MCA15

KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Housefed Complex, Dispur, Guwahati - 781 006

Master of Computer Applications

SYSTEM PROGRAMMING

CONTENTS

Subject Expert

Prof. Anjana Kakati Mahanta, Deptt. of Computer Science, Gauhati University
Prof. Jatindra Kr. Deka, Deptt. of Computer Science and Engineering, Indian Institute of Technology, Guwahati
Prof. Diganta Goswami, Deptt. of Computer Science and Engineering, Indian Institute of Technology, Guwahati

Course Coordinator

Tapashi Kashyap Das, Assistant Professor, Computer Science, KKHSOU
Arabinda Saikia, Assistant Professor, Computer Science, KKHSOU

SLM Preparation Team

Units	Contributors	Content Editor
1	**Binod Deka** JRF, IIT Guwahati	**Prof. Jyotiprokash Goswami** Deptt. of Computer Applications, Assam Engineering College, Guwahati
2 & 4	**Sangeeta Kakoty**, Lecturer Deptt. of Computer Science Jagiroad College	
3 & 5	**Navanath Saharia** Research Scholar, Tezpur University	

January 2011

The university acknowledges with thanks the financial support provided by the **Distance Education Council, New Delhi**, for the preparation of this study material.

Printed and published by Registrar on behalf of the Krishna Kanta Handiqui State Open University.

Housefed Complex, Dispur, Guwahati- 781006; Web: www.kkhsou.org

MASTER OF COMPUTER APPLICATIONS

SYSTEM SOFTWARE

DETAILED SYLLABUS

Marks

Unit 1 : Introduction to Language Processor **20**

Language Processor: Activities, Phase and Pass; Types of Language Processor: Compiler, Interpreter, Assembler; Phases of Compiler; Programming Language Grammer; Terminal Symbols, Alphabets and Strings.

Unit 2 : Assembler **20**

Features of Assembly Language Programming, Statement Format, Assembly Language statements, Advantage of Assembly Language, Design specification of Assembler.

Unit 3 : Linker and Loader **20**

Linker, Loader, Two Pass Linking, Loader Scheme: General Scheme Loader, Compile and Go loader, Bootstrap Loader; Object Files; Object Code Library; Concept of Relocation; Symbol and Symbol Resolution; Overlay; Dynamic Linking.

Unit 4 : Compilers and Interpreters **20**

Aspects of Compilation, Phases of a Compiler: Analysis and Synthesis Phase; Memory Allocation: Static and Dynamic; Compilation of Expressions; Interpreters.

Unit 5 : Scanning and Parsing **20**

Finite State Automaton: DFA and NFA; Regular Expression; Parsing: Role of Parsing, Parse Tree; Top-Down Parsing:Recursive Descent Parsing, Predictive Parsing; Bottom-Up Parsing; Operator precedence grammars; LR Parsing.

COURSE INTRODUCTION

This course on "***System Programming***" deals with the activity of programming system softwares. System programming aims to produce software which provides services to the computer hardware.

The course consists of five units which are as follows :

Unit 1 This unit introduces the concept of the language processor. Different types of language processors like compiler, interpreter, assembler are briefly described in this unit.

Unit 2 This unit is on assembler. With this unit, learners will be acquainted with assembly language programming along with the design specification of the assembler.

Unit 3 This unit deals with the concept of the linker and the loader. Learners will be acquainted with different loader scheme, object code library and concept of relocation.

Unit 4 This unit focuses on the compiler and the interpreter. Concept of memory allocation is also discussed in this unit.

Unit 5 This unit the last unit of this course. Various important concepts like finite state autometa, regular expressions etc. are discussed in this unit. The concept of parse tree and different parsing techniques are also presented in this unit.

Each unit of this course intends to include some boxes along-side the main sections, to help you know some of the difficult, unfamiliar terms. Some "EXERCISE" have also been included to help you apply your own thoughts. You may find some boxes marked with: "LET US KNOW". These boxes will provide you with some additional interesting and relevant information. Again, you will get "CHECK YOUR PROGRESS" questions. These have been designed to self-check your progress of study. It will be helpful for you if you solve the problems put in these boxes immediately after you go through the sections of the units and then match your answers with "ANSWERS TO CHECK YOUR PROGRESS" given at the end of each unit.

UNIT 1 : INTRODUCTION TO LANGUAGE PROCESSOR

UNIT STRUCTURE

1.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- learn about language processors
- learn the various phases and passes of a language processor
- learn about the types of language processor
- learn the programming language grammar
- define the terminal symbols, alphabets and grammar.

1.2 INTRODUCTION

To make you understand about *language processor* we can take the help of one real life example. Suppose you are talking with a stranger who knows

only English. But you are not good at English. Then whenever the stranger speaks, you will first try to understand the type of language he is speaking i.e. your brain will process (try to understand) it and convert (make understandable) the language into your own language. So we have seen that the main purpose of language processor is to understand the language that is speaking to us. Similar activity is going inside a computer. As we know a computer does not understand our language. Subsequently, computer scientists have developed so many languages (like high level language C/C++) that can be understood by the computer so that we can interact with the Computer to meet our requirements.

In order to process them by the computer we need to have some kind of mechanism like our brain. So, we can describe this mechanism as a set of instructions that perform the processing task. Let us now discuss the basics of language processor in this unit.

1.3 LANGUAGE PROCESSORS

A formal definition of language processor is given bellow :

"A program that performs tasks, such as translating and interpreting, required for processing a specified programming language". Examples of language processors include a FORTRAN processor and a COBOL processor etc.

Language Processors has mainly three purposes :

- bridge gap between Application Domain and Execution Domain
- translation from one language to another
- to detect errors in source code during translation.

1.3.1 Language Processing Activities

Two language processing activities are :

a) Program generation

b) Program execution

Main purpose of the **program generation activities** is to generate program automatically. Program generator is nothing but software which accepts the specification of a program to be generated as an input, then generates the required program in the target language.

Program execution activity organizes the execution of a program written in a programming language on a computer system. During this activity, it determines how to fetch a particular instruction, how to decode these instruction in order to determine the required operations to be performed etc.

1.3.2 Phase and Pass of a Language Processor

There are mainly two phases of language processor which are :

a) Analysis phase and

b) Synthesis phase

The objective of the analysis phase is to break up the source code and imposes into a grammatical structure to create an *intermediate representation* (IR). In addition to it, it also collects information like labels from source code and stores in a *symbol table*.

Whereas the synthesis phase, construct the **object code** from the intermediate representation and symbol table. Here the object code means the code produced after the compilation; in other words the low level machine code is termed as object code. Some other objectives of this phase includes: obtain machine code from mnemonics table, check the address of an operand from symbol table and synthesize a machine instruction. We will discuss later more about this phase while discussing the complier.

Pass of a language processor describes how many times it processes the source program. In simple term, a pass of a language processor indicates the processing of every statements once in a source program.

There are two passes of language processors :

Pass 1 : Perform the analysis of the source program and collect the important information from it.

Pass 2 : Perform the synthesis of the target programmers.

1.4 TYPES OF LANGUAGE PROCESSOR

Language processor is divided into three types, they are :

- Compiler
- Interpreter
- Assembler

1.4.1 Compiler

This language processor translates the complete source program as a whole into machine code before execution. Examples: C and C++ compilers.

Fig. 1.1

If there are errors in the source code, the compiler identifies the errors at the end of compilation. Such errors must be removed to enable the compiler to compile the source code successfully. The object program generated after compilation can be executed a number of times without translating it again.

1.4.2 Interpreter

It translates each statement of source program into machine code and executes it immediately before to translate the next statement. If there is an error in the statement the interpreter terminates its translating process at that statement and displays the error message. The GWBASIC is an example of interpreter.

Generally Programmers write program for specific purposes in high level language which the CPU directly cannot execute. So this high level code has to be converted into machine code. This conversion is done by an interpreter.

Interpreter does not produce the target executable code for the computer. The operations performed by an interpreter are:[* Internal error: Invalid file format. | In-line.WMF *]

a) It directly executes the input code (i.e. source code) line by line by using the given inputs and producing the desired outputs.

b) It may translate source code into some intermediate language and then execute this code immediately. Examples of such language are: Python, Perl etc.

c) It can also execute previously stored previously compiled code, made by compiler that is part of interpreter system. Java is the best example of this type language.

1.4.3 Assembler

Before discussing the assembler you should know what an assembly language is. *Assembly language* is the symbolic representation of a computer's binary encoding - **machine language** . Assembly language is more readable than machine language because it uses symbols instead of bits. The symbols in assembly language name commonly occurring bit patterns, such as opcodes and register specifiers, so programmer can read and remember it. In addition, assembly language permits programmers to use *labels* to identify and name particular memory words that hold instructions or data.

A tool called an **assembler** translates assembly language into binary instructions. Assemblers provide a friendlier representation than a computer's 0s and 1s that simplifies writing and reading programs. Symbolic names for operations and locations are one facet of this representation. Another facet is programming facilities that increase a program's clarity.

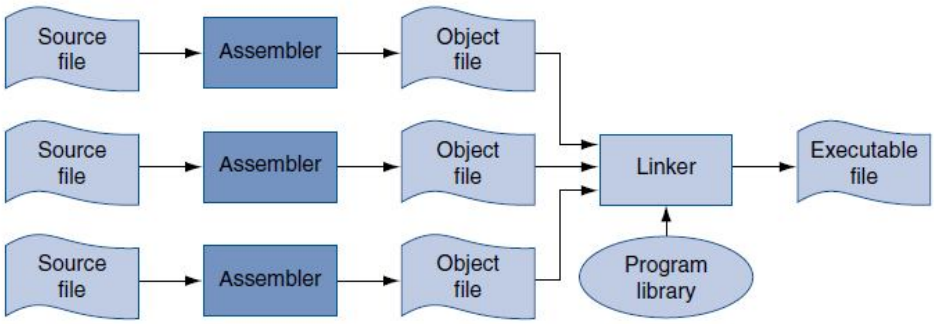


Fig. 1.2

An assembler reads a single assembly language *source file* and produces an *object file* containing machine instructions as shown in the above figure. Another tool, called a **linker,** combines a collection of object and library files into an *executable file,* which a computer can run.

1.5 PHASES OF A COMPILER

A compiler operates in phases, each of which transforms the source program from one representation to another. A typical decomposition of a compiler is shown in the figure below. The first three phase, forms the analysis portion of a compiler and the last three phase forms the synthesis phase of a compiler. A brief description of these phases is given the unit 4.

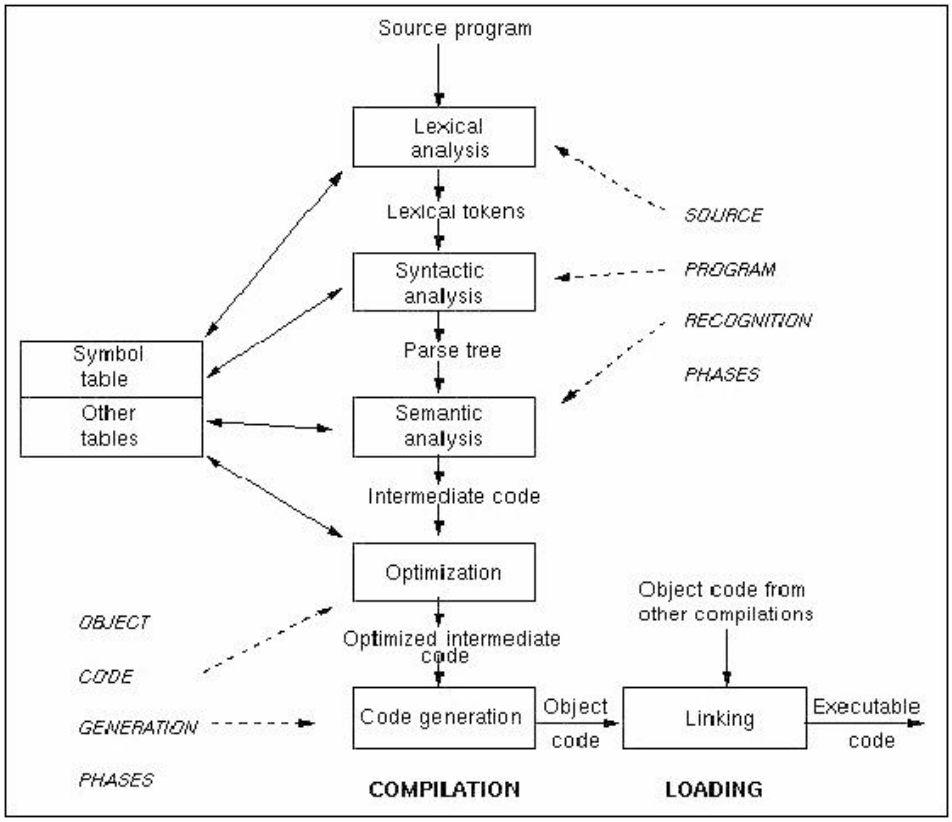


Fig. 1.3

Symbol Table : Compiler uses symbol table to keep track of scope and binding information about names found in the source code. The table is changed every time a name is encountered. Changes to this table occur i) if a new name is discovered ii) if new information about an existing name is discovered.

CHECK YOUR PROGRESS

1. State true or false of the following:
   a) COBOL processor is one example of a language processor.
   b) Translation from one language to another language is one of the purposes of language processor.
   c) Program generation is not the activity of language processors.
   d) Lexer is responsible for generating the token from the input.

1.6 PROGRAMMING LANGUAGE GRAMMAR

The lexical and syntactic features of any programming language are determined from its grammar. Here we are going to discuss about the basics of grammar required for any programming language. As we already know that a language **L** is nothing but the collection of some meaningful sentences. Each sentence in the language consist of sequence of valid words and each word consist of some letters and /or symbols with respect to the particular language **L**. Such type of language can be termed as ***formal language***.

So, we can define the Grammar for such formal language L as the set of rules which precisely specify the sentences. You may think that our natural language (English,MIL) are also called formal language but it is not true. We can not call it since natural language vocabulary is so rich to remember all. But we can call programming languages as formal languages.

1.7 TERMINAL SYMBOLS, ALPHABET AND STRINGS

Before going to the details of Grammar, we have to understand the basic terminology which is used with grammar like terminals, alphabet etc. We are discussing about the $\sum$ grammar of a programming language L. The **alphabet** of a language L is denoted by is the collections of symbol in its characters set.

The lower case letters like **a, b, c, d ….etc** are used to denote the symbols in $\sum$. A symbol in the alphabet is termed as **terminal** symbol (denoted by T) of a language L.

In mathematical notation alphabet representation is shown as below :

$\sum$ = { a, b, c, d, e.....z, 0, 1, 2, 3, 4.....9}

A **string** is defined as the finite sequence of symbols which is represented by α, β, γ... etc. α = **axy** is a string over. The length of a string is the number of symbols in it. ε used to denote the **null string**, meaning that

string which has no symbols. To build larger string from the existing string concatenation operations is used.

Grammar has mainly four components: **Terminal** (denoted by lower case letter), **Non-Terminal** (These are the symbols which can be replaced, generally denoted by capital letter), **Start Symbol** (Generally a non-terminal symbol) and **production rules** (some rules to create new strings).

Production rules are often written in the form: head $\rightarrow$ body; e.g., the rule z0 $\rightarrow$ z1 specifies that z0 can be replaced by z1.

In order to generate a specific string in the language say L, we must begin with the start symbol that leads us to that string by using the production rules. The production rules are applied any order until a string that contains neither the start symbol nor the designated non-terminal is produced. Any particular sequence of production rules on the start symbol yields a unique string in that particular language. If there are multiple way of using production rules for that string then we call that grammar as ambiguous.

For example, suppose the alphabet are a and *b*, the start symbol is *S*, and the production rules:

1. S $\rightarrow$ aSb
2. S $\rightarrow$ b

here a,b are the terminals and S is non-terminal as well as the start symbol. Suppose using the above production rules we want to generate a string **aabbb.** So, starting with S we get the string as:

S => aSb => aaSbb => aabbb

Like by using specific ordering of the production rules (if more than one production rule exist) we can generate any string for a particular language.

CHECK YOUR PROGRESS

2. Fill in the blanks
   a) Terminals are denoted by _____.
   b) _____ are the set of rules which precisely specify the sentences.
   c) Natural languages are also called _____.
   d) Grammar has mainly _____ components.

1.8 LET US SUM UP

- Main purpose of language processor is to make a particular language understandable to the computer.
- Language processor can translate from one language to another language and it can also detect the errors in the source code.
- Compiler, Interpreter and Assembler are different types of language processor.
- Language processor has mainly two phases namely analysis phase and synthesis phase.
- Interpreter translates each statement of source program into machine code and executes it immediately before to translate the next statement.
- Assembler is for low-level programming language, while a compiler is the translator program for high-level programming languages.
- Compiler uses symbol table to keep track of scope and binding information about names found in the source code
- A Grammar has mainly four elements terminals, non-terminals start symbol and production rules.

1.9 ANSWERS TO CHECK YOUR PROGRESS

1. a) True b) True c) False d) True
2. a) T b) Grammar c) formal language
   d) four

1.10 FURTHER READINGS

- *System Programming and Operating Systems*, D. M. Dhamdhere, Tata Mc-Graw Hill. Operating.
- *Systems and System Programming*, P. Balakrishna Prasad, Scitech.

1.11 PROBABLE QUESTIONS

1. What do you mean by language processor? What is the purpose of it.?
2. Mention two language process activities.
3. What are the phases and passes of a language processors?
4. Explain the various types of language processors?
5. Define compiler. Mention the different phases of a compiler.
6. What is the purpose of Symbol table?

UNIT 2 : ASSEMBLER

UNIT STRUCTURE

2.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- learn about assembly language programming
- define the basic features of assembly language programming
- describe formats and characteristics of assembly language programming as well as their advantages
- learn about design specification of assembler.

2.2 INTRODUCTION

We have already come across the concept of language processor and its types. We are also acquainted with compiler, interpreter and assembler with their functionalities. We have also learnt the basic idea of system programming, their components and phases, how they processed, what are different rules like semantic, syntax and lexical, their grammar, symbol, alphabets etc.

In this unit, we will discuss about assembly language programming, their formats of writing, and design specification of assembler.

2.3 BASIC FEATURES OF ASSEMBLY LANGUAGE PROGRAMMING

As we know that assembly language is a low level language, so is machine dependent. It means that the codes of assembly language are different in different computer system or in different family of computers. Assembler is a program that accepts assembly language as input and produce corresponding machine codes as outputs. It has three basic features to simplify programming:

1. **Mnemonic Operation Codes (*mnemonic opcodes*):** Instead of writing only the combination of 0's and 1's, now machine is using few operation codes to eliminate the difficulties of using numeric codes, which is called mnemonic operation code. It can memorize as well as can understand by user and machine also. It also enables the assembler to provide helpful diagnostics.
2. **Symbolic Operands**: Instead of specifying the memory address of the operand/data directly in the program, symbolic names are used in the assembly language program and the assembler takes care of translating these symbolic names. This leads to a very important practical advantage during program modification.
3. **Data Declarations:** Data can be declared in a variety of notations to avoid manual conversations of constants. The declaration may be in decimal, octal and hexadecimal notations.

2.4 STATEMENT FORMAT

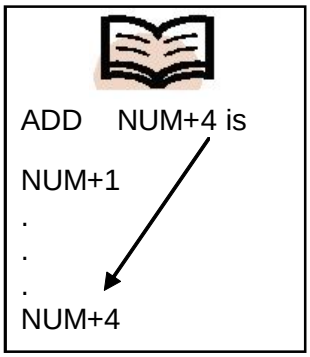
An assembly language statement has the following format :

[Label] <Opcode> <Op. Spec> [,<Op. Spec>.....]

Where <Op. Spec> is the operand specification and has the following syntax:

<Symbolic name> [+<displacement>][(<index register>)]

If a label is mentioned in a statement, it has to be associated with symbolic name generated by operand specification. Some possible operand forms are :



ADD	**NUM**	where NUM refers to the memory word associated with.
ADD	**NUM+5**	where memory word associated is 5 location away from memory word NUM is associated. So, 5 is the displacement or offset from NUM.
ADD	**NUM(4)**	here 4 is the index register no., not value

We can use any combination of that forms also like NUM+4(4) etc. Following are the mnemonic codes used in assembly language for machine instructions :

Instruction Opcode	**Assemble mnemonic**	**Remarks**
00	STOP	Stop execution
01	ADD	
02	SUB	
03	MULT	
04	MOVER	move memory to Register
05	MOVEM	move register to memory
06	COMP	comparison
07	BC	branch on condition
08	DIV	
09	READ	
10	PRINT	

Mnemonics ADD, SUB, DIV, and MULT are for arithmetic operation and are performed in register. The comparison instruction sets a condition code analogous to a subtract instruction without affecting the values of its operands. Condition code can be tested by BC instruction. The assembly instruction format of BC is

BC <condition code specif>, <memory address>

The condition code specifications are the character code used for simplicity. They are :

LT	1	less than
LE	2	less than equal
EQ	3	equal
GT	4	greater than
GE	5	greater than equal
ANY	6	unconditional transfer

The general format for machine instruction is :

sign **opcode** **reg. operand** **mem. operand**

Opcode occupies	2 digits
Register operand occupies	1 digit
Memory operand occupies	3 digits

Sign is not a part of the instruction.

2.5 ASSEMBLY LANGUAGE STATEMENTS

The three kinds of statements in assembly language are *Imperative statements*, *Declarative statements* and *Assembler Directives*.

- **Imperative Statements:**

  It indicates the action to be performed during the execution of the assembled program. Each imperative statement translates into one machine instruction. For example, ADD, MOVE, READ etc.

- **Declarative Statements:**

  It declares constants and storage areas in a program. The syntax of writing declarative statement is :

[Label]	DS	<constant>
[Label]	DC	'<value>'

  Where DS represents Declare Storage and DC represents Declare Constant. These are used for intermediate code generation. Consider the following instruction format,

			Memory area occupies
A	DS	1	A
N	DC	'1'	N 1

Here DS occupies the memory area to store one word memory and DC occupies to store value 1. Using DS we can reserve a blocks of memory also. To reserve a block of 100 memory words, the statement to be,

A DS 100

If we want to retrieve the fifth word of the memory that already specified, then have to write A+4, which is the 5th position and A is the memory location.

For DC, the programmer can declare in different forms like decimal, binary, hexadecimal etc. The assembler converts them to the appropriate internal form. Actually DC is mainly used to initialize the memory words to given values. These values may change inside the program if we give new values to it. Assembler does not have any protection for that.

In HLL, DC can implement in two ways: as **immediate operands** and as **literals**. Immediate operands can be used only if the target machine has the necessary features and it is written as

ADD AREAG, 5

When it will be translated into machine language, it will have two operands: AREAG and the value '5' as immediate operands. This feature does not support by simple assembly language program of other machines except Intel 8086.

Literals are operands and are different from constant because its value does not change during the execution of a program and its location cannot be specified in the assembly program. It is written as :

ADD AREAG, = '5'

The equivalent statements by using DC is :

```
ADD         AREAG, FIVE
.....
.....
FIVE        DC          '5'
```

It is different from immediate operands because it does not require architectural provision of target machines.

- **Assembler Directives:**

  It directs the assembler to take certain action during the process of assembling the program. Assembler directives are START and END and are used as

  START **<constant>**

  END **[<operand specification>]**

Constant of START directives indicate that the target program generated by assembler should place in that memory location. The operand specification of END directives is optional. It indicates the beginning location of the program where it executed, but by default it takes always the first instruction of the assembled program.

Here we are giving an assembly language program and its corresponding machine language program to find the factorial of N (N!) value for your understanding.

Assembly Program			**Machine Lang. Prog.**			
	START	101				
	READ	N	101)	+ 09	0	113
	MOVER	BREG, ONE	102)	+ 04	2	115
	MOVEM	BREG, TM	103)	+ 05	2	116
AG	MULT	BREG, TM	104)	+ 03	2	116
	MOVER	CREG, TM	105)	+ 04	3	116
	ADD	CREG, ONE	106)	+ 01	3	115
	MOVEM	CREG, TM	107)	+ 05	3	116
	COMP	CREG, N	108)	+ 06	3	113
	BC	LE, AG	109)	+ 07	2	104
	MOVEM	BREG, RST	110)	+ 05	2	114
	PRINT	RST	111)	+ 10	0	114
	STOP		112)	+ 00	0	000
N	DS	1	113)			
RST	DS	1	114)			

ONE	DC	'1'	115)	+ 00	0	000
TM	DS	1	116)			
	END					

Fig. 2.1 : An Assembly and its equivalent machine language program

There is no machine language for assembler directives i.e. for START and END there is no corresponding machine language.

2.6 ADVANTAGES OF ASSEMBLY LANGUAGE

The advantages of using assembly language program over machine language are :

1) Machine language programs are very difficult to read, write and understand whereas assembly language programs are easy to read, write and understand.

2) Machine language programs are difficult to remember, whereas mnemonics of assembly language are easy to remember. For example, to remember ST is easier than 01010000.

3) Machine language addresses are absolute whereas in assembly language addresses are symbolic

4) Introduction of data to program is easier in Assembly language than machine language.

CHECK YOUR PROGRESS

1. Fill in the blanks:
   a) An Assembler is _____.
   b) Mnemonics are _____.
   c) Opcode uses _____ in machine instruction.
   d) DS is _____.
   e) Literals are _____.
   f) For assembler directive END, _____ is optional.

g) DC is used to _____ memory words.

h) DS is used to reserve _____.

i) Assembly program contains _____ of statements.

j) ADD, MOVE, READ are example of _____ statements.

2. Write True or False:

a) An assembler converts assembly language to machine code.

b) Mnemonics are machine operation code.

c) In assembler, data can be declared only in decimal notation.

d) Declarative statements are used for immediate code generation.

e) DS does not have corresponding machine language program.

2.7 DESIGN SPECIFICATION OF ASSEMBLER

Here in this section we will give preliminary idea to design of an assembler. It is a four step approach.

1. Identify the information necessary to perform a task.
2. Design a suitable data structure to record the information
3. Determine the processing necessary to obtain and maintain the information
4. Determine the processing necessary to perform the task.

There are two intermediate phases to get the target program from source program. So, the general model for the translation process can be represented as follows :

Analysis of source text + synthesis of target text

= translation from source to target

The fundamental information requirements arise in the synthesis phase of an assembler. In the analysis phase, we are concerned with the determination of meaning of a source language text. The functions of these two phases are :

- **Analysis phase:**

  The primary function of this phase is the building of symbol table. For this, it must determine the addresses with which the symbolic names used in a program are associated. The functions are:

  — Isolate the table, mnemonic operation code and operand fields of a statement

  — Enter the symbol found in label field (if any) and address of the next available machine word into the symbol table.

  — Validate the mnemonic operation code by looking it up in the mnemonic table.

  — Determine the storage requirements of the statement by considering the mnemonic operation code and operand fields of the statement.

  — Calculate the address of the first machine word following the target code generated for this statement. This is called the location counter.

For example, consider the assembly statement :

AGAIN LOAD RESULT + 4

Our first work is to separate the codes and here we found that AGAIN appears in the label field, LOAD in the opcode mnemonic field and RESULT + 4 in the operand field. Then will find the rules of meaning for the codes. We have thus completed analysis of the source statement. Then the work of synthesis phase will start.

- **Synthesis phase :**

  In this phase we will select the appropriate machine operation code for the mnemonics of the statement and place it in the machine instruction's opcode field. Then find the address of the corresponding operand and place it in the address field. The detail functions are :

- Obtain the machine opcode corresponding to the mnemonic operation code by searching the mnemonics table.
- Obtain address of the operand from the symbol table.
- Synthesize the machine instruction or the machine form of the constant, as the case may be.

For the assembly statement considering in the analysis phase, now the work of the synthesis phase is that, first to select the appropriate machine instruction's opcode for the mnemonic LOAD and place it in the machine instruction's opcode field. Then will evaluate the address corresponding to the operand expression 'RESULT + 4' and place it in the address field of the machine instruction. This would complete the translation of this assembly language statement.

- **Data structure of an assembler**

  To implement memory allocation, a data structure called *location counter* (LC) is introduced. The LC is always contains the address of the next memory word in the target program, so it points to the new location. It is initialized to the START statement. Whenever the analysis phase sees the label in the assembly statement, it enters the label where LC is pointing in the symbol table. It then finds the number of memory word required by the assembly statement and update the position of LC. To update the contents of LC, the analysis phase needs to know the lengths of different instructions. Accordingly the mnemonic table also can be extended and the information will include in the new field called ***length***. The data structure of analysis and synthesis phase is :

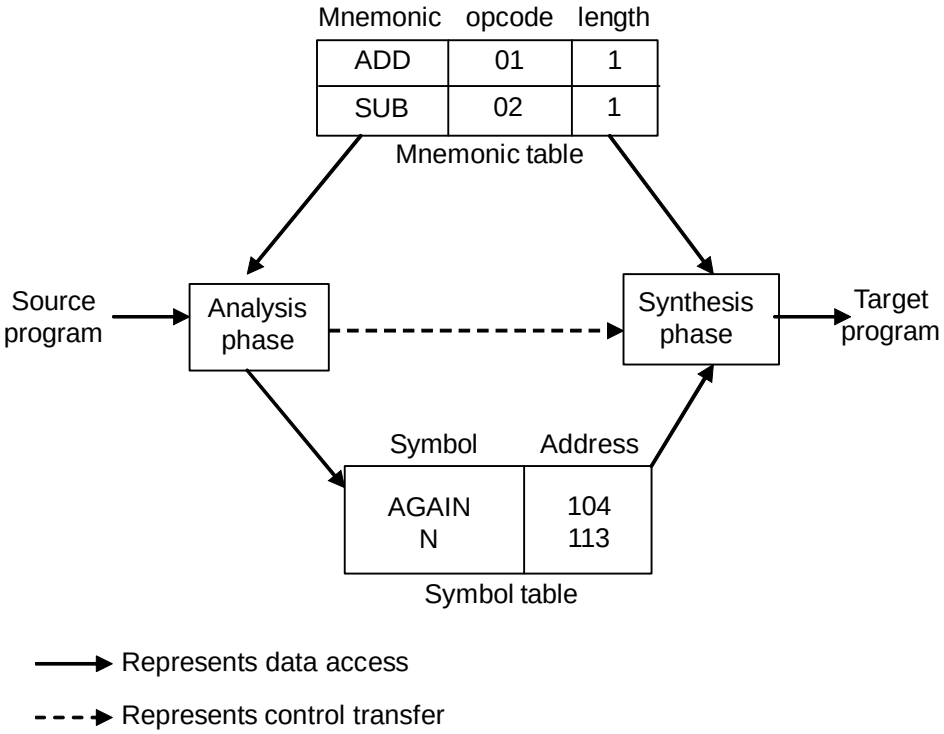
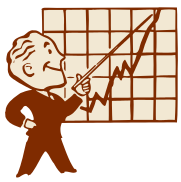


Fig. 2.2 : Data structure of assembler

The processing involved in maintaining the location counter is called ***LC processing***. Mnemonic table is a fixed table which is merely accessed by the analysis and synthesis phases while the symbol table is constructed during analysis and synthesis phase.



CHECK YOUR PROGRESS

3. Write True or False:
   a) Each entry of symbol table has two primary fields.
   b) Mnemonic table is frequently accessed by the analysis and synthesis phases.
   c) There are two intermediate phases to get the target program from source program.
   d) Location counter always contains the address of the previous memory word in the target program
   e) Name and address are the primary fields of symbol table.
   f) Mnemonic table has primary fields Mnemonics and Opcode.

2.8 LET US SUM UP

- Mnemonic Operation Codes, symbolic operand and data declarations are three basic features of Assembly language programming.
- The format of an assembly language statement is :

  [Label] <Opcode> <Op. Spec> [,<Op. Spec>.....]

  Where <Op. Spec> is the operand specification and has the following syntax:

  <Symbolic name> [+<displacement>][(<index register>)]
- All arithmetic are performed in register and set a condition code. The condition code is tested by a Branch of Condition (BC) instruction.
- An assembly language contains three kinds of statements: Imperative, Declarative and Assembler Directives.
- Generally assembly language programmers write the value of a constant operand as a part of the instruction that used it. To avoid this, define the constant and makeup a label for it. Such operand is called **literal**. It is different from constant.
- Synthesis phase and analysis phase are two phases of design specification of an assembler. In synthesis phase two data structures: symbol table and mnemonic table are used. In analysis phase, assembler isolates the mnemonic opcodes, label and operands.
- LC processing is done in the analysis phase.

2.9 ANSWERS TO CHECK YOUR PROGRESS

1. a) Program b) Operation Code c) 2 digits
   d) Declare Storage e) Operands
   f) Operand specification g) Initialize h) Memory area
   i) 3 kinds j) imperative Statements
2. a) True b) True c) False d) True e) True
3. a) True b) False c) True d) False e) True
   f) True.

2.10 FURTHER READINGS

- System Programming and Operating Systems, D. M. Dhamdhere, Tata Mc-Graw Hill. (Second Edition)
- Operating Systems and System Programming, P. Balakrishna Prasad, Scitech. (Second Edition)

2.11 PROBABLE QUESTIONS

1. Compare machine language and assembly language.
2. Explain three basic features of assembly language programming.
3. Show with example the writing format of assembly language statement.
4. What are different kinds of assembly language statement? Explain.
5. Discuss the advanced assembler directives with example.
6. Discuss different data structure used by an assembler.
7. How literals are different from constant.
8. Discuss the functions of both phases of an assembler during translating a source program to its processed form.

UNIT 3 : LINKER AND LOADER

UNIT STRUCTURE

3.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- learn the basics of loader and linker
- learn about two pass linking
- describe different scheme of loading
- illustrate the working strategy of Linker and Loader

- learn about basics object file
- learn the basic concept of relocation
- describe about object code library
- define symbol and symbol resolution.

3.2 INTRODUCTION

In this unit, we will discuss the concept of linking and loading. Linking is a process of combining various pieces of code, module or data together to form a single executable unit that can be loaded in memory. Linking can be done at compile time, at load time (by loaders) and also at run time (by application programs). A loader is a program that takes object program and prepares it for execution. Once such executable file has been generated, the actual object module generated by the linker is deleted. Thus, although the linker itself produces an intermediate relocatable module, the more usual result of a linker call is an executable file that is subsequently produced by the Loader.

3.3 LINKER

A linker reads all of the symbol tables in the input module and extracts the useful information, which is sometimes all of the incoming information but frequently just what's needed to link. Then it builds the link-time symbol tables and uses those to guide the linking process. Depending on the output file format, the linker may place some or all of the symbol information in the output file. Linkers handle a variety of symbols. All linkers handle symbolic references from one module to another. Each input module includes a symbol table. The symbols include the following:

- Global symbols defined and perhaps referenced in the module.
- Global symbols referenced but not defined in the module (generally called externals).
- Segment names, which are usually also considered to be global symbols defined to be at the beginning of the segment.
- Non-global symbols, usually for debuggers and crash dump analysis (optional). These are not really symbols needed for the linking

process, but sometimes they are mixed in with global symbols so the linker has to at least skip over them. In other cases, they may be in a separate table in the file or in a separate debug information file.

Line number information, to tell source language debuggers the correspondence between source lines and object code (optional).

3.4 LOADER

Loader is the program that accomplished the loading task. Loading is the process of bringing a program into main memory so that it can run. On most modern systems, each program is loaded into a fresh address space, which means that all programs are loaded at a known fixed address, and can be linked for that address. Loading is pretty simple from this point and requires the following steps:

- Read enough header information from the object file to find out how much address space is needed.
- Allocate that address space, in separate segments if the object format has separate segments.
- Read the program into the segments in the address space.
- Zero out any BSS (Block Started by Symbol, a portion of data segment containing statically allocated variable) space at the end of the program if the virtual memory system does not do so automatically.
- Create a stack segment if the architecture needs one.
- Set up any run-time information such as program arguments or environment variables.
- Start the program.

If the program is not mapped through the virtual memory system, reading in the object file just means reading in the file with normal read system calls. On systems that support shared read-only code segments, the system needs to check whether there's already a copy of the code segment loaded in and uses that rather than making another copy.

3.5 COMPILER, LINKER AND LOADER IN ACTION : THE BASICS

To keep things simple and understandable, I target all my discussions to ELF (executable and linking format) executables on the x86 architecture (Linux) and use the GNU compiler (GCC) and linker (ld). However, the basic concepts of linking remain the same, regardless of the operating system, processor architecture or object file format being used.

Let us consider two program files, ***a.c*** and ***b.c***. As we invoke the GCC on ***a.c*** and ***b.c*** at the shell prompt, the following actions take place :

gcc a.c

cpp other-command-line options a.c /tmp/a.i

cc1 other-command-line options /tmp/a.i -o /tmp/a.s

as other-command-line options /tmp/a.s -o /tmp/a.o

Run preprocessor on a.c and store the result in intermediate preprocessed file.

Run compiler proper on a.i and generate the assembler code in a.s

Run assembler on a.s and generate the object file a.o

cpp, **cc1** and **as** are the *GNU's preprocessor*, *compiler proper* and *assembler* respectively. They are a part of the standard GCC distribution.

If we repeat the above steps for file b.c, we have another object file, b.o. The linker's job is to take these input object files (a.o and b.o) and generate the final executable :

ld other-command-line-options /tmp/a.o /tmp/b.o -o a.out

The final executable (a.out) is then ready to be loaded. To run the executable, we type its name at the shell prompt :

./a.out

The shell invokes the loader function, which copies the code and data in the executable file a.out into memory, and then transfers control to the beginning of the program. The loader is a program called execve, which

loads the code and data of the executable object file into memory and then runs the program by jumping to the first instruction.

a.out was first coined as the Assembler OUTput in a.out object files. Since then, object formats have changed variedly, but the name continues to be used.

Linker and Loader perform several related but conceptually separate actions.

- Program loading– This includes copy a program from secondary storage to main memory so that it is ready to execute. In some cases loading includes allocating storage, setting protection bits, or mapping virtual addresses to disk pages.
- Relocation– Compilers and assemblers generate the object code for each input module with a starting address of zero. Relocation is the process of assigning load addresses to different parts of the program by merging all sections of the same type into one section. The code and data section also are adjusted so they point to the correct runtime addresses.
- Symbol resolution– A program is made up of multiple subprograms; reference of one subprogram to another is made through symbols. A linker's job is to resolve the reference by noting the symbol's location and patching the caller's object code.

Although there is considerable, overlap exists between the functions of linkers and loaders. One way to think of them is: the loader does the program loading; the linker does the symbol resolution; and either of them can do the relocation, as you can not draw a concrete line between them.

3.6 TWO PASS LINKING

Now we move to general structure of linkers. Input to the linker may be a set of object files, libraries or a command files and produces an object file or some auxiliary information like load map, file containing debugger symbol etc.

Each input file contains a set of *segments,* contiguous block of code or data to be placed in the output file. Each input file also contains at least one symbol table. Some symbols are exported-defined within the file for use in other files, generally the names of routines within the file that can be called from elsewhere. Other symbols are imported-used in the file but not defined, generally, the names of routines called from but not present in the file.

When a linker runs, it first has to scan the input files to find the sizes of the segments and to collect the definitions and references of all the symbols. It creates a symbols table listing all the segments defined in the input file and a symbol table with all the symbols imported or exported. Using the data from the first pass the linker assigns numeric locations to symbols, determines the sizes and location of the segments in the output address space and figures out where everything goes in the output file.

The second pass uses the information collected in the first pass to control the actual linking process. It reads and relocates the object code, substituting numeric address for symbol references and adjusting memory addresses in code and data to reflect relocated segment addresses, and writes the relocated codes to the outputs file. It then writes the output file generally with header information, the relocated segments and symbol table information. If the program uses dynamic linking the symbol table contains the information, the run time linker will need to resolve dynamic symbols.

A few linkers appear to work in one pass. They do so by buffering some or all of the contents of the input files in memory or disk during the linking process then reading the buffered material latter.

3.7 LOADER SCHEME

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

3.7.1 General Scheme Loader

In this loader scheme, some translator (assembler) converts the source program to object program. 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.

3.7.2 '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 that is simply 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.

3.7.3 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 programmer knows the starting address of every module, the 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.

Advantages :

- It is simple to implement and process of execution is efficient.
- 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.
- The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory.

Disadvantages :

- 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.

3.7.4 Direct Linking Loader

The direct linking loader is the most common type of relocatable loader. The loader cannot 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 :

- The length of the object code segment
- The list of all the symbols which are not defined 111 the current segment but can be used in the current segment.
- The list of all the symbols that 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 table. The USE table holds the information such as name of the symbol, address, address relativity.

3.7.5 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.

3.8 OBJECT FILES

Compilers and assemblers create object files containing the generated binary code and data for a source file. Linkers combine multiple object files into one file; loaders take object files and load them into memory. In this section, we delve into the details of object file formats and contents. An object file comes in three forms:

- Relocatable object file, which contains binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file.
- Executable object file, which contains binary code and data in a form that can be directly loaded into memory and executed.
- Shared object file, which is a special type of relocatable object file that can be loaded into memory and linked dynamically, either at load time or at run time.

Compilers and assemblers generate relocatable object files (also shared object files). Linkers combine these object files together to generate executable object files.

3.8.1 What Goes into an Object File?

An object file contains five basic kinds of information. (Some object files contain even more than this, but these categories are plenty to keep us occupied in this chapter.)

- Header information: This is overall information about the file, such as the size of the code, the name of the source file it was translated from, and the creation date.
- Object code: This is binary instructions and data generated by a compiler or assembler.
- Relocation information: This is a list of the places in the object code that have to be fixed up when the linker changes the addresses of the object code.
- Symbols: These include global symbols defined in this module and symbols to be imported from other modules or defined by the linker.
- Debugging information: This includes other information about the object code that is not needed for linking but that is of use to a debugger, such as source file and line number information, local symbols, and descriptions of data structures used by the object code such as C structure definitions.

Not all object formats contain all of these kinds of information, and it's possible to have quite useful formats with little or no information beyond the object code.

3.9 OBJECT CODE LIBRARIES

All linkers support object code libraries in one form or another, with most also providing support for various kinds of shared libraries. The basic principle of an object code library is simple enough (Fig.4.1). A library is little

more than a set of object code files. (Indeed, on some systems you can literally concatenate a group of object files together and use the result as a link library.) If any imported names remain undefined after the linker processes all of the regular input files, it runs through the library or libraries and links in any of the files in the library that export one or more undefined names.

Shared libraries complicate this task a little by moving some of the work from link time to load time. The linker identifies the shared libraries that resolve the undefined names in a linker run, but rather than linking anything into the program, the linker notes in the output file the names of the libraries in which the symbols were found, so that the shared library can be bound in when the program is loaded.

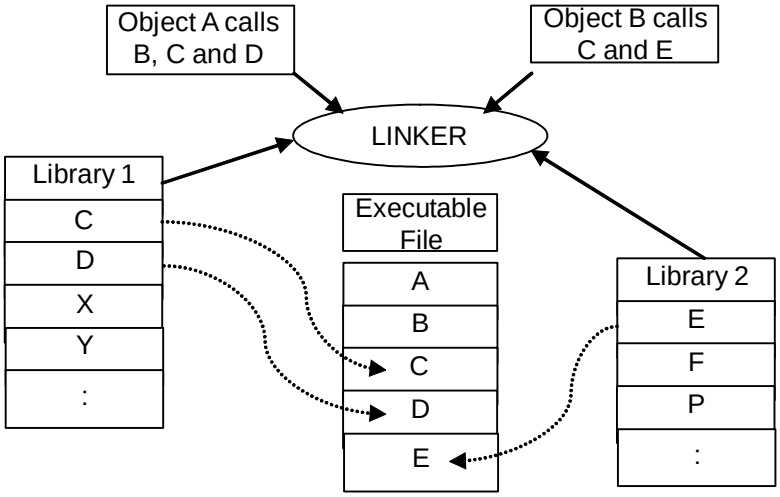


Fig. 3.1 : Object code library call

3.10 CONCEPT OF RELOCATION

Once a linker has scanned all of the input files to determine segment sizes and symbol definitions and symbol references, figured out which library modules to include, and decided where in the output address space all of the segments will go, the next stage is the heart of the linking process: relocation. We use the term relocation to refer both to the process of adjusting program addresses to account for nonzero segment origins and to the process of resolving references to external symbols, because the two are frequently handled together.

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. 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 linker's first pass lays out the positions of the various segments and collects the segment-relative values of all global symbols in the program. Once the linker determines the position of each segment, it potentially needs to fix up all storage addresses to reflect the new locations of the segments. On most architecture, addresses in data are absolute, while those embedded in instructions may be absolute or relative. The linker needs to fix them up accordingly. The linker also resolves stored references to global symbols to the symbol addresses.

3.11 SYMBOLS AND SYMBOL RESOLUTION

Every relocatable object file has a symbol table and associated symbols. In the context of a linker, the following kinds of symbols are present :

- Global symbols defined by the module and referenced by other modules. All non-static functions and global variables fall in this category.
- Global symbols referenced by the input module but defined elsewhere. All functions and variables with extern declaration fall in this category.
- Local symbols defined and referenced exclusively by the input module. All static functions and static variables fall here.

The linker resolves symbol references by associating each reference with exactly one symbol definition from the symbol tables of its input relocatable object files. Resolution of local symbols to a module is straightforward, as a module cannot have multiple definitions of local symbols. Resolving references to global symbols is trickier, however. At compile time, the compiler exports each global symbol as either strong or weak. Functions and initialized global variables get strong weight, while global uninitialized variables are weak. Now, the linker resolves the symbols using the following rules:

1. Multiple strong symbols are not allowed.
2. Given a single strong symbol and multiple weak symbols, choose the strong symbol.
3. Given multiple weak symbols, choose any of the weak symbols.

3.12 OVERLAY

Overlay is a technique that dates back to before 1960, and are still in use in some memory-constrained environments. Several MS-DOS linkers in the 1980 supported them in a form nearly identical to that used 25 years earlier on mainframe computers. Although overlays are now little used on conventional architectures, the techniques that linkers use to create and manage overlays remain interesting. Also, the inter-segment call tricks developed for overlays point the way to dynamic linking. Overlaid programs divide the code into a tree of segments, such as the one in Fig. 3.1. The programmer manually assigns object files or individual object code segments to overlay segments. Sibling segments in the overlay tree share the same memory. In the example, segments A and D share the same memory, B and C share the same memory, and E and F share the same memory. The sequence of segments that lead to a specific segment is called a path, so the path for E includes the root, D, and E.

When the program starts, the system loads the root segment which contains the entry point of the program. Each time a routine makes a “downward” inter-segment call, the overlay manager ensures that the path to the call target is loaded. For example, if the root calls a routine in segment

A, the overlay manager loads section A if it's not already loaded. If a routine in A calls a routine in B the manager has to ensure that B is loaded, and if a routine in the root calls a routine in B, the manager ensures that both A and B are loaded. Upwards calls don't require any linker help, since the entire path from the root is already loaded.

For example, linking the following two programs produces link-time errors:

```
/* first.c */          /* second.c */
int foo () {               int foo () {
  return 0;                  return 1;
        }                          }
int main () {
      foo ();
        }
```

The linker will generate an error message because foo (strong symbol as its global function) is dfined twice.

gcc first.c second.c

/tmp/ccM1DKre.o: In function 'foo':

/tmp/ccM1DKre.o(.text+0x0): multiple definition of 'foo'

/tmp/cclhvEMn.o(.text+0x0): first defined here

collect2: ld returned 1 exit status

Collect2 is a wrapper over linker ld that is called by GCC.

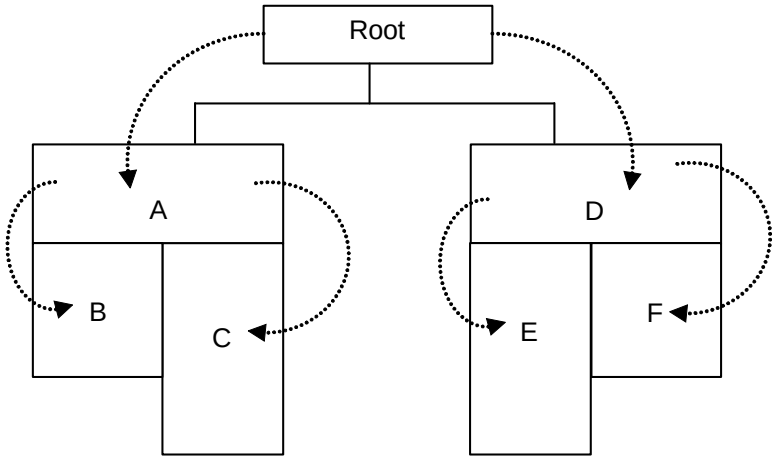


Fig.3.2 : Typical overlay tree format

3.13 DYNAMIC LINKING

Dynamic linking defers much of the linking process until a program starts running or sometimes even later. It provides a variety of benefits that are hard to get otherwise :

- Dynamically linked shared libraries are easier to create than statically linked shared libraries.
- Dynamically linked shared libraries are easier to update than statically linked shared libraries.
- The semantics of dynamically linked shared libraries can be much closer to those of unshared libraries.
- Dynamic linking permits a program to load and unload routines at run time, a facility that can otherwise be very difficult to provide.

There are a few disadvantages, of course. The run-time performance costs of dynamic linking are substantial compared to those of static linking, because a large part of the linking process has to be redone every time a program runs. Every dynamically linked symbol used in a program has to be look up in a symbol table and resolved.

CHECK YOUR PROGRESS

1. Multiple Choice Questions:
   i) The output of the linker (LINK command) is stored in a file with the extension
      a) .lis b) .obj c) .exe d) .lnk
   ii) Resolution of externally defined symbols is performed by:
      a) Compiler b) Assembler c) Linker d) Loader
   iii) Relocatable programs:
      a) Can be loaded at a specific location in memory
      b) Can be loaded almost anywhere in memory
      c) Cannot be used with virtual memory management scheme
      d) All of above

iv) A program that set up an executable program in main memory for execution is

a) Compiler b) Assembler c) Linker d) Loader

v) Which of the following about the loader is/are incorrect?

a) Loader brings object program into memory for execution.

b) Linkage editors perform linking after loading.

c) Dynamic linking schemes delay linking until execution time.

d) Absolute loader modifies the object program so that it can be loaded at any address location.

2. Define absolute loader? With a procedure show the design of an absolute loader?

.....

.....

.....

.....

3.14 LET US SUM UP

In this unit we have learnt the working strategy of linker and loader. *Linking* is the process of combining various pieces of code and data together to form a single executable that can be loaded in memory. Linking can be done at compile time, at load time (by loaders) and also at run time (by application programs). Loading is the process of bringing a program into main memory so that it can run. After that we have acquainted with concept of relocation. We have also learnt various important concepts like symbol resolution overlay and dynamic linking.

3.15 ANSWERS TO CHECK YOUR PROGRESS

1. i) c ii) b iii) c iv) c v) b
2. 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.

Procedure 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

A. For each program segment do Begin
   1. Read the first line from object module to obtain information about memory location. The starting address, S in corresponding object module is the memory location where executable code is to be placed. Hence
   2. Memory_location = S
   3. Line counter = 1, as it is first line While (! end of file)

B. For the current object code do Begin
   1. Read next line
   2. Write line into location S
   3. S = S + 1
   4. Line counter Line counter + 1.

3.16 FURTHER READINGS

- Linker & Loader, John R. Levine, Morgan Kaufmann Publishers.

3.17 PROBABLE QUESTIONS

1. Define loader? What are the functions of loader?
2. Define loading, relocating and linking?
3. What is bootstrap loader?
4. Differentiate between
   - Machine dependent and machine Independent loader feature.
   - Dynamic Linking and Dynamic loading
5. Define relocating loader?
6. Define relative loader?
7. Briefly explain a simple bootstrap loader, with an algorithm show the design of a bootstrap loader?

UNIT 4 : COMPILERS AND INTERPRETERS

UNIT STRUCTURE

4.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- define and describe compilers
- describe the phases of compilation process
- learn about the memory allocation
- define and distinguish interpreters from compilers.

4.2 INTRODUCTION

In our previous unit, we discussed about assemblers. From that we understood that assembler is a translation program that translates assembly language program to their corresponding source code for computers. It means, assemblers are translators for lower level assembly languages. Now in this unit, we are going to discuss another one translator called compiler which is mainly for translating higher level languages.

To execute an higher level language program, it should be converted to its machine language. A compiler is a program that can translates an higher level language program to its machine form. We will discuss here how a compiler converts a program, what strategy they are taking while compiling, how they allocate memory etc. We will also define Interpreter which can analyze a source program statement by statement.

4.3 ASPECTS OF COMPILATION

From the introduction we came to know that compilers are like bridge between programming language (PL) domain and an execution domain. The main aspects are :

1. Generate code to implement meaning of a source program in the execution domain.
2. Provide diagnostics for violations of PL semantics in a source program.

Before discussing about compilers, we should understand few terms related to PL domain that contributes to the semantic gap between PL domain and an execution domain. Those are:

Data types

These are the specification of legal values for variables and legal operations on legal values of a variable. For example, in C language to declare a variable x,

int x ; where ‘int’ is the data type and x is the legal variable. Legal operations are the assignment operation and data manipulation operation. A compiler can ensure by the following tasks that the data types are assigned through legal operations :

1. Checking legality of an operation for the types of its operands. This ensures that a variable is subjected to legal operations of its type.
2. Use type conversion operations to convert values of one type into values of another type wherever necessary and permissible according to the rule of PL.

3. Use appropriate instruction sequences of the target machine to implement the operations of a type.

At the time of type conversion operation, compiler generates a **type specific code** to implement an operation and manipulate the values of the variable through that code. This ensures execution efficiency since type related issues do not need explicit handling in the execution domain.

Data and Control Structures

Data structures used by a PL are arrays, stacks, queues, list, records etc.

When a compiler compiles the data structure used by the PL, it develops memory mapping to access the memory word allocated to the element. A record is a heterogeneous data structure that require complex memory mapping.

The control structure of a language is the collection of language features for altering the flow of control during the execution of a program. That are execution control, procedure calls, conditional transfer etc. the compiler must ensure that the source program does not violate the semantics of a control structure.

Scope Rules

Scope rule is the rule that determines the accessibility of an entity in a program execution. The compiler performs scope analysis and name resolution to determine the data item designated by the use of a name in the source program. The generated code simply implements the results of the analysis.

4.4 PHASES OF A COMPILER

We have already mentioned about the various phase of a compiler. In this section, we will briefly describe about the various phase of a compiler. Recall that the whole compilation process consists of two phase : analysis phase and synthesis phase. Analysis Phase again consist of three sub phases:

- Lexical analysis
- Syntax analysis
- Semantic analysis

Synthesis Phase also have three sub phases :

- Intermediate code generation
- Code optimization
- Code generation

4.4.1 Analysis Phase

Lexical Analysis :

A program which is responsible for converting sequence of characters into sequence of tokens from the source program is known as Lexical analyzer or scanner or lexer. This entire process is called Lexical analysis. By the term Token here means the unit of information present in the source code.

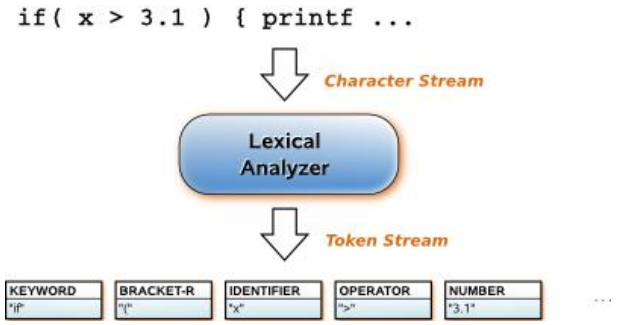


Fig. 4.1

To clarify the concept of token look at the example shown in the above diagram. Each unit of the information's of the C program part are the token. For example "If", "x" "(" etc are the example of tokens. Each token consists at least of a type identifier e.g. "if" is keyword, ">" is an operator, "(","),"{" are the brackets,3.1 is the floating constant etc.

So the main purpose of the Lexical Analysis phase of a compiler is:

- Read the stream of input character from the source code.
- Produce as output as a sequence of tokens.

The lexical analyzer is the first phase of a compiler. As we know it produces a sequence of tokens and the parser uses these token for syntax analysis.

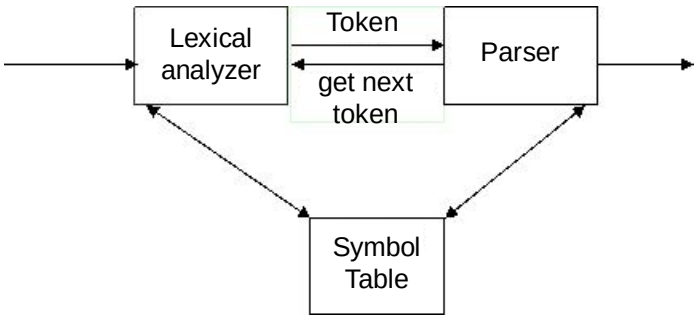


Fig. 4.2

As shown in the above figure, upon receiving a “get next token” command from the parser the lexical analyzer reads input characters until it can identify the next token.

Syntax analysis or parsing :

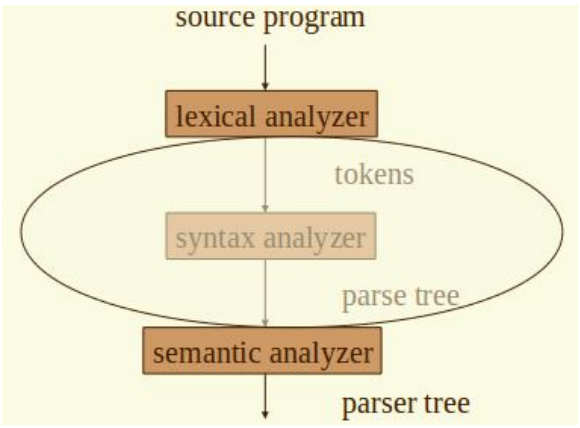


Fig. 4.3

As the lexical analyzer generates tokens from the source code, in this phase it constructs parse tree from the tokens obtain from the previous lexical analyses phase. Syntax analysis is about discovering structure in text and is used to determine whether or not a text conforms to an expected format. “Is this a textually correct C program?”

So the main objective of this phase is to check whether the source code written by us is properly correct or not according to the rules as specified by the particular language. It checks this by constructing the parse tree from the tokens.

For example, consider the following part of a program :

…….

position : = initial + rate * 60

…….

After the lexical analysis phase of the above source we got :

Identifier :

position, initial, rate

Operators :

: = , + , *

Constant :

60

After this we have to determine whether the above statements are structurally correct statement or not by constructing parse tree in this phase.

Parse tree for the above statements is :

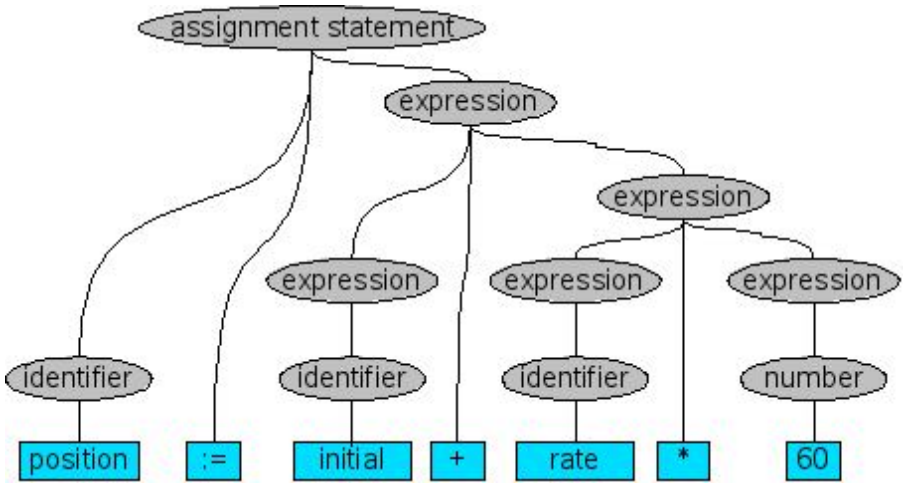


Fig. 4.4

From the parse tree we can say that an identifier, constant (here 60) individually is an expression. In addition to that we can also conclude that initial **+ rate * 60** is an expression. Finally the entire expression is an assignment statement which is proved to be valid as the parse tree shows. This is what the parse tree wants to show.

The above rules as described “an identifier is expression” etc are called grammar rules and the collection of such rules is called a context free grammar or simply a grammar. Context free grammar will be described later in detail.

Semantic analysis :

As we have discussed in the above sections syntax analysis only verifies that the program which consists of tokens arranged in a syntactically valid combination or not. Now in this phase we will go for deeper to check whether they form a sensible set of instructions in the programming language.

For example, any old noun phrase followed by some verb phrase makes a syntactically correct English sentence, a semantically correct one has subject-verb agreement, proper use of gender, and the components go together to express an idea that makes sense.

For a Computer program, to be semantically valid, all variables, functions etc. must be properly defined, expressions and variables must be used in ways that respect the type system, access control must be respected, and so forth.

Here in this phase, we need to ensure that program is sound enough to carry on to code generation. In many languages like C,C++ etc, identifiers have to be declared before they are used. As the compiler encounters a new declaration, it records the type information assigned to that identifier. Then, as it continues examining the rest of the program, it verifies that the type of an identifier is respected in terms of the operations being performed. For example, the type of the right side expression of an assignment statement should match the type of the left side, and the left side needs to be a properly declared and assignable identifier. The parameters of a function should match the arguments of a function call in both number and type. These are the examples of the things checked in the semantic analysis phase.

To take an example of semantic-analysis phase, consider the following C program part :

```
void main()
    {
    ……..............
    int  kkhsou_var,result;
    kkhsou_var=10;
    result = kkhsou_var +100;   // assignment statements
    …..............
    }
```

As we have seen the above assignment statement, the data type of both ‘kkhsou_var’ and ‘result’ must be same, so declared as both type as integer. Such type of checking is one example of semantic-analysis phase functions. So to summarize the above explanations regarding this phase we can conclude the following type of error check are done:

a) Multiple declarations :

   A variable should be declared (in the same scope) at most once.

b) Undeclared variable :

   A variable should not be used before being declared.

c) Type mismatch :

   Type of left hand side of an expression should match the type of right hand side.

d) Wrong arguments :

   Function should be called with right number and types of arguments.

4.4.2 Synthesis Phase

Intermediate code generation :

After the syntax and semantic-analysis phases of a compiler, intermediate code are generated from the source code. We can think of intermediate code as the specific code for an abstract machine. Intermediate code are easy to produce and easy to translate into target program. Intermediate code are also known as **three address code**.

To make the above explanations more clear consider the following example:

…….

position : = initial + rate * 60

…….

after lexical analysis the above statements are converted as bellow

id_1 := id_2 + Id_3 * 60; where “id” indicates the identifiers.

So in this phase the above code becomes

temp1 : = inttoreal(60)

temp2 : = id_3 * temp1

temp3 : = id_2 + temp2

id_1 : = temp3

Code generated during this phase has some special property:

a) Each code has at most one operator in addition to the assignment operator.

b) The compiler must generate temporary name to hold the value computed by each instructions.

c) Some of the three address code has fewer than three operands.

Main advantages of this phase is that it allows machine abstraction, separate high level operation from their low level implementation. Also it allows to reuse of front-ends and back-ends. Also it has some disadvantages; First, it implies an additional pass for the compiler. Second, It complicates optimizations specific to the target architecture. Third, It usually is orthogonal to the target machine, translation to a specific architecture will take longer and be inefficient.

Code optimization :

During this phase, the code optimizer optimizes the code produced by the intermediate code generator in terms of time and space. The main objective of this phase is to improve the intermediate code so that faster running machine code will result. Code optimizations is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output.

Optimizations can be done by a compiler in three places: In the **source code**, in the **intermediate code** and in the **machine code.**

It is best to perform optimizations in the intermediate language, as the optimizations can be shared among all the compilers that use the same intermediate language. Also, the intermediate language is typically simpler than both the source language and the machine language.

One example of code optimization technique is **common sub expression elimination**. As the name implies it searches for the common expression in a program then replaces this common expression with another temporary variable. Consider the following example:

```
a = b * c + g;
d = b * c * d;
```

We can replace the common expression b*c with temp as shown in below.

```
temp = b * c;
a = temp + g;
d = temp * d;
```

Constant propagation is also one way to optimize the code. It is a process of substituting the values of known constants in expressions at compile time as shown in below.

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

Propagating x yield:

```
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

Code generation :

Code generation is the process by which a compiler converts some intermediate representation of source code into machine language that can be easily executed by computer. During this phase the

code generator takes the parse tree as input from the previously executed phases and generates the machine level code.

A compiler translates a program written in a higher level language, such as C, C++, Java etc, into an equivalent machine language or assembly language program.

Consider the following assignment statement.

x := y + 13

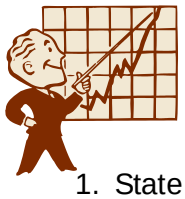
After the code generator executes the above code we get the following :

load y

add constant 13

store x

where x, y, and constant13 are symbolic addresses.



CHECK YOUR PROGRESS

1. State true or false :
   a) Tokens are formed during the lexical analysis phase.
   b) Parse tree is constructed in the semantic analysis phase.
   c) Synthesis phase construct the object code from the intermediate representation (IR) and symbol table.
   d) Intermediate code is generated after the lexical analysis phase.
   e) Code optimization phase produce a faster running machine code.
2. Fill in the blanks :
   a) Compilers are _____ between the PL domain and execution domain.
   b) _____ are legal operations of legal values of the type.
   c) To compile any data structure of a PL, compilers develop _____ to each element.

d) A record is a _____ data structure.

e) Record perform _____ memory mapping.

f) _____ of a language is to alter the flow of control during execution.

4.5 MEMORY ALLOCATION

During semantic analysis of data declaration statement, a compiler allocates memory for their instruction and data separately. This is called memory allocation and used to perform memory binding. Memory allocation is important because,

1. Determine the amount of memory required to represent the value of a data item
2. Use an appropriate memory allocation model to implement the lifetimes and scope of data item.
3. Determine appropriate memory mappings to access the values in a non scalar data item.

4.5.1 Static and Dynamic Memory Allocation

Memory binding is an association between the 'memory address' attribute of a data item and address of a memory area.

In compiler the word *'static'* means *before* the execution of a program begins and *'dynamic'* means *during* the execution of a program. Thus, static allocation implies that storage is allocated to all program variables before the start of program execution. The binding of data item also takes place at compile time. Since no memory is allocated or deallocated during execution, variables are permanent in a program. On the other hand, in dynamic storage allocation, binding is affected during program execution. Static allocation is widely used in scientific languages like FORTRAN and dynamic allocation is used by block-structured languages like PL, ALGOL, PASCAL etc.

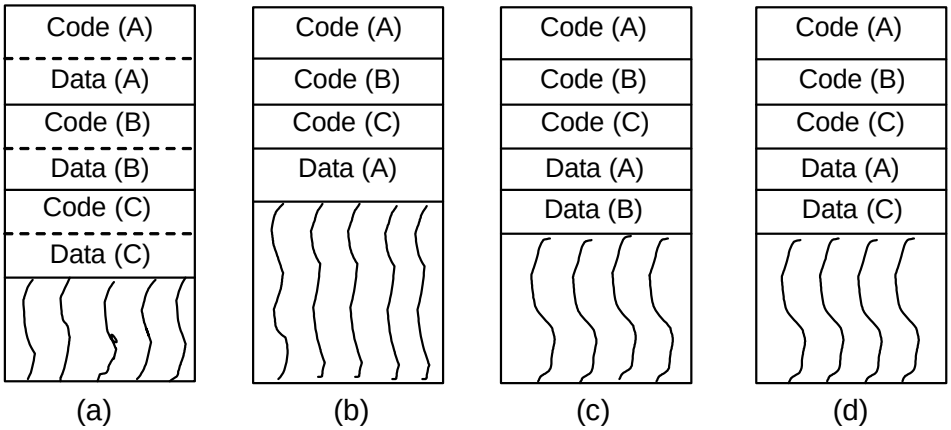


Fig 4.5 : Static and dynamic memory allocation

Figure shows static and dynamic allocation of three program units A, B and C. Part (a) shows static memory allocation. Part (b) shows dynamic allocation when only program unit A is active. Part (c) shows the situation after A calls B. Part (d) shows the situation after B returns to A and A calls C. C has been allocated a part of the memory of B that has deallocated. It is clear from the diagram that static allocation needs more memory than dynamic allocation only when those all program units are active.

Dynamic allocation is again of two types: automatic and program controlled allocation. In **automatic dynamic allocation**, memory is allocated when the program unit is entered during execution and is deallocated when the program unit is exited. Different memory area may be allocated for the same variable if the variable has different activation operation in the program. On the other hand, in **program controlled dynamic allocation**, memory can allocate or deallocate at arbitrary point during execution. It is obvious that in both allocations, address of the memory area allocated cannot be determined at compile time. Automatic allocation is implemented using **stack** since entry and exit follows the LIFO in nature. When a program unit is entered during execution of a program, a record is created in the stack to contain its variables. A pointer is set to point to this record. The program controlled allocation is implemented using a **heap**. A pointer is needed to point to each allocated memory area.

The advantages of dynamic allocation are :

- Recursion can be implemented easily because memory is allocated when the program unit is entered during execution.
- It can support data structure whose size is determined dynamically, e.g. an array declaration.

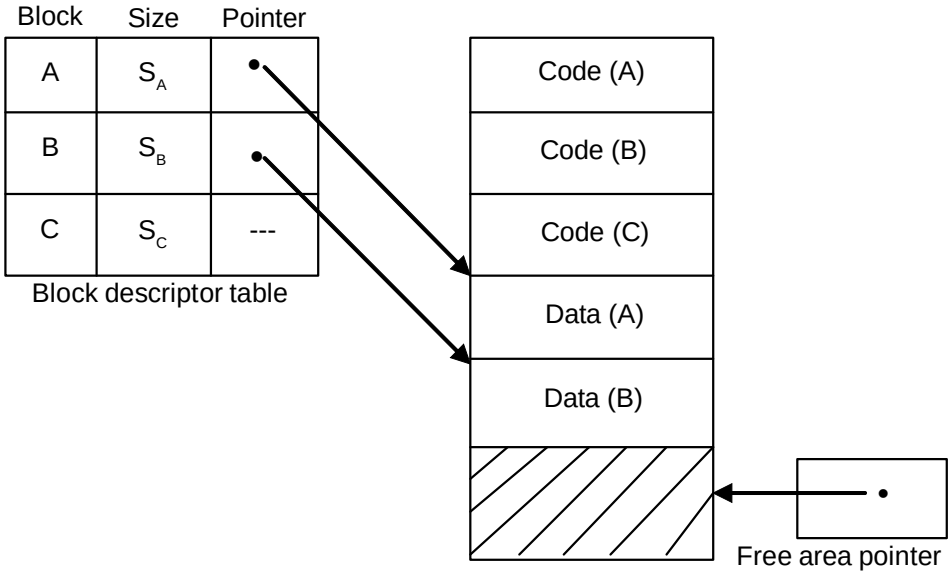


Fig 4.6 : A schematic for managing dynamic storage allocation

At the time of accessing variables and allocating memory dynamically, a compiler should perform many works and as such the extended stack model requires the ***symbol table***. When a particular block is encountered during compilation, a new record is pushed on the stack that contains the nesting level and the symbol table of that block. Each entry in the symbol table contains a variable's name, type, length and displacement in the AR. At the time of searching a name in the symbol table, the topmost record of the stack is searched first.

CHECK YOUR PROGRESS

3. Fill in the blanks :
   a) _____ is an association between the address of memory area and the attribute of data item.
   b) In static memory allocation, variables are allocated _____ the execution of a program.

c) In _____ a program can allocate or deallocate memory at arbitrary point during execution.

d) A _____ is a program unit that can contain data declaration.

4. State True or False :

a) In dynamic allocation, memory binding is done before execution of a program.

b) At the time of allocating memory dynamically, compilers require symbol table.

c) Stack based memory allocation model is adequate for program controlled memory allocation.

d) A Symbol Table contains scope information for variables.

e) Dynamic allocation is widely used by scientific languages.

4.6 COMPILATION OF EXPRESSIONS

In HLL, expressions occur in assignment statements, conditional statement and IO statements etc. In processing these statements, a significant amount of compilation time is spent. Expression processing actions can be grouped as:

1. Activities preparatory to code generation
2. Activities concerned with code generation for a given target machine.

Group activities are mainly semantic and therefore machine independent in nature. Here in this section we will discuss the code generation expression for intermediate code.

Intermediate Code forms for expressions

Because of some reason, expression compilation is rarely completed in a single pass of compilation. Here we will discuss the code forms of (i) postfix notation which is popularly used in multi pass non-optimizing compilers, (ii) triples and quadruples which are used in optimizing compilers and (iii) expression trees which are good for optimizing the use of machine registers in expression evaluation.

Postfix notation

In the postfix notation operator immediately appears after its last operand. Operators can be evaluated in the order in which they appear in the string. It is a popular intermediate code in non optimizing compilers due to ease of generation and use. This code generation is performed by using stack of operand descriptors. Operand descriptors are pushed on the stack as operand appears in the string and operator descriptors are popped from the stack. A descriptor for the partial result generated by the operator is now pushed on the stack. Consider the following expression that describes the stack operation for code generation in postfix notation.

2 1 5 4 3

Source string is a + b * c + d * e ↑ f (4.1)

The postfix notation of this string is a b c * + d e f ↑ * +

Here the operand stack will contain descriptors for a, b and c when operator '*' is encountered. Then contain the descriptors for the operand 'a' and the partial result of b*c when '+' is encountered. The stack occupation for this code generation is as:

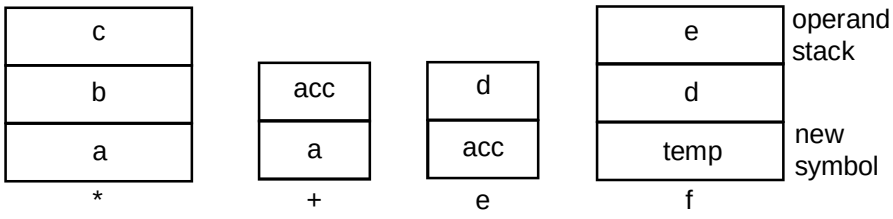


Fig 4.5 : Some steps of code generation for the above postfix string

Triples and quadruples

A *triple* is a representation of an elementary operation in the form of a pseudo-machine instruction which consists of an operator and upto two operands of that operator. The form is:

Operator	Operand 1	Operand 2

For the triple '* a b', the operation represents as a * b. Each operand of a triple is either a variable/constant or the result of some evaluation represented by some other triple. In the later case, the operand field contains that triple's number. A program representation called indirect triples is useful

in optimizing compilers. In this representation, a table is built to contain all distinct triples in the program. A program statement is represented as a list of triple number. This arrangement is useful to detect the occurrences of identical expressions in a program. For efficiency reason, a hash organization can be used for the table of triples.

	operator	*operand1*	*operand2*
1	*	b	c
2	+	(1)	a
3	↑	e	f
4	*	d	(3)
5	+	(2)	(4)

Fig 4.6 : Triples for string 4.1

The symbol (1) in the operand filed of triple 2 indicates that the operand is the value of b * c represented by triple number 1.

During optimization, expressions are often moved around in the program. This requirement cannot handle by indirect triple. For this an extension of triple is used which is called *quadruples*. Each quadruple has the form

Operator	Operand 1	Operand 2	Result name

Result name designated the result of the evaluation. It can be used as the operand of another quadruple. This is more convenient than use a number like triple.

	operator	*operand1*	*operand2*	*result name*
1	*	b	c	t_1
2	+	t_1	a	t_2
3	↑	e	f	t_3
4	*	d	t_3	t_4
5	+	t_2	t_4	t_5

Fig 4.7 : Quadraple

Here t_1, t_2, ……, t_5 are result names, not temporary location for holding partial results. Now, even if a quadruple is moved, it would still be referenced by its result name which remains unchanged. Some of the result names become temporary locations if common subexpressions are detected.

Expression Trees

Expression tree is an abstract syntax tree which depicts the structure of an expression. This representation simplifies the analysis of an expression to determine the best evaluation order. This is used in the expression where bottom up parser cannot lead as efficient code. Consider the instruction sequences (a) and (b):

LOAD	A	LOAD	C
ADD	B	SUB	D
STORE	$TEMP_1$	STORE	$TEMP_1$
LOAD	C	LOAD	A
SUB	D	ADD	B
STORE	$TEMP_2$	DIV	$TEMP_1$
LOAD	$TEMP_1$		
DIV	$TEMP_2$		
(a)		(b)	

Both instruction sequences (a) and (b) can evaluate the same expression (A + B) / (C - D) on a machine having single register. Second instruction sequence is shorter and faster. LOAD-STORE optimization is the process of minimizing the number of load and store instructions by utilizing CPU registers and instruction peculiarities. Instead of generating always in a right-to-left manner, it can be sometimes more efficient to generate code from left-to-right. The guiding rule to identify such possibilities is that “if the register requirements of both subtrees of an operation are identical, first evaluate the right subtree”. This rule reduces the LOAD-STORE requirements because the result of the left subtree can be immediately used in evaluation of the parent operator.

CHECK YOUR PROGRESS

5. Write True or False :
   a) In most of the cases, expression compilation is completed in a single pass of compilation.
   b) Postfix notation is also called polish notation.
   c) In polish notation, operator appears just after the operands.
   d) Quadruple is a representation having one operator and upto two operands of that operator.
   e) Indirect triples representation is useful in optimizing compilers.
6. Fill in the blanks :
   a) Code generation is performed by using _____ of operand descriptors.
   b) _____ has one operator and maximum of two operands.
   c) _____ is good for optimizing the use of machine registers in expression evaluation.
   d) Triple and quadruples are used in optimizing _____.
   e) In _____ form, a statement is represented as a sequence of elementary operations.

4.7 INTERPRETERS

Both compiler and interpreter analyze a source statement to determine its meaning. But interpreter does not translate the source program, it execute the source program without translating them to the machine language. It simply determines its meaning and then carries it out by itself. An interpreter avoids the overheads of compilation which is the main advantage during program execution. But interpretation is expensive in terms of CPU time, because each statement is subjected to the interpretation cycle. Following figure shows the broad schematic of an interpreter which

is much similar to the schematic of execution of a machine language program by a CPU.

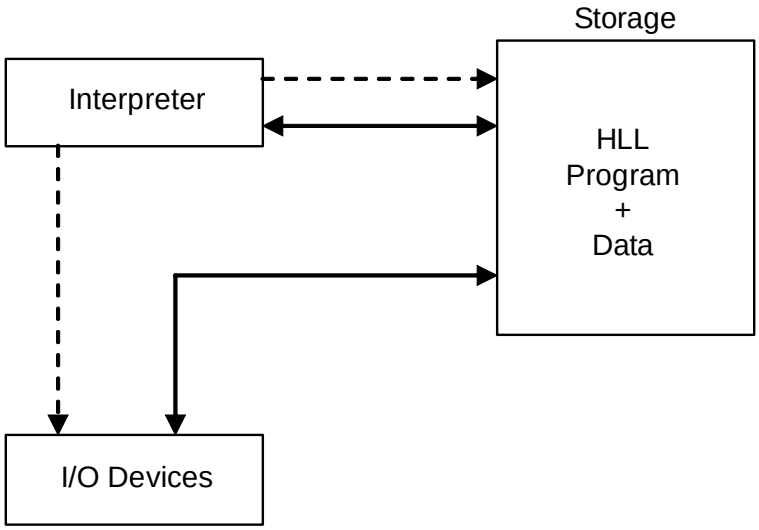


Fig 4.8 : Interpretation of high level program

An interpreter consists of three main components :

1. **Symbol table :** it holds information concerning entities in the source program
2. **Data store :** it contains values of the data item determined in the program being interpreted. It is a set of component which is an array containing elements of a distinct type.
3. **Data manipulation routine:** data manipulation routine contains a set of routine for every legal data manipulation action in the source language.

Use of Interpreter :

Because of its efficiency and simplicity, interpreter is use in many programming languages. Since interpretation does not perform code generation, it is simpler to develop interpreter than compiler. This simplicity makes interpretation more attractive in situations where programs or commands are not executed repeatedly. Hence interpretation is a popular choice for commands to an operating system or an editor. For the same reason, many software packages use interpretation for their user interfaces. Avoiding generation of machine language instructions helps to make the interpreter portable.

Pure and impure interpreter :

In *pure interpreter*, the source program submitted by the programmer is retained in the source form through the interpretation. Here no pre-processing is performed on the source program before interpretation begins. This has the *advantage* that no processing overheads are incurred for a change made in the source program. The *disadvantage* of this interpretation is that a statement has to be reprocessed every time it is visited during interpretation.

An *impure interpreter* performs some preliminary processing of the source program to reduce the analysis overheads during interpretation. Here the processor converts the source program to its intermediate representation (IR) which is used during interpretation. This speeds up interpretation, but the use of intermediate code implies that the entire program has to be preprocessed after any modification.

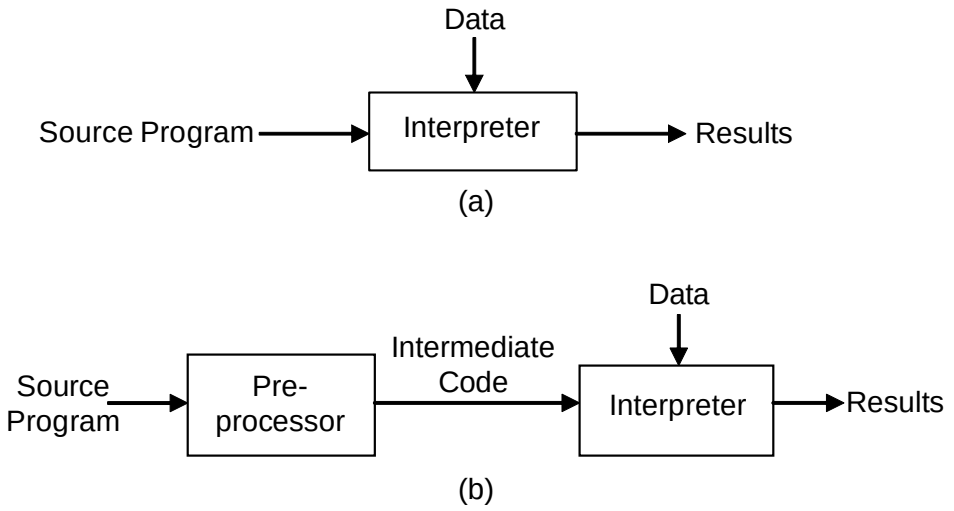


Fig 4.9 : Schematic for Interpreter (a) Pure Interpreter
(b) Impure Interpreter

Intermediate code used in an impure interpreter is so designed as to make interpretation very fast. Postfix notation is a popular intermediate code for interpreters. The intermediate code for a source string A + B * C could look like as follows:

S#17	S#4	S#29	*	+

CHECK YOUR PROGRESS

7. Fill in the blanks :
   a) _____ avoids the overhead of compilers in a program.
   b) _____ and ____ are the main motivating factor of using interpreter in PL.
   c) Data store contains values of _____ declared in the program being interpreted.
8. State True or False :
   a) Interpretation is not expensive in terms of CPU time.
   b) Both compilers and interpreters analyze a source statement to determine its meaning.

4.8 LET US SUM UP

- Compiler bridges the semantic gap between programming language (PL) domain and an execution domain. The features that contribute in this semantic gap are the **data type, data structure, scope rule and control structure** of PL.
- The whole compilation process consists of two phase : **analysis phase** which includes lexical analysis, syntax analysis and semantic analysis and **synthesis phase** which includes intermediate code generation, code optimization and code generation.
- At the time of type conversion operation, compiler generates a **type specific code** to implement an operation and manipulate the values of the variable through that code.
- When a compiler compiles the data structure used by the PL, it develops memory mapping to access the memory word allocated to the element.
- Memory allocation and memory bindings are important during semantic analysis of a program. A compiler allocates memory for their instruction and data separately. Memory binding perform by compilers in two ways: one is *before* execution of a program called ***'static***

binding' and *during* the execution of a program called ***'dynamic binding'***. FORTRAN use Static allocation dynamic allocation is used by block-structured languages like PL, ALGOL, PASCAL etc.

- Symbol table is required by a compiler at the time of allocating memory dynamically to perform their task. Each entry in the symbol table contains a variable's name, type, length and displacement in the AR.
- Compilers use stack for allocating memory which have some limitations.
- Though both compilers and interpreters analyze source program to execute, compilers translate it to its machine code form, but interpreter execute without translating them. Interpreter determines its meaning where source program will be same after interpreting.

4.9 ANSWERS TO CHECK YOUR PROGRESS

1. a) T b) F c) T d) F e) T
2. a) bridge b) data types c) memory mapping
   d) heterogeneous e) complex f) control structure
3. a) memory binding b) before
   c) program controlled dynamic allocation d) block
4. a) F b) T c) F d) T
   e) T f) F
5. a) F b) T c) T d) F e) T
6. a) stack b) Triple c) expression trees
   d) compilers e) Intermediate code
7. a) interpreter b) efficiency, simplicity c) data item.
8. a) F b) T

4.10 FURTHER READINGS

- System Programming and Operating Systems, D. M. Dhamdhere, Tata Mc-Graw Hill. (Second Edition)
- Operating Systems and System Programming, P. Balakrishna Prasad, Scitech. (Second Edition)

4.11 PROBABLE QUESTIONS

1. Define compiler. Explain the different phases of a compiler.
2. What is memory allocation? Discuss static and dynamic memory allocation during compilation of a program.
3. Why symbol table is required?
4. How intermediate code for expression is handled by compiler?.
5. Why code optimization is used? How it is achieved?
6. What is the function of interpreter? Why it is used?
7. What is the difference between compiler and interpreter?
8. Discuss pure and impure interpreter.

UNIT 5 : SCANNING AND PARSING

UNIT STRUCTURE

5.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- learn the basics of lexical analysis
- define the meaning of finite state automata
- design non-deterministic and deterministic finite state automata
- learn how to define a regular expression
- learn about parsing.

5.2 INTRODUCTION TO SCANNING

The scanning or lexical analysis is the first phase of designing a compiler, to identify a set of valid tokens/words of a language that occur in the input string. Words or tokens are sequence of characters that represents a unit of information in the source program. A program or function, which performs lexical analysis, is called a lexical analyzer, lexer or scanner. Thus scanner performs the pattern matching task. Since we are mostly concerned with the programming languages, the possible set of words in such language is finite such as keywords, name of constant and variable, subprograms etc. To design a recognizer or scanner for these types of languages, we can use finite state machine to determine whether a given word belongs to the language.

Lexical analyser acts as an interface between the input source program to be compiled and the later stages of the compiler. During lexical analysis, the input sequences of characters are read until a token is encountered based on request from syntax analyser. The parser looks into the sequence of tokens and identifies the language construct occurring in the input program. Apart from that, the lexical analyser also participates in creation and maintenance of symbol table. For example, consider the valid C statement,

int a = 2 ;

Lexeme	**Token type**
Int	Keyword
a	Identifier
=	Operator
2	Integer constant
;	Statement terminator symbol

So lexical analysis phase tokenize the input statement *int a = 2 ;* like the tabulated token type and store it in the symbol table.

CHECK YOUR PROGRESS

1. State true or false:
   i) A token is a sequence of related words.
   ii) Lexical analyser is the first phase of designing a complier.
   iii) A finite state machine is used to recognize a token in lexical analyser.
   iv) Lexical anlyser acts as an interface between scanner and parser.

5.2 FINITE STATE AUTOMATON

Modern computers are capable of performing a wide variety of computations. An *abstract machine* reads in an input string and, depending on the input, outputs true (*accept*), outputs false (*reject*), or gets stuck in an infinite loop and outputs nothing. We say that a machine *recognizes* a particular language, if it outputs *true* for any input string in the language and *false* otherwise. The artificial restriction to such *decision problems* is purely for notational convenience. Virtually all computational problems can be recast as language recognition problems. For example, to determine whether an integer 97 is prime, we can ask whether 97 is in the language consisting of all primes {2, 3, 5, 7, 13, ... } or *Is a LINUX OS is more powerful than a Windows OS? Can Java do more things than C++?* To accomplish this, we define a notion of *power*. We say that machine A is at least as *powerful* as machine B if machine A can be “programmed’” to recognize all of the languages that B can. Machine A is more powerful than B, if in addition, it can be programmed to recognize at least one additional language. Two machines are *equivalent* if they can be programmed to recognize precisely the same set of languages. Using this definition of power, we will classify several fundamental machines. Naturally, we are interested in designing the most powerful computer, i.e., the one that can solve the widest range of language recognition problems. Note that our notion of power does not say anything about how fast a computation can be done. Instead, it reflects a more fundamental notion

of whether or not it is even possible to perform some computation in a finite number of steps.

A ***finite state automaton*** (***FSA***) is a mathematical model of the machine described above, consists of a finite set states and a set of transitions from state to state that occur on input symbols chosen from an alphabet ". For each input symbol, there is exactly one transition out of each state. One state usually denoted q_0, is the initial state, in which automaton starts. As the input symbols are read in one at a time, the FSA changes from one state to another in a pre-specified way. The new state is completely determined by the current state and the symbols just read in. There is always an initial state and one or more than one final or accepting state. Thus state and transition is the heart of a FSA. In compilation state may be a character for lexical analysis phase or a words or token in syntax and semantic analysis phase. The token obtaining during lexical analysis is recognize using FSA.

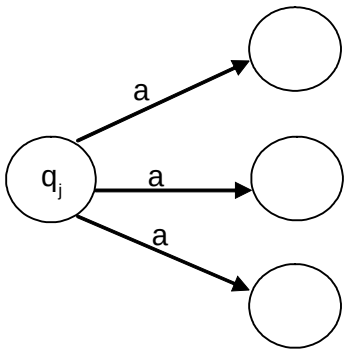
There are two types of finite automata :

- Non-deterministic finite automata (NFA)
- Deterministic finite automata (DFA).

An input string is said to be accepted by the automata, if there exist at least one path from the start state of the machine to its final states whose transitions are governed by the input string.

5.3.1 Non-Deterministic Finite Automata (NFA)

A *non-deterministic finite automaton* (NFA) is a state machine that reads an input string and decide whether to accept it. NFA allows transitions from state q_i on symbol *a* to many states (or none at all).



Formally, a non-deterministic finite automaton (NFA) is a 5-tuple *(Q, Σ, δ, q_0, F)* where

Q is a finite set of **states**,

Σ is a finite **input alphabet**,

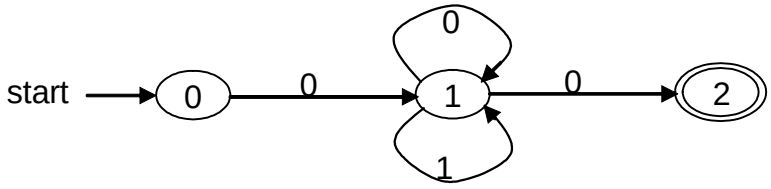
δ : Q × Σ → P(Q) is the **transition function**

q_0 ε Q is the **start state** and

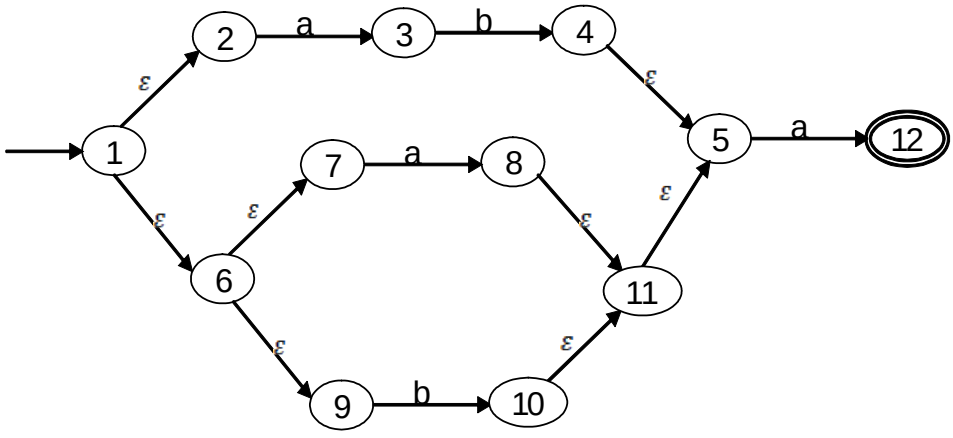
F ⊆ Q is the set of **final** or **accepting states**

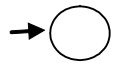
The basic feature of non-deterministic finite automata is its ability to have more than one possible transition from a state on the same input symbol. For example, from state q_i there may be transition to two distinct states q_j and q_k label by the same input symbol. The actual transition occurring is to be chosen non-deterministically by the machine. Another non-determinism lies with the transitions labeled by **ε**, known as **ε** -transiiton. If there is a -transiiton form state q_i to q_j, the machine can do such a transition without consuming any input.

Example : Draw a NFA to represent the regular expression 0(0|1)*0

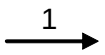


Example : Draw a NFA to represent the regular expression ab|(a|b)a



In the above two examples,  implies starting state,

 implies a state,

 transition on input 1,

 accepting state.

5.3.2 Deterministic Finite Automata (DFA)

A *deterministic finite automaton* (DFA) represents a finite state machine that recognizes a *regular expression*. The concept of regular expression will be discussed in the next section.

The next state of a DFA is uniquely determined by the current state and the input symbol. As compared to NFA, the transitions are deterministic in DFA. Thus, there can not be more than one state labeled by same input. Characteristics of a DFA are enumerated bellow:

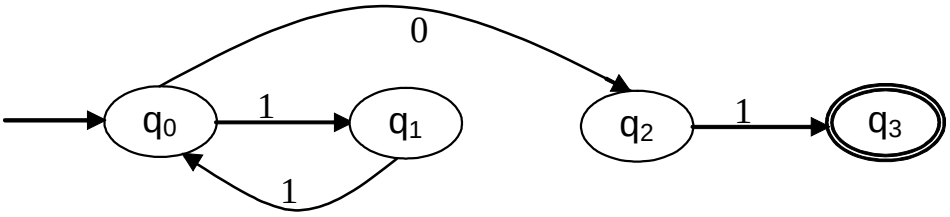
1. A DFA has no ε transition.
2. A DFA can not have more than one transition out of a state with same label.
3. There is exactly one accepting state.

Example : Formal notation of this FSA is denoted by M = {Q, Σ, δ, q_0, F}, where Q = {q_0 q_1 q_2 q_3 ……. q_n}, Σ = {0,1}, F = {q_0} and δ =

States	Input	
	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

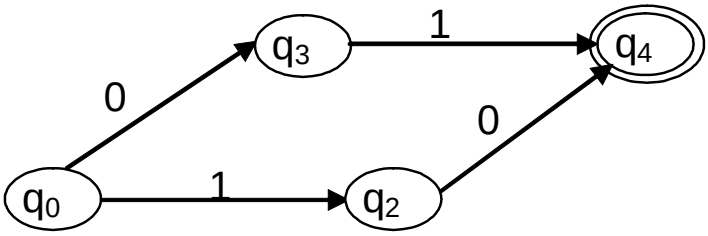
Example : Draw the DFA for input string1101.

Solution :



Example : Draw a DFA for regular expression 01|10.

Solution :



5.4 REGULAR EXPRESSION

In this section we introduce the regular expression, the standard notation for characterizing text sequences. The regular expression is used for specifying test strings in all sorts of text processing information extraction applications.

In arithmetic, we can use the operation + and × to build up expression such as

(a + b) × c ----------(1)

We can calculate the value of equation (1) for a = 5, b = 2 and c = 3 is 21. Similarly, we can use regular operations to build up expressions describing languages, which are called regular expressions. For example-

(0 U 1)0* ----------(2)

Here the value is the language consisting of all string starting with a 0 or a 1 followed by any number of 0s. Let us analyse expression (2). The symbol 0 and 1 are shorthand for the set {0} and {1}. So (0 U 1) means ({0} U {1}). The value of ({0} U {1}) is {0,1}. The part of 0* means {0}*, and its value is the language consisting of all strings containing any numbers of 0s. A language denoted by a regular expression is said to be a *regular set*. If two regular expressions *r* and *s* denote the same language, we say that

r and *s* are equivalent. Regular expressions have an important role in computer science applications, like search important pattern form a text document.

Formally, a regular expression is an algebraic notation for characterizing a set of strings. Thus, they can specify search strings as well as define a language in a formal way. *R* is said to be a regular expression, if

- *a* for some a in the alphabet Σ,
- ε the language containing single string i.e., empty string
- Φ the language that does not contain any string
- (R_1 U R_2), where R_1 and R_2 are regular expressions
- (R_1 o R_2), where R_1 and R_2 are regular expressions
- (R_1*), where R_1 is a regular expressions

Example : Let Σ = {a,b}.

- The regular expression (a|b) denotes the set {a,b}
- The regular expression (a|b) (a|b) denotes {aa, ab, ba, bb}, the set of all strings of a's and b's.
- The regular expression (a*) denotes {ε, a, aa, aaa,..... }

CHECK YOUR PROGRESS

2. Fill up the blanks
   i) _____ has multiple transitions on a start state for the same input symbol.
   ii) A language denoted by regular language is called _____.
   iii) Regular expression r and s is said to be equivalent if r and s denote same _____.

5.5 PARSING

Syntactic analysis is the second phase of compilation process. While the first phase (that is lexical analyser) reads an input source program and

produces a sequence of token, the syntactic analyser verifies whether the tokens are properly sequence in accordance with the grammar of the language. The process of analysis the syntax of the language is done by a module in the compiler is called parser and the analyzing or verifying process whether an input string matches the grammar of the language is called parsing. Thus the input to the parsing system is the output of lexical analyser or scanner and output is a form a tree known as parse tree.

5.5.1 Role of Parser

The parser looks into the sequence of tokens returned by the lexical analyser and extract the constructs of the language appearing in the sequence. Thus, the role of parser is divided into two categories–

1. To identify the language constructs presents in a given input program. If the parser determines the input to be a valid one, it outputs the representation of the input in the form of a parse tree.
2. If the input is grammatically incorrect, parser tries to recover the error, if possible. If not, parser outputs syntax error message to the user.

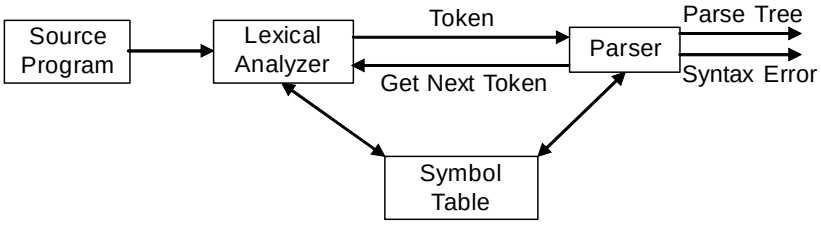


Fig. 5.1 : Role of Parser

Parser can be categorized into three broad categories :

1. Universal parsing, that perform parsing using Cocke-Kasami-Younger(CKY) or Earley's algorithm as they can parse any grammar.
2. Top-down parsing, build parse tree starting from the root node and work up to the leaves.
3. Bottom-up parsing, build parse tree starting from the leaves and work up to the root node.

5.5.2 Parse Tree

A parse tree is a pictorial presentation for a derivation or more simply a graphical representation of input or processed statement. A derivation is the sequence of intermediately strings generated to expand the start symbol of the grammar to a desired string of terminals.

Example : Let us consider the following grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid id$$

The sentence id + id * id has two distinct leftmost derivations–

$E \Rightarrow E * E$
$\Rightarrow E + E * E$
$\Rightarrow id + E * E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$

$E \Rightarrow E + E$
$\Rightarrow id + E$
$\Rightarrow id + E * E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$

For these two derivations, we can draw the following parse tree–

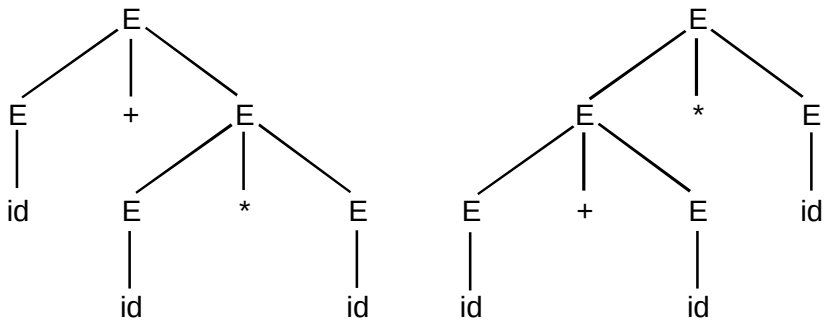


Fig. 5.2 : Two parse tree for id + id * id

LET US KNOW

Leftmost derivation : The derivation where the leftmost nonterminal is replaced at each step is called leftmost derivation.

Rightmost derivation : the derivation in which the rightmost nonterminal is replaced at each step is called rightmost derivation.

Ambiguous grammar : A grammar is said to be ambiguous grammar if there exist more than one parse tree for the same input sentence. That is an ambiguous grammar can have more than one leftmost and rightmost derivation.

Left recursion : A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \rightarrow A\alpha$ for some string α.

5.6 TOP DOWN PARSING

A top-down parsing algorithm parses input string of tokens by tracing out the steps in a leftmost derivation. It is termed as top-down as the parse tree is traversing in a preordered way from root node towards the leaf nodes. Top-down parser comes in two forms. They are–

1. Backtracking parser
2. Non-backtracking parsers

Backtracking parser includes recursive descent parser, whereas non-backtracking parser includes predictive parsers like table driven parsers or LL(k) parsers. Backtracking parsers tries different possibilities for parsing an input string, by backing up an arbitrary amount in the input if any possibility fails. A predictive parser attempts to predict the next construction in the input string using one or more lookahead tokens. Backtracking parsers are more powerful but slower than predictive parsers, as they require exponential time to parse. They are also suitable for handwritten parsers. LL(k) parser gets its name, as it process the input string as left-to-right and leftmost derivations. The number “k” in the parenthesis means that it uses *k* symbol lookahead for parsing.

5.6.1 Recursive Descent Parsing

This parser consists of a set of mutually recursive routines that may require backtracking to create the parse tree. Thus, it may require repeated scanning of the input.

Example : Consider the grammar

$S \rightarrow abA$

$A \rightarrow cd \mid c \mid \varepsilon$

For the input string ab, the recursive descent parser starts by constructing a parse tree representing S → abA as shown in figure 5.3(a). In figure 5.3(b) the tree expand with the production A → cd. Since it does not match the string ab the parser backtracks and then tries the alternatives A → c as shown in figure 5.3(c). Here also the parse tree does not match the string ab. So the parser backtracks and tries the alternative A → ε. This time, it finds a match [figure 5.3(d)]. Thus, the parsing is complete and successful.

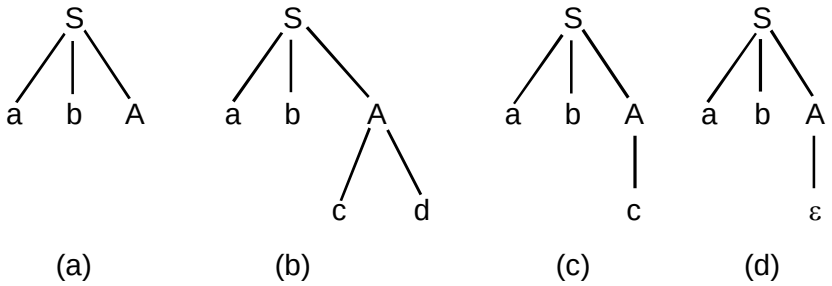


Fig. 5.3 : Example of parse tree construction

The main pitfall of recursive parsing are :

1. Left recursion, where the grammar has production of the form A → Aε, that leads an infinite loop
2. Backtracking, when there is one alternative in the production to be tried while parsing the input string.

In recursive descent parsing, elimination of left recursion and left factoring is used to prevent the above mentioned drawbacks.

5.6.2 Predictive Parsing

Recursive descent parsing can be performed using a stack of activation record without backtracking. Those types of parsers are known as predictive parsers. Thus predictive parsers are special kind of recursive descent parser. To construct a predictive parser we must know the proper alternatives by looking the first symbol it derives. A predictive parser based on transition diagram attempts to match terminal symbol against the input, and make a potentially recursive

procedure call whenever it has to follow an edge labeled by a non-terminal. A non-recursive predictive parser can be obtained by stacking the states s, when there is a transition on a non-terminal out of s, and poping the stack, when the final stack for a non-terminal is reached. Thus non-recursive predictive parser maintains a stack of activation records explicitly rather than maintaining a stack implicitly via recursive call.

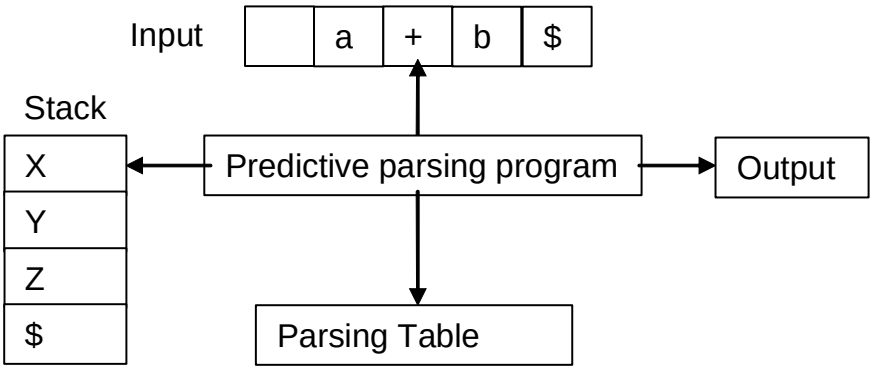


Fig. 5.4 : Model of non-recursive predictive parser

Non-recursive predictive or table driven predictive parser has an input buffer, a stack, and a parsing table. The input buffer contains the string to be parsed, followed by $, to indicate the end of input string. The stack contains a sequence of grammar symbols with $ at the end of the stack, indicating the bottom of the stack. Initially the stack contains the start symbol of the grammar on the top of $. The parsing table is constructed using FIRST and FOLLOW function. The steps to construct the predictive parse table for a grammar G are enumerated bellow-

1. Eliminate left recursion in grammar G.
2. Perform left factoring in grammar G.
3. Find FIRST and FOLOW in grammar G.
4. Construct the predictive parse table.
5. Check if the parser can accept the given input string.

We leave the working strategy of FIRST and FOLLOW function to the advanced compiler design course.

CHECK YOUR PROGRESS

3. Multiple Choice Questions:

i) A predictive parser has a procedure for every _____.

a) Terminal b) Production

c) non-terminals d) grammar

ii) Main drawbacks of recursive parsers are _____, _____.

a) Handle and left recursion

b) Left recursion and left factoring

c) Backtracking and left factoring

d) Left recursion and Backtracking

iii) _____ is a special kind of recursive descent parser that needs no backtracking.

a) LR b) LL c) Predictive d) Top-down

iv) Non-recursive predictive parser uses _____ explicitly.

a) Stack b) handle c) Operator d) None of these

5.7 BOTTOM UP PARSING

In bottom-up parsing, the parse tree for the input string is constructed beginning at the bottom nodes (leaves) and working up towards the root, the reverse process of top-down parsing approach. Thus in bottom-up parsing instead of expanding the successive non-terminals according to production rules, a current string or right sentential form is collapsed each time until the start non-terminals is reached to predict the legal next symbol. This can be regarded as a series of reductions. This approach is also known as shift-reduce parsing. This is the primary method for many compilers, mainly because of its speed and the tools, which automatically generate a parser, based on the grammar. For example, consider the following set of productions:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

Bottom-up parsing of the input sentence " id + id * id " is done as follows:

Right sentential form	Substring	Reducing production
id + id * id	Id	$E \rightarrow id$
E + id * id	Id	$E \rightarrow id$
E + E * id	E + E	$E \rightarrow E + E$
E * id	Id	$E \rightarrow id$
E * E	E * E	$E \rightarrow E * E$
E		Start symbol. Hence the input is accepted

Parse tree representation for the above input is shown in fig. 5.5:

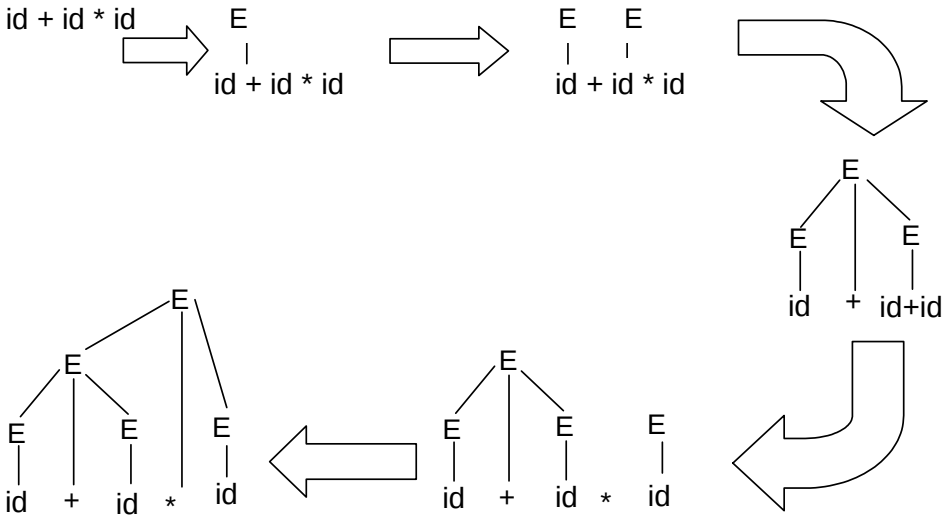


Fig. 5.5 : Construction of bottom-up parsing

5.8 OPERATOR PRECEDENCE GRAMMARS

A grammar is said to be operator grammar if there does not exist any production rule with right hand side. Thus operator grammar has two characteristics :

1. There is no ε production in this grammar
2. No production would have two adjacent non-terminals.

An example of Operator grammar is E → E + E | E – E | id | (E)

Operator precedence parsing, a shift-reduce parsing method that can be applied to the operator grammars. This type of parsing face two main challenges-

1. Identification of the correct handle in the reduction steps, such that the given input can be eventually reduced to the start symbol
2. Identification of which production to use for reducing in the reduction steps, such hat we can correctly reduce the given input to the start symbol.

The operator precedence parser uses operator precedence relationship table (Fig.5.6) for making decisions to identify the correct handle in the reduction step.

	+	-	*	/	Id	(	)	$
+	·>	·>	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	·>	·>
Id	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	≐	
)	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·		

Fig. 5.6 : Operator precedence relationship table

The precedence relationship between the operators a and b are enumerated bellow–

1. The relation *a* <· *b* implies that the terminal *a* has lower precedence than *b.*
2. The relation a ·> b implies that terminal *a* has higher precedence than *b*.
3. The relation $a \doteq b$ implies that the terminal a has same precedence as b.

To locate handle the following steps are followed in operator precedence parsing :

1. Scan for *right-to-left* until the right most ·> is encountered.
2. Scan towards left over all equal precedence until the first <· precedence is encountered.
3. Everything between <· and ·> is a handle.
4. Once the handle is identified, reduce it.

CHECK YOUR PROGRESS

4. Fill up the blank
   i) Operator precedence parser is a type of _____.
   ii) A relation *a* <· *b* implies that the terminal *a* has _____ than *b.*
   iii) Everything between <· and ·> is a _____, in operator precedence parser.

5.9 LR PARSING

LR parsing is one of the best methods for syntactic recognition of programming language. An LR (*Left-to-Right* and *Rightmost* derivation) parser uses the *shift-reduce* technique. In general when we talk about LR parsing we mean LR(1) parsing, that is LR parsing with one symbol lookahead. (*We have LR(k) parsing with k lookahead*). Figure 5 pictorially represent the structure of LR parser. The following are the enumerated advantage of LR parser.

1. An LR parser can recognize virtually all programming language constructs written in context free grammars.
2. It can detect the syntax error quickly.
3. It can be implemented in a very efficient manner.
4. It is the most general non-backtracking technique.

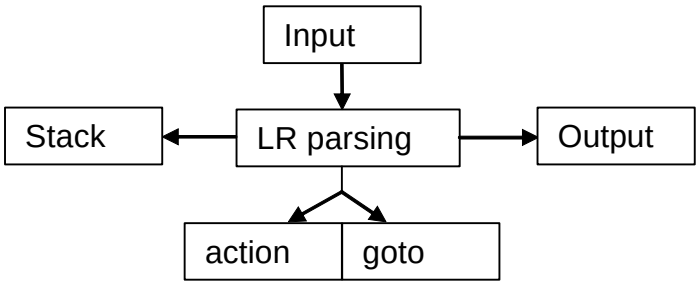


Fig. 5.7 : Model of an LR parser

Basic LR parsing model consists of two parts – *action* and *goto*. The algorithm used is much similar as shift-reduce parsing method. The stack for an LR parser consists of grammar symbol and states. The states summarize what is bellow that state on the stack and pairing the state on the top of the stack with the next input symbol indices into the parsing table to determine the next action. The *action* portion consist of

1. Shift action – pushes the next input symbol and next state onto the stack.
2. Reduce action – matches the *handle* to some production $A \rightarrow \beta$. It then pops 2 * | β | elements off the stack and pushes A and a states.
3. Accept – means the parsing action is finished.
4. Error – causes the parser to call an error handling routine.

LET US KNOW

Handle : A handle of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in the rightmost derivation of γ. That is if $S \Rightarrow \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta w$ then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta w$.

Augmented grammar : If *G* is a grammar with start symbol *S*, the augmented grammar for *G*, is *G'* with a new start symbol *S'* and production $S' \rightarrow S$.

Algorithm 5.1 : LR parsing

```
Input : LR parsing table and string w
Output :
Begin
    Push initial state
    Repeat
    Let  s = top of the stack
         a = next input symbol
    if action[s,a] = shift s2  then
            push a and s2
            advance input pointer to the next input symbol
    else if action[s,a] = reduce A'!β then
            pop 2 * | β | symbols
            s'  = top of the stack
            push A  and goto[s', A]
    else if action[s,a] = accept then
            return true
    else
            return error
End
```

There are three different methods to perform LR parsing

1. SLR (Simple LR) – easy to implement but less powerful.
2. CLR (Canonical LR) – most general and powerful but tedious to implement.
3. LALR (Lookahead LR) – a hybrid method consisting SLR and CLR.

5.10 LET US SUM UP

In this unit, we have learnt to design a finite state machine, non-deterministically and deterministically. After that we have learn about regular expression. We have also looked at the various parsing approach and basic working strategy of the parsers.

5.11 ANSWERS TO CHECK YOUR PROGRESS

1. i) False ii) True iii) True iv) False
2. i) NFA ii) Regular set iii) language
3. i) b ii) d iii) c iv) a
4. i) Bottom-up parser ii) Lower precedence iii) handle

5.12 FURTHER READINGS

- Aho, Sethi and Ullman, *Compilers – Principles, Techniques, and Tools*.

5.13 MODEL QUESTIONS

1. What is scanning and parsing? Differentiate between them.
2. Draw a DFA for bit strings with at least one 0 and at least one 1.
3. Define NFA. Draw a NFA that matches all strings that contain either a multiple of 3 1's or a multiple of 5 1's. *Hint*: Use 3 + 5 + 1 = 9 states and one epsilon transition.
4. Draw a NFA that recognize the language of all strings that end in aaab.
5. What do you mean by parsing? Define parse tree. Note down the role of parser in compiler design.
6. Differentiate between top-down parsing and bottom-up parsing.
7. Define LR grammar. Enumerate different types of LR grammar.
8. Explain no-recursive predictive parsing.