L(G) then left-most derivation of ω,

error otherwise.

Initial State : $S on stack (with S being start symbol)

ω$ in the input buffer

SET ip to point the first symbol of ω$.

repeat

let X be the top stack symbol and a the symbol pointed by ip.

if X∈ Vt or $

if X = a

POP X and advance ip.

else

error()

endif

else /\* X is non-terminal \*/

if M[X,a] = X → Y1, Y2,... Yk

POP X

PUSH Yk, Yk-1,... Y1 /\* Y1 on top \*/

Output the production X → Y1, Y2,... Yk

else

error()

endif

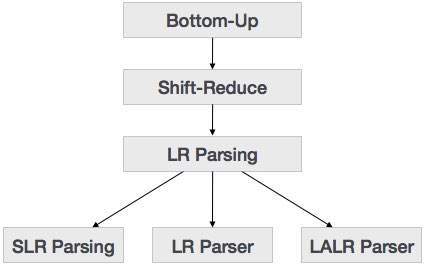
endif

until X = $ /\* empty stack \*/

A grammar G is LL(1) if A → α | β are two distinct productions of G:

* for no terminal, both α and β derive strings beginning with a.
* at most one of α and β can derive empty string.
* if β → t, then α does not derive any string beginning with a terminal in FOLLOW(A).

**Bottom-up parsing** starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.



Shift-Reduce Parsing

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

* **Shift step**: The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
* **Reduce step** : When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

* SLR(1) – Simple LR Parser:
  + Works on smallest class of grammar
  + Few number of states, hence very small table
  + Simple and fast construction
* LR(1) – LR Parser:
  + Works on complete set of LR(1) Grammar
  + Generates large table and large number of states
  + Slow construction
* LALR(1) – Look-Ahead LR Parser:
  + Works on intermediate size of grammar
  + Number of states are same as in SLR(1)

LR Parsing Algorithm

Here we describe a skeleton algorithm of an LR parser:

token = next\_token()

repeat forever

s = top of stack

if action[s, token] = “shift si” then

PUSH token

PUSH si

token = next\_token()

else if action[s, token] = “reduce A::= β“ then

POP 2 \* |β| symbols

s = top of stack

PUSH A

PUSH goto[s,A]

else if action[s, token] = “accept” then

return

else

error()

* **Recursive descent parsing** : It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.
* **Backtracking** : It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Bottom-up Parsing

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

**Example:**

Input string : a + b \* c

Production rules:

S → E

E → E + T

E → E \* T

E → T

T → id

Let us start bottom-up parsing

a + b \* c

Read the input and check if any production matches with the input:

a + b \* c

T + b \* c

E + b \* c

E + T \* c

E \* c

E \* T

E

S

**Semantic analysis**

We have learnt how a parser constructs parse trees in the syntax analysis phase. The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.

For example

E → E + T

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production.

Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

For example:

int a = “value”;

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

* Scope resolution
* Type checking
* Array-bound checking

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

* Type mismatch
* Undeclared variable
* Reserved identifier misuse.
* Multiple declaration of variable in a scope.
* Accessing an out of scope variable.
* Actual and formal parameter mismatch.

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

**Example:**

E → E + T { E.value = E.value + T.value }

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

S → ABC

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

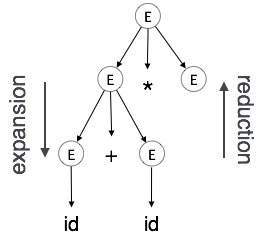
Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

S → ABC

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

**Expansion** : When a non-terminal is expanded to terminals as per a grammatical rule



**Reduction** : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).

Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

For example:

int value = 5;

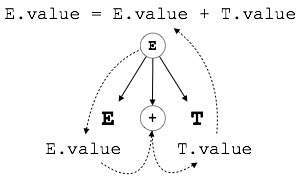
<type, “integer”>

<presentvalue, “5”>

For every production, we attach a semantic rule.

S-attributed SDT

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT

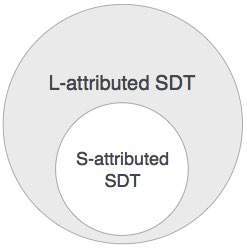
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

S → ABC

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

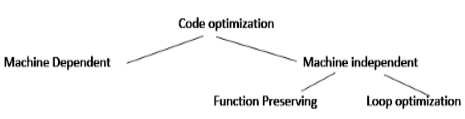
**Optimization**

* Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.
* In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.
* A code optimizing process must follow the three rules given below:
  1. The output code must not, in any way, change the meaning of the program.
  2. Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
  3. Optimization should itself be fast and should not delay the overall compiling process.
* Efforts for an optimized code can be made at various levels of compiling the process.

1. At the beginning, users can change/rearrange the code or use better algorithms to write the code.
2. After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
3. While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

* Optimization can be categorized broadly into two types:

Machine Independent and Machine Dependent.



**Machine Independent Optimization**

* In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations.
* For example:
* do
* {
* item = 10;
* value = value + item;
* } while(value<100);
* This code involves repeated assignment of the identifier item, which if we put this way:
* Item = 10;
* do
* {
* value = value + item;
* } while(value<100);

should not only save the CPU cycles, but can be used on any processor.

**Machine Dependent Optimization**

* Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture.
* It involves CPU registers and may have absolute memory references rather than relative references.
* Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

**Machine independence includes two types**

i. Function Preserving

ii. Loop optimization

* **Function preserving**
* **Common Sub Expression Elimination**

The expression that produces the same results should be removed out from the code

**Example**

|  |  |
| --- | --- |
| BO | AO |
| T1 = 4+i | T1 = 4+i |
| T2 = T2 +T1 | T2 = T2 +T1 |
| T3 = 4 \* i | T4 = T2 + T1 |
| T4 = T2 + T3 |  |

* **Constant folding**

If expression generates a constant value then instead of performing its calculation again and again we calculate it once and assign it.

**Example**

|  |  |
| --- | --- |
| **BO** | **AO** |
| T1 = 512 | T1 = 2.5 |

* **Copy Propagation**

In this propagation a F value is been send to G and G value is been send to H We can eliminate G variable directly assigning the value of F to H.

|  |  |
| --- | --- |
| **BO** | **AO** |
| T1 = X | T2 = T3 + T2 |
| T2 = T3 + T2 | T3 = T1 |
| T3 = X |  |

* **Dead Code Elimination**

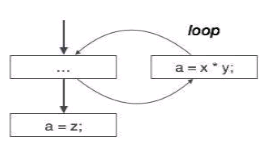
Dead code is one or more than one code statements, which are:

* + Either never executed or unreachable,
  + Or if executed, their output is never used.

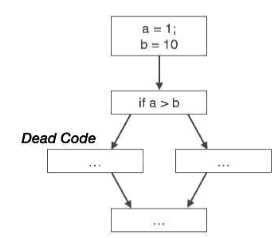
Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

**Partially dead code Elimination**

* There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not.
* Such codes are known as partially dead-code.



* The above control flow graph depicts a chunk of program where variable ‘a’ is used to assign the output of expression ‘x \* y’.
* Let us assume that the value assigned to ‘a’ is never used inside the loop.
* Immediately after the control leaves the loop, ‘a’ is assigned the value of variable ‘z’, which would be used later in the program.
* We conclude here that the assignment code of ‘a’ is never used anywhere, therefore it is eligible to be eliminated.
* Likewise, the picture below depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.



**3. Loop Optimization**

We are going to perform optimization on loops.

* Code Motion

It specifies on a condition if we perform some operations to be carried out and then compare for a condition.

Instead of that perform the calculation outside the loop and assign a value in the calculation.

|  |  |
| --- | --- |
| **BO** | **AO** |
| While(i < = limit-2) { ….. …… … } | T1 = limit – 2 While (i< =t1) { ………………… ………… ………….. } |

* **Strength Reduction**

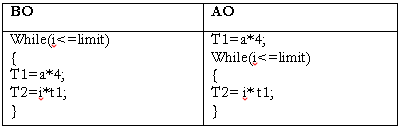
It specifies the operators such as multiplication and division can be replaced by a addition and subtraction respectively.

The multiplication operator can be easily replaced by left shift operator a<<1 Division can be replaced by a a>>1 operator.

|  |  |
| --- | --- |
| **BO** | **AO** |
| T1 = a \* 2 | a<<1 |
| T1 = a / 2 | a >> 1 |

* **Frequency Reduction**

In this case if a expression inside a loop is not dynamically affected by a loop we calculate it outside the loop and use the value inside the loop.



* **Loop Distribution**

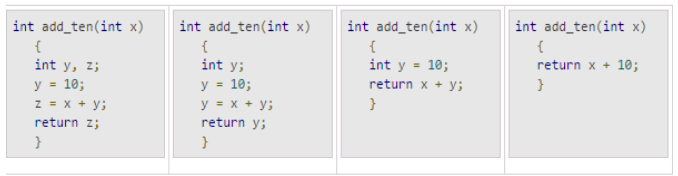
It specifies the values in a particular loop to be assigned to a array keeps of varing i.e the array location in which a loop need to be work again and again. We can use two different loops which allows loop distribution

**Peephole Optimization**

* This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand.
* These methods can be applied on intermediate codes as well as on target codes.
* A bunch of statements is analyzed and are checked for the following possible optimization:

**1. Redundant instruction elimination**

* At source code level, the following can be done by the user:



At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

* MOV x, R0
* MOV R0, R1

We can delete the first instruction and re-write the sentence as:

MOV x, R1

**2. Unreachable code**

* Unreachable code is a part of the program code that is never accessed because of programming constructs.
* Programmers may have accidently written a piece of code that can never be reached.

**Example:**

void add\_ten(int x)

{

return x + 10;

printf(“value of x is %d”, x);

}

* In this code segment, the printf statement will never be executed as the program control returns back before it can execute, hence printf can be removed.

• In this code segment, the printf statement will never be executed as the program control returns back before it can execute, hence printf can be removed.

**3. Flow of control optimization**

* There are instances in a code where the program control jumps back and forth without performing any significant task.
* These jumps can be removed. Consider the following chunk of code:
* ...
* MOV R1, R2
* GOTO L1
* ...
* L1: GOTO L2
* L2: INC R1
* In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:
* ...
* MOV R1, R2
* GOTO L2
* ...
* L2: INC R1

**4. Algebraic expression simplification**

* There are occasions where algebraic expressions can be made simple. For example, the expression a = a + 0 can be replaced by a itself and the expression a = a + 1 can simply be replaced by INC a.

**5. Strength reduction**

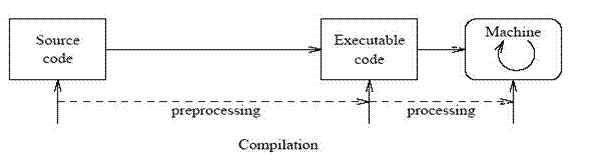
* There are operations that consume more time and space. Their ‘strength’ can be reduced by replacing them with other operations that consume less time and space, but produce the same result.
* For example, x \* 2 can be replaced by x << 1, which involves only one left shift. Though the output of a \* a and a2 is same, a2 is much more efficient to implement.

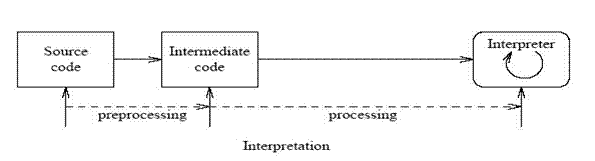
**6. Accessing machine instructions**

* The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently.
* If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient results.
* **SECTION SYSTEM PROGRAMMMING**
* I copy
* **Compiler vs Interpreter**

**Compiler**

* A [compiler](http://www.engineersgarage.com/articles/what-is-compiler-tutorial) is a piece of code that translates the high level language into machine language. When a user writes a code in a high level language such as Java and wants it to execute, a specific compiler which is designed for Java is used before it will be executed. The compiler scans the entire program first and then translates it into machine code which will be executed by the computer processor and the corresponding tasks will be performed.

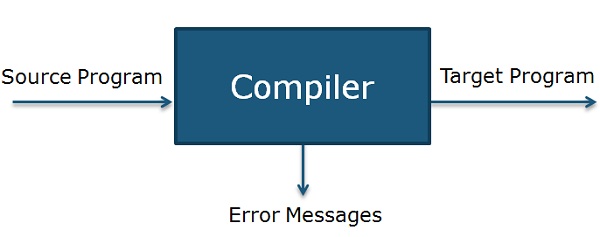


* Shown in the figure is basic outline of the compilation process, here program written in higher level language is known as source program and the converted one is called object program.
* **Interpreter**
* **Interpreters are not much different than compilers**. They also convert the high level language into machine readable binary equivalents. Each time when an interpreter gets a high level language code to be executed, it converts the code into an intermediate code before converting it into the machine code. Each part of the code is interpreted and then execute separately in a sequence and an error is found in a part of the code it will stop the interpretation of the code without translating the next set of the codes.
* 
* Outlining the basic working of the interpreter the above figure shows that first a source code is converted to an intermediate form and then that is executed by the interpreter.

|  |  |  |
| --- | --- | --- |
| **BASIS FOR COMPARISON** | **COMPILER** | **INTERPRETER** |
| Input | It takes an entire program at a time. | It takes a single line of code or instruction at a time. |
| Output | It generates intermediate object code. | It does not produce any intermediate object code. |
| Working mechanism | The compilation is done before execution. | Compilation and execution take place simultaneously. |
| Speed | Comparatively faster | Slower |
| Memory | Memory requirement is more due to the creation of object code. | It requires less memory as it does not create intermediate object code. |
| Errors | Display all errors after compilation, all at the same time. | Displays error of each line one by one. |
| Error detection | Difficult | Easier comparatively |
| Pertaining Programming languages | C, C++, C#, Scala, typescript uses compiler. | Java, PHP, Perl, Python, Ruby uses an interpreter. |

**Definition of Compiler**

A compiler is a program that reads a program written in the high-level language and converts it into the machine or low-level language and reports the errors present in the program. It converts the entire source code in one go or could take multiple passes to do so, but at last, the user gets the compiled code which is ready to execute.



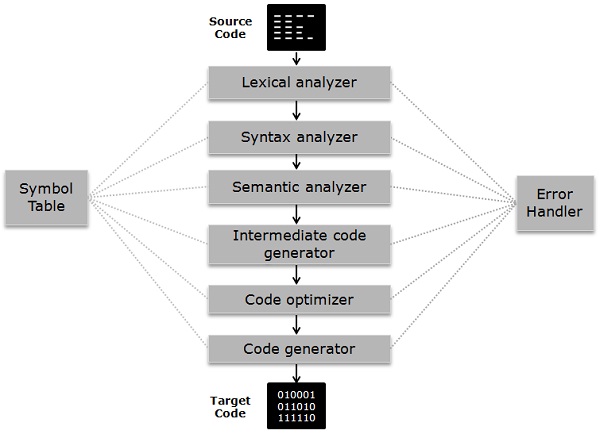
Compiler operates on phases; various stages can be grouped into two parts that are:

* **Analysis Phase** of the compiler is also referred to as the front end in which program is divided into fundamental constituent parts and checks grammar, semantic and syntax of the code after which intermediate code is generated. Analysis phase includes lexical analyzer, semantic analyzer and syntax analyzer.
* **Synthesis phase** of the compiler is also known as the back end in which intermediate code is optimized, and target code is generated. Synthesis phase includes code optimizer and code **generator.**

**PHASES OF COMPILER**

Now let’s understand the working of each stage in detail.

1. **Lexical Analyzer**: It scans the code as a stream of characters, groups the sequence of characters into lexemes and outputs a sequence of tokens with reference to the programming language.
2. **Syntax Analyzer**: In this phase, the tokens that are generated in the previous stage are checked against the grammar of programming language, whether the expressions are syntactically correct or not. It makes parse trees for doing so.
3. **Semantic Analyzer**: It verifies whether the expressions and statements generated in the previous phase follow the rule of programming language or not and it creates annotated parse trees.
4. **Intermediate code generator**: It generates equivalent intermediate code of the source code. There are many representations of intermediate code, but TAC (Three Address Code) is the used most widely.
5. **Code Optimizer**: It improves time and space requirement of the program. For doing so, it eliminates the redundant code present in the program.
6. **Code generator**: This is the final phase of the compiler in which target code for a particular machine is generated. It performs operations like memory management, Register assignment, and machine specific optimization.



The **symbol table** is somewhat a data structure which manages the identifiers along with the relevant type of data it is storing.**Error Handler** detect, report, correct the errors encountering in between the different phases of a compiler.

Definition of Interpreter

The interpreter is an alternative for implementing a programming language and does the same work as a compiler. Interpreter performs **lexing**, **parsing** and **type checking** similar to a compiler. But interpreter processes syntax tree directly to access expressions and execute statement rather than generating code from the syntax tree.

An interpreter may require processing same syntax tree more than once that is the reason why interpretation is comparatively slower than executing the compiled program.

Compilation and interpretation probably combined to implement a programming language. In which a compiler generates intermediate-level code then the code is interpreted rather than compiled to machine code.

Employing an interpreter is advantageous during program development, where the most important part is to be able to test a program modification rapidly rather than run the program efficiently.

**Key Differences Between Compiler and Interpreter**

Let’s look at major differences between Compiler and Interpreter.

1. The compiler takes a program as a whole and translates it, but interpreter translates a program statement by statement.
2. Intermediate code or target code is generated in case of a compiler. As against interpreter doesn’t create intermediate code.
3. A compiler is comparatively faster than Interpreter as the compiler take the whole program at one go whereas interpreters compile each line of code after the other.
4. The compiler requires more memory than interpreter because of the generation of object code.
5. Compiler presents all errors concurrently, and it’s difficult to detect the errors in contrast interpreter display errors of each statement one by one, and it’s easier to detect errors.
6. In compiler when an error occurs in the program, it stops its translation and after removing error whole program is translated again. On the contrary, when an error takes place in the interpreter, it prevents its translation and after removing the error, translation resumes.
7. In a compiler, the process requires two steps in which firstly source code is translated to target program then executed. While in Interpreter It’s a one step process in which Source code is compiled and executed at the same time.
8. The compiler is used in programming languages like C, C++, C#, Scala, etc. On the other Interpreter is employed in languages like Java, PHP, Ruby, Python, etc.

Conclusion

Compiler and interpreter both are intended to do the same work but differ in operating procedure, Compiler takes source code in an aggregated way whereas Interpreter takes constituent parts of source code, i.e., statement by statement.

Although both compiler and interpreter have certain advantages and disadvantages like Interpreted languages are considered as cross-platform, i.e., the code is portable. It also doesn’t need to compile instruction previously unlike compiler which is time-saving. Compiled languages are faster regarding compilation process.

II copy

### What is a compiler?

The simplest definition of a compiler is a program that translates code written in a high-level programming language (like JavaScript or Java) into low-level code (like Assembly) directly executable by the computer or another program such as a virtual machine.

For example, the Java compiler converts [Java](https://en.wikipedia.org/wiki/Java) code to [Java Bytecode](https://en.wikipedia.org/wiki/Java_bytecode) executable by the [JVM](https://en.wikipedia.org/wiki/Java_virtual_machine) (Java Virtual Machine). Other examples are [V8](https://en.wikipedia.org/wiki/V8_%28JavaScript_engine%29), the JavaScript engine from Google which converts JavaScript code to machine code or [GCC](https://en.wikipedia.org/wiki/GNU_Compiler_Collection) which can convert code written in programming languages like C, C++, Objective-C, Go among others to native machine code.



### What’s in the black box?

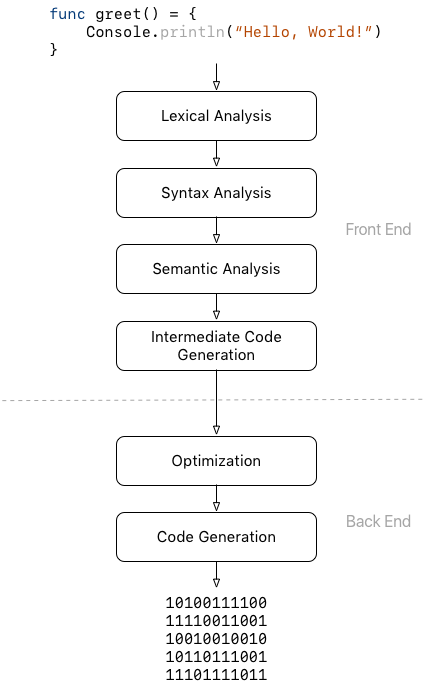
So far we’ve looked at a compiler as a magic black box which contains some spell to convert high-level code to low-level code. Let’s open that box and see what’s inside.

A compiler can be divided into 2 parts.

* The first one generally called **the front end** scans the submitted source code for syntax errors, checks (and infers if necessary) the type of each declared variable and ensures that each variable is declared before use. If there is any error, it provides informative error messages to the user. It also maintains a data structure called **symbol table** which contains information about all the symbols found in the source code. Finally, if no error is detected, another data structure, an intermediate representation of the code, is built from the source code and passed as input to the second part.
* The second part, the **back end** uses the intermediate representation and the symbol table built by the front end to generate low-level code.

Both the front end and the back end perform their operations in a sequence of phases. Each phase generates a particular data structure from another data structure emitted by the phase before it.

The phases of the front end generally include **lexical analysis**, **syntax analysis**, **semantic analysis** and **intermediate code generation**while theback end includes **optimization** and **code generation**.



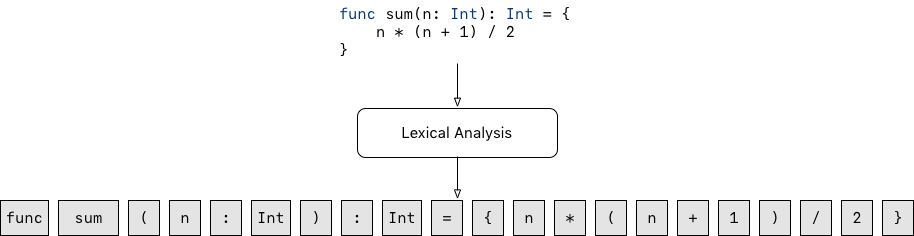
**Structure of a compiler**

#### Lexical Analysis

The first phase of the compiler is the lexical analysis. In this phase, the compiler breaks the submitted source code into meaningful elements called **lexemes** and generates a sequence of **tokens** from the lexemes.

A lexeme can be thought of as a uniquely identifiable string of characters in the source programming language, for example, keywords such as if, whileor func, identifiers, strings, numbers, operators or single characters like (, ), . or :.

A token is an object describing a lexeme. Along with the value of the lexeme(the actual string of characters of the lexeme), it contains information such as its type (is it a keyword? an identifier? an operator? …) and the position (line and/or column number) in the source code where it appears.

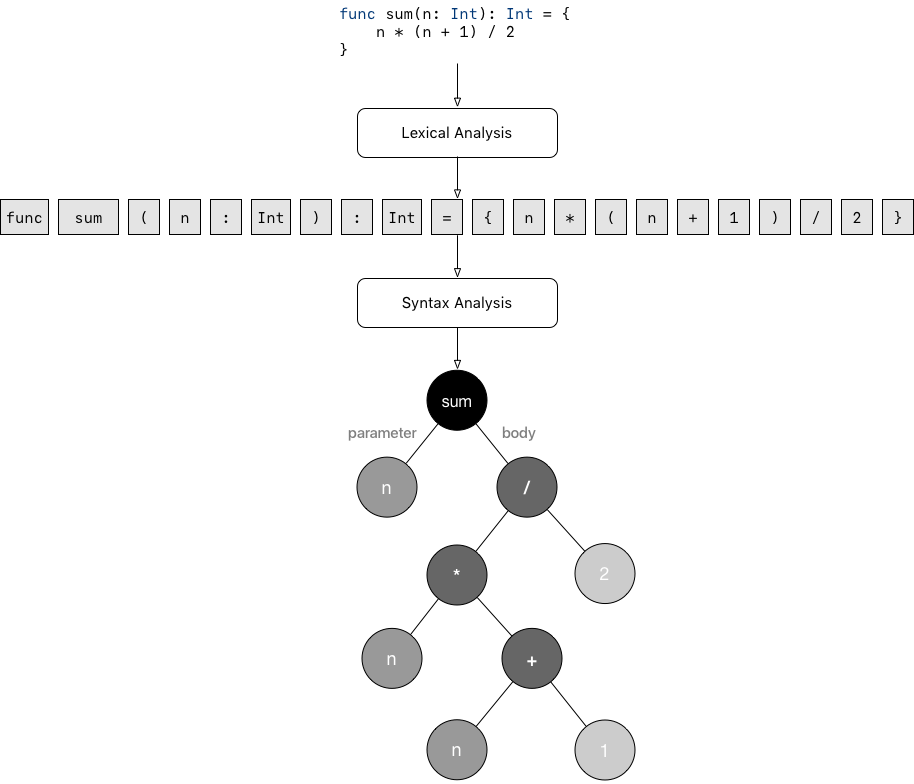


Sequence of lexemes generated during lexical analysis

If the compiler encounters a string of characters for which it cannot create a token, it will stop its execution by throwing an error; for example, if it encounters a malformed string or number or an invalid character (such as a non-ASCII character in Java).

#### Syntax Analysis

During syntax analysis, the compiler uses the sequence of tokens generated during the lexical analysis to generate a tree-like data structure called **Abstract Syntax Tree**, **AST** for short. The AST reflects the syntactic and logical structure of the program.



**Abstract Syntax Tree generated after syntax analysis**

Syntax analysis is also the phase where eventual syntax errors are detected and reported to the user in the form of informative messages. For instance, in the example above, if we forget the closing brace } after the definition of the sum function, the compiler should return an error stating that there is a missing } and the error should point to the line and column where the } is missing.

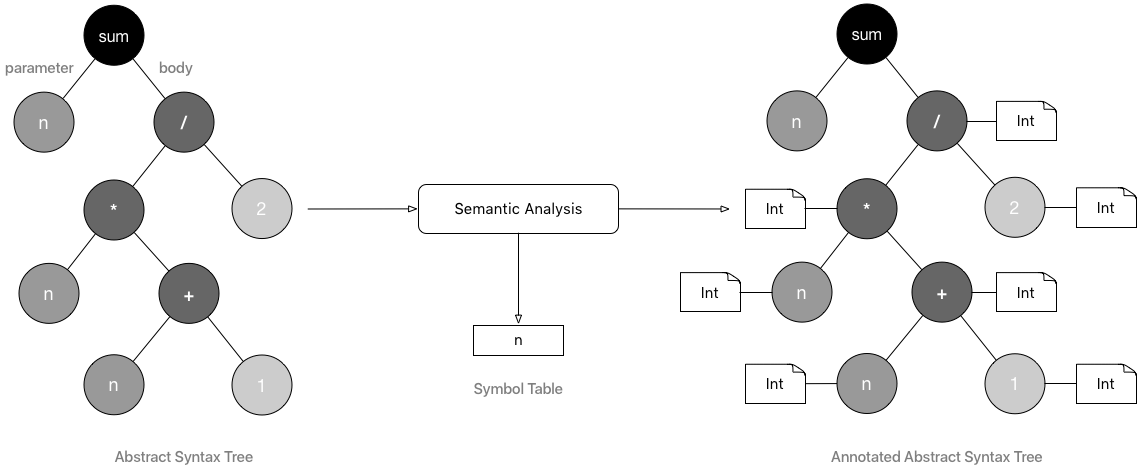
If no error is found during this phase, the compiler moves to the semantic analysis phase.

#### Semantic Analysis

During semantic analysis, the compiler uses the AST generated during syntax analysis to check if the program is consistent with all the rules of the source programming language. Semantic analysis encompasses

* **Type inference**. If the programming language supports type inference, the compiler will try to infer the type of all untyped expressions in the program. If a type is successfully inferred, the compiler will **annotate** the corresponding node in the AST with the inferred type information.
* **Type checking**. Here, the compiler checks that all values being assigned to variables and all arguments involved in an operation have the correct type. For example, the compiler makes sure that no variable of type String is being assigned a Double value or that a value of type Bool is not passed to a function accepting a parameter of type Double or again that we’re not trying to divide a String by an Int, "Hello" / 2 (unless the language definition allows it).
* **Symbol management**. Along with performing type inference and type checking, the compiler maintains a data structure called **symbol table**which contains information about all the symbols (or names) encountered in the program. The compiler uses the symbol table to answer questions such as Is this variable declared before use?, Are there 2 variables with the same name in the same scope? What is the type of this variable? Is this variable available in the current scope? and many more.

The output of the semantic analysis phase is an **annotated AST** and the **symbol table**.

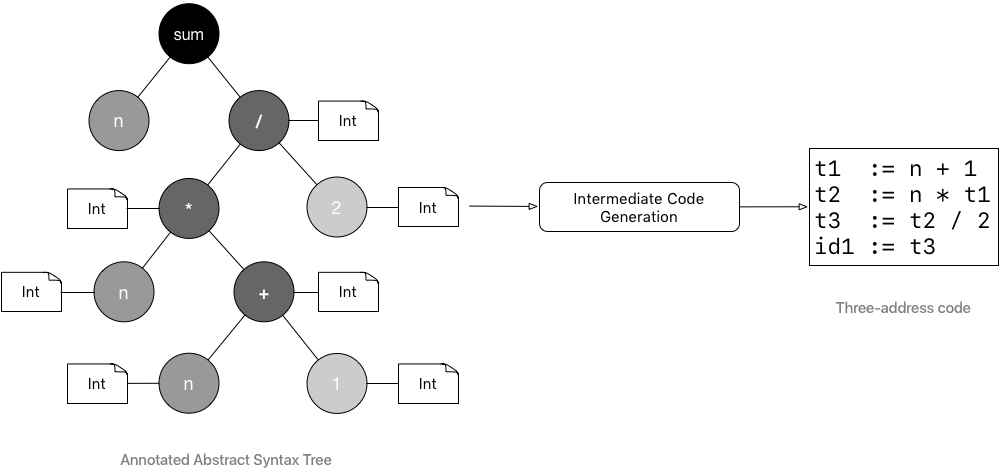


#### Intermediate Code Generation

After the semantic analysis phase, the compiler uses the annotated AST to generate an intermediate and machine-independent low-level code. One such intermediate representation is the [**three-address code**](https://en.wikipedia.org/wiki/Three-address_code).

The three-address code (3AC), in its simplest form, is a language in which an instruction is an assignment and has at most 3 operands.

Most instructions in 3AC are of the form a **:=** b **<operator>** c or a **:=** b.



The above drawing depicts a 3AC code generated from an annotated ASTcreated during the compilation of the function

**func** sum(n: **Int**): **Int** = {  
 n \* (n + 1) / 2  
}

The intermediate code generation concludes the front end phase of the compiler.

#### Optimization

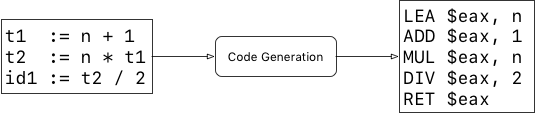
In the optimization phase, the first phase of the back end, the compiler uses different optimization techniques to improve on the intermediate code generated by making the code faster or shorter for example.

For example, a very simple optimization on the 3AC code in the previous example would be to eliminate the temporary assignment t3 := t2 / 2 and directly assign to id1 the value t2 / 2.



#### Code Generation

In this last phase, the compiler translates the optimized intermediate code into machine-dependent code, Assembly or any other target low-level language.



### Compiler vs. Interpreter

Let’s conclude this article with a note about the difference between compilers and interpreters.

Interpreters and compilers are very similar in structure. The main difference is that an interpreter directly executes the instructions in the source programming language while a compiler translates those instructions into efficient machine code.

An interpreter will typically generate an efficient intermediate representation and immediately evaluate it. Depending on the interpreter, the intermediate representation can be an AST, an annotated AST or a machine-independent low-level representation such as the three-address code.

**Section B Part II**

# What are the loader schemes?

Loading schemes:

1.Absolute loader. 2.Relocating loader. 3.Direct linking loader. 4.Dynamic Loading. 5.Dynamic linking.

(1 )Absolute loader: The task of an absolute loader is virtually trivial.The loader simply accepts machine language code and places it into main memory specified by the assembler.

(2) Relocating loader: The task of relocating loader is to avoid reassembling of of all subroutines when a subroutine is changed and to perform tasks of allocation and linking for programmer.

(3) Dynamic loading: In order to overlay structure to work it is necessary for the module loader to load the various procedures as they are needed. There are many binders capable of processing and allocating overlay structure. the portion of the loader that actually intercepts calls and loads necessary procedure is called overlay supervisor of simply flipper. this overall scheme is called dynamic loading or load on call.

(4) Dynamic linking: This is mechanism by which loading and linking of external references are postponed until execution time. This was made to sort out disadvantage of previous loading schemes like subroutine is referenced and never executed

System Programming

**Loaders and Linkers**

Introduction:

In this chapter we will understand the concept of linking and loading. As discussed earlier the source program is converted to object program by assembler. The loader is a program which takes this object program, prepares it for execution, and loads this executable code of the source into memory for execution.

**Definition of Loader:**

Loader is utility program which takes object code as input prepares it for

execution and loads the executable code into the memory. Thus loader is

actually responsible for initiating the execution process.

**Functions of Loader**:

The loader is responsible for the activities such as allocation, linking,

relocation and loading

1) It allocates the space for program in the memory, by calculating the

size of the program. This activity is called allocation.

2) It resolves the symbolic references (code/data) between the object

modules by assigning all the user subroutine and library subroutine

addresses. This activity is called linking.

3) There are some address dependent locations in the program, such

address constants must be adjusted according to allocated space, such

activity done by loader is called relocation.

4) Finally it places all the machine instructions and data of corresponding

programs and subroutines into the memory. Thus program now becomes

ready for execution, this activity is called loading.

**Loader Schemes:**

Based on the various functionalities of loader, there are various types of

loaders:

**1) “compile and go” loader:** in this type of loader, the instruction is read

line by line, its machine code is obtained and it is directly put in the main

memory at some known address. That means the assembler runs in one

part of memory and the assembled machine instructions and data is

directly put into their assigned memory locations. After completion of

assembly process, assign starting address of the program to the location

counter. The typical example is WATFOR-77, it’s a FORTRAN compiler

which uses such “load and go” scheme. This loading scheme is also

called as “assemble and go”.

**Advantages:**

• This scheme is simple to implement. Because assembler is placed at one

part of the memory and loader simply loads assembled machine

instructions into the memory.

**Disadvantages:**

• In this scheme some portion of memory is occupied by assembler which

is simply a wastage of memory. As this scheme is combination of

assembler and loader activities, this combination program occupies large

block of memory.

• There is no production of .obj file, the source code is directly converted

to executable form. Hence even though there is no modification in the

source program it needs to be assembled and executed each time, which

then becomes a time consuming activity.

• It cannot handle multiple source programs or multiple programs written

in different languages. This is because assembler can translate one source

language to other target language.

• For a programmer it is very difficult to make an orderly modulator

program and also it becomes difficult to maintain such program, and the

“compile and go” loader cannot handle such programs.

• The execution time will be more in this scheme as every time program

is assembled and then executed.

Complier and GO to Scheme Dia

|  |  |
| --- | --- |
| User program | Compile & go loader |

|  |
| --- |
|  |
|  |
| Compile & go loader |
|  |
| Execute code of user program |
|  |

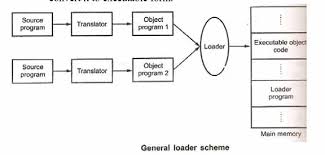
2) **General Loader Scheme:** in this loader scheme, the source program is converted to object program by some translator (assembler). The loader accepts these object modules and puts machine instruction and data in an executable form at their assigned memory. The loader occupies some portion of main memory.

**Advantages:**

• The program need not be retranslated each time while running it. This is because initially when source program gets executed an object program gets generated. Of program is not modified, then loader can make use of this object program to convert it to executable form.

• There is no wastage of memory, because assembler is not placed in the memory, instead of it, loader occupies some portion of the memory. And size of loader is smaller than assembler, so more memory is available to the user.

• It is possible to write source program with multiple programs and multiple languages, because the source programs are first converted to object programs always, and loader accepts these object modules to convert it to executable form.



**3) Absolute Loader:** Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in the memory. This type of loader is called

absolute because no relocation information is needed; rather it is obtained from the programmer or assembler. The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file. In this scheme, the programmer or assembler should have knowledge of memory management. The resolution of external references or linking of different subroutines are the issues which need to be handled by the programmer. The programmer should take care of two things: first thing is : specification of starting address of each module to be used. If some modification is done in some module then the length of that module may vary. This causes a change in the starting address of immediate next . modules, its then the programmer's duty to make necessary changes in the starting addresses of respective modules.

Second thing is ,while branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction. For example

Line no

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | Main | START | 1000 |  |
|  |  |  |  |  |
|  |  |  |  |  |
| 1 |  | JMP | 5000 |  |
| 16 |  | STORE  END | : INSTRUCTION AT LOCATION 2000 | |
| 1 |  | SUM | START 1000 |  |
| 2 |  |  |  |  |
| 20 |  | JMP | 2000 |  |
| 21 |  | END |  |  |

In this example there are two segments, which are interdependent. At line number 1 the assembler directive START specifies the physical starting address that can be used during the execution of the first segment MAIN.

Then at line number 15 the JMP instruction is given which specifies the physical starting address that can be used by the second segment. The assembler creates the object codes for these two segments by considering the stating addresses of these two segments. During the execution, the first segment will be loaded at address 1000 and second segment will be loaded at address 5000 as specified by the programmer. Thus the problem of linking is manually solved by the programmer itself by taking care of the mutually dependant dresses. As you can notice that the control is correctly transferred to the address 5000 for invoking the other segment, and after that at line number 20 the JMP instruction transfers the control to the location 2000, necessarily at location 2000 the instruction STORE of line number 16 is present. Thus resolution of mutual references and linking is done by the programmer. The task of assembler is to create the object codes for the above segments and along with the information such as starting address of the memory where actually the object code can be placed at the time of execution. The absolute loader accepts these object modules from assembler and by reading the information about their starting addresses, it will actually place (load) them in the memory at specified addresses.

The entire process is modeled in the following figure.i.e. **Process of Absolute Loading**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Absolute Loader |  |
| SEGMENT1 |  | ASSEMBLER |  | Object code |  |
|  | Starting address | Object code for segment1 |
| ….. |  |  |  |  |  |
| SEGMENT2 |  | ASSEMBLER |  | Object code |  |
|  | Starting address | Object code for segment2 |
| ….. |  |  |  |  |  |
| SEGMENT n |  | ASSEMBLER |  | Object code |  |
|  | Starting address | Object code for segment n |

‘ Absolute Loader Main memory

Thus the absolute loader is simple to implement in this scheme-

l) Allocation is done by either programmer or assembler

2)Linking is done by the programmer or assembler

3)Resolution is done by assembler

4)Simply loading is done by the loader

As the name suggests, no relocation information is needed, if at all it is

required then that task can be done by either a programmer or assembler

**Advantages:**

1. It is simple to implement

2. This scheme allows multiple programs or the source programs written different languages. If there are multiple programs written in different languages then the respective language assembler will convert it to the language and a common object file can be prepared with all the ad resolution.

3. The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory.

4. The process of execution is efficient.

**Disadvantages:**

1. In this scheme it is the programmer's duty to adjust all the inter segment addresses and manually do the linking activity. For that, it is necessary for a programmer to know the memory management.

If at all any modification is done the some segments, the starting

addresses of immediate next segments may get changed, the programmer

has to take care of this issue and he needs to update the corresponding

starting addresses on any modification in the source.

**Algorithm for absolute Loader**

Input: Object codes and starting address of program segments.

Output: An executable code for corresponding source program. This executable code is to be placed in the main memory

**Method: Begin**

For each program segment

do Begin

Read the first line from object module to obtain information about memory location. The starting address say S in corresponding object module is the memory location where executable code is to be placed.

Hence

Memory\_location = S

Line counter = 1; as it is first line While (! end of file)

For the curent object code

do Begin

1. Read next line

2. Write line into location S

3. S = S + 1

4. Line counter Line counter + 1

**Subroutine Linkage:** To understand the concept of subroutine linkages, first consider the following scenario:

"In Program A a call to subroutine B is made. The subroutine B is not written in the program segment of A, rather B is defined in some another program segment C"

Nothing is wrong in it. But from assembler's point of view while generating the code for B, as B is not defined in the segment A, the assembler can not find the value of this symbolic reference and hence it will declare it as an error. To overcome problem, there should be some

mechanism by which the assembler should be explicitly informed that segment B is really defined in some other segment C. Therefore whenever segment B is used in segment A and if at all B is defined in C, then B must -be declared as an external routine in A.

To declare such subroutine as external, we can use the assembler directive EXT. Thus the statement such as EXT B should be added at the beginning of the segment A. This actually helps to inform assembler that B is defined somewhere else. Similarly, if one subroutine or a variable is defined in the current segment and can be referred by other segments then those should be declared by using pseudo-ops INT. Thereby the assembler could inform loader that these are the subroutines or variables used by other segments. This overall process of establishing the relations between the subroutines can be conceptually called a\_ subroutine linkage.

For example

MAIN START

EXT B

.

.

.

CALL B

.

.

END

B START

.

.

RET

END

At the beginning of the MAIN the subroutine B is declared as external. When a call to subroutine B is made, before making the unconditional jump, the current content of the program counter should be stored in the system stack maintained internally. Similarly while returning from the subroutine B (at RET) the pop is performed to restore the program

counter of caller routine with the address of next instruction to be executed.

**Concept of relocations:**

Relocation is the process of updating the addresses used in the address sensitive instructions of a program. It is necessary that such a modification should help to execute the program from designated area of the memory.

The assembler generates the object code. This object code gets executed after loading at storage locations. The addresses of such object code will get specified only after the assembly process is over. Therefore, after loading,

**address of object code = Mere address of object code + relocation constant.**

There are two types of addresses being generated: Absolute address and, relative address. The absolute address can be directly used to map the object code in the main memory. Whereas the relative address is only after the addition of relocation constant to the object code address. This kind of adjustment needs to be done in case of relative address before actual execution of the code. The typical example of relative reference is :

addresses of the symbols defined in the Label field, addresses of the data which is defined by the assembler directive, literals, redefinable symbols.

Similarly, the typical example of absolute address is the constants which are generated by assembler are absolute.

The assembler calculates which addresses are absolute and which addresses are relative during the assembly process. During the assembly process the assembler calculates the address with the help of simple expressions.

For example

LOADA(X)+5

The expression A(X) means the address of variable X. The meaning of the above instruction is that loading of the contents of memory location which is 5 more than the address of variable X. Suppose if the address of X is 50 then by above command we try to get the memory location

50+5=55. Therefore as the address of variable X is relative A(X) + 5 is also relative. To calculate the relative addresses the simple expressions are allowed. It is expected that the expression should possess at the most addition and multiplication operations. A simple exercise can be carried out to determine whether the given address is absolute or relative. In the expression if the address is absolute then put 0 over there and if address is relative then put lover there. The expression then gets transformed to sum of O's and l's. If the resultant value of the expression is 0 then expression is absolute. And if the resultant value of the expression is 1 then the expression is relative. If the resultant is other than 0 or 1then the

expression is illegal. For example:

|  |  |  |
| --- | --- | --- |
| **EXPRESSION** | **COMPUTATION** | **RELOCATION ATTRIBUTE** |
| A-B | 1-1 =0 | ABSOLUTE |
| A+B-C | 1+1-1=1 | RELATIVE |
| A-B+5 | 1-1+0=0 | ABSOLUTE |
| A+B | 1+1=2 | IILEGAL |

In the above expression the A, Band C are the variable names. The assembler is to c0l1sider the relocation attribute and adjust the object code by relocation constant. Assembler is then responsible to convey the information loading of object code to the loader. Let us now see how

assembler generates code using relocation information.

**Direct Linking Loaders**

In the above expression the A, Band C are the variable names. The

assembler is to c0l1sider the relocation attribute and adjust the object

code by relocation constant. Assembler is then responsible to convey the

information loading of object code to the loader. Let us now see how

assembler generates code using relocation information.

**Direct Linking Loaders**

The direct linking loader is the most common type of loader. This type of loader is a relocatable loader. The loader can not have the direct access to the source code. And to place the object code in the memory there are two situations: either the address of the object code could be absolute which then can be directly placed at the specified location or the address

can be relative. If at all the address is relative then it is the assembler who informs the loader about the relative addresses.

The assembler should give the following information to the loader

1) The length of the object code segment

2) The list of all the symbols which are not defined 111 the current segment but can be used in the current segment.

3) The list of all the symbols which are defined in the current segment but can be referred by the other segments.

The list of symbols which are not defined in the current segment but can be used in the current segment are stored in a data structure called USE table. The USE table holds the information such as name of the symbol, address, address relativity.

The list of symbols which are defined in the current segment and can be referred by the other segments are stored in a data structure called DEFINITION table. The definition table holds the information such as symbol, address.

**Overlay Structures and Dynamic Loading:**

Sometimes a program may require more storage space than the available one Execution of such program can be possible if all the segments are not required simultaneously to be present in the main memory. In such situations only those segments are resident in the memory that are actually needed at the time of execution But the question arises what will happen if the required segment is not present in the memory? Naturally the execution process will be delayed until the required segment gets loaded in the memory. The overall effect of this is efficiency of execution process gets degraded. The efficiency can then be improved by carefully selecting all the interdependent segments. Of course the assembler can not do this task. Only the user can specify such dependencies. The inter dependency of the segments can be specified by a tree like structure called static overlay structures. The overlay structure contain multiple root/nodes and edges. Each node represents the segment. The specification of required amount of memory is also essential in this structure. The two segments can lie simultaneously in the main memory if they are on the same path. Let us take an example to understand the concept. Various segments along with their memory requirements is as

shown below.

S1 (16K)

S3 (8k)

S2 (14k)

S4 (20k)

S6(16K)

S5 (12K)

S7(16K)

S8(12K)

S9(18K)

**Automatic Library Search:**

Previously, the library routines were available in absolute code but now the library routines are provided in relocated form that ultimately reduces their size on the disk, which in turn increases the memory utilization. At

execution time certain library routines may be needed. Keeping track of which library routines are required and how much storage is required by these routines, if at all is done by an assembler itself then the activity of

automatic library search becomes simpler and effective. The library routines can also make an external call to other routines. The idea is to make a list of such calls made by the routines. And if such list is made available to the linker then linker can efficiently find the set of required routines and can link the references accordingly.

For an efficient search of library routines it desirable to store all the calling routines first and then the called routines. This avoids wastage of time due to winding and rewinding. For efficient automated search of

library routines even the dictionary of such routines can be maintained. A table containing the names of library routines and the addresses where they are actually located in relocatable form is prepared with the help of

translator and such table is submitted to the linker. Such a table is **called subroutine directory**. Even if these routines have made any external calls the -information about it is also given in subroutine directory. The linker

searches the subroutine directory, finds the address of desired library routine (the address where the routine is stored in relocated form).Then linker prepares a load module appending the user program and necessary library routines by doing the necessary relocation. If the library routine contains the external calls then the linker searches the subroutine directory finds the address of such external calls, prepares the load module by resolving the external references. Linkage Editor: The execution of any program needs four basic functionalities and those are allocation, relocation, linking and loading. As we have also seen in direct linking loader for execution of any program each time these four functionalities need to be performed. But performing all these functionalities each time is time and space consuming task. Moreover if the program contains many subroutines or functions and the program needs to be executed repeatedly then this activity becomes annoyingly complex .Each time for execution of a program, the allocation, relocation linking and -loading needs to be done. Now doing these activities each time increases the time and space complexity. Actually, there is no need to redo all these four activities each time. Instead, if the results of some of these activities are stored in a file then that file can be used by other activities. And performing allocation, relocation, linking and loading can be avoided each time. The idea is to separate out these activities in separate groups. Thus dividing the essential four functions in groups reduces the overall time complexity of loading process. The program which performs allocation, relocation and linking is called binder. The binder performs relocation, creates linked executable text and stores this text in a file in some systematic manner. Such kind of module prepared by the binder execution is called

load module. This load module can then be actually loaded in the main memory by the loader. This loader is also called as module loader. If the binder can produce the exact replica of executable code in the load module then the module loader simply loads this file into the main memory which ultimately reduces the overall time complexity. But in this process the binder should knew the current positions of the main memory.

Even though the binder knew the main memory locations this is not the only thing which is sufficient. In multiprogramming environment, the region of main memory available for loading the program is decided by

the host operating system. The binder should also know which memory area is allocated to the loading program and it should modify the relocation information accordingly. The binder which performs the linking function and produces adequate information about allocation and relocation and writes this information along with the program code in the file is called linkage editor. The module loader then accepts this rile as input, reads the information stored in and based on this information about allocation and relocation it performs the task of loading in the main memory. Even though the program is repeatedly executed the linking is done only once. Moreover, the flexibility of allocation and relocation helps efficient utilization of the main memory.

Direct linking: As we have seen in overlay structure certain selective subroutines can be resident in the memory. That means it is not necessary to resident all the subroutines in the memory for all the time. Only necessary routines can be present in the main memory and during execution the required subroutines can be loaded in the memory. This process of postponing linking and loading of external reference until execution is called dynamic linking. For example suppose the subroutine main calls A,B,C,D then it is not desirable to load A,B,C and D along with the main in the memory. Whether A, B, C or D is called by the main or not will be known only at the time of execution. Hence keeping these routines already before is really not needed. As the subroutines get

executed when the program runs. Also the linking of all the subroutines has to be performed. And the code of all the subroutines remains resident in the main memory. As a result of all this is that memory gets occupied

unnecessarily. Typically 'error routines' are such routines which can be invoked rarely. Then one can postpone the loading of these routines during the execution. If linking and loading of such rarely invoked external references could be postponed until the execution time when it was found to be absolutely necessary, then it increases the efficiency of overhead of the loader. In dynamic linking, the binder first prepares a load module in which along with program code the allocation and relocation information is stored. The loader simply loads the main module in the main memory. If any external ·reference to a subroutine comes, then the execution is suspended for a while, the loader brings the required subroutine in the main memory and then the execution process is resumed. Thus dynamic linking both the loading and linking is done

dynamically.

**Advantages**

1. The overhead on the loader is reduced. The required subroutine will be load in the main memory only at the time of execution.

2. The system can be dynamically reconfigured.

**Disadvantages** The linking and loading need to be postponed until the execution. During the execution if at all any subroutine is needed then the process of execution needs to be suspended until the required subroutine

gets loaded in the main memory.

**Bootstrap Loader**: As we turn on the computer there is nothing meaningful in the main memory (RAM). A small program is written and stored in the ROM. This program initially loads the operating system from secondary storage to main memory. The operating system then takes the overall control. This program which is responsible for booting up the system is **called bootstrap loader**. This is the program which must be executed first when the system is first powered on. If the program starts from the location x then to execute this program the program counter of this machine should be loaded with the value x. Thus the task of setting

the initial value of the program counter is to be done by machine hardware. The bootstrap loader is a very small program which is to be fitted in the ROM. The task of bootstrap loader is to load the necessary portion of the operating system in the main memory .The initial address at which the bootstrap loader is to be loaded is generally the lowest (may be at 0th location) or the highest location. .

Concept of Linking: As we have discussed earlier, the execution of

program can be done with the help of following steps

1. Translation of the program(done by assembler or compiler)

2. Linking of the program with all other programs which are needed for

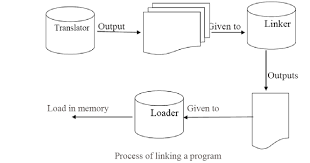
execution. This also involves preparation of a program called load module.

3. Loading of the load module prepared by linker to some specified

memory location.

The output of translator is a program called object module. The linker processes these object modules binds with necessary library routines and prepares a ready to execute program. Such a program is called binary

program. The "binary program also contains some necessary information about allocation and relocation. The loader then load s this program into memory for execution purpose.



Various tasks of linker are -

1. Prepare a single load module and adjust all the addresses and subroutine references with respect to the offset location.

2. To prepare a load module concatenate all the object modules and adjust all the operand address references as well as external references to the offset location.

3. At correct locations in the load module, copy the binary machine instructions and constant data in order to prepare ready to execute module.

The linking process is performed in two passes. Two passes are necessary because the linker may encounter a forward reference before knowing its address. So it is necessary to scan all the DEFINITION and USE table at least once. Linker then builds the Global symbol table with the help of USE and DEFINITION table. In Global symbol table name of each externally referenced symbol is included along with its address relative to beginning of the load module. And during pass 2, the addresses of external references are replaced by obtaining the addresses from global symbol table.

**SECTION A PART II**

CONCEPT OF OS

**Features of ML**

* 1. Instructions are written using binary logic.  
     b) Requires many logic lines to accomplish a given task.  
     c) Easily executable by CPU but difficult for humans to understand.

**Assembly language program structure**

Assembly language contains the following code like structure it mostly works with instructions like ADD(adding a number from one register to the other),LXI,LHLD,SUB etc.

It’s structure looks like:

**Structure of Assembly Language Programming**

An assembly language program has six fields: Memory Address, Machine Code, Opcode, Operands, and Comments. **Memory Address:** These are 16-bit addresses of the user memory in the system, where the machine code of the program is stored. The beginning address shown as in the format “XX00”; the symbol XX represents the page number and 00 represents the line number.

**Machine Code:** Also called as instruction code. These are the hexadecimal numbers represents instructions that are stored in the respective memory addresses.

**Label:** A label is a symbol or group of symbols used to represent an address of specific statement. Labels are usually followed by a colon. Labels are not required in a statement; they are just inserted where they are needed.

**Opcode (Operation Code):** An instruction is divided into two parts: Opcode and Operand. Opcode are the abbreviated symbols to indicate the type of operation or function that will perform by the machine code.

**Operand:** The operand field of the statement contains the 8-bit or 16-bit data, the memory address, the port address, or the name of the registers on which the instruction is to be performed.

An instruction, called a mnemonic or mnemonic instruction is formed by combining Opcode and Operand. A mnemonic is just the letters which are usually initials or a shortened form of the English words for the operation performed by the instruction. For example, the mnemonic for subtract is SUB, the mnemonic for exclusive or is XOR, and the mnemonic for the instruction to copy data from one location to another is MOV.

**Comments:**This field is not become a part of the program, it simply a part of the proper documentation of a program to explain or remind of the function that this instruction or group of instructions performs in the program. These are separated by a semicolon (;) from the instruction on the same line.

The statement format and example are as follows.

|  |
| --- |
|  |
| **Memory Address (Hex)** | **Machine Code (Hex)** | **Label** | **Opcode** | **Operand** | **Comments** |
| 2000 | 06 | START: | MVI | B, 37H | ; Load register B with data 37H |
| 2001 | 37 |  |  |  |  |
| 2002 | 78 |  | MOV | A, B | ; Copy data from register B in to A |
| 2003 | D3 |  | OUT | PORT 1 | ; Display accumulator (A) contents |
| 2004 | PORT 1 |  |  |  | ; (37H) at port1 |
| 2005 | C3 |  | JMP | START | ; go back to the beginning and read the data 37 H again |
| 2006 | 00 |  |  |  |  |
| 2007 | 20 |  |  |  |  |

**Syntax of Assembly Language Statements**

Assembly language statements are entered one statement per line. Each statement follows the following format:

[label] mnemonic [operands] [;comment]

The fields in the square brackets are optional. A basic instruction has two parts, the first one is the name of the instruction (or the mnemonic) which is to be executed, and the second are the operands or the parameters of the command.

Following are some examples of typical assembly language statements:

INC COUNT ; Increment the memory variable

COUNT MOV TOTAL, 48 ; Transfer the value 48 in the memory variable

TOTAL ADD AH, BH ; Add the content of the ; BH register into the AH register

AND MASK1, 128 ; Perform AND operation on the ; variable MASK1 and 128

ADD MARKS, 10 ; Add 10 to the variable MARKS

MOV AL, 10 ; Transfer the value 10 to the AL register

The Hello World Program in Assembly

The following assembly language code displays the string 'Hello World' on the screen:

section .text

global main ; must be declared for linker (ld)

main: tells linker entry point mov edx,len ;message length mov ecx,msg ;message to write mov ebx,1 ; file descriptor (stdout)

mov eax,4 ; system call number (sys\_write)

int 0x80 ; call kerne

**SECTION A ASSEMBLER**

**Describe different types of Assemblers.**

The assemblers can be of any one of the following types:

1) Single pass assemblers

2) Two pass assemblers and

3) Multi pass assemblers

Pass is a terminology used in system software. Each reading of a program can be called as a ***pass.***The assembly process can be done within a pass, if so, such assemblers are single pass assemblers. If the assembly is done two passes, then those assemblers are called two pass assemblers. The translation of the assembly language program can be done in several passes. Such assemblers are called multi pass assemblers.

***Single Pass Assemblers:***

In single pass assemblers, the entire translation of assembly language program into object program is done in only one pass. The source program is read only once. These are also called as ‘one pass assembler’.

The object code can be produced in the single pass assemblers in two different ways. In the first way, the object code is directly loaded into the main memory for execution. Here, no loader is required. This type of loading scheme is called ‘compile and loading scheme’.

In the second way, the object program will be loaded into the main memory for execution later as necessity arises. Here, a separate loader program is necessary.

An assembler, which goes through an assembly language program only once, is known as One-pass assembler. It suffers from forward reference problem. This is faster because they scan the program only once. This does not have many features supported by the two pass assemblers.

***Two Pass Assemblers:***

The two pass assemblers are widely used and the translation process is done in two passes. The two pass assemblers resolve the problem of forward references conveniently. An assembler, which goes through an assembly language program twice, is called a two ***pass assembler***. During the first pass it collects all labels. During the second pass it produces the machine code for each instruction and assigns address to each of them. It assigns addresses to labels by counting their position from the starting address. It provides many features than the single pass assembler. It is widely used.

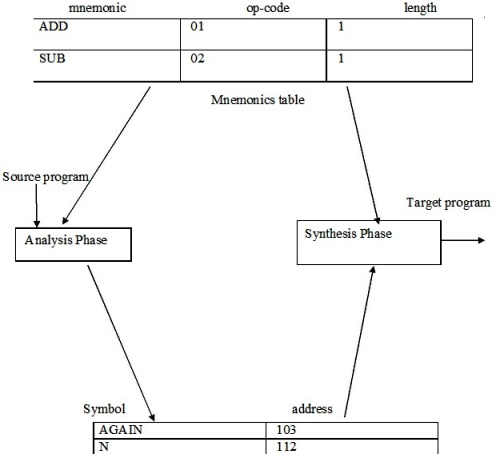
***Multi Pass Assemblers:***

If the assembly process is done more than two passes then those assemblers are called as multi pass assemblers. One reading of a program (source program or any program in concern) can be called as a pass. In assemblers, the passes are necessary to solve the problem of forward references. In order to process the entire symbol definitions several passes are made over the source program.

**Compare single pass assembler and two pass assembler.**

|  |  |
| --- | --- |
| **Two pass translation** | **Single pass translation** |
| 1. Two pass translations consist of pass I and pass II. | 1. A one pass assembler requires 1 scan of the source program to generate machine code. |
| 2. LC processing is performed in the first pass and symbols defined in the program are entered into the symbol table, hence first pass performs analysis of the source program. | 2. The process of forward references is talked using a process called back patching. |
| 3. So, two pass translation of assembly lang. program can handle forward reference easily. | 3. The operand field of an instruction containing forward references is left blank initially |
| 4. The second pass synthesizes the target form using the address information found in the symbol table. | 4. A table of instruction containing forward references is maintained separately called table of incomplete instruction (TII). |
| 5 First pass constructs an intermediate representation of the source program and that will be used by second pass. | 5 This table can be used to fill-up the addresses in  incomplete instruction. |
| 6 IR (intermediate representation) consists of two main components: data structure + IC (intermediate code). | 6 The address of the forward referenced symbols is put in the blank field with the help of back patching list. |

**Explain two pass assembler in detail with suitable example.**



**Analysis Phase**

* The primary function performed by the analysis phase is the building of the symbol table.
* For this purpose it must determine address of the symbolic name.
* It is possible to determine some address directly, however others must be inferred. And this function is called memory allocation.
* To implement memory allocation a data structure called location counter (LC) is used, it is initialized to the constant specified in the START statement.
* We refer the processing involved in maintaining the location counter as LC processing.

Tasks of Analysis phase

* Isolate the label, mnemonics op-code, and operand fields of a constant.
* If a label is present, enter the pair (symbol, <LC content>) in a new entry of symbol table.
* Check validity of mnemonics op-code.

           Perform LC processing.

**Synthesis Phase**

* Consider the assembly statement,

               MOVER BREG, ONE

* We must have following information to synthesize the machine instruction corresponding to this statement:
  1. Address of name ONE
  2. Machine operation code corresponding to mnemonics MOVER.
* The first item of information depends on the source program; hence it must be available by analysis phase.
* The second item of information does not depend on the source program; it depends on the assembly language.
* Based on above discussion, we consider the use of two data structure during

**Synthesis phase:**

1. Symbol table: Each entry in symbol table has two primary field-name and address. This table is built by analysis phase
2. Mnemonics table: An entry in mnemonics table has two primary field-mnemonics and op-code.

Tasks of Synthesis phase

1. Obtain machine op-code through look up in the mnemonics table.
2. Obtain address of memory operand from the symbol table.
3. Synthesize a machine instruction

.

**Define forward references. How it can be solved using back-patching? Explain with example.**

Forward Reference: - A forward reference occurs when a label is used as an operand, For example as a branch target, earlier in the code than the definition of the label.

                                    The assembler cannot know the address of the forward reference label until it reads the definition of the label.

The assembler implements the back-patching technique as follows:

* It builds a table of incomplete instructions (TII) to record information about instructions whose operand fields were left blank.
* Each entry in this table contains a pair of the form (instruction address, symbol) to indicate that the address of symbol should be put in the operand field of the instruction with the address instruction address.
* By the time the END statement is processed, the symbol table would contain the addresses of all symbols defined in the source program and TII would contain information describing all forward references.
* The assembler can now process each entry in TII to complete the concerned instruction.
* Alternatively, entries in TII can be processed on the fly during normal processing.
* In this approach, all forward references to a symbol symb would be processed when the statement that defines symbol symbi is encountered.
* The instruction corresponding to the statement

                MOVER BREG, ONE

             Contains a forward reference to ONE.

* Hence the assembler leaves the second operand field blank in the instruction that is assembled to reside in location 101of memory, and makes an entry (101, ONE) in the table of incomplete instructions (TII).
* While processing the statement
* After the END statement is processed, the entry (101, ONE) would be processed by obtaining the address of ONE from the symbol table and inserting it in the second operand field of the instruction with assembled address 101.

**Explain The Working Of Single Pass Assembles. Show The Structures Of Its Databases Used.**

Assembler translate assembly language programs to machine   
code is called an Assembler.

**One-pass assemblers:** It is an assembler that generally generates the object code directly in memory for immediate execution.

It parses through your source code only once and you are done.

• **One-pass assemblers are used when**

it is necessary or desirable to avoid a second pass over the source program the external storage for the intermediate file between two passes is slow or is inconvenient to use Main problem: forward references to both data and instructions One simple way to eliminate this problem: require that all areas be defined before they are referenced. It is possible, although inconvenient, to do so for data items. Forward jump to instruction items cannot be easily eliminated.

**Data structures for assembler:**

**Op code table**   
Looked up for the translation of mnemonic code key: mnemonic code result: bits Hashing is usually used once prepared, the table is not changed efficient lookup is desired since mnemonic code is predefined, the hashing function can be tuned a priori The table may have the instruction format and length to decide where to put op code bits, operands bits, offset bits for variable instruction size used to calculate the address **Symbol table**   
Stored and looked up to assign address to labels efficient insertion and retrieval is needed deletion does not occur Difficulties in hashing non random keys Problem the size varies widely pass 1: loop until the end of the program   
1. Read in a line of assembly code   
2. Assign an address to this line increment N (word addressing or byte addressing) 3. Save address values assigned to labels in symbol tables 4. Process assembler directives constant declaration space reservation **Algorithm for Pass 1 assembler:** begin if starting address is given   
LOCCTR = starting address;   
else   
LOCCTR = 0; while OPCODE != END do ;; or EOF   
begin   
read a line from the code   
if there is a label if this label is in SYMTAB, then error   
else insert (label, LOCCTR) into SYMTAB search OPTAB for the op code   
if found   
LOCCTR += N ;; N is the length of this instruction (4 for MIPS)   
else if this is an assembly directive   
update LOCCTR as directed   
else error   
write line to intermediate file end program size = LOCCTR - starting address;   
end

**Load-and-go assembler** • Load-and-go assembler generates their object code in memory for immediate execution. • No object program is written out, no loader is needed. • It is useful in a system oriented toward program development and testing such that the efficiency of the assembly process is an important consideration

**Forward Reference:** **Load-and-go assembler**

Omits the operand address if the symbol has not yet been defined Enters this undefined symbol into SYMTAB and indicates that it is undefined Adds the address of this operand address to a list of forward references associated with the SYMTAB entry Scans the reference list and inserts the address when the definition for the symbol is encountered. Reports the error if there are still SYMTAB entries indicated undefined symbols at the end of the program Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error.