Uttar Pradesh Rajarshi Tandon Open University

BCA-1.18

Bachelor in Computer Applications

Uttar Pradesh Rajarshi Tandon
Open University

BCA-1.18

Bachelor in Computer Applications

Block

1

System Software

Introduction to System Software and software tools

Printed and Published by Dr. Arun Kumar Gupta, Registrar, **Uttar Pradesh Rajarshi Tandon Open University, 2019.**

Printed By : Chandrakal Universal Pvt Ltd. 42/7 Jawahar Lal Neharu Road, Prayagraj, 211002

SYSTEM SOFTWARE

Course Introduction

The objective of this course is to have an understanding of foundations of design of assemblers, loaders, linkers, and macro processors. You should be able to understand the relationship between system software and machine architecture, know the design and implementation of assemblers, know the design and implementation of linkers and loaders, and have an understanding of macro-processors and system software tools.

The course is organized into following blocks:

UNIT - 1 : LANGUAGE PROCESSORS

Structure

1.0 INTRODUCTION

In this unit, the focus is to understand the system software that supports the operation of a computer. You will be introduced with insights and the concepts of language processors and the language processor development tools.

1.1 Objectives

- To understand the relationship between system software and machine architecture.
- To know the language processors
- To have an understanding of system software tools.

1.2 Language Processing Activities

Programmers found it difficult to write or read programs in machine language. In a quest for a convenient language, they began to use a mnemonic (symbol) for each machine instructions which would subsequently be translated into machine language.

Such a mnemonic language is called Assembly language.

Programs known as Assemblers are written to automate the translation of assembly language into machine language.

Assembler: Translates Assembly language program into Machine language program

Fundamental functions:

1. Translating mnemonic operation codes to their machine language equivalents.

2. Assigning machine addresses to symbols used by the programmers.

HIGH-LEVEL LANGUAGE TRANSLATORS

It is the program that takes an input program in one language and produces an output in another language. It bridges the semantic gap between a programming language domain and the execution domain.

A compiler is a language program that translates programs written in any high-level language into its equivalent machine language program.

Two aspects of compilation are:

I.Generate code to increment meaning of a source program in the execution domain.

II.Provide diagnostics for violation of programming language, semantics in a source program. The program instructions are taken as a whole.

1.3 Fundamentals of Language Processing & Language Specification

Compilers translate high-level language programs (ie programs written in C, C++) into underlying machine-language to enable it to run on hardware.

Assemblers translate assembly language programs into machine language.

Language Interpreters

It is a translator program that translates a statement of high-level language to machine language and executes it immediately. The program instructions are taken line by line. The interpreter reads the source program and stores it in memory.

During interpretation, it takes a source statement, determines its meaning and performs actions which increments it. This includes computational and I/O actions.

Program counter (PC) indicates which statement of the source program is to be interpreted next. This statement would be subjected to the interpretation cycle.

The interpretation cycle consists of the following steps:

I. Fetch the statement.

II. Analyze the statement and determine its meaning.

III. Execute the meaning of the statement.

The source program is retained in the source form itself, no target program exists. A statement is analyzed during the interpretation.

Loaders and Linkers- The loader is system software which brings the object program (i.e. translated by compiler or interpreter or assembler) into memory for execution. Linker is system software which performs linking of modules (including libraries) and program relocation.

The study of system software deals with principles of the design of language processors-compilers, assemblers etc.

The distinction between system software and application software is aspect of machine dependency. The system software specializes to deal with machine dependent features whereas application software concerns dealing with running of some application.

For example, assemblers translate mnemonic instructions into machine code. The instruction formats, addressing modes which are machine dependent features are of direct concern in assembler design.

There are some aspects of system software that do not directly depend upon the type of computing system being supported. These are known as machine-independent features deals with the general design and logic of an assembler is basically the same on most computers.

Check Your Progress :

What is difference between language Compiler and Interpreter?

1.4 Language Processor Development Tools

In this section, you will be introduced to software tools to make a language processor easy to use. Another aim of tools is to make it possible for the resources of the system to be used efficiently.

Compilers	*Assemblers*	*Macroprocessor*	
Loaders	*Text Editors*	*Searching/Sorting*	*Debugger*

Text Editor is software that can compose a string, or modify an existing one. Two common types of editors:

i. line editors: lines of text are numbered automatically, and modified by giving the number of the line as part of the editing command
ii. Screen editors: text is displayed on the screen, and the cursor can be moved to where the changes are going to be made.

Macro Instructions : A macro instruction (macro) is simply a notational convenience for the programmer to write a shorthand version of a program. It represents a commonly used group of statements in the source program. It is replaced by the macro processor with the corresponding group of source language statements. This operation is called “expanding the macro”

Macro Processors: Its functions essentially involve the substitution of one group of characters or lines for another. Macro processors are used in

i. assembly language
ii. high-level programming languages, e.g., C or C++
iii. OS command languages
iv. general purpose

Debuggers: Debugger is a software tool used to locate the errors in source programs. User interfaces of debuggers may use: a command language, or windowing system and menus.

Functions of a debugger :

i. Execution sequencing: observation and control of the flow of execution.

ii. Tracing and traceback: Tracing is used to track the flow of execution and data modifications Traceback can show the path by which the current statement was reached. It can also show which statements have modified a given variable or parameter.

1.5 Summary

In this unit we presented Language Translators and the Tools for Language development. The later part of course you will study its design and understands its machine dependency aspects for its categorization as system software.

1.6 Terminal Questions

1. Define system software.
2. What is a Language Translator?
3. What do you understand by Machine Dependency of System Software?
4. What are Language Development tools?

UNIT - 2 : DATA STRUCTURES FOR LANGUAGE PROCESSING

Structure

2.0 INTRODUCTION

The search operation is frequently used by a Language Processor. This makes the design of data structures a crucial issue in language processing activities.

2.1 OBJECTIVES

In this unit we shall discuss the data structure requirements of Language Processor and suggest efficient data structure to meet there requirements.

2.2 SEARCH DATA STRUCTURES

Program components in languages such as C are normally compiled into object files, which are combined into an executable file by linkage editor or linking loader. The linkage editor adjusts addresses as needed when it combines the object modules, and it also puts in the addresses where a module references a location in another module (eg. Function call)

Criteria for Classification of Data Structure of Language Processor:

1. Nature of Data Structure: whether a “linear” or “nonlinear”.

   Example-linear=array, stack etc-Non-linear=Tree, Graph etc

RECALL : An array is a collection of similar data types, stored into a common variable. The collection forms a data structure where objects are stored **linearly**, one after another in memory.

2. Purpose of Data Structure(DS) : whether a “search” DS or an “allocation” DS.

   Example - Search = Binary Search tree, etcallocation = stacks, heaps etc

RECALL : Binary search trees (BST) are ordered or sorted binary trees data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key.

Stack is used for static memory allocation and Heap for dynamic memory allocation, both stored in the computer's RAM.

3. Life time of a data structure: whether used during language processing or during target program execution.

Example- Lang Proc = Object based data model Target Prog = Hash tables

Recall: Object based Data Models are based on concept of using the entities in the real world. At this stage, applications are not bothered about what data value is stored, what is the size of each data etc.

Hash table is a data structure used to implement an associative array, a structure that can map keys to values. Hash tables are more efficient than any other table lookup structure. For this reason, they are widely used during program execution time.

A search data structure (or search structure) is a set of entries accommodating the information concerning one entity. Each entity is assumed to contain a key field which forms the basis for search.

Linear Data Structure

Linear data structure consists of a linear arrangement of elements in the memory.

Advantage : Facilitates Efficient Search.

Dis-Advantage: Require a contagious area of memory.

The problem with contagious memory area allocation is related to fact that the size of a data structure is difficult to predict. So designer is forced to overestimate the memory requirements of a linear Data Structure to ensure that it does not outgrow the allocated memory. Disadvantage: Wastage Of Memory.

Non Linear Data Structures

Overcomes the disadvantage of Linear Data Structures using elements of Non Linear DS are accessed using pointers. Hence the elements need not occupy contiguous areas of memory.

Disadvantage : Non Linear DS leads to lower search efficiency.

Search Data Structures

Search Data Structures are used during Language Processing to maintain attribute information concerning different entities in source program. An entity is created only once but may be searched for large number of times. Search efficiency is therefore very important.

Allocation Data Structures

The address of memory area allocated to an entity is known to the user(s) of that entity. Means, no search operations are conducted on them. So the important criteria for allocation data structures:-

– Speed of allocation and deallocation

– Efficiency of memory utilization

Use of Search and Allocation DS: Language Processors uses both search Data Structures and allocation Data Structures during its operation.

- Use of Search DS : To constitute various tables of information.
- Use of Allocation DS : To handle programs with nested structures of some kind.
- Target program rarely uses search DS.

Example Consider Following Pascal Program :

```
Program Sample (input, output);

varx,y                  : real;
i                       : integer;
Procedure calc(vara, b : real);
varsum                  : real;
begin
sum                     : = a+b;
---
endcalc;
begin {Main Program}
----
end.
```

The definition of procedure 'calc' is nested inside the main program. Symbol tables need to be created for the main program as well as for procedure 'calc'. We call them Symtabsample and Symtabcalc. The Data Structures these symbol tables are Search Data Structures. During compilation, the attributes of a symbol are obtained by searching appropriate symbol table. Now, memory needs to be allocated to Symtabsample and Symtabcalc using an Allocation Data Structure. The addresses of these tables are noted in a suitable manner. Hence no searches are involved in locating Symtabsample and Symtabcalc.

Check your progress

The symbols used in calc-procedure a,b above need to be stored in a table. Why?

Example : Consider Following Pascal and C segments:

```
Pascal: varp : integer;
begin
new (p);
```

```
C: float *ptr;
ptr = (float*)calloc(5,sizeof(float));
```

The Pascal call new(p): allocates sufficient memory to hold an integer value and puts the address of this

memory area in p. The C statement ptr=… : allocates a memory area sufficient to hold 5 float values and puts its address in ptr.

Means, access to these memory areas are implemented through pointers. i.e p and ptr. Conclusion: No search is involved in accessing the allocated memory.

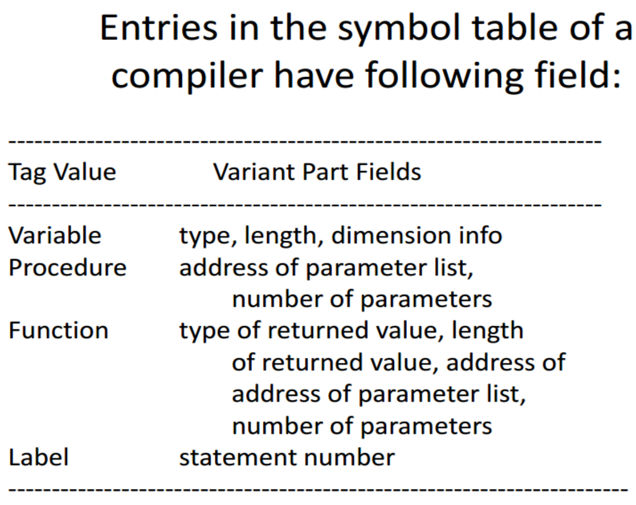
SEARCH DATA STRUCTURES

Search Data Structures	Topic List
Entry Formats	Sequential Search Org
Fixed and variable length entries	Binary Search Org
Hybrid entry formats	Hash table Org
Operations on search structures	Hashing functions
Generic Search	Collision handling methods
Procedures	Linked list
Table organizations	Tree Structured Org

- In ‘Search’ the basic requirement is the ‘Key’. Key is the symbol field containing name of an entity.

- Search Data Structure (also called search structure) is a set of entries, each entry accommodating the information concerning one entity. Each entry in search structure is a set of fields i.e. a record, a row. Each entry is divided into two parts:–Fixed Part–Variant Part

- The value in fixed (tag) part determines the information to be stored in the variant part of the entry.



Entries in the symbol table of a compiler have following field:

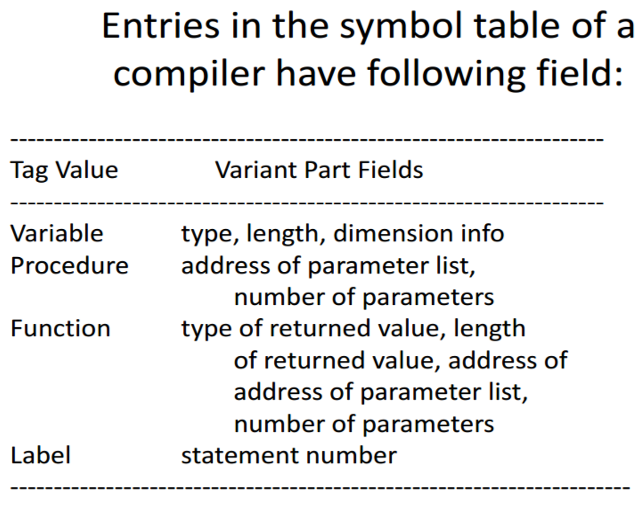
Tag Value	Variant Part Fields
Variable	type, length, dimension info
Procedure	address of parameter list, number of parameters
Function	type of returned value, length of returned value, address of address of parameter list, number of parameters
Label	statement number

Fixed Length Entry : Each entry has same type and size DS such as array.

Examples - 1. Symbol 2. Class 3. Type 4. Length 5. Dimension Information 6. Parameter List 7. No. of Parameters 8. Type of returned value 9. Length of returned value 10. Statement number.

Variable Length Entry : Type and size of each record could be different. Example- 1. Name 2. Class 3. Statement Number

When class = label, all fields excepting name, class and statement number are redundant. Here, Search method may require knowledge of length of entry.



Entries in the symbol table of a compiler have following field:

Tag Value	Variant Part Fields
Variable	type, length, dimension info
Procedure	address of parameter list, number of parameters
Function	type of returned value, length of returned value, address of address of parameter list, number of parameters
Label	statement number

Operations on Search Structures:

1. Operation add : Add the entry of a symbol. Entry of symbol is created only once.

2. Operation search : Search and locate the entry of a symbol. Searching may be performed for more than once.

3. Operation delete : Delete the entry of a symbol. Uncommon operation.

Binary Search Organization

All entries in a table are assumed to satisfy an ordering relation. '<' relation implies that the symbol occupying an entry is 'smaller than' the symbol occupying the next entry.

Algorithm (Binary Search)

```
1.    start : =1; end : = f ;
2.    while start <= end
– (a) e:= [(start+end)/2]; where [] implies a rounded quotient. Exit
      with success if s=se.
– (b)if s < sethen end : = e-1 ; else start : = e+1 ;
3. Exit with failure.
```

Binary search organization is suitable only for a table containing a fixed set of symbols.

Hash Table Organization :

There are three possibilities exist concerning the predicted entry

–Entry may be occupied by s

–Entry may be occupied by some other symbol

–Entry may be empty.

Linked List & Tree Structure Organizations

Each entry in linked list organization contains a single pointer field.

List has to be searched sequentially.

Hence its performance is identical with that of sequential search tables.

Binary Trees

Each node in the tree is a symbol entry with two pointer fields i.e. Left Pointer and Right Pointer.

Algorithm (Binary Search Tree)

```
1.    current_node_pointer := address of root
2.    if s = (current_node_pointer)*.symbol then exit with success;
3.    if s < (current_node_pointer)*.symbol then
      current_node_pointer : = (current_node_pointer) *. left_pointer;
      elsecurrent_node_pointer:= (current_node_pointer) *. right_pointer;
4.    If current_node_pointer := nill thenexit with failure. elsegoto step 2.
```

Check your Progress

When of the following search structures are used Binary Search and Hash Table. Why?

2.3 Allocation Data Structure

Important allocation data structures are –Stacks, Extended Stacks and Heaps.

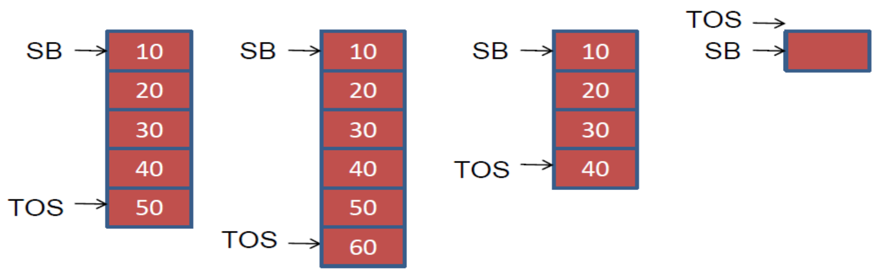
Stack: is a linear data structure which satisfies following properties:

1. Allocation and de-allocation are performed in a LIFO manner.
2. Only last element is accessible at any time.
   - SB –Stack Base points to first word of stack.
   - TOS –Top Of Stack points to last entry allocated to stack.

Extended Stack Model: is needed for handling variable length record. All entries may not be of same size. A Record consists of a set of consecutive stack entries. Two new pointers exist in the model other than SB and TOS :-

1. RB Record Base pointing to the first word of the last record in stack.
2. 'Reserve Pointer', the first word of each record.

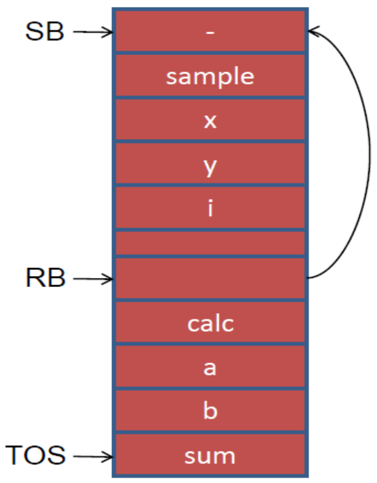
The allocation an



Now let us see an implementation of this model in a Pascal program that contains nested procedures where many symbol table must co-exist during compilation.

Example : Consider Pascal Program

Program Sample(input,output);

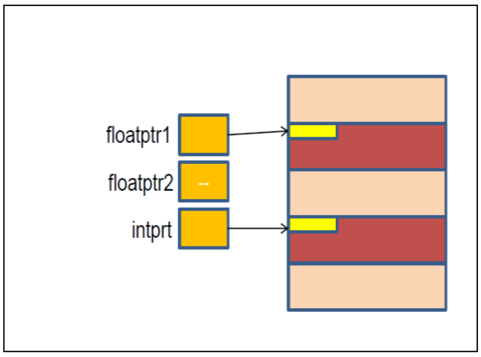


```
varx,y : real;
i : integer;
Procedure calc(vara,b : real);
var sum : real;
begin
sum := a+b;
---
---
endcalc;
begin {Main Program}
----
----
end.
```

Heaps: is Non Linear Data Structure which Permits allocation and de-allocation of entities in random order. Heap DS does not provide any specific means to access an allocated entity. Hence, allocation request returns pointer to allocated area in heap. Similarly, de-allocation request must present a pointer to area to be de-allocated. So, it is assumed that each user of an allocated entity maintains a pointer to the memory area allocated to the entity.

Example: Status of Heap after c program execution

```
float *floatptr1,*floatptr2;
Int *intprt;
floatptr1=(float *)calloc(5,sizeof(float));
floatptr2=(float *)calloc(5,sizeof(float));
intptr=(int *)calloc(5,sizeof(int));
free(floatptr2);
```



•This creates a ‘hole’ in the allocation.

2.4 Summary

In this unit you have studied the requirement of search data structures and allocation data structures for language processors.

2.5 Terminal Questions

1. Which operation is frequently used by a Language Processor?
2. What are the criteria for classification of Data Structures for Language Processors?
3. What are the differences between Search Data Structures and Allocation Data Structures in Language Processors?

\

UNIT - 3 : SOFTWARE TOOLS

Structure

3.0 Introduction

Software development tool is a program/application that software developers use to create, debug, maintain, programs and applications. Software development tools can be roughly divided into the following categories:

1) Performance analysis tools
2) Debugging tools
3) Static analysis and formal verification tools
4) Correctness checking tools
5) Memory usage tools
6) Application build tools
7) Integrated development environment

3.1 Objectives

In objective of this unit is to learn the details of software development tools such as- text editors, debuggers and programming environments from perspective of system software design.

3.2 Software Tools for Program Development

An Interactive text editor has become an important part of almost any computing environment. Text editor acts as a primary interface to the computer for all type of program development. An interactive debugging system provides programmers with facilities that aid in testing and debugging of programs. Our discussion is broad in scope, giving the overview of Text Editors, Debugging systems etc for Program Development.

3.3 Text Editor

Text Editor is software that can compose a string, or modify an existing one. Two common types of editors:

- line editors: lines of text are numbered automatically, and modified by giving the number of the line as part of the editing command
- screen editors: text is displayed on the screen, and the cursor can be moved to where the changes are going to be made.

Command language:

i. text commands
ii. function keys
iii. menus

Example - Line Editor Basic Commands :

- Edit	E <filename>
- Display	L <line no>*or* L <line no 1> - <line no2>(eg. L 100 or L 100-130)
- Insert	I <line no>(eg. I 20 inserts lines after line 20)
- Delete	D <line no>*or*D <line no 1> - <line no 2>(eg. D 100 or D 100-125)
- Save	S

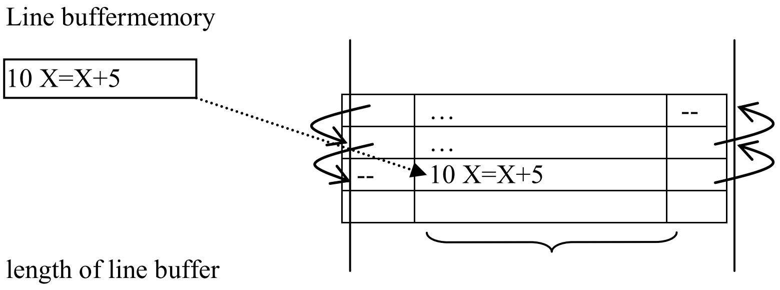
Command Processing

1. If file size is smaller than memory buffer

i.The entire file can be kept in memory.

ii.Line is composed into line buffer, which will be handled by the buffer manager. It puts the line into memory buffer.

iii. Doubly linked lists are used



The disadvantage of this method is, it leaves unused space in the memory due to the deletions. To avoid this, if a block of lines is deleted, the editor may keep track of these spaces and reuse them if necessary. This method is called garbage collection.

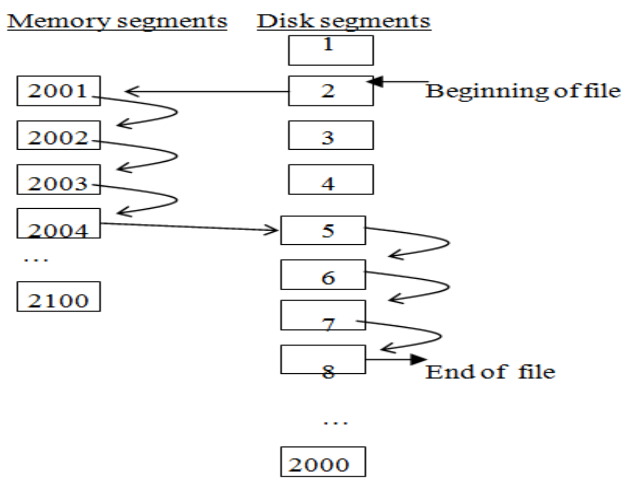
2. If file size is larger than memory buffer

i. Parts of the file are moved into the memory to work on. Rest of the file is kept on disk.
ii. If the required line is already in memory, no extra processing is necessary.
iii. If the required line is on disk,

   a. write the segments in the memory buffer onto disk

   b. read a block of segments which include the required line, from disk and place them in the memory. It is better to choose the block of segments in a way that the line to be accessed will be in the middle of them.

Following table keeps the used and free segments in memory and on disk. In this example, size of a data segment is the same as the size of a sector on the disk (eg. 256 bytes). Disk segments are numbered between1-2000 and memory segments are numbered between 2001-2100. The segments are in the form of a doubly linked list.



Memory		Disk	
in use	free	in use	free
2001	2005	2	1
2002	2006	5	3
2003	...	6	4
2004	...	7	9
	...	8	...
	2100		...
			...
			2000

Check your Progress

What is purpose of text-editor?

Screen Editors

The screen is the viewing window for the text file. User positions the cursor on the screen where editing is to take place. There is an area of RAM, which corresponds to what is seen on the screen. Each byte in this area corresponds to a particular character position on the screen. The editor can cause a character to appear at a particular place on the screen by writing its ASCII code into the corresponding position in RAM. A special hardware displays this area on the screen and refreshes it 30 times per second.

Detailed of Editor

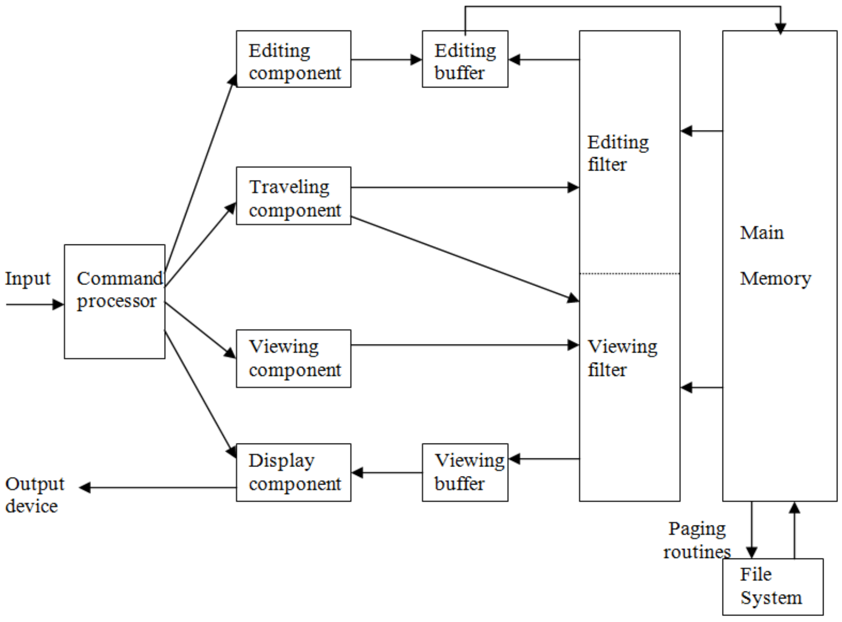


Figure 3.1 Command processor**:** accepts user commands and analyzes their syntactic structures.

Editing component**:** is the collection of modules dealing with editing. It maintains the current editing pointer, which shows the start of the editing area.

Editing filter**:** filters the document to generate a new editing buffer. It gathers the selection of contiguous characters beginning at the current point. It may also gather portions of the document that are not contiguous.

Traveling component**:** performs the setting of the current editing and viewing pointers.

Viewing component: is the collection of modules responsible for determining the next view. It maintains the current viewing pointer, which shows the start of the area to be viewed.

Viewing filter: filters the document to generate a new viewing buffer. In line editors it contains the current line, in screen editors it contains a rectangular cutout of the plane of text.

Display component: takes the viewing buffer and produces a display by mapping the buffer to a rectangular subset of the screen, called *window*.

Paging: If loading the entire document into main memory is not feasible, this problem is solved either by the editor paging routines (where portions of the document, called pages, are read into memory when needed) or by the virtual memory of the operating system.

3.4 Debug Monitors

Debugging Systems

A debugging system (DB) provides programmers with facilities that aid in testing and debugging of programs. Many such systems are available during these days.

Debugging Functions and Capabilities :

One important requirement of any DB is the observation and control of the flow of program execution.

Setting break points– execution is suspended use debugging commands to analyze the progress of the program, résumé execution of the program.

Setting some conditional expressions, evaluated during the debugging session, program execution is suspended, when conditions are met, analysis is made, later execution is resumed.

A Debugging system should also provide functions such as tracing and trace back.

- Tracing can be used to track the flow of execution logic and data modifications. The control flow can be traced at different levels of detail – procedure, branch, individual instruction, and so on.
- Trace back can show the path by which the current statement in the program was reached. It can also show which statements have modified a given variable or parameter. The statements are displayed rather than as hexadecimal displacements

Program-Display capabilities

A debugger should have good program-display capabilities.

- Program being debugged should be displayed completely with statement numbers.
- The program may be displayed as originally written or with macro expansion.
- Keeping track of any changes made to the programs during the debugging session.

Support for symbolically displaying or modifying the contents of any of the variables and constants in the program. Resume execution – after these changes.

To provide these functions, a debugger should consider the language in which the program being debugged is written.

The context being used has many different effects on the debugging interaction.

Sometimes the language translator itself has debugger interface modules that can respond to the request for debugging by the user. The source code may be displayed by the debugger in the standard form or as specified by the user or translator.

It is also important that a debugging system be able to deal with optimized code. Many optimizations like

- Invariant expressions can be removed from loops
- Separate loops can be combined into a single loop
- Redundant expression may be eliminated
- Elimination of unnecessary branch instructions

Leads to rearrangement of segments of code in the program. All these optimizations create problems for the debugger, and should be handled carefully

Relationship with Other Parts of the System :

- The important requirement for an interactive debugger is that it always be available. It must appear as part of the run-time environment and an integral part of the system.
- When an error is discovered, immediate debugging must be possible. The debugger must communicate and cooperate with other operating system components such as interactive subsystems.
- Debugging is more important at production time than it is at application-development time. When an application fails during a production run, work dependent on that application stops.
- The debugger must also exist in a way that is consistent with the security and integrity components of the system.
- The debugger must coordinate its activities with those of existing and future language compilers and interpreters

User-Interface Criteria :

- Debugging systems should be simple in its organization and familiar in its language, closely reflect common user tasks.
- The simple organization contributes greatly to ease of training and ease of use.
- The user interaction should make use of full-screen displays and windowing-systems as much as possible.
- With menus and full-screen editors, the user has far less information to enter and remember. There should be complete functional equivalence between commands and menus – user where unable to use full-screen debugging systems may use commands.
- The command language should have a clear, logical and simple syntax.
- Command formats should be as flexible as possible.
- Any good debugging systems should have an on-line HELP facility. HELP should be accessible from any state of the debugging session.

DEBUGGING METHODS

- Debugging by Induction
- Debugging by Deduction
- Debugging by backtracking

Debugging by Induction

It should be obvious that careful thought will find most errors without the debugger even going near the computer. One particular thought process is induction, where you move from the particulars of a situation to the whole. That is, start with the clues (the symptoms of the error, possibly the results of one or more test cases) and look for relationships among the clues.

The steps are as follows :

1. Locate the pertinent data. A major mistake debuggers make is failing to take account of all available data or symptoms about the problem. The first step is the enumeration of all you know about what the program did correctly and what it did incorrectly
2. Organize the data. Remember that induction implies that you‘re processing from the particulars to the general, so the second step is to structure the pertinent data to let you observe the patterns.
3. Devise a hypothesis. Next, study the relationships among the clues and devise, using the patterns that might be visible in the structure of the clues, one or more hypotheses about the cause of the error.
4. Prove the hypothesis. However, it is vital to prove the reasonableness of the hypothesis before you proceed. If you skip this step, you‘ll probably succeed in correcting only the problem symptom, not the problem itself.
5. Fix the Error. If hypothesis is proved successfully then go on to fix the error else repeat steps 3 and

Debugging by Deduction

The process of deduction proceeds from some general theories or premises, using the processes of elimination and refinement, to arrive at a conclusion (the location of the error).

The steps are as follows :

1. Enumerate the possible causes or hypotheses. The first step is to develop a list of all conceivable causes of the error. They don‘t have to be complete explanations; they are merely theories to help you structure and analyze the available data.
2. Use the data to eliminate possible causes. Carefully examine all of the data, particularly by looking for contradictions, and try to eliminate all but one of the possible causes. If all are eliminated, you need more data through additional test cases to devise new theories.
3. Refine the remaining hypothesis. The possible cause at this point might be correct, but it is unlikely to be specific enough to pinpoint the error. Hence, the next step is to use the available clues to refine the
4. Prove the remaining hypothesis. This vital step is identical to step 4 in the induction method.

Debugging by Backtracking

- An effective method for locating errors in small programs is to backtrack the incorrect results through the logic of the program until you find the point where the logic went astray.

In other words, start at the point where the program gives the incorrect result—such as where incorrect data were printed. At this point you deduce from the observed output what the values of the program‘s variables must have been.

By performing a mental reverse execution of the program from this point and repeatedly using the process of —if this was the state of the program at this point, then this must have been the state of the program up here, if you can quickly pinpoint the error.

Debugging By Testing

- Test cases for debugging: purpose is to provide information useful in locating a suspected error.
- This method is used in conjunction with the induction method

Debugging By Brute Force

- Requires little thought
- Inefficient and generally unsuccessful.

Can be partitioned into three categories:

- Debugging with a storage dump
- Debugging according to the common suggestion to —scatterprint statements throughout the program.
- Debugging with automated debugging tools.

Check your Progress

What is main purpose of debugmonitor?

3.5 Programming Environments

Conceptual model of the editing system provides an easily understood abstraction of the target document and its elements.

For example, Line editors – simulated the world of the key punch – 80 characters card, single line or an integral number of lines.

Screen editors – Document is represented as a quarter-plane of text lines, unbounded both down and to the right. User sees, through a cutout, only a rectangular subset of the quarter plane on a multi-line display terminal.

The user interface is concerned with, the input devices, the output devices and, the interaction language. The input devices are used to enter elements of text being edited, to enter commands. The output devices, lets the user view the elements being edited and the results of the editing operations and, the interaction language provides communication with the editor.

- text devices
- button devices
- Locator devices.

1. Text Devices are keyboard. Button Devices are special function keys, symbols on the screen. Locator Devices are mouse, data tablet. There are voice input devices which translates spoken words to their textual equivalents.
2. Output Devices are Tele typewriters(first output devices), Glass teletypes (Cathode ray tube (CRT) technology), Advanced CRT terminals, Thin-film-transistor (TFT) Monitors and Printers (Hard-copy).
3. The interaction language could be, typing oriented or text command oriented and menu-oriented user interface. Typing oriented or text command oriented interaction was with oldest editors, in the form of use of commands, use of function keys, control keys etc.
4. Menu-oriented user interface has menu with a multiple choice set of text strings or icons. Display area for text is limited. Menus can be turned on or off.

3.6 User Interfaces

USER-INTERFACE CRITERIA for Interactive Debugging System

The interactive debugging system should be user friendly. The facilities of debugging system should be organized into few basic categories of functions which should closely reflect common user tasks.

Full – screen displays and windowing systems

The user interaction should make use of full-screen display and windowing systems.

The advantage of such interface is that the information can be should displayed and changed easily and quickly.

Menus

- With menus and full screen editors, the user has far less information to enter and remember
- It should be possible to go directly to the menus without having to retrace an entire hierarchy.
- When a full-screen terminal device is not available, user should have an equivalent action in a linear debugging language by providing commands.

Command language

- The command language should have a clear, logical, simple syntax. Parameters names should be consistent across set of commands

  Parameters should automatically be checked for errors for type and range values.
- Defaults should be provided for parameters.
- Command language should minimize punctuations such as parenthesis, slashes, and special characters.

On Line HELP facility

- Good interactive system should have an on-line HELP facility that should provide help for all options of menu
- Help should be available from any state of the debugging system.

3.7 Summary

We have studied about tools for program development such as Debuggers, Text Editors, and User Interface etc.

3.8 Terminal Questions

1. What are the types of text editors?
2. What are debuggers function in optimization of program development?
3. How programming environment is useful to the user?
4. What is the main difference between line editor and stream editor?
5. Which are the four basic categories in which programming environment can be classified?

UNIT - 4 : ASSEMBLERS

Structure

4.0 INTRODUCTION

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

4.1Objective

1.In this unit, you will be introduced to the concept of assemblers:

a) learn about assembly language programming

b) learn the basic features of assembly language programming

c) describe assembly language programming

d) as well as their advantages

e) learn about design specification of assembler.

4.2 Elements of Assembly Language Programming

An assembler is a program that accepts as input an assembly language program (source) and produces its machine language equivalent (object code) along with the information for the loader.

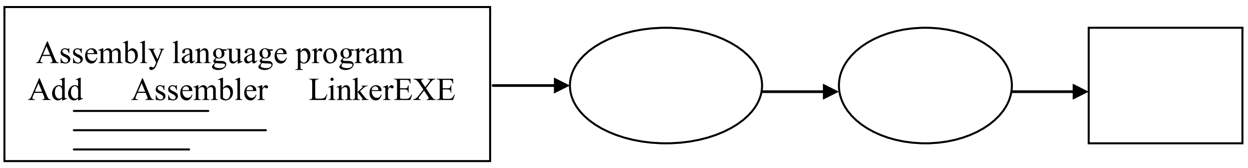


Figure 4.1 Executable program generation from an assembly source code

Advantages of coding in assembly language are: Provides more control over handling particular hardware components, generate smaller, more compact executable modules and often results in faster execution Disadvantages: not portable, more complex and requires understanding of hardware details.

An assembler does the following:

1. Generate machine instructions: evaluate the mnemonics to produce their machine code, evaluate the symbols, literals, addresses to produce their equivalent machine addresses and convert the data constants into their machine representations

2. Process pseudo operations

   Encoding instructions as binary numbers is natural and efficient for computers. Humans, however, have a great deal of difficulty understanding and manipulating these numbers. People read and write symbols (words) much better than long sequences of digits.

The language translation is needed by which a human-readable program is translated into a form that a computer can execute.

An assembler is software tool that translates assembly language program into binary form. Assemblers provide a user friendlier representation called assembly language than a computer's 0s and 1s that simplifies writing and reading programs.

In assembly language, mnemonic is a symbolic names for operations and locations becomes one facet of this representation. Another facet is assembly programming facilities that increase a program's clarity.

An assembler reads a single assembly language source file and produces an object file containing machine instructions and bookkeeping information that helps combine several object files into a program.

In a typical assembly programming environment, most programs consist of several files—also called modules— that are written and assembled independently. A program may also use prewritten routines supplied in a program library. A module typically contains references to subroutines and data defined in other modules and in libraries. The code in a module cannot be executed when it contains unresolved references to labels in other object files or libraries.

Another tool, called a linker, combines a collection of object and library files into an executable file resolving issues, which a computer can run.

An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file. The important functions of an assembler are following:-

1) Assembler is a program to handle all the tedious mechanical translations

2) Allows using:
   (i) Symbolic opcodes
   (ii) Symbolic operand values
   (iii) Symbolic addresses

3) The Assembler keeps track of the numerical values of all symbols translates symbolic values into numerical values.

4) Time Periods of the Various Processes in Program Development.

5) The Assembler Provides:

   a. Access to all the machine's resources by the assembly program.

      This includes access to the entire instruction set of the machine.

   b. A means for specifying run-time locations of program and data in memory.

   c. Provide symbolic labels for the representation of constants and addresses.

   d. Perform assemble-time arithmetic.

   e. Provide for the use of any synthetic instructions.

   f. Emit machine code in a form that can be loaded and executed.

   g. Report syntax errors and provide program listings

   h. Provide an interface to the module linkers and program loader.

   i. Expand programmer defined macro routines.

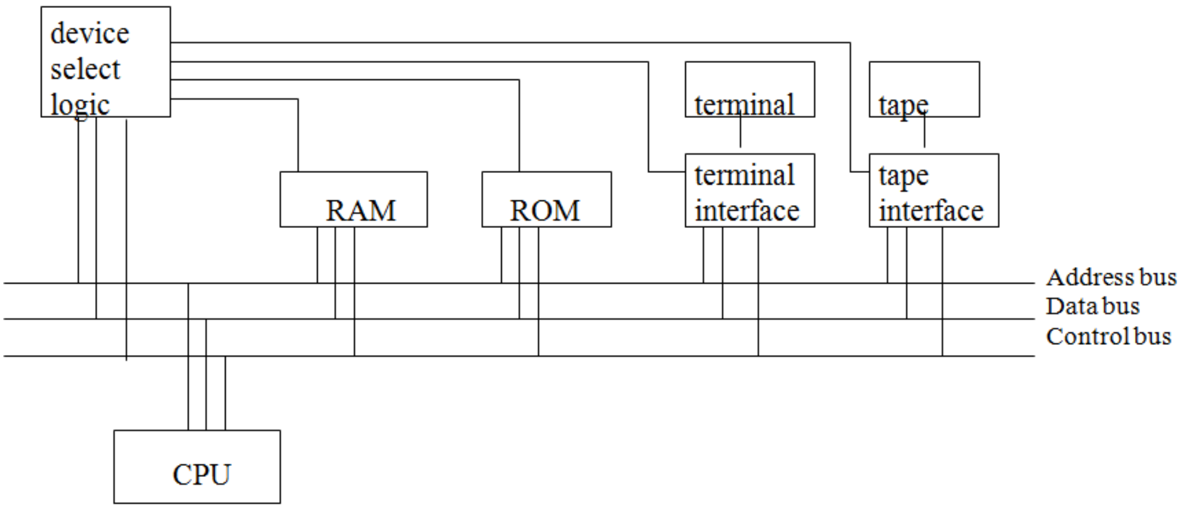


Figure 4.2 : Interconnection of computer components

Central Processing Unit (CPU) controls the operation of the computer and processes data.

Major components of CPU are:Arithmetic Logic Unit (ALU): performs arithmetic and logic operations, Control Unit: controls the operation of CPU,Registers: Storage space inside the CPU, Interconnection mechanism for communication among ALU, control unit and registers.

Main Memory, Instructions and data are stored in main memory.A memory word is a group of bits that are written in or read out of the memory as a unit. The addressable unit in main memory is a word.

Example:A memory module has word length of 32-bits and addressed by 16 bits. The size of this memory is 1 Kbytes.

System Bus is the communication path that connects CPU, memory and I/O units. Bus lines can be classified into three functional groups: data lines, address lines,control lines

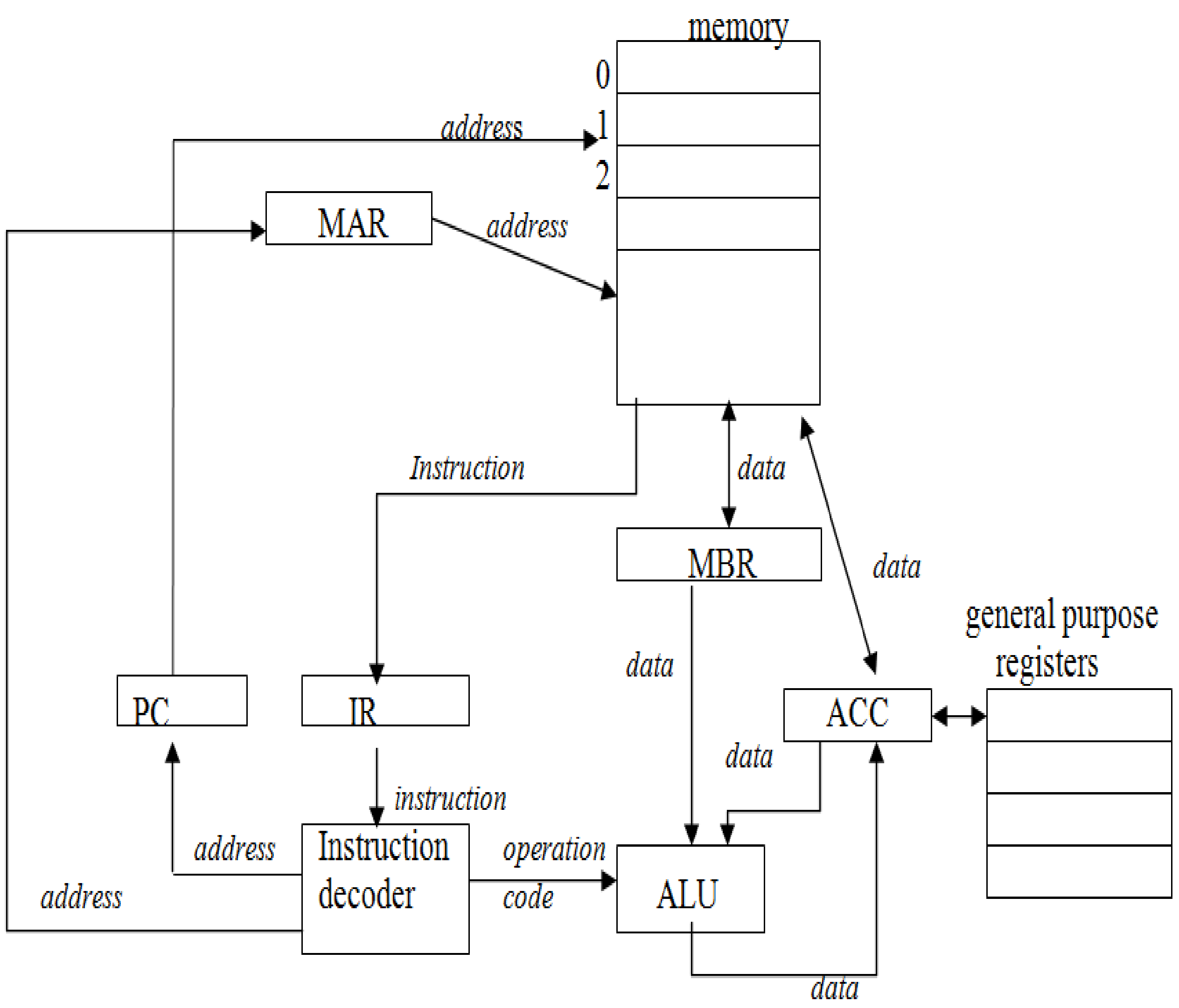


Figure 4.3: Information flow between the components of a computer system

PC (program counter) : contains the location of the next instruction to be executed in memory. IR (instruction register) : contains the current instruction. WR (working registers): serve as "scratch pads" for the instruction interpreter. GR (general registers): used by the programmer as storage locations for special functions. MAR (memory address register): contains the address of the memory location that is to be read from or stored into MBR (memory buffer register): contains a copy of the designated memory location specified by the MAR after a read operation or the new contents of the memory location prior to a write operation. Memory controller: transfers data between the MBR and the core memory location, the address of which is in the MAR. I/O channels: used to input and output the information from memory

Instruction execution cycle :

1. Fetch instruction: Read the next instruction into IR

2. Decode instruction: Determine the opcode and the operand specifiers
3. Calculate effective address and fetch operands: Calculate the address of the operands and fetch the operands from memory into registers
4. Execute instruction: Perform the indicated instruction
5. Write result: Store the result in memory

Example : Execution of ADD instruction

Op-code	memory location	memory location
	(addr1)	(addr2)

ADD 010A, 010C

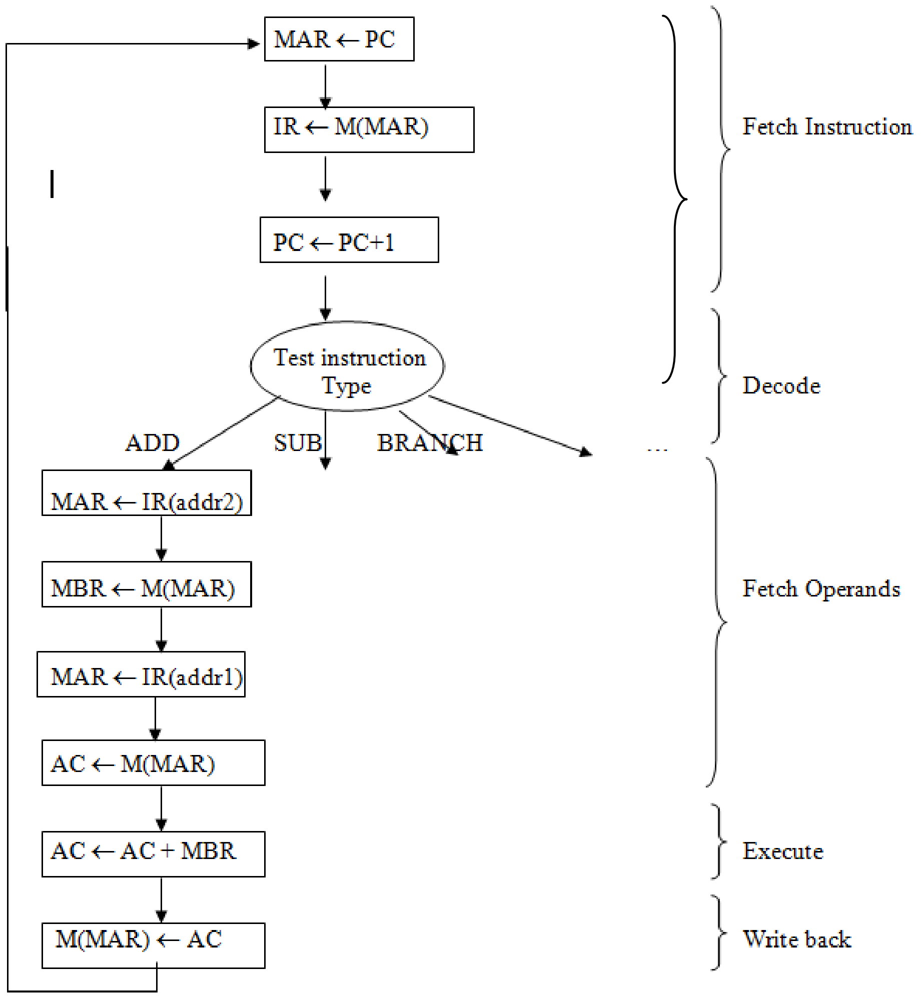


Figure 4.4: Flowchart for the execution of ADD instruction

Instruction length should either be equal to the memory transfer length or a multiple of it.
Fields of instruction format:

1. Operation code (opcode) field: specifies the operation to be performed
2. Address field: designates a memory address or a processor register
3. Mode field: specifies the way the operand or effective address is determined

Op-code	mode	data/address

Example : ADDR1, R2
CLR A

Operation type	**Examples**
Arithmetic and logical	Integer arithmetic and logical operations: add, subt., and, or
Data transfer	Loads/stores (move instructions)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management inst.
Floating point	Floating point operations: add, multiply
Decimal	Decimal: add, multiply, decimal to char conversion
String	String: move, compare, search

The first three (arithmetic and logic, data transfer, and control) are basic operations which exist in all instruction sets.

Operands

3-operand format (a result and 2 source operands)

2-operand format (one of the operands is both the result and the source operand)

1-operand format (the operand is both the result and the source operand)

0-operand format (no operands)

register-register (all operands are in registers),

register-memory (some operand are in memory and some operands are in registers),
memory-memory (all operands are in memory)

Addressing Modes:

Addressing mode of an operation show how the CPU will access the operands.

Effective Address (EA): The actual memory address specified by an addressing mode

Types of Addressing Modes:

Implied addressing: The operand is implied in the opcode

Immediate addressing: The operand is specified in the instruction itself

Direct addressing: Effective address is equal to the address part of the instruction

Indirect addressing: The address field of the instruction gives the address where the effective address is stored in memory

Register addressing: The operands are in registers

Register indirect: Instruction specifies a register whose contents give the address of the operand in memory

Indexed addressing: The content of the index register is added to the address part of the instruction to obtain the effective address

Base register addressing: the content of a base register is added to the address part of the instruction to obtain the effective address

Auto-increment or auto-decrement: similar to register indirect except that the register is incremented or decremented after its value is used to access memory

Relative addressing: The content of the program counter is added to the address part of the instruction to obtain the effective address

Addressing Mode	**Example instruction**	**Meaning**	**When used**
Register	Add R4, R3	R4←R4+R3	When a value is in a register
Immediate or Literal	Add R4,#3	R4←R4+3	For constants.
Displacement or based	Add R4,100(R1)	R4←R4+M[100+R1]	Accessing local variables
Register deferred or indirect	Add R4, (R1)	R4←R4+M[R1]	Accessing using a pointer-computed address
Indexed	Add R3, (R1+R2)	R3←R3+M[R1+R2]	Useful in array addressing
Direct or absolute	Add R1, (1001)	R1←R1+M[1001]	Static data access
Indirect or deferred	Add R1, @(R3)	R1←R1+M[M[R3]]	R3 contains the address of the operand

Auto-increment	Add R1, (R2)+	R1←R1+M[R2]
		R2←R2+d
			Useful for stepping thru. arrays within a loop. R2, start of array

Auto-decrement	Add R1, -(R2)	R2←R2-d
		R1←R1+M[R2]
			Same use as auto-incr., they are also used in stack operations

Scaled or Index	Add R1,100(R2)[R3]	R1←R1+ M[100+R2+R3*d]	Used to index arrays. any base addr. mode

Addressing modes with examples, meaning and usage

Example:

PC = 200
R1 = 400 (processor register)
XR=100 (index register)

Address	**Memory**
....	
600	900
......	
702	325
.....	

800	300
.....	
200	LDA, M, 500
202	Next instruction
....	
3FF	450
400	700
....W	
500	800

Direct address :	AC = 800	(EA=500)
Immediate :	AC=500	(EA=operand field)
Indirect :	AC=300	(EA=800)
Relative :	AC=325	(EA=500+200)
Indexed :	AC=900	(EA=XR+500)
Register :	AC=400	(RI)

Instruction Set : We can distinguish 3 main instruction set architectures.

1. Accumulator architecture*:* one operand is implicitly the accumulator.

   Ex*:* C=A+B

   ```
   LOAD    A
   ADD     B
   STORE   C
   ```

2. Stack architecture: the operands are implicitly on the top of stack.

   Ex: C=A+B

   ```
        PUSH    A
        PUSH    B
   ADD
        POP     C
   ```

3. General purpose register architectures*:* have only explicit operands-either registers or memory locations. Ex : C=A+B

   ```
             LOAD R2, A
         ADD R2, B
         STOREC, R2
   ```

Assembler Syntax and Directives :

Syntax : Label OPCODE Op1, Op2, ... ; Comment field

Pseudo-operations (sometimes called "pseudos," or directives) are "opcodes" that are actually instructions to the assembler and that do not result in code being generated.

Assembler data structures

- Table that maps text of opcodes to op number and instruction format(s)

- "Symbol table" that maps defined symbols to their value

  Disadvantages of Assembly
- programmer must manage movement of data items between memory locations and the ALU.
- programmer must take a "microscopic" view of a task, breaking it down to manipulate individual memory locations.
- assembly language is machine-specific.
- statements are not English-like (Pseudo-code)

Directives Assembler

1. Directives are commands to the Assembler
2. They tell the assembler what you want it to do, e.g. a. Where in memory to store the code b. Where in memory to store data c. Where to store a constant and what its value is d. The values of user-defined symbols

Object File Format

Assemblers produce object files.

An object file on UNIX operating system contains six distinct sections:

- The object file header describes the size and position of the other pieces of the file.
- The text segment contains the machine language code for routines in the source file. These routines may be non-executable because of unresolved references.
- The data segment contains a binary representation of the data in the source file. The data also may be incomplete because of unresolved references to labels in other files.
- The relocation information identifies instructions and data words that depend on absolute addresses. These references must change if portions of the program are moved in memory.
- The symbol table associates addresses with external labels in the source file and lists unresolved references.
- The debugging information contains a concise description of the way in which the program was compiled, so a debugger can find which instruction addresses correspond to lines in a source file and print the data structures in readable form.

The assembler produces an object file that contains a binary representation of the program and data and additional information to help link pieces of a program. This relocation information is necessary because the assembler does not know which memory locations a procedure or piece of data will occupy after it is linked with the rest of the program. Procedures and data from a file are stored in a contiguous piece of memory, but the assembler does not know where this memory will be located. The assembler also passes some symbol table entries to the linker. In particular, the assembler must record which external symbols are defined in a file and what unresolved references occur in a file.

Macros

Macros are a pattern-matching and replacement facility that provides a simple mechanism to name a frequently used sequence of instructions. Instead of repeatedly typing the same instructions every time macros are used, a programmer invokes the macro and the assembler replaces the macro call with the corresponding sequence of instructions. Macros, like subroutines, permit a programmer to create and name a new abstraction for a common operation. Unlike subroutines, however, macros do not cause a subroutine call and return when the program runs since a macro call is replaced by the macro's body when the program is assembled. After this replacement, the resulting assembly is indistinguishable from the equivalent program written without macros.

The 2-Pass Assembly Process

- Pass 1:

1. Initialize location counter (assemble-time "PC") to 0
2. Pass over program text: enter all symbols into symbol table
   a. May not be able to map all symbols on first pass
   b. Definition before use is usually allowed
3. Determine size of each instruction, map to a location
   a. Uses pattern matching to relate opcode to pattern
   b. Increment location counter by size
   c. Change location counter in response to ORG pseudos

- Pass 2 :

1. Insert binary code for each opcode and value
2. "Fix up" forward references and variable-sizes instructions

- Examples include variable-sized branch offsets and constant fields

A two-pass assembler performs two sequential scans over the source code:

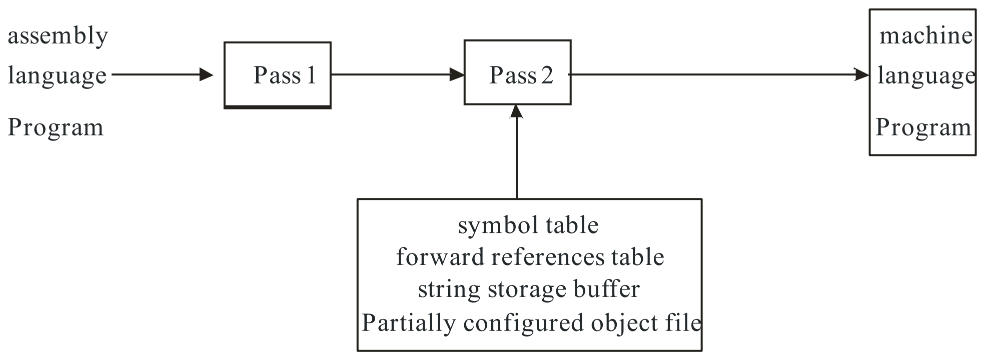
Pass 1: symbols and literals are defined

Pass 2: object program is generated

Parsing: moving in program lines to pull out op-codes and operands

Data Structures :

- Location counter (LC): points to the next location where the code will be placed
- Op-code translation table: contains symbolic instructions, their lengths and their op-codes (or subroutine to use for translation)
- Symbol table (ST): contains labels and their values
- String storage buffer (SSB) contains ASCII characters for the strings
- Forward references table (FRT) : contains pointer to the string in SSB and offset where its value sill be inserted in the object code



-

A simple two pass assembler.

Example : Decrement number 5 by 1 until it is equal to zero.

assembly language program	memory	object code address	in memory	
-------------	----------	↓	↓	
START	0100H			
LDA	#5	0100	01	
		0101	00	
		0102	05	
LOOP : SUB	#1	0103	1D	
		0104	00	
		0105	01	
COMP	#0	0106	29	
		0107	00	
		0108	00	
JGT	LOOP	0109	34	} ←
		010A	01	placed in Pass 1
		010B	03	
RSUB		010C	4C	
		010D	00	
		010E	00	
END				

Op-code Table

Mnemonic	***Addressing mode***	***Opcode***
LDA	immediate	01
SUB	immediate	1D
COMP	immediate	29
LDX	immediate	05
ADD	indexed	18
TIX	direct	2C
JLT	direct	38
JGT	direct	34

RSUB	implied	4C

Symbol Table

Symbol	*Value*
LOOP	0103

```
assembly language                              memory                 object code
        program                                address                 in memory
----------------------
START
LDA    #0              0000                                             01
                       0001                                             00
                       0002                                             00
LDX    #0              0003                                             05
                       0004                                             00
                       0005                                             00
LOOP: ADD    LIST,X    0006                                             18
                       0007                                             00
                       0008                                             12
TIX  COUNT             0009                                             2C
        000A           00
        000B           15
JLT   LOOP       000C  38
        000D           00
        000E           06
RSUB             000F  4C
                                                                        001000
                                                                        001100
        LIST:          WORD                     200                     001200
                                                                        001302
                                                                        001400
        COUNT: WORD   6                                                 001500
                                                                        001600
                                               0017                         06
```

Example: Sum 6 elements of a list which starts at location 200.

```
assembly language                         memory                       object code
   program                                address                      in memory
-----------           ------------           ↓                              ↓
   START               0100H
     LDA                 #0                  0100                           01
                                             0101                           00
                                             0102                           00
     LDX                 #0                  0103                           05
                                             0104                           00
                                             0105                           00
LOOP : ADD             LIST, X               0106                           18
                                             0107                           01 } placed in Pass 2
                                             0108                           12 } ←
TIX                    COUNT                 0109                           2C
                                             010A                           01  placed in Pass 2
                                             010B                           15  ←
JLT                     LOOP                 010C                           38
```

```
                                   010D                        01} placed in Pass 1
                                   010E                        06} <----
RSUB                               010F                        4C
                                   0110                        00
                                   0111                        00
LIST: WORD        200              0112                        00
                                   0113                        02
                                   0114                        00
COUNT: WORD         6              0115                        00
                                   0116                        00
                                   0117                        06
                  END
```

Symbol Table

Symbol	*Address*
LOOP	0106
LIST	0112
COUNT	0115

Forward References Table

Offset	*SSB pointer for the symbol*
0007	DC00
000A	DC05

SSB

DC00	4CH	
DC01	49H	ASCII for L,I,S,T
DC02	53H	
DC03	54H	
DC04	5EH	ASCII for separation character
DC05		

Pass1

- All symbols are identified and put in ST
- All op-codes are translated
- Missing symbol values are marked

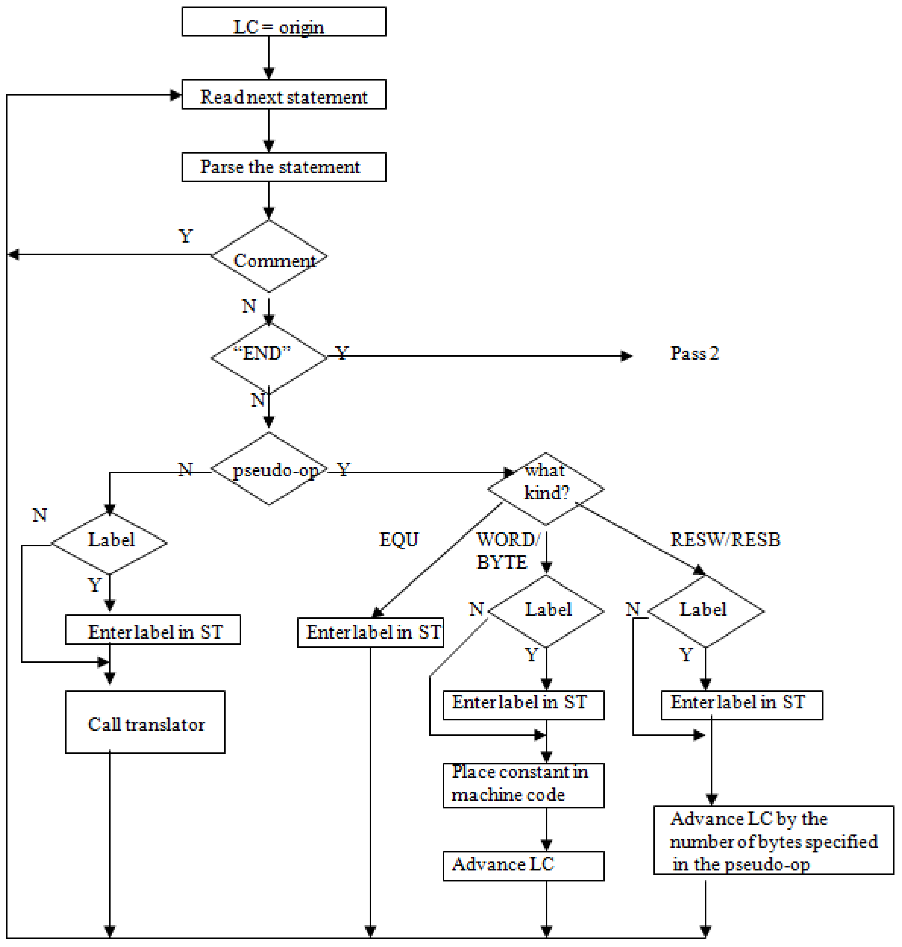


Figure 4.5: First pass of a two-pass assembler.
Translator Routine

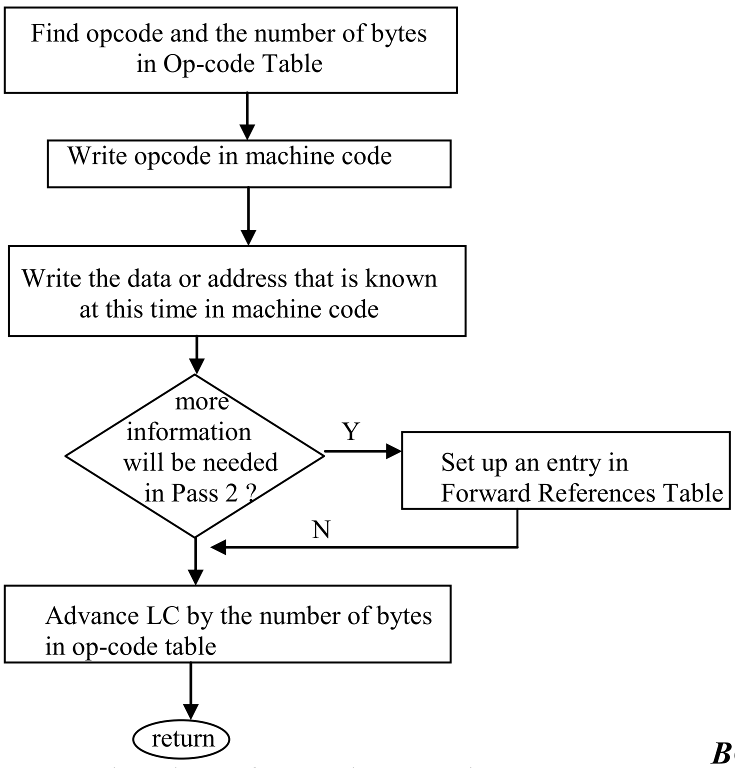

Find opcode and the number of bytes in Op-code Table
Write opcode in machine code
Write the data or address that is known at this time in machine code
more information will be needed in Pass 2 ?
Y
Set up an entry in Forward References Table
N
Advance LC by the number of bytes in op-code table
return

Figure 4.6: Flowchart of a translator routine

Check your progress

1. Assembler is a program that automate the translation of assembly language into machine language. How the assembler translates?
2. Mnemonic a memory trick is used in Assembly language. Easily relocatable language is Assembly Language or machine language
3. Which of the following is called low level languages?- a. Machine language b. Assembly language (or both)

Pass 2- Fills addresses and data that was unknown during Pass 1.

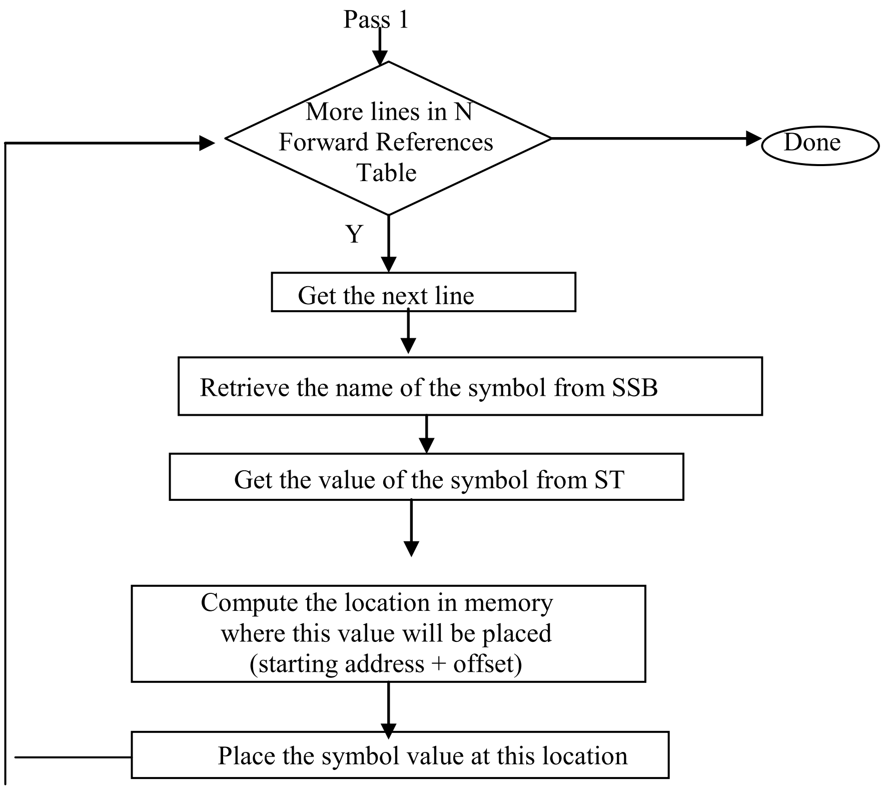

Pass 1
More lines in Forward References Table
N
Done
Y
Get the next line
Retrieve the name of the symbol from SSB
Get the value of the symbol from ST
Compute the location in memory where this value will be placed (starting address + offset)
Place the symbol value at this location

Figure 4.7: Second pass of a two-pass assembler

Relocatable Code : Producing an object code, this can be placed to any specific area in memory.

Direct Address Table (DAT) : contains offset locations of all direct addresses in the program (e.g., 8080 instructions that specify direct addresses are LDA, STA, all conditional jumps...). To relocate the program, the loader adds the loading point to all these locations.

$$\text{assembly language program} \longrightarrow \boxed{\text{Assembler}} \longrightarrow \text{machine language program and DAT}$$

Assembler output for a relocatable code.

Example: Following relocatable object code and DAT are generated for LIST, COUNT, LOOP direct addresses used in ADD, TIX and JLT in following example program.

```
        assembly language                                 memory      object code
             program                                      address     in memory
----------------------------------------------              ↓             ↓
          START
          LDA               #0                            0000          01
                                                          0001          00
                                                          0002          00
          LDX               #0                            0003          05
                                                          0004          00
                                                          0005          00
LOOP : ADD      LIST,       X                             0006          18
                                                          0007          00
                                                          0008          12
          TIX      COUNT                                  0009          2C
                                                          000A          00
                                                          000B          15
          JLT      LOOP                                   000C          38
                                                          000D          00
                                                          000E          06
          RSUB                                            000F          4C
                                                          0010          00
                                                          0011          00
LIST :    WORD              200                           0012          00
                                                          0013          02
                                                          0014          00
COUNT:WORD                  6                             0015          00
                                                          0016          00
                                                          0017          06
          END
```

DAT

0007
000A
000D

Forward and backward references in the machine code are generated relative to address 0000. To relocate the code, the loader adds the new load-point to the references in the machine code which are pointed by the DAT.

One-Pass Assemblers

Two methods can be used:

- Eliminating forward references

  Either all labels used in forward references are defined in the source program before they are referenced, or forward references to data items are prohibited.

- Generating the object code in memory

    No object program is written out and no loader is needed. The program needs to be re-assembled every time.

Multi-Pass Assemblers: Make as many passes as needed to process the definitions of symbols.

Example:

A	EQU	B	
	B	EQU	C
	C	DS	1

3 passes are required to find the address for A

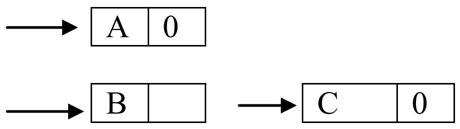
Such references can also be solved in two passes: entering symbol definitions that involve forward references in the symbol table. Symbol table also indicates which symbols are dependent on the values of others.

Example:

A	EQU	B	
	B	EQU	D
	C	EQU	D
	D	DS	1

At the end of Pass1:
Symbol Table

A	&1	B	0
B	&1	D	
C	&1	D	0
D		200	



After evaluating dependencies:
Symbol Table

A		200	0
B		200	0
C		200	0
D		200	0

4.7 Summary

Assembly language is a high level language needs software tool called assembler to translate into machine language. The design of assembler comprises of two-passes-which defines user-defined-symbols into table called symbol table and then translates assembly program into machine language program.

4.8 Terminal Questions

1. What is the main purpose of assembler?
2. What is an assembler directives?
3. Does the directives appear in object program?
4. Where does assembler stores all the names and their corresponding values?
5. What kind of assembler used to overcome the problems of the assembler in dealing with branching code?

UNIT - 5 : MACRO PROCESSORS

Structure

5.0 INTRODUCTION

Macro Processor is a program that lets you define the code that is reused many times giving it a specific Macro name and reuse the code by just writing the Macro name only. And the most important thing is that, macro processing is done before the compilation replacing the code in place of macro calls and deleting macro definitions.

5.1 Objectives

In this unit, you will learn the basics of macro-definition, macro-call, macro-expansion and advanced macro functionalities as well as design of macro processors.

5.2 Macros and Macro Processors

Macro Instructions

A macro instruction (macro) is simply a notational convenience for the programmer to write a shorthand version of a program.

It represents a commonly used group of statements in the source program. It is replaced by the macro processor with the corresponding group of source language statements. This operation is called "expanding the macro"

Macro Processors

A macro processor: Its functions essentially involve the substitution of one group of characters or lines for another.

Normally, it neither performs analysis of the text it handles nor does it concern the meaning of the involved statements during macro expansion.

Therefore, the design of a macro processor generally is machine independent.

Macro processors are used in

–assembly language

–high-level programming languages, e.g., C or C++

–OS command languages

–general purpose

Basic Functions

- Macro Definition
- Macro Invocation
- Macro Expansion
- One-Pass Algorithm
- Data Structure

Macro Definition

- Two new assembler directives are used in macro definition:

– MACRO : identify the beginning of a macro definition

– MEND : identify the end of a macro definition

- Prototype (pattern) for the macro:

– Each parameter begins with '&'

```
label    op       operands
     name  MACRO  parameters
           :
          body
           :
         MEND
```

Body: the statements that will be generated as the expansion of the macro.

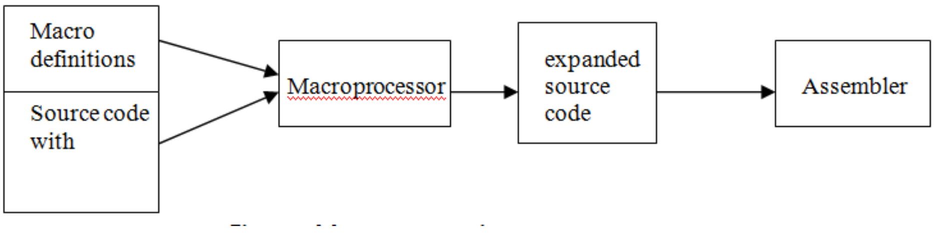


Figure 5.1 : Macro expansion on a source program.

Macro Invocation

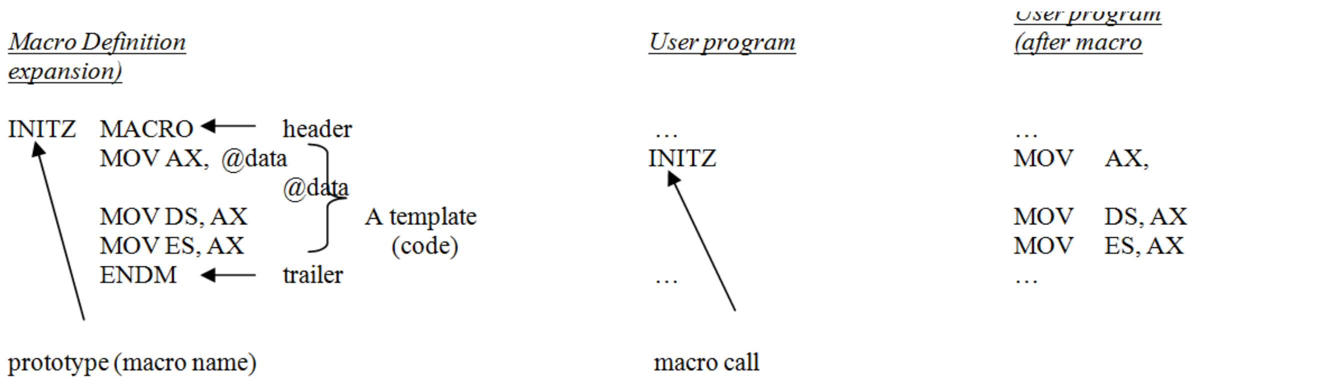
A macro invocation statement (a macro call) gives the name of the macro instruction being invoked and the arguments in expanding the macro. The processes of macro invocation and subroutine call are quite different.

–Statements of the macro body are expanded each time the macro is invoked.

–Statements of the subroutine appear only one, regardless of how many times the subroutine is called.

Macro Expansion

Each macro invocation statement will be expanded into the statements that form the body of the macro.



Arguments from the macro invocation are substituted for the parameters in the macro prototype.

The arguments and parameters are associated with one another according to their positions. The first argument in the macro invocation corresponds to the first parameter in the macro prototype, etc. This allows the programmer to use a macro instruction in exactly the same way as an assembler language mnemonic.

```
Example: displaying a message MES2.
```

Macro Definition

```
PROMPT  MACRO MESSGE     ;dummy argument
        MOV   AH, 09H
        LEA   DX, MESSGE
        INT   21H
        ENDM              ;end of macro
```

User program

```
…
PROMPT MES2
…
…
MES2 DB 'Enter the
          data as mm/dd/yy'
mm/dd/yy'
```

User program (after macro expansion)

```
…
MOV   AH, 09H
LEA   DX, MES2
INT   21H
…
MES2  DB 'Enter the
       data as
```

Conditional assembly:

Only part of the macro is copied out into the code. Which part is copied out will be under the control of the parameters in the macro call.

CONMB (condition) branch address

Ex: *Line*

no.Assembly instructions

……

8CONMB(&C>2), 15

9

……

15

……

If condition is true, skip the code up to line 15.

If condition is false, expansion continues from line 9.

Two-Pass Macro Processor

- Two-pass macro processor

- Pass 1:

  Process macro definition
- Pass 2 :

  Expand all macro invocation statements
  - Problem
- This kind of macro processor cannot allow recursive macro definition, that is, the body of a macro contains definitions of other macros (because all macros would have to be defined during the first pass before any macro invocations were expanded).

One-Pass Macro Processor

- A one-pass macro processor that alternate between macro definition and macro expansion in a recursive way is able to handle recursive macro definition.
- Because of the one-pass structure, the definition of a macro must appear in the source program before any statements that invoke that macro.

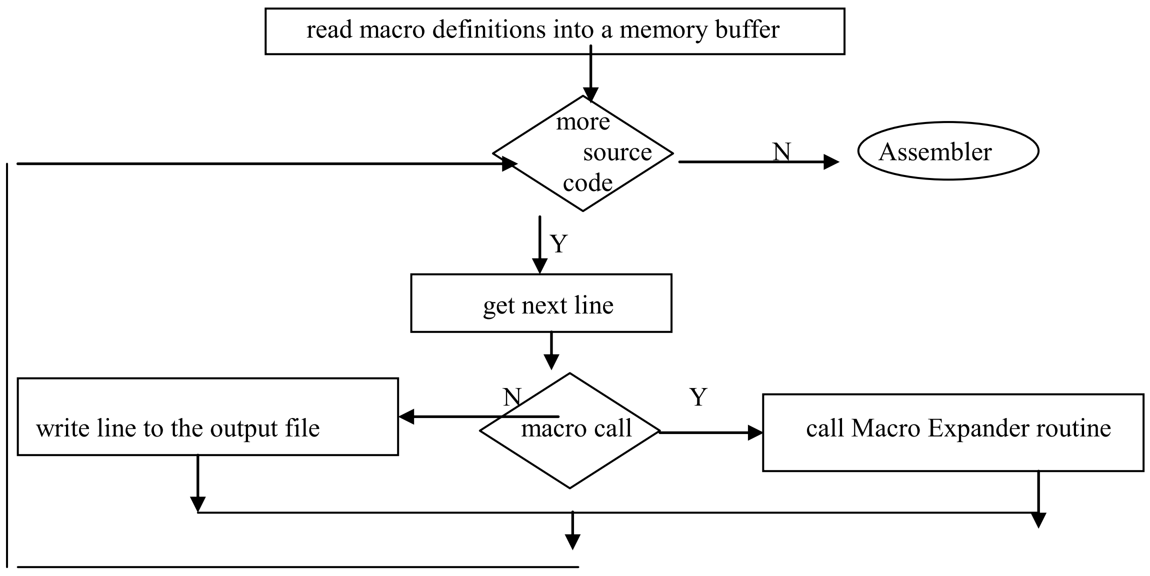
Data Structures

- DEFTAB (definition table)
- Stores the macro definition including
  - macro prototype
  - macro body
- Comment lines are omitted.
- References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.
  - NAMTAB
- Stores macro names
- Serves an index to DEFTAB
  - pointers to the beginning and the end of the macro definition
  - ARGTAB
- Stores the arguments of macro invocation according to their positions in the argument list
- As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body

Check your progress

Explain the reasons of advantage(s) of incorporating the macro processor into pass 1 is/ are:

A. many functions do not have to be implemented twice
B. Functions are combined and it is not necessary to create intermediate files as output from the macro processor and input to the assembler
C. more flexibility is available to the programmer in which he/she may use all the features of the assembler in conjunction with macros


read macro definitions into a memory buffer
more source code
N
Assembler
Y
get next line
N
Y
write line to the output file
macro call
call Macro Expander routine

A macroprocessor front end for an assembler

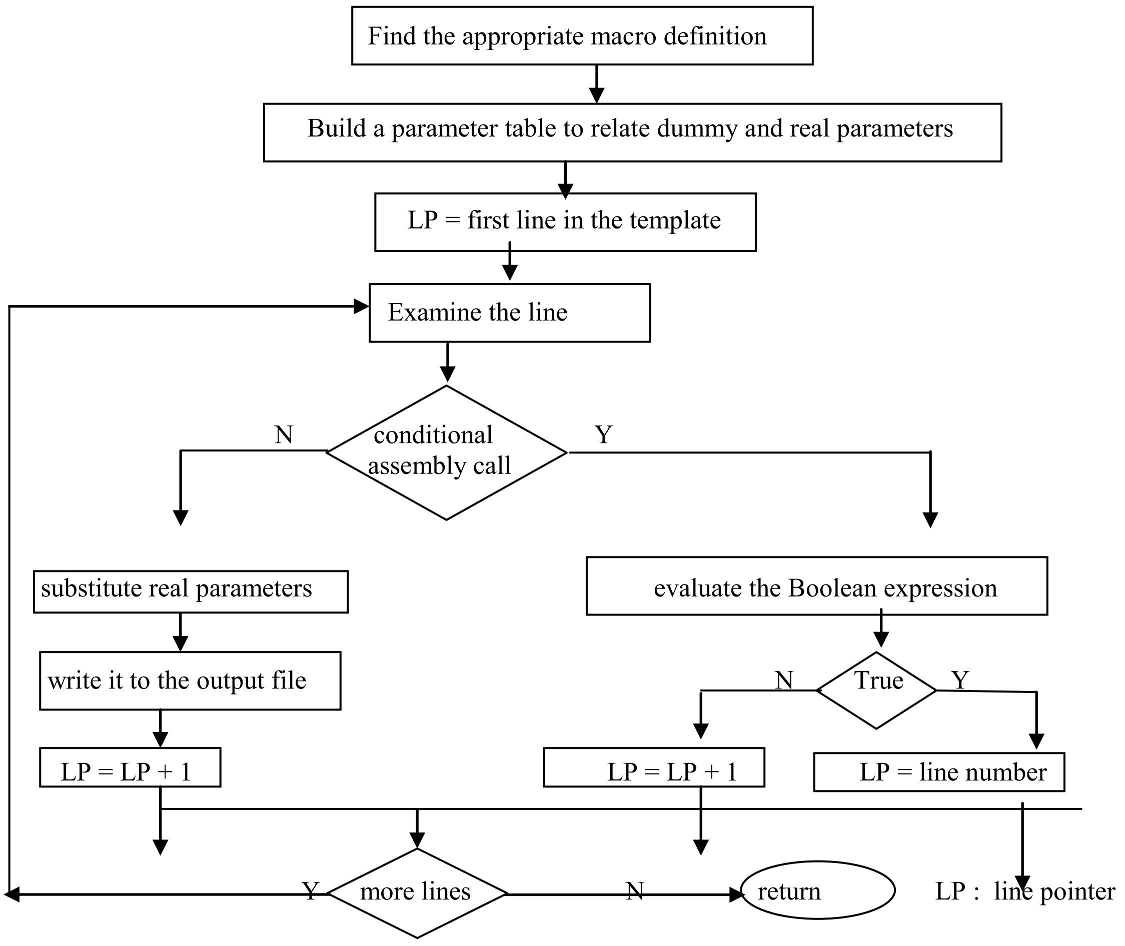

Find the appropriate macro definition
Build a parameter table to relate dummy and real parameters
LP = first line in the template
Examine the line
N
conditional assembly call
Y
substitute real parameters
evaluate the Boolean expression
write it to the output file
N
True
Y
LP = LP + 1
LP = LP + 1
LP = line number
Y
more lines
N
return
LP : line pointer

Figure 5.2: Macro Expander Routine

x:

Source program	*Macro definition*
-------- ALPHA A, 2, C --------	ALPHA MACRO ARG1, ARG2, ARG3 ---------- ENDM

```
                                   …        ALPHA     MACRO    ARG1,     ARG2,     ARG3
ALPHA           A, 2, C            …                    …
                                                       ENDM
```

Parameter table

Dummy parameter	Real parameter
ARG1	A
ARG2	2
ARG3	C

Macro calls within Macros: Using a stack to keep the order of the macro calls, create a parameter table at each macro call, if a dummy parameter appears in the real parameter field of a called macro, search the parameter table of the calling macro and replace it with the appropriate real parameter.

5.4 Summary

In summary, we have learnt the basic functions of macro processors. In order to use macros we have learnt macro definitions and macro expansions, its features such as-labels, nested definitions, recursive invocations, conditional macro processing, keyword parameters. We have learnt data structures and algorithms such as-NAMTAB, DEFTAB, ARGTAB whereas for recursive invocation-STACK.

In this unit we have understood relation between macro processors and assemblers.

5.5 Terminal Questions

1. How are MACROS defined?

2. What are positional parameters? How do they differ from keyword parameters?

3 State and explain the algorithm for an one pass macro processor.

4 What is a general purpose macro processor? Explain

Uttar Pradesh Rajarshi
Tandon Open University

BCA-1.18

Bachelor in Computer Applications

Block

2

System Software

Compilers and Interpreters

Printed and Published by Dr. Arun Kumar Gupta, Registrar, **Uttar Pradesh Rajarshi Tandon Open University, 2019.**

Printed By : Chandrakal Universal Pvt Ltd. 42/7 Jawahar Lal Neharu Road P

UNIT - 6 : COMPILER- LEXICAL ANALYSIS

Introduction to NFA and DFA, Lexical Analysis: Role of a Lexical analyzer, input buffering, specification and recognition of tokens, Finite Automata, Designing a lexical analyzer generator, Pattern matching based on NFA's.

Structure

6.0 INTRODUCTION

To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.

Once decided the possible tokens of language, we need a mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

6.1 Objectives

In this unit you will learn

- Theory of lexical analysis
- Techniques for developing lexical analyzers

6.2 Introduction to NFA and DFA

FINITE AUTOMATA (FA) :

In the quest to transform regular expressions into efficient lexical analyzer programs, we use a stepping stone finite automata. The nondeterministic version ie Nondeterministic finite automata are easy to use but not close to real machines so these can be transformed into deterministic finite automata, which are easily and efficiently executable on normal hardware.

A finite automaton is a machine (in abstract sense) that has a finite number of states and a finite number of transitions between these. A transition between states is usually labelled by a character from the input alphabet (or epsilon).

A finite automaton can be used to decide if an input string is a member in some particular set of strings. To do this, we select one of the states of the automaton as the starting state. We start in this state and in each step, we can do one of the following:

- o Follow an epsilon transition to another state, or
- o Read a character from the input and follow a transition labeled by that character.

When all characters from the input are read, we see if the current state is marked as being accepting. If so, the string we have read from the input is in the language defined by the automaton. We may have a choice of several actions at each step:

- o choose between either an epsilon transition or
- o a transition on an alphabet character, and if there are several transitions with the same symbol, we can choose between these.

This makes the automation nondeterministic, as the choice of action is not determined solely by looking at the current state and input. It may be that some choices lead to an accepting state while others do not.

Finite automata is of two types : -

(a) Nondeterministic finite automata (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and E, the empty string(epsilon), is a possible label.

(b) Deterministic finite automata (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

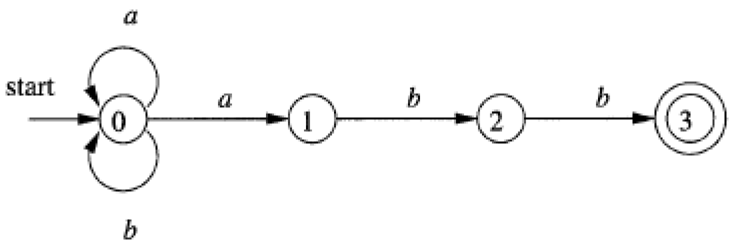
Both deterministic and nondeterministic finite automata are capable of recognizing the same languages. In fact these languages are exactly the same languages, called the regular languages, that regular expressions can describe.

NFA :

A nondeterministic finite automaton (NFA) consists of:

1. A finite set of states S.
2. A set of input symbols C, the input alphabet. We assume that E, which stands for the empty string, is never a member of C.
3. A transition function that gives, for each state, and for each symbol a set of next states.
4. A state so from S that is distinguished as the start state (or initial state).
5. A set of states F, a subset of S, that is distinguished as the accepting states (or final states).

We can represent either an NFA or DFA by a transition graph, where the nodes are states and the labeled edges represent the transition function. There is an edge labeled a from state s to state t if and only if t is one of the next states for state s and input a



DFA :

A deterministic finite automaton (DFA) is a special case of an NFA where:

1. There are no moves on input

2. For each state s and input symbol a, there is exactly one edge out of s labeled a.

If we are using a transition table to represent a DFA, then each entry is a single state. we may therefore represent this state without the curly braces that we use to form sets.

Deterministic Finite Automata

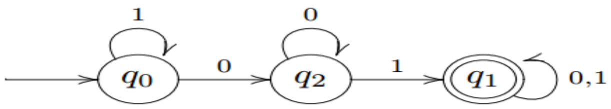
How to present a DFA? With a *transition table*

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

The $\rightarrow$ indicates the *start* state: here q_0

The $*$ indicates the final state(s) (here only one final state q_1)

This defines the following *transition diagram*



While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language, the DFA is a simple, concrete algorithm for recognizing strings. It is fortunate indeed that every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers.

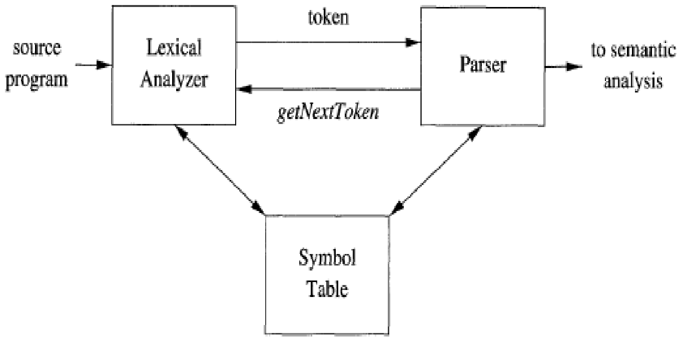
Check your progress 1

What is a full form of DFA and NFA?

6.3 Lexical Analysis

THE ROLE OF THE LEXICAL ANALYZER

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the interactions is suggested in Figure below. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the get Next Token command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.



Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes.

One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).

Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.

In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

ISSUES IN LEXICAL ANALYSIS:

1. Simplicity of design: It is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer. If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.

2. Compiler efficiency is improved: A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

3. Compiler portability is enhanced: Input-device-specific peculiarities can be restricted to the lexical analyzer.

TOKENS, PATTERNS, AND LEXEMES:

When discussing lexical analysis, we use three related but distinct terms:

Token :

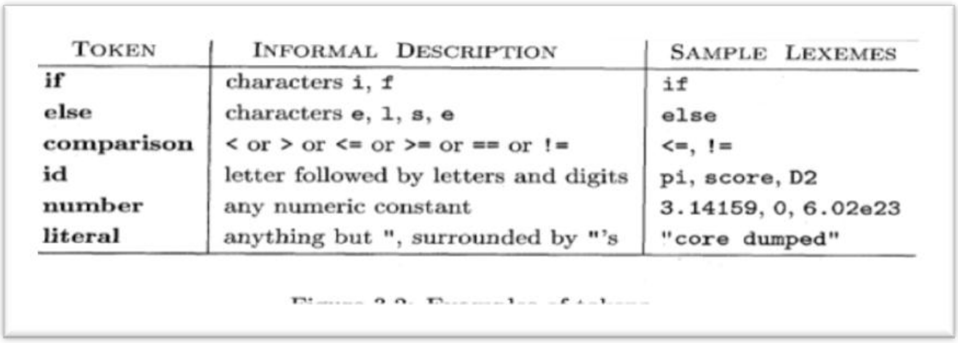
A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

Pattern :

A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Lexeme :

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.



TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

1. Tokens are treated as terminal symbols in the grammar for the source language using boldface names to represent tokens.
2. The lexemes matched by the pattern for the tokens represent the strings of characters in the source program that can be treated together as a lexical unit
3. In most of the programming languages keywords, operators, identifiers, constants, literals and punctuation symbols are treated as tokens.
4. A pattern is a rule describing the set of lexemes that can represent a particular token in the source program.
5. In many languages certain strings are reserved ie their meanings are predefined and cannot be changes by the users
6. If the keywords are not reserved then the lexical analyzer must distinguish between a keyword and a user-defined identifier

ATTRIBUTES FOR TOKENS:

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse. We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. The most important example is the token id, where we need to associate with the token a great deal of information. Normally, information about an identifier - e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) - is kept in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Example : The token names and associated attribute values for the Fortran statement are written below as a sequence of pairs.

<id, pointer to symbol-table entry for E>

< assign-op >

<id, pointer to symbol-table entry for M>

<mult -op>

<id, pointer to symbol-table entry for C>

<exp-op>

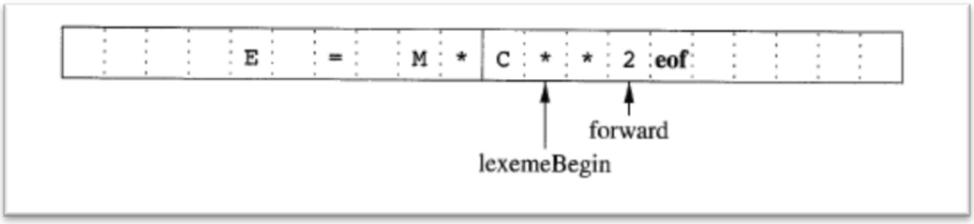
<number , integer value 2 >

Note that in certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value. In this example, the token number has been given an integer-valued attribute. In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for number a pointer to that string.

INPUT BUFFERING :

During lexical analyzing, to identify a lexeme, it is important to look ahead at least one additional character. Specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character

An important scheme involves two buffers that are alternatively reloaded



Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character represented by of marks the end of the source file and is different from any possible character of the source program.

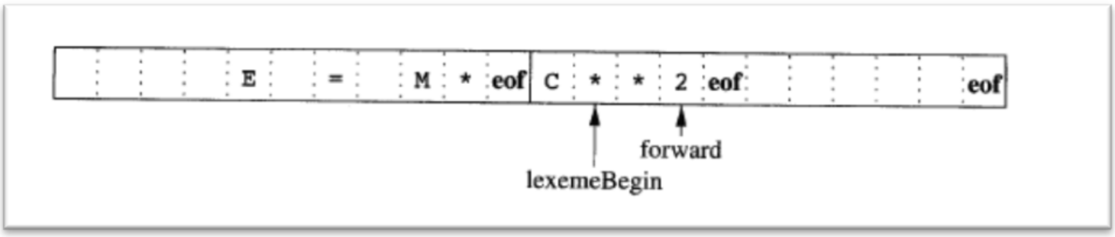
Two pointers to the input are maintained:

I. Pointer lexemeBegin, marks the beginning of the current lexeme, whose extent we are attempting to determine.

2. Pointer forward scans ahead until a pattern match is found; Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set to the character immediately after the lexeme just found. In Fig shown, we see forward has passed the end of the next lexeme, ** (the Fortran exponentiation operator), and must be retracted one position to its left.

Sentinels

we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof. Note that eof retains its use as a marker for the end of the entire input. Any eof that appears other than at the end of a buffer means that the input is at an end. Figure below summarizes the algorithm for advancing forward. Notice how the first test, which can be part of a multiway branch based on the character pointed to by forward, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.



SPECIFICATION OF TOKENS :

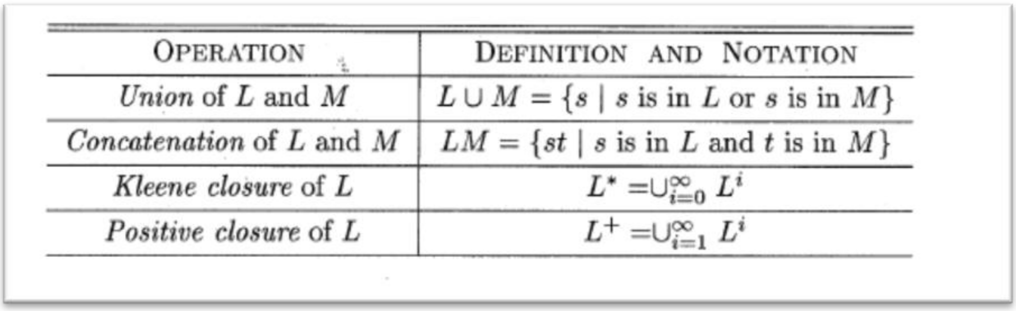
Regular languages are an important notation for specifying lexeme patterns

Strings and Languages:

- An alphabet is any finite set of symbols ex: Letters, digits and punctuation
- The set {01) is the binary alphabet
- A string over an alphabet is a finite sequence of symbols drawn from the alphabet
- The length of the string represented as |s|, is the number of occurrences of symbols in s.
- The empty string denoted as € is the string of length 0
- A language is any countable set of strings over some fixed alphabet ex: abstract languages
- If x and y are strings then the concatenation of x and y denoted by xy is the string formed by appending y to x. For eg if x=cse and y=department , then xy=cse department .

Operation on Languages :

In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in Fig. shown. Union is the familiar operation on sets. The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them. The (Kleene) closure of a language L, denoted L*, is the set of strings you get by concatenating L zero or more times. Finally, the positive closure, denoted L+, is the same as the Kleene closure, but without the term L_o. That is, E will not be in L+ unless it is in L itself.



OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Here are some other languages that can be constructed from languages L and D, using the operators shown in Fig:

1. L U D is the set of letters and digits - strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. LD is the set df 520 strings of length two, each consisting of one letter followed by one digit.
3. L4 is the set of all 4-letter strings.
4. L* is the set of ail strings of letters, including e, the empty string.
5. L(LUD)* is the set of all strings of letters and digits beginning with a letter.
6. D+ is the set of all strings of one or more digits.

REGULAR EXPRESSIONS :

Suppose we wanted to describe the set of valid C identifiers. It is almost exactly the language described in item (5) above; the only difference is that the underscore is included among the letters.

We were able to describe identifiers by giving names to sets of letters and digits and using the language operators union, concatenation, and closure. This process is so useful that a notation called regular expressions has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet. In this notation, if letter is established to stand for any letter or the underscore, and digit is established to stand for any digit, then we could describe the language of C identifiers by:

letter(letter|digit)*

The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of," and the juxtaposition of letter with the remainder of the expression signifies concatenation. The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language L(r), which is also defined recursively from the languages denoted by r's subexpressions. Here are the rules that define the regular expressions over some alphabet ∑ and the languages that those expressions denote.

There are rules that form the regular expression :

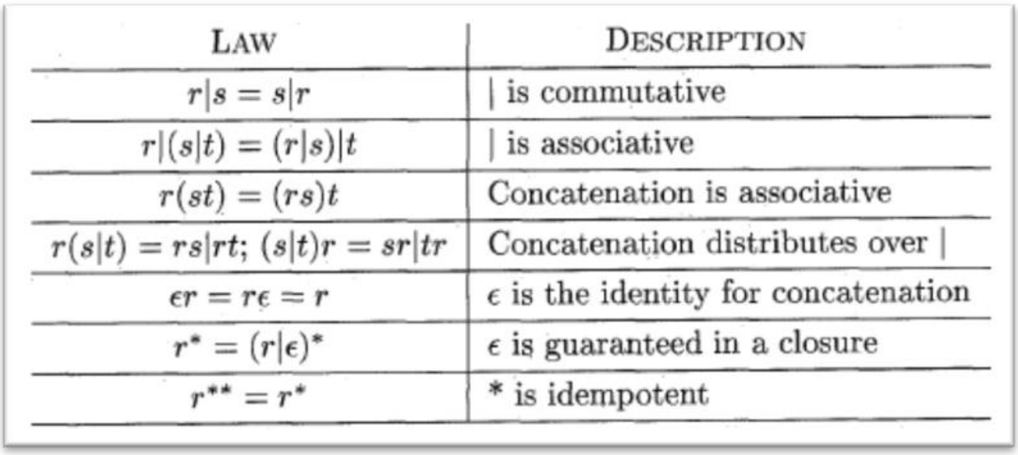
1. If ϵ is a regular expression, and L(E) is {ϵ} , that is, the language whose sole member is the empty string.
2. If a is a symbol in ∑, then a is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with a in its one position.

   Suppose r and s are regular expressions denoting languages L(r) and L(s), respectively.

3. (r)|(s) is a regular expression denoting the language L(r)U L(s).
2. (r)(s) is a regular expression denoting the language L(r) L(s).
3. (r)* is a regular expression denoting (L(r)) * .
4. (r) is a regular expression denoting L(r).
5. The unary operator * has highest precedence and is left associative.
6. Concatenation has second highest precedence and is left associative.
7. | has lowest precedence and is left associative.

Under these conventions, for example, we may replace the regular expression (a)|((b)*(c)) by a|b*c. Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.

Figure below shows some of the algebraic laws that hold for arbitrary regular expressions r, s, and t.



LAW	DESCRIPTION
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over |
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

REGULAR DEFINITIONS

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If ∑ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$$\begin{array}{ccc} t_1 & \to & r_1 \\ t_2 & \to & r_2 \\ & \cdots & \\ t_n & \to & r_n \end{array}$$

Where :

1. Each t_i is a new symbol, not in $\sum$ and not the same as any other of the t's

2. Each r_i is a regular expression over the alphabet $\sum$ U $\{t_l, t_2, . . . , t_{i-l})$.

   By restricting r_i to $\sum$ and the previously defined t's, we avoid recursive definitions, and we can construct a regular expression over $\sum$ alone, for each r_i.

6.4 Pattern matching based on NFA's

RECOGNITION OF TOKENS :

In the previous section we learned how to express patterns using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns. Our discussion will make use of the following running example.

stmt -> **if** *expr* **then** *stmt*

I **if** *expr* **then** *stmt* **else** *stmt*

I ***E.***

expr -> *term* **relop** *term*

I *term*

term -> **id**

I **number**

The patterns for these tokens are described using regular definitions

$$\begin{array}{rcl}
\mathit{digit} & \to & [\texttt{0-9}] \\
\mathit{digits} & \to & \mathit{digit}^{+} \\
\mathit{number} & \to & \mathit{digits}\ (\texttt{.}\ \mathit{digits})?\ (\ \texttt{E}\ [\texttt{+-}]?\ \mathit{digits}\ )? \\
\mathit{letter} & \to & [\texttt{A-Za-z}] \\
\mathit{id} & \to & \mathit{letter}\ (\ \mathit{letter} \mid \mathit{digit}\ )^{*} \\
\mathit{if} & \to & \texttt{if} \\
\mathit{then} & \to & \texttt{then} \\
\mathit{else} & \to & \texttt{else} \\
\mathit{relop} & \to & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}$$

For this language, the lexical analyzer will recognize the keywords if , then, and else, as well as lexemes that match the patterns for relop, id, and number.

Token we is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. It is the following token that gets returned to the parser.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

TRANSITION DIAGRAM :

As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "transition diagrams.

State:

Transition diagrams have a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

Edges:

Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols. If we are in some states, and the next input symbol is a, we look for an edge out of states labeled

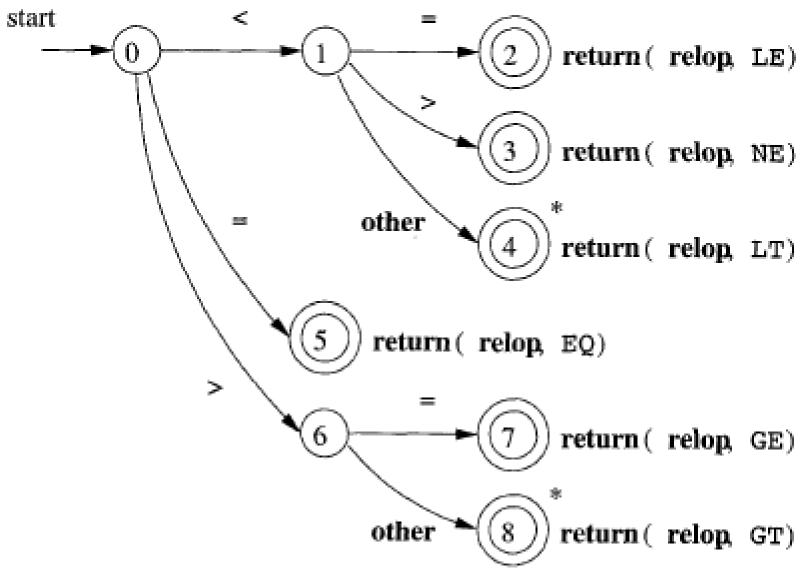
by a (and perhaps by other symbols, as well). If we find such an edge, we advance the forward pointer arid enter the state of the transition diagram to which that edge leads.

We shall assume that all our transition diagrams are deterministic, meaning that there is never more than one edge out of a given state with a given symbol among its labels. We shall relax the condition of determinism, making life much easier for the designer of a lexical analyzer, although trickier for the implementer.

Some important conventions about transition diagrams are :

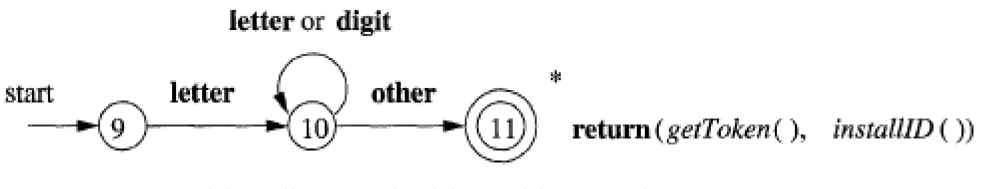
1. Certain states are said to be accepting, or final. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the LexemeBegin and forward pointers. We always indicate an accepting state by a double circle, and if there is an action to be taken - typically returning a token and an attribute value to the parser - we shall attach that action to the accepting state.
2. In addition, if it is necessary to retract the forward pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract forward by more than one position, but if it were, we could attach any number of *'s to the accepting state.
3. One state is designated the start state, or initial state; it is indicated by an edge, labeled "start ," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

Example : Transition diagrams for relational operations.



RECOGNIZING KEYWORDS AND IDENTIFIERS :

Recognizing keywords and identifiers presents a problem. Usually, keywords like if or then are reserved (as they are in our running example), so they are not identifiers even though they look like identifiers. Thus, although we typically use a transition diagram like that of Fig shown to search for identifier lexemes, this diagram will also recognize the keywords if, then, and else of our running example.



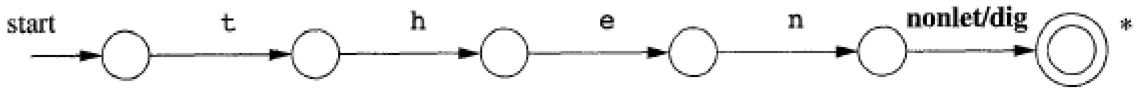
A transition diagram for id's and keywords

There are two ways that we can handle reserved words that look like identifiers:

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this

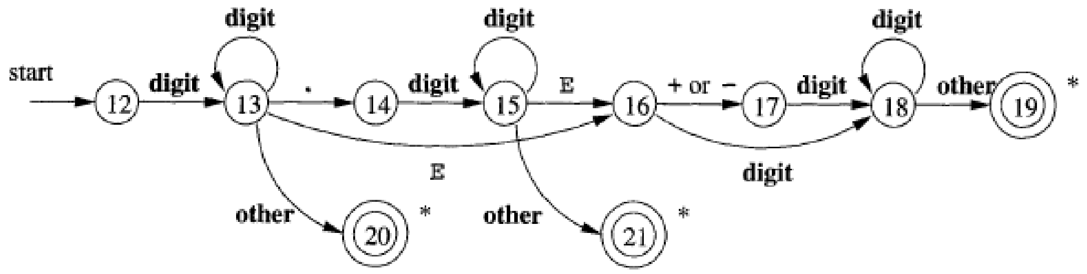
method is in use in Fig shown. When we find an identifier, a call to installID places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is id. The function get Token examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents - either id or one of the keyword tokens that was initially installed in the table.

2. Create separate transition diagrams for each keyword; an example for the keyword then is shown in Fig. Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token then in situations where the correct token was id, with a lexeme like then extvalue that has then as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when the lexeme matches both patterns.



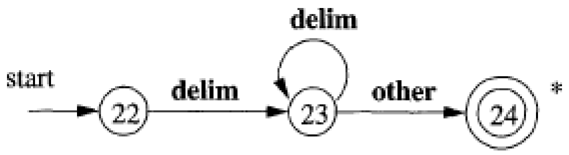
Hypothetical transition diagram for the keyword then

The transition diagram for a token number is given below



- If a dot is seen we have an optional fraction
- State 14 is entered and we look for one or more additional digits
- State 15 is used for this purpose
- If we see an E , we have an optional exponent , states 16 through 19 are used to recognize the exponent value
- In the state 15 , if we see anything other than E or digit , then 21 is the end of the accepting state

A transition diagram for whitespace is given below



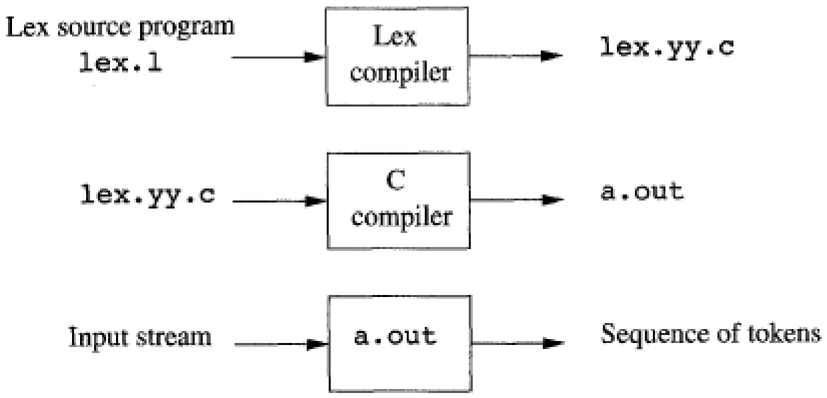
In the diagram we look for one or more whitespace characters represented by delim in the diagram typically these characters would be blank, tab, newline

6.5 Designing a lexical analyzer generator-LEX

In this section, we introduce a tool called Lex, or in a more recent implementation Flex, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens. The input notation for the Lex tool is referred to as the *Lex language* and the tool itself is the *Lex compiler.* Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex . yy . c, that simulates this transition diagram. The mechanics of how this translation from regular expressions to transition diagrams occurs is the subject of the next sections; here we only learn the Lex language.

Use of Lex :

An input file lex1 is written in the lex language and describes the lexical analyzer to be generated . The Lex compiler transforms lex1 to a c program in a file that is always named lex.yy.c



STRUCTURE OF A LEX PROGRAM :

A Lex program has the following form :

declarations

%%

translation rules

%%

auxiliary functions

The declarations section includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions

The translation rules each have the form

Pattern {Action)

Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C.

The third section holds whatever additional functions are used in the actions.

Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

The lexical analyzer created by Lex behaves as follows :

1. When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns Pi.

2. It then executes the associated action Ai. Typically, Ai will return to the parser, but if it does not (e.g., because Pi describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser.

3. The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable yylval to pass additional information about the lexeme found, if needed.

LEX PROGRAM FOR TOKEN :

```
%{
/* definitions of manifest constants

LT, LE, EQ, NE, GT, GE,

IF, THEN, ELSE, ID, NUMBER, RELOP */

%}

/* regular definitions */

delim [ \t\nl

ws (delim)+

letter [A-Za-z]

digit [o-9]

id {letter} {(letter) | {digit})*

number {digit)+ (\ . {digit}+)? (E [+-] ?{digit}+)?

%%

{ws} (/* no action and no return */)

if{return(IF) ; }

then{return(THEN) ; }

else{return(ELSE) ; }

{id}{yylval = (int) installID(); return(ID);}

{number}{yylval = (int) installNum() ; return(NUMBER) ; }

"<"{yylval = LT; return(REL0P); }

"<="{yylval = LE; return(REL0P); }

"="{yylval = EQ ; return(REL0P); }

"<>"{yylval = NE; return(REL0P);}

">"{yylval = GT; return(REL0P);}

">="{yylval = GE; return(REL0P);}

%%

int installID0 {/* function to install the lexeme, whose
first character is pointed to by yytext,
and whose length is yyleng, into the
symbol table and return a pointer
thereto */
}
int installNum() {/* similar to installID, but puts numerical
constants into a separate table */
}
```

In the declarations section we see a pair of special brackets, %(and %). Anything within these brackets is copied directly to the file lex.yy.c, and is not treated as a regular definition. The manifest constants are placed inside it. Also the languages occur as a sequence of regular definitions.

Regular definitions that are used in later definitions or in the patterns of the translation rules are surrounded by curly braces. Thus, for instance, delim is defined to be a shorthand for the character class consisting of the blank, the tab, and the newline; the latter two are represented, as in all UNIX commands, by backslash followed by t or n, respectively

In the auxiliary-function section, we see two such functions, installID( )and installNum(). Like the portion of the declaration section that appears between everything in the auxiliary section is copied directly to file lex.yy.c, but may be used in the actions.

First, ws, an identifier declared in the first section, has an associated empty action. If we find whitespace, we do not return to the parser, but look for another lexeme. The second token has the simple regular expression pattern if. Should we see the two letters if on the input, and they are not followed by another letter or digit (which would cause the lexical analyzer to find a longer prefix of the input matching the pattern for id), then the lexical analyzer consumes these two letters from the input and returns the token name IF, that is, the integer for which the manifest constant IF stands. Keywords then and else are treated similarly. The fifth token has the pattern defined by id. Note that, although keywords like i f match this pattern as well as an earlier pattern, Lex chooses whichever pattern is listed first in situations where the longest matching prefix matches two or more patterns. The action taken when id is matched is given as follows:

1. Function installID( ) is called to place the lexeme found in the symbol table.
2. This function returns a pointer to the symbol table, which is placed in global variable yylval, where it can be used by the parser or a later component of the compiler. Note that installID () has available to it two variables that are set automatically by the lexical analyzer that Lex generates:
   (a) yytext is a pointer to the beginning of the lexeme
   (b) yyleng is the length of the lexeme found.
3. The token name ID is returned to the parser.

Check your progress

What is the purpose of LEX?

6.6 Summary

In this unit, we have learnt about abstract machines-FA, NFA,DFA which recognizes regular expression. LEX a lexical analyzer generator, generates lexical analyzer which accepts tokens of Programming Language.

6.7 Terminal Questions

1. What is a lexical analyzer?
2. Differentiate between NFA and DFA.
3. Explain how LEX is used for developing lexical analyzer?
4. Can a DFA simulate NFA?
5. What is the regular expressions that denotes a language comprising all possible strings of even length over the alphabet (0, 1)?

UNIT 7 : COMPILER- SYNTAX ANALYSIS

Syntax Analysis: Role of Parser, Top-down parsing, recursive descent and predictive parsers (LL), Bottom-Up parsing, Operator precedence parsing, LR, SLR and LALR parsers.(First and follow technique for generating a parse table is to be taught), Phases of the Compiler, Aspects of compilation, Memory allocation. Compilation of expressions and control structures.

Structure

7.0 Introduction

Syntax analyzer verifies whether the sequence of tokens (output of lexical analyzer) are in accordance with the grammar of the language. The analysis of program according to the syntax of programming language in the compiler is called parsing. The output of parser is a form a tree known as parse tree.

7.1 Objectives

- To study Syntax Analysis(SA): second-phase of compilation
- Role of Parser in SA
- Types of Parsers
- Construction of Parsing Tables

7.2 Role of Parser

Parser for any grammar is program that takes as input string w (obtain as tokens from the lexical analyzer) and produces as output either a parse tree for w, if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar. The goal of the parser is to determine the syntactic validity of a source string is valid; a tree is built for use by the subsequent phases of the computer. The parse tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string. Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementary subtree:

1. By deriving a string from a non-terminal or
2. By reducing a string of symbol to a non-terminal.

The two types of parsers employed in compilers are:

a. Top down parser: which build parse trees from top(root) to bottom(leaves)
b. Bottom up parser: which build parse trees from leaves and work up the root.

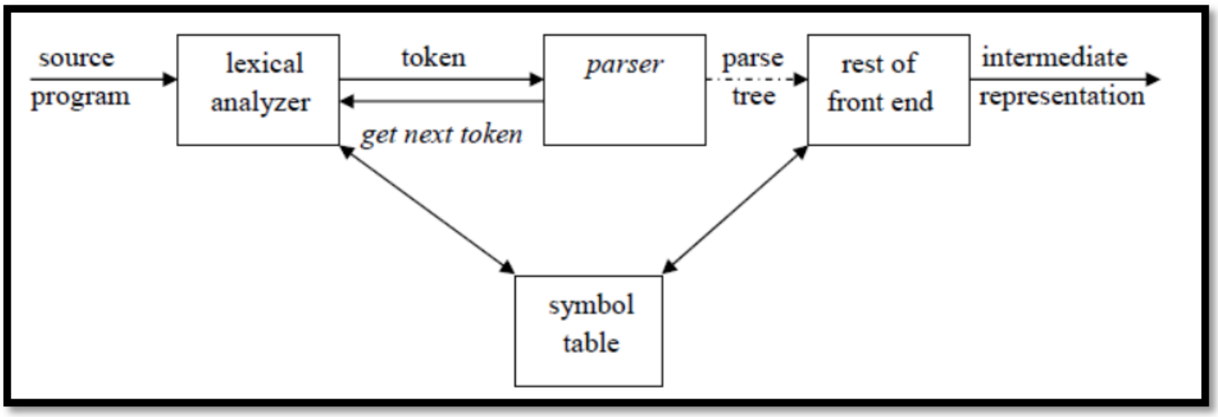


Fig : Role of parser in compiler model.

7.3 Top-down parsing

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

1. Backtracking- ex. recursive descent parsers
2. Non-backtracking parsers- ex. predictive parsers

Backtracking parser includes, 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.

The parser uses symbol-look-ahead and an approach called top-down parsing without backtracking. Top-down parsers check to see if a string can be generated by a grammar by creating a parse tree starting from the initial symbol and working down.

Top-down parsing can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing :

1. Recursive descent parsing
2. Predictive parsing

7.4 Recursive descent parser

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 : For the following grammar and the input string ab.

S -> abA

A -> cd | c | ϵ

The recursive descent parser starts by constructing a parse tree representing S -> abA, matched ab part of input. On the tree expand with the production A -> cd, fails to match the string ab. The parser backtracks and then tries with the alternative A -> c again the parse tree does not match the string ab. So the parser backtracks again and tries the last alternative A->ϵ. This time, it finds a match. Thus, the parsing is complete and successful. The main pitfall of recursive parsing are :

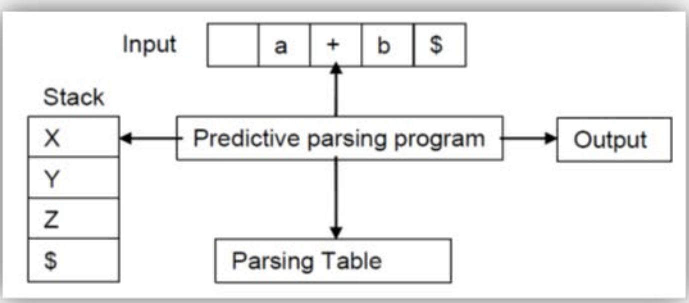
1. Left recursion, where the grammar has production of the form A->Aϵ leads an infinite loop
2. Backtracking, when there is one alternative in the production to be tried while parsing the input string.

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

7.5 Predictive parsers (LL)

Top-down 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 alternatives by looking ahead 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 nonterminal.

A non-recursive predictive parser can be obtained by pushing into stack 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.



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 (ie the stack bottom). Initially the stack contains the start symbol of the grammar on the top of $. The parsing table is constructed using FIRST and FOLLOW function. Steps to construct the predictive parse table for grammar G-

1. Eliminate left recursion in G.
2. Perform left factoring in G.
3. Find FIRST and FOLLOW in G.
4. Construct predictive parsing table.
5. Verify if the parser can accept the input string.

This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for leftmost-derivation, and k indicates k-symbol look ahead. Therefore, a parser using the single symbol look-ahead method and top-down parsing without backtracking is called LL(1) parser.

A usual implementation of an LL(1) parser is:

1. initialize its data structures,
2. get the lookahead token by calling scanner routines, and
3. call the routine that implements the start symbol.

FIRST AND FOLLOW

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or e can be added to any FIRST set.

1. If X is terminal, then FIRST(X) is {X}.
2. If X->e is a production, then add e to FIRST(X).
3. If X is nonterminal and X->Y1Y2...Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi) and e is in all of FIRST(Y1),...,FIRST(Yi-1) that is, Y1.......Yi-1=*>e. If e is in FIRST(Yj) for all j=1,2,...,k, then add e to FIRST(X). For example, everything in FIRST(Yj) is surely in FIRST(X). If y1 does not derive e, then we add nothing more to FIRST(X), but if Y1=*>e, then we add FIRST(Y2) and so on.

To compute the FIRST(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place $ in FOLLOW(S), where S is the start symbol and $ in the input right endmarker.
2. If there is a production A=>aBs where FIRST(s) except e is placed in FOLLOW(B).
3. If there is aproduction A->aB or a production A->aBs where FIRST(s) contains e, then

everything in FOLLOW(A) is in FOLLOW(B).

Example :

Consider the following grammar :

E → E+T | T
T → T*F | F
F → (E) | id

After eliminating left-recursion the grammar is

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

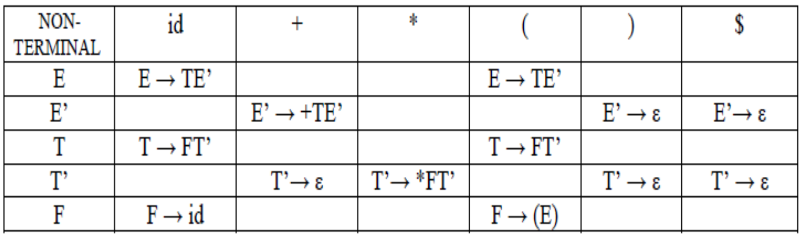
Computing First( ) :

FIRST(E) = { (, id}
FIRST(E') ={+ , ε }
FIRST(T) = { (, id}
FIRST(T') = {*, ε }
FIRST(F) = { (, id }

Computing Follow( ):

FOLLOW(E) = { $,) }
FOLLOW(E') = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T') = { +, $,) }
FOLLOW(F) = {+, * , $,) }

Predictive parsing table :



NON-TERMINAL	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E'→ ε
T	T → FT'			T → FT'		
T'		T'→ ε	T'→ *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Stack implementation:

stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E → TE'
$E'T'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T → FT'
$E'T'id	id*id $	F → id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

LL(1) grammar :

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar:

S → iEtS | iEtSeS | a
E → b

After eliminating left factoring, we have

S → iEtSS' | a
S'→ eS | ε
E → b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) = { i, a }
FIRST(S') = {e, ε }
FIRST(E) = { b}
FOLLOW(S) = { $,e }
FOLLOW(S') = { $,e }
FOLLOW(E) = {t}

Parsing table:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → b				

Since there are more than one production, the grammar is not LL(1) grammar.

ERROR RECOVERY IN PREDICTIVE PARSING

The stack of a nonrecursive predictive parser makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. We shall therefore refer to symbols on the parser stack in the following discussion. An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and the parsing table entry M[A,a] is empty.

Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of

synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice. Some heuristics are as follows

As a starting point, we can place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.

It is not enough to use FOLLOW(A) as the synchronizingset for A. Fo example , if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchica structure on constructs in a language; e.g., expressions appear within statement, which appear within bblocks,and so on. We can add to the synchronizing set of a lower construct the symbols that begin higher constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generaitn expressions.

If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

If a nonterminal can generate the empty string, then the production deriving e can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.

If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

Check your Progress

1.Life 79uehepeveneparsing techniques.

2.With drawback of recursive parching technique.

7.6 BOTTOM-UP PARSING

In bottom-up parsing the parse tree for an input string is constructed beginning at the bottom nodes (leaves) and going towards the root ie 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 reduced each time until the start non-terminals is reached. 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.

Consider the following grammar and given input string abbcde:

S → aABe

A → Abc | b

B → d

Handle	**REDUCTION (LEFTMOST)**
a**b**bcde	(A → b)
a**Abc**de	(A → Abc)
aA**d**e	(B → d)
aABe	(S → aABe)

The reductions trace out the right-most derivation in reverse ie construction of bottom-up parse tree.

Handle : A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation. In the above derivation the underlined substrings are called handles.

Handle pruning : A rightmost derivation in reverse can be obtained by handle pruning ie if w is a sentence or string of the grammar at hand, then $w = \gamma_n$, where γ_n is the nth right sentinel form of some rightmost derivation.

Actions in shift-reduce parser :

i. shift – The next input symbol is shifted onto the top of the stack.

ii. reduce – The parser replaces the handle within a stack with a non-terminal.

iii. accept – The parser announces successful completion of parsing.

iv. error–The parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

1. Shift-reduce conflict : The parser cannot decide whether to shift or to reduce.
2. Reduce-reduce conflict : The parser cannot decide which of several reductions to make.

1. Shift-reduce conflict :

Example :

Consider the grammar: E→E+E | E*E | id and input id+id*id

Stack	Input	Action	Stack	Input	Action
$ E+E	*id $	Reduce-by	$E+E		*id $ Shift
		E→E+E			
$ E	*id $	Shift	$E+E*	id $	Shift
$ E*	**id** $	Shift	$E+E*id	$	Reduce by
					E→id
$E***id**	$	Reduce by	$E+E*E	$	Reduce by
		E→id			E→E*E
$E*E	$	Reduce by	$E+E	$	Reduce by
		E→E*E			E→E*E
$E			$E		

2. Reduce-reduce conflict :

Consider the grammar: M → R+R | R+c | R R → c and input c+c

Stack	Input	Action	Stack	Input	Action
$	**c+c $**	**Shift**	**$**	**c+c $**	**Shift**
$ c	**+c $**	**Reduce by**	**$ c**	**+c $**	**Reduce by**
		R→c			**R→c**
$ R	**+c $**	**Shift**	**$ R**	**+c $**	**Shift**
$ R+	**c $**	**Shift**	**$ R+**	**c $**	**Shift**
$ R+c	**$**	**Reduce by**	**$ R+c**	**$**	**Reduce by**
		R→c			**M→R+c**
$ R+R	**$**	**Reduce by**	**$ M**	**$**	
				M→R+R	
$ M	**$**				

Viable prefixes:

i. α is a viable prefix of the grammar if there is w such that αw is a right sentinel form.

ii. The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.

iii. The set of viable prefixes is a regular language.

7.7 Operator precedence parsing

An efficient way of constructing shift-reduce parser is called operator-precedence parsing. Operator precedence parser can be constructed from a grammar called Operator-grammar. These grammars have the property that no production on right side is ε or has two adjacent nonterminals. Thus operator grammar has two characteristics:

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

Example:

Consider the grammar(NOT a operator grammar):

E → EAE | (E) | -E | id

A → + | - | * | / | ↑

Since the right side EAE has three consecutive non-terminals, the grammar can be written as follows (Operator grammar):

E → E+E | E-E | E*E | E/E | E↑E | -E | id

Operator precedence relations:

There are three disjoint precedence relations namely

<. -less than

= -equal to

.> - greater than

The relations give the following meaning:

a <. b – a yields precedence to b

a = b – a has the same precedence as b

a .> b – a takes precedence over b

The operator precedence parser uses operator precedence relationship table shown below for making decisions to identify the correct handle in the reduction step

TABLE : Operator-precedence relations

	+	-	*	/	↑	id	(	)	$
+	.>	.>	<.	<.	<.	<.	<.	.>	.>
-	.>	.>	<.	<.	<.	<.	<.	.>	.>
*	.>	.>	.>	.>	<.	<.	<.	.>	.>
/	.>	.>	.>	.>	<.	<.	<.	.>	.>
↑	.>	.>	.>	.>	<.	<.	<.	.>	.>
id	.>	.>	.>	.>	·.>			.>	.>
(	<.	<.	<.	<.	<.	<.	<.	=	
)	.>	.>	.>	.>	.>			.>	.>
$	<.	<.	<.	<.	<.	<.	<.		

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.

Stack implementation of operator precedence parsing :

Operator precedence parsing uses a stack and precedence relation table. It is a shift-reduce parsing containing all four actions shift, reduce, accept and error.

The initial configuration of an operator precedence parsing is

STACK INPUT

$ w $

where w is the input string to be parsed.

Example :

Consider the grammar E → E+E | E-E | E*E | E/E | E↑E | (E) | id. Input string is id+id*id .The implementation is as follows:

STACK	INPUT	COMMENT
$	<· id+id*id $	shift id
$ id	·> +id*id $	pop the top of the stack id
$	<· +id*id $	shift +
$ +	<· id*id $	shift id
$ +id	·> *id $	pop id
$ +	<· *id $	shift *
$ + *	<· id $	shift id
$ + * id	·> $	pop id
$ + *	·> $	pop *
$ +	·> $	pop +
$	$	accept

Advantages of operator precedence parsing:

1. It is easy to implement.
2. Once an operator precedence relation is made between all pairs of terminals of a grammar, the grammar can be ignored. The grammar is not referred anymore during implementation.

Disadvantages of operator precedence parsing:

1. It is hard to handle tokens like the minus sign (-) which has two different precedence.
2. Only a small class of grammar can be parsed using operator-precedence parser.

7.8 LR PARSERS

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. In LR(k), the 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and the 'k' for the number of input symbols. When 'k' is omitted, it is assumed to be 1.

Advantages of LR parsing:

i. It recognizes virtually all programming language constructs for which CFG can be written.
ii. It is an efficient non-backtracking shift-reduce parsing method.
iii. A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
iv. It detects a syntactic error as soon as possible.

Drawbacks of LR method :

It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

1. SLR- Simple LR: Easiest to implement, least powerful.
2. CLR- Canonical LR: Most powerful, most expensive.
3. LALR- Look-Ahead LR: Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

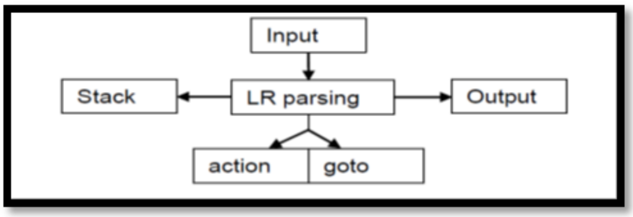
It consists of : an input, an output, a stack, a driver program, and a parsing table that has two parts (action and goto).

i. The driver program is the same for all LR parser.
ii. The parsing program reads characters from an input buffer one at a time.
iii. The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a state.
iv. The parsing table consists of two parts : action and goto functions.

Action : The parsing program determines s_m, the state currently on top of stack, and a_i, the current input symbol. It then consults action[s_m,a_i] in the action table which can have one of four values :

1. shift s, where s is a state,
2. reduce by a grammar production $A \rightarrow \beta$,
3. accept, and
4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.



LR Parsing algorithm :

Input : An input string w and an LR parsing table with functions action and goto for grammar G.

Output : If w is in L(G), a bottom-up-parse for w; otherwise, an error indication.

Method : Initially, the parser has s_0 on its stack, where s_0 is the initial state, and w$ in the input buffer. The parser then executes the following program :

```
set ip to point to the first input symbol of w$;
repeat forever begin
        let s be the state on top of the stack and
        a the symbol pointed to by ip;
if action[s, a] = shift s' then begin
        push a then s' on top of the stack;
        advance ip to the next input symbol
        end
else if action[s, a] = reduce A→β then begin
        pop 2* | β | symbols off the stack;
        let s' be the state now on top of the stack;
        push A then goto[s', A] on top of the stack;
        output the production A→ β
end
else if action[s, a] = accept then
```

```
return
else error( )
end
```

7.9 First and follow technique for parse table

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute goto(I,X), where, I is set of items and X is grammar symbol.

LR(0) items :

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :

A → . XYZ

A → X . YZ

A → XY . Z

A → XYZ .

Closure operation :

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

1. Initially, every item in I is added to closure(I).
2. If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation :

Goto(I, X) is defined to be the closure of the set of all items [A→ αX . β] such that[A→ α . Xβ] is in I.

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G'
2. Construct the canonical collection of set of items C for G'
3. Construct the parsing action function action and goto using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G'

Output : The SLR parsing table functions action and goto for G'

Method :

1. Construct C = {I_0, I_1, …. I_n}, the collection of sets of LR(0) items for G'.
2. State i is constructed from Ii.. The parsing functions for state i are determined as follows:
   (a) If [A→α·aβ] is in Ii and goto(I_i,a) = I_j, then set action[i,a] to "shift j". Here a must be terminal.
   (b) If [A→α·] is in I_i , then set action[i,a] to "reduce A→α" for all a in FOLLOW(A).

(c) If [S'→S.] is in Ii, then set action[i,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The goto transitions for state i are constructed for all non-terminals A using the rule: If goto(I_i,A) = I_j, then goto[i,A] = j.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing [S'→.S].

Example for SLR parsing :

Construct SLR parsing for the following grammar :

G : E → E + T | T

T → T * F | F

F → (E) | id

The given grammar is :

G : E → E + T ------ (1)

E →T ------ (2)

T → T * F ------ (3)

T → F ------ (4)

F → (E) ------ (5)

F → id ------ (6)

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar :

E' → E /* making G into augmented G'

E → E + T

E → T

T → T * F

T → F

F → (E)

F → id

Step 2 : Find LR (0) items.

I_0 : E' → . E

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → . id

GOTO (I_0 , E) GOTO (I_4 , id)

$I_1 : E' \rightarrow E .$ $I_5 : F \rightarrow id .$

$E \rightarrow E . + T$

GOTO (I_6 , T)

GOTO (I_0 , T) $I_9 : E \rightarrow E + T .$

$I_2 : E \rightarrow T .$ $T \rightarrow T . * F$

$T \rightarrow T . * F$

GOTO (I_6 , F)

GOTO (I_0 , F) $I_3 : T \rightarrow F .$

$I_3 : T \rightarrow F .$

GOTO (I_6 , ( )

$I_4 : F \rightarrow (. E)$

GOTO (I_0 , ( ) GOTO (I_6 , id)

$I_4 : F \rightarrow (. E)$ $I_5 : F \rightarrow id .$

$E \rightarrow . E + T$

$E \rightarrow . T$ GOTO (I_7 , F)

$T \rightarrow . T * F$ $I_{10} : T \rightarrow T * F .$

$T \rightarrow . F$

$F \rightarrow . (E)$ GOTO (I_7 , ( )

$F \rightarrow . id$ $I_4 : F \rightarrow (. E)$

$E \rightarrow . E + T$

GOTO (I_0 , id) $E \rightarrow . T$

$I_5 : F \rightarrow id .$ $T \rightarrow . T * F$

$T \rightarrow . F$

GOTO (I_1 , +) $F \rightarrow . (E)$

$I_6 : E \rightarrow E + . T$ $F \rightarrow . id$

$T \rightarrow . T * F$

$T \rightarrow . F$ GOTO (I_7 , id)

$F \rightarrow . (E)$ $I_5 : F \rightarrow id .$

$F \rightarrow . id$

GOTO (I_8 ,))

GOTO (I_2 , *) I11 : $F \rightarrow (E) .$

$I_7 : T \rightarrow T * . F$

$F \rightarrow . (E)$ GOTO (I_8 , +)

$F \rightarrow . id$ $I_6 : E \rightarrow E + . T$

$T \rightarrow . T * F$

GOTO (I_4 , E) $T \rightarrow . F$

$I_8 : F \rightarrow (E .)$ $F \rightarrow . (E)$

E → E . + T F → . id

GOTO (I_4 , T) GOTO (I_9 , *)

I_2 : E →T . I_7 : T → T * . F

T → T . * F F → . (E)

F → . id

GOTO (I_4 , F)

I_3 : T → F

GOTO (I_4 , ( )

I_4 : F → (. E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → id

FOLLOW (E) = { $,) , +)

FOLLOW (T) = { $, + ,) , * }

FOLLOW (F) = { * , + ,) , $ }

SLR parsing table :

	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
I0	s5			s4			1	2	3
I1		s6				ACC			
I2		r2	s7		r2	r2			
I3		r4	r4		r4	r4			
I4	s5			s4			8	2	3
I5		r6	r6		r6	r6			
I6	s5			s4				9	3
I7	s5			s4					10
I8		s6			s11				
I9		r1	s7		r1	r1			
I10		r3	r3		r3	r3			
I11		r5	r5		r5	r5			

***Blank entries are error** entries.*

Stack implementation:

*Check whether the input id + id * id is valid or not.*

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I0 , id) = s5 ; shift
0 id 5	+ id * id $	GOTO (I5 , +) = r6 ; reduce by F→id

0 F 3	+ id * id $	GOTO (I0 , F) = 3
		GOTO (I3 , +) = r4 ; reduce by T → F
0 T 2	+ id * id $	GOTO (I0 , T) = 2
		GOTO (I2 , +) = r2 ; reduce by E → T
0 E 1	+ id * id $	GOTO (I0 , E) = 1
		GOTO (I1 , +) = s6 ; shift
0 E 1 + 6	id * id $	GOTO (I6 , id) = s5 ; shift
0 E 1 + 6 id 5	* id $	GOTO (I5 , *) = r6 ; reduce by F → id
0 E 1 + 6 F 3	* id $	GOTO (I6 , F) = 3
		GOTO (I3 , *) = r4 ; reduce by T → F
0 E 1 + 6 T 9	* id $	GOTO (I6 , T) = 9
		GOTO (I9 , *) = s7 ; shift
0 E 1 + 6 T 9 * 7	id $	GOTO (I7 , id) = s5 ; shift
0 E 1 + 6 T 9 * 7 id 5		$ GOTO (I5 , $) = r6;reduce by F → id
0 E 1 + 6 T 9 * 7 F 10	$	GOTO (I7 , F) = 10
* F		GOTO (I10 , $) = r3 ; reduce by T → T
0 E 1	+ 6 T 9 $	GOTO (I6 , T) = 9
T		GOTO (I9 , $) = r1 ; reduce by E → E +
0 E 1	$	GOTO (I0 , E) = 1
		GOTO (I1 , $) = accept

7.10 Phases of the Compiler

We will briefly describe about the various phase of a compiler. The whole compilation process consists of two phases: analysis phase and synthesis phase.

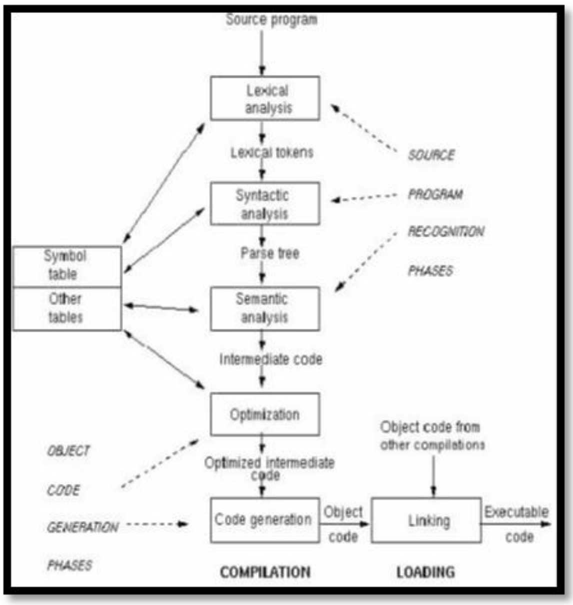
Analysis Phase again consist of three sub phases:

1. Lexical analysis(Tokenizing) Scans program, groups characters into tokens.
2. Syntax analysis(Parsing) Structures token sequence according to grammar rules of the language
3. Semantic analysis Checks semantic constraints of the language.

Synthesis Phase also have three sub phases:

1. Intermediate code generation: Translates to lower-level machine independent representation
2. Code optimization: Improves code quality
3. Code generation: Generates target code.

The figure below shows phases of compilation.



7.11 Aspects of compilation

we have seen that that compilers are tools which allows humans to read and write programs in high-level language and translates into machine language which can run on computer. This translation is called as compilation.

The main aspects of compilation are :

1. Generate an executable code of source program.
2. Provide the diagnostics for violations of syntax(according to grammar) and semantics in a source program.

Thus, the compilers supports features of high-level programming that contributes to the semantic gap between executable code such as:

Data types: These are the specification of user defined names for variables and operations on 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 language.
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 : When a compiler compiles the data structure used by the language, 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.

Memory allocation

During semantic analysis, on encountering data declaration statement, compiler allocates memory for their instruction and data separately. This is called memory allocation and used to perform memory binding. Memory allocation is important due to 3 reasons:-

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

Types of memory allocation : (i) Static and (ii) Dynamic

Static : means before the execution of a program begins. Static Memory allocation allocates storage to all program variables before the start of program execution. The binding of data item takes place at compile time. Since no memory is allocated or deallocated during execution, variables are permanent in a program.

Dynamic : means during the execution of a program. In dynamic storage allocation, binding time is during program execution. The advantages of dynamic allocation are :

(a) Recursion can be implemented easily because memory is allocated when the program unit is entered during execution.

(b) Support data structure whose size is determined dynamically.

At the time of accessing variables and allocating memory dynamically, a compiler performs 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.

Dynamic allocation is again of two types : (a) automatic and (b) program controlled allocation.

Automatic dynamic allocation **:** Memory is allocated when the program unit is entered during execution and deallocated when program unit is exited. Different memory area may be allocated for the same variable if the variable has different activation operation in the program. Automatic memory allocation is implemented using stack as entry and exit follows the LIFO. 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.

Program controlled dynamic allocation **:** memory allocation/deallocation at arbitrary point during execution. The program controlled allocation is implemented using a heap. A pointer is needed to point to each allocated memory area.

These are some other types of representations that the compilers use.

Symbol Table : Compiler uses symbol table to keep track of scope and binding information about names. Symbol table is changed every time a name is encountered in the source. Such as-a new name is discovered, new information about an existing name is discovered.

Symbol table must have mechanism such as-add new entries, find existing information efficiently.

Two common mechanism used - (i) linear lists, simple to implement, poor performance (ii) hash tables, greater programming/space overhead, good performance Compiler should be able to grow symbol table dynamically. If size is fixed, it must be large enough for the largest program.

A compiler uses a symbol table to keep track of scope and binding information about names. It is filled after the abstract syntax tree is made by walking through the tree, discovering and assimilating information about the names. There should be two basic operations- to insert a new name or information into the symbol table as and when discovered and to efficiently lookup a name in the symbol table to retrieve its information.

Symbol Table Entries :

For each declaration of a name, there is an entry in the symbol table. Different entries need to store different information because of the different contexts in which a name can occur. An entry corresponding to a particular name can be inserted into the symbol table at different stages depending on when the role of the name becomes clear. The various attributes that an entry in the symbol table can have are lexeme, type of name, size of storage and in case of functions - the parameter list etc.

There might be multiple entries in the symbol table for the same name, all of them having different roles. It is quite intuitive that the symbol table entries have to be made only when the role of a particular name becomes clear. The lexical analyzer therefore just returns the name and not the symbol table entry as it cannot determine the context of that name. Attributes corresponding to the symbol table are entered for a name in response to the corresponding declaration. There has to be an upper limit for the length of the lexemes for them to be stored in the symbol table.

Information about the storage locations that will be bound to names at run time is kept in the symbol table. If the target is assembly code, the assembler can take care of storage for various names. All the compiler has to do is to scan the symbol table, after generating assembly code, and generate assembly language data definitions to be appended to the assembly language program for each name. If machine code is to be generated by the compiler, then the position of each data object relative to a fixed origin must be ascertained. The compiler has to do the allocation in this case. In the case of names whose storage is allocated on a stack or heap, the compiler does not allocate storage at all, it plans out the activation record for each procedure.

The entries in the symbol table are for declaration of names. When an occurrence of a name in the source text is looked up in the symbol table, the entry for the appropriate declaration, according to the scoping rules of the language, must be returned. A simple approach is to maintain a separate symbol table for each scope most closely nested scope rule can be implemented in data structures.

Most closely nested scope rules can be implemented by adapting the data structures discussed in the previous section. Each procedure is assigned a unique number. If the language is block-structured, the blocks must also be assigned unique numbers. The name is represented as a pair of a number and a name. This new name is added to the symbol table. Most scope rules can be implemented in terms of following operations:

a) Lookup - find the most recently created entry.

b) Insert - make a new entry.

c) Delete - remove the most recently created entry.

Storage class: A storage class is an extra keyword at the beginning of a declaration which modifies the declaration in some way. Generally, the storage class (if any) is the first word in the declaration, preceding the type name. Ex. static, extern etc.

Scope: The scope of a variable is simply the part of the program where it may be accessed or written. It is the part of the program where the variable's name may be used. If a variable is declared within a function, it is local to that function. Variables of the same name may be declared and used within other functions without any conflicts. For instance,

```
int fun1()
{
   int a;
   int b;
   ....
}
int fun2()
{
   int a;
   int c;
   ....
}
```

Visibility : The visibility of a variable determines how much of the rest of the program can access that variable. You can arrange that a variable is visible only within one part of one function, or in one function, or in one source file, or anywhere in the program

Local and Global variables: A variable declared within the braces {} of a function is visible only within that function; variables declared within functions are called local variables. On the other hand, a variable declared outside of any function is a global variable , and it is potentially visible anywhere within the program

Automatic Vs Static duration: How long do variables last? By default, local variables (those declared within a function) have automatic duration : they spring into existence when the function is called, and they (and their values) disappear when the function returns. Global variables, on the other hand, have static duration : they last, and the values stored in them persist, for as long as the program does. (Of course, the values can in general still be overwritten, so they don't necessarily persist forever.) By default, local variables have automatic duration. To give them static duration (so that, instead of coming and going as the function is called, they persist for as long as the function does), you precede their declaration with the static keyword: static int i; By default, a declaration of a global variable (especially if it specifies an initial value) is the defining instance. To make it an external declaration, of a variable which is defined somewhere else, you precede it with the keyword extern: extern int j; Finally, to arrange that a global variable is visible only within its containing source file, you precede it with the static keyword: static int k; Notice that the static keyword can do two different things: it adjusts the duration of a local variable from automatic to static, or it adjusts the visibility of a global variable from truly global to private-to-the-file.

A major consideration in designing a symbol table is that insertion and retrieval should be as fast as possible. We talked about the one dimensional and hash tables a few slides back. Apart from these balanced binary trees can be used too. Hashing is the most common approach.

7.12 Search Data structures

Hashed local symbol table

Hash tables can clearly implement 'lookup' and 'insert' operations. For implementing the 'delete', we do not want to scan the entire hash table looking for lists containing entries to be deleted. Each entry should have two links:

a) A hash link that chains the entry to other entries whose names hash to the same value - the usual link in the hash table.

b) A scope link that chains all entries in the same scope - an extra link. If the scope link is left undisturbed when an entry is deleted from the hash table, then the chain formed by the scope links will constitute an inactive symbol table for the scope in question.

Global Symbol table structure

The global symbol table will be a collection of symbol tables connected with pointers. The exact structure will be determined by the scope and visibility rules of the language. Whenever a new scope is encountered a new symbol table is created. This new table contains a pointer back to the enclosing scope's symbol table and the enclosing one also contains a pointer to this new symbol table. Any variable used inside the new scope should either be present in its own symbol table or inside the enclosing scope's symbol table and all the way up to the root symbol table. A sample global symbol table is shown in the slides later.

The variable names have to be translated into addresses before or during code generation. There is a base address and every name is given an offset with respect to this base which changes with every invocation. The variables can be divided into four categories:

a) Global Variables

b) Global Static Variables

c) Stack Variables

d) Stack Static Variables

Global variables, on the other hand, have static duration (hence also called static variables): they last, and the values stored in them persist, for as long as the program does. (Of course, the values can in general still be overwritten, so they don't necessarily persist forever.) Therefore they have fixed relocatable address or offset with respect to base as global pointer . By default, local variables (stack variables) (those declared within a function) have automatic duration : they spring into existence when the function is called, and they (and their values) disappear when the function returns. This is why they are stored in stacks and have offset from stack/frame pointer. Register allocation is usually done for global variables. Since registers are not indexable, therefore, arrays cannot be in registers as they are indexed data structures. Graph coloring is a simple technique for allocating register and minimizing register spills that works well in practice. Register spills occur when a register is needed for a computation but all available registers are in use. The contents of one of the registers must be stored in memory to free it up for immediate use. We assign symbolic registers to scalar variables which are used in the graph coloring.

word boundaries - the most significant byte of the object must be located at an address whose two least significant bits are zero relative to the frame pointer half-word boundaries - the most significant byte of the object being located at an address whose least significant bit is zero relative to the frame pointer sort variables by the alignment they need

Large local data structures require large space in local frames and therefore large offsets. You can either allocate another base register to access large objects or you can allocate space in the middle or elsewhere and then store pointers to these locations starting from at a small offset from the frame pointer, fp.

7.13 Compilation of expressions and control structures

In High-Level Language, expressions occur in assignment statements, conditional statement and IO statements etc. Expression processing actions can be grouped as:

1. Intermediate code generation
2. 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: Compilation of expression 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. The postfix notation of a + b * c + d * e ↑ f 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.

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.

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. Syntax directed translation of expression into 3-address code

$S \rightarrow id := E$	S.code = E.code ||
	gen(id.place:= E.place)
$E \rightarrow E_1 + E_2$	E.place:= newtmp
	E.code:= E_1 .code || E_2 .code ||
	gen(E.place := E_1 .place + E_2 .place)
$E \rightarrow E_1 * E_2$	E.place:= newtmp
	E.code := E_1 .code || E_2 .code ||
	gen(E.place := E_1 .place * E_2 .place)

Three-address code is a sequence of statements of the general form X := y *op* z Where x, y and z are names, constants, or compiler generated temporaries. op stands for any operator, such as fixed- or floating-point arithmetic operator, or a logical operator on Boolean-valued data. Note that no built up arithmetic expression are permitted, as there is only one operator on the right side of a statement. Thus a source language expression like x + y * z might be translated into a sequence t1 := y * z

t2 := x + t1

where t1 and t2 are compiler-generated temporary names. This unraveling of complicated arithmetic expression and of nested flow-of-control statements makes three- address code desirable for target code generation and optimization. The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged unlike postfix notation. We can easily generate code for the three-address code given above. The S-attributed definition above generates three-address code for assigning statements. The synthesized attribute *S.code* represents the three-address code for the assignment S. The nonterminal E has two attributes:

. *E.place* , the name that will hold the value of E, and
. *E.code* , the sequence of three-address statements evaluating E.
The function *newtemp* returns a sequence of distinct names t1, t2,.. In response to successive calls.
Syntax directed translation of expression .

$E \rightarrow -E_1$	E.place := newtmp
	E.code := E_1 .code ||
	gen(E.place := - E_1 .place)
$E \rightarrow (E_1)$	E.place := E_1 .place
	E.code := E_1 .code
$E \rightarrow$ id	E.place := id.place
	E.code := ' '

Example: For a = b * -c + b * -c following code is generated
t 1 = -c
t 2 = b * t 1
t 3 = -c
t 4 = b * t 3
t 5 = t 2 + t 4
a = t 5

$S \rightarrow$ while E do S_1	S.begin := newlabel
S. begin :	S.after := newlabel
E.code	
if E.place = 0 goto S.after	S.code := gen(S.begin:) ||
S_1 .code	E.code ||
goto S.begin	gen(if E.place = 0 goto S.after) ||
S.after :	S_1 .code || gen(goto S.begin) ||
	gen(S.after:)

A new label S *.begin* is created and attached to the first instruction for E. Another new label S. *after* is created. The code for E generates a jump to the label if E is true, a jump to S.next if E is false; again, we set E.false to be S.next. After the code for S1 we place the instruction goto S.begin, which causes a jump back to the beginning of the code for the Boolean expression. Note that S1.next is set to this label S.begin, so jumps from within S1.code can directly to S.begin.

Flow of Control

S $\rightarrow$ if E then S_1 else S_2	S.else := newlabel
	S.after := newlabel
E.code	S.code = E.code ||
if E.place = 0 goto S.else	gen(if E.place = 0 goto S.else) ||
S_1 .code	S $_1$ .code ||
goto S.after	gen(goto S.after) ||
S.else:	gen(S.else :) ||
S $_2$ .code	gen(S.after :)
S.after:	

In translating the if-then-else statement the Boolean expression E jumps out of it to the first instruction of the code for S1 if E is true, and to first instruction of the code for S2 if E is false, as illustrated in the figure above. As with the if-then statement, an inherited attribute s.next gives the label of the three-address instruction to be executed next after executing the code for S. An explicit goto S.next appears after the code for S1, but not after S2.

Declarations

For each name create symbol table entry with information like type and relative address

P $\rightarrow$ {offset=0} D	
D $\rightarrow$ D ; D	
D $\rightarrow$ id : T	enter(id.name, T.type, offset);
	offset = offset + T.width
T $\rightarrow$ integer	T.type = integer; T.width = 4
T $\rightarrow$ real	T.type = real; T.width = 8

In the translation scheme nonterminal P generates a sequence of declarations of the form id : T . Before the first declaration is considered, offset is set to 0. As each new name is seen, that name is entered in the symbol table with the offset equal to the current value of offset, and offset is incremented by the width of the data object denoted by the name.

The procedure - enter (name, type, offset) creates a symbol-table entry for *name* , gives it type and relative address offset in its data area. We use synthesized attributes *type* and *width* for nonterminal T to indicate the type and Width, or number of memory units for nonterminal T to indicate the type and width, or number of memory units taken by objects of that type. If type expressions are represented by graphs, then attribute *type* might be a pointer to the node representing a type expression. We assume integers to have width of 4 and real to have width of 8. The width of an array is obtained by multiplying the width of each element by the number of elements in the array. The width of each pointer is also assumed to be 4.

P -> D

D -> D; D | id : T | proc id ; D ;S

Whenever a procedure declaration D proc id ; D1 ; S is processed, a new symbol table with a pointer to the symbol table of the enclosing procedure in its header is created and the entries for declarations in D1 are created in the new symbol table. The name represented by id is local to the enclosing procedure and is hence entered into the symbol table of the enclosing procedure.

Type conversion within assignments

$E \rightarrow E_1 + E_2$

E.place= newtmp;

if E_1 .type = integer and E_2 .type = integer

then emit(E.place ':=' E_1 .place 'int+' E_2 .place);

E.type = integer;

similar code if both E_1 .type and E_2 .type are real

else if E_1 .type = int and E_2 .type = real

then

u = newtmp;

emit(u ':=' inttoreal E_1 .place);

emit(E.place ':=' u 'real+' E_2 .place);

E.type = real;

similar code if E_1 .type is real and E_2 .type is integer

When a compiler encounters mixed type operations it either rejects certain mixed type operations or generates coercion instructions for them.

Semantic action for E -> E1+ E2:

E.place= newtmp;

if E1.type = integer and E2.type = integer

then emit(E.place ':=' E1.place 'int+' E2.place);

E.type = integer;

..

similar code if both E1.type and E2.type are real

else if E1.type = int and E2.type = real

then

u = newtmp;

emit(u ':=' inttoreal E1.place);

emit(E.place ':=' u 'real+' E2.place);

E.type = real;

similar code if E1.type is real and E2.type is integer

The three address statement of the form u ':=' inttoreal E1.place denotes conversion of an integer to real. int+ denotes integer addition and real+ denotes real addition.

Code generation is done along with type checking and if type resolution can be done in a single pass no intermediate representation like an abstract syntax tree would be required.

Syntax directed translation of boolean expressions

E $\longrightarrow$ E_1 or E_2

E.place := newtmp

emit(E.place ':=' E_1 .place 'or' E_2 .place)

E$\longrightarrow$ E_1 and E_2

E.place:= newtmp

emit(E.place ':=' E_1 .place 'and' E_2 .place)

E $\longrightarrow$ not E_1

E.place := newtmp

emit(E.place ':=' 'not' E_1 .place)

E $\longrightarrow$ (E_1) E.place = E_1 .place

The above written translation scheme produces three address code for Boolean expressions. It is continued to the next page.

Syntax directed translation of boolean expressions

E$\longrightarrow$ id1 relop id2

E.place := newtmp

emit(if id1.place relop id2.place goto nextstat+3)

emit(E.place = 0) emit(goto nextstat+2)

emit(E.place = 1)

$E \longrightarrow$ true

E.place := newtmp

emit(E.place = '1')

$E \longrightarrow$ false

E.place := newtmp

emit(E.place = '0')

In the above scheme, nextstat gives the index of the next three address code in the output sequence and emit increments nextstat after producing each three address statement.

Flow of control statements

$S \longrightarrow$ if E then S_1

| if E then S_1 else S_2

| while E do S_1

We can translate a boolean expression into three-address code without generating code for boolean operators and without evaluating the entire expression. Take the case of the previous example, here we can tell the value of t by whether we reach statement 101 or 103, so the value of t becomes redundant. Similarly, for larger expressions the value can be determined without having to evaluate the expression completely. However, if a part of the expression having a side effect is not evaluated, the side effect will not be visible.

Now we will consider the translation of boolean expressions into three address code generated by the following grammar:

S -> if E then S1

| if E then S1 else S2

| while E do S1

where E is the boolean expression to be translated.

Consider the following:

newlabel - returns a new symbolic label each time it is called.

E.true - the label to which control flows if E is true.

E.false - the label to which control flows if E is false.

For *if-then*

S -> if E then S1

E.true = newlabel //generate a new label for E.true

E.false = S.next //jump to S.next if E is false

S1.next = S.next

S.code = E.code || gen(E.true ':') || S1.code // associate the label created for E.true with the

// three address code for the first statement for S1

For *while-do*

S -> while E do S1

S.begin = newlabel

E.true = newlabel

E.false = S.next

S1.next = S.begin

S.code = gen(S.begin ':') || E.code || gen(E.true ':') || S1.code || gen(goto S.begin)

In case of while-do statement a new label S.begin is created and associated with the first instruction of the code for E, so that control can be transferred back to E after the execution of S1. E.false is set to S.next, so that control moves out of the code for S in case E is false. Since S1.next is set to S.begin, jumps from the code for S1 can go directly to S.begin.

Control flow translation of boolean expression

$E \longrightarrow E_1$ or E_2	E_1 .true := E.true
	E_1 .false := newlabel
	E_2 .true := E.true
	E_2 .false := E.false
	E.code := E_1 .code || gen(E_1 .false) || E_2 .code
$E \longrightarrow E_1$ and E_2	E_1 .true := new label
	E_1 false := E.false
	E_2 .true := E.true
	E_2 false := E.false
	E.code := E_1 .code || gen(E_1 .true) || E_2 .code

E is translated into a sequence of conditional and unconditional jumps to one of the two locations: E.true or E.false depending if E is true or false. If E is of the form E1 *or* E2, then if E1 is true then we immediately know that E itself is true, so E1.true is the same as E.true. If E1 is false then E2 must be evaluated, so E1.false is the label of the first statement of E2. If E2 is evaluated and E2 is true, it implies that E is true, so E2.true is set to E.true. Similarly, if E2 is evaluated and it is false, the entire expression is false.

E -> E1 or E2 E1.true := E.true

E1.false := newlabel

E2.true := E.true

E2.false := E.false

E.code := E1.code || gen(E1.false) || E2.code

Analogously E1 *and* E2 can also be translated. Here if E1 is false then there need be no further considerations.

E -> E1 and E2 E1.true := new label

E1 false := E.false

E2.true := E.true

E2 false := E.false

E.code := E1.code || gen(E1.true) || E2.code

Procedure Calls

S $\longrightarrow$ call id (Elist)

Elist $\longrightarrow$ Elist , E

Elist $\longrightarrow$ E

The parameters are passed by reference and have statically allocated storage.

Code Generation

S $\longrightarrow$ call id (Elist)

for each item p on queue do emit('param' p)

emit('call' id.place)

Elist $\longrightarrow$ Elist , E

append E.place to the end of queue

Elist $\longrightarrow$ E

initialize queue to contain E.place

For code generation, we generate three address code which is needed to evaluate arguments that are in fact expressions. As a result a list of param three address statements are generated. This is a syntax-directed translation and gives: param p_1 ; param p_2 ; param p_n ; call id.place

7.14 Summary

In this unit, we have learnt the various parsing approach and basic working strategy of the parsers.

7.15 Terminal Questions

1.What do you mean by parsing? Define parse tree. Note down the role of parser in compiler design.

2. Differentiate between top-down parsing and bottom-up parsing.

3. Define LR grammar. Enumerate different types of LR grammar.

4. Explain non-recursive predictive parsing.

UNIT - 8 : COMPILER-CODE GENERATION

Intermediate languages : graphical representations, DAGs, Three address code, types of three address statements, syntax directed translation into three address code, implementation of three address statements.

Structure

8.0 Introduction

In this unit you will learn about code generation using syntax directed translation. This is also called as intermediate code, i.e. machine independent code generation.

8.1 Objective

The learning objectives are following:-

- o Intermediate code representations
- o Types of intermediate code
- o Syntax directed translation

8.2 Intermediate Languages

In Intermediate code generation we use syntax directed methods to translate the source program into an intermediate form programming language constructs such as declarations, assignments and flow-of-control statements. Intermediate code is the output of the Parser and the input to the Code Generator. It is relatively machine-independent which allows the compiler to be retargeted and relatively easy to manipulate (optimize). Advantages of Using an Intermediate Language are:-

i. Retargeting - Build a compiler for a new machine by attaching a new code generator to an existing front-end.

ii. Optimization - reuse intermediate code optimizers in compilers for different languages and different machines.

There are three types of intermediate representation:-

1. Syntax Trees

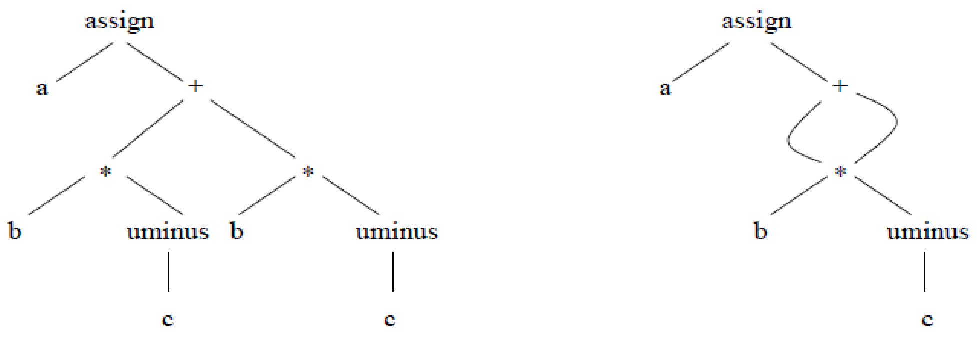
2. Postfix notation

3. Three Address Code

Semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees of for generating postfix notation.

8.3 Graphical Representations

A syntax tree depicts the natural hierarchical structure of a source program. A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified. A syntax tree and dag for the assignment statement a:=b*-c+b*-c appear in the figure.



(a) Syntax tree **(b) Dag**

8.4 Postfix notation

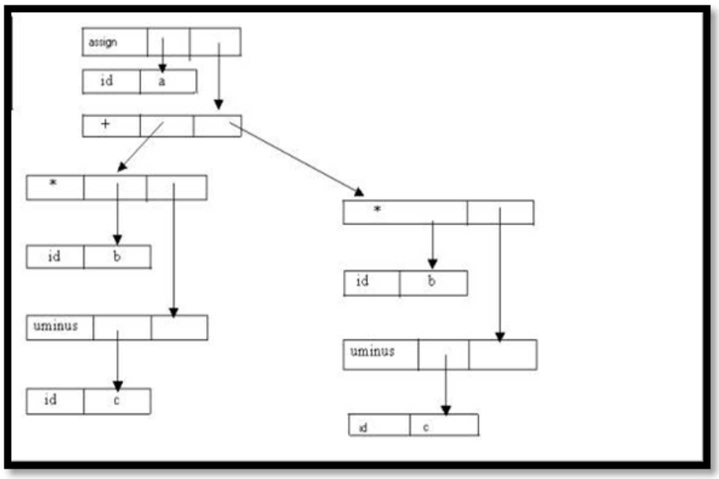
Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the in which a node appears immediately after its children. The postfix notation for the syntax tree in the fig is
a b c uminus + b c uminus * + assign
The edges in a syntax tree do not appear explicitly in postfix notation. They can be recovered in the order in which the nodes appear and the no. of operands that the operator at a node expects.
Syntax tree for assignment statements are produced by the syntax directed definition in fig.
ProductionSemantic Rule

S -> id := ES.nptr := mknode('assign', mkleaf(id, id.place), E.nptr)
E -> E1 + E2E.nptr := mknode('+', E1.nptr ,E2.nptr)
E -> E1 * E2E.nptr := mknode('* ', E1.nptr ,E2.nptr)

E -> - E1E.nptr := mkunode('uminus', E1.nptr)
E -> (E1)E.nptr := E1.nptr
E -> idE.nptr := mkleaf(id, id.place)

8.5 DAGs

This same syntax-directed definition will produce the dag if the functions mkunode(op, child) and mknode(op, left, right) return a pointer to an existing node whenever possible, instead of constructing new nodes. The token id has an attribute place that points to the symbol-table entry for the identifier id.name, representing the lexeme associated with that occurrence of id. lf the lexical analyzer holds all lexemes in a single array of characters, then attribute name might be the index of the first character of the lexeme. Two representations of the syntax tree in above fig appear in below figures. Each node is represented as a record with a field for its operator and additional fields for pointers to its children. In Fig, nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position IO.



0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11	……		

8.6 Three-address code

Three-address code is a sequence of statements of the general form

X:= Y Op Z where x, y, and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean-valued data. Note that no built-up arithmetic expressions are permitted, as there is only one operator on the right side of a statement. Thus a source language expression like x+y*z might be translated into a sequence

t1 := y * z

t2 : = x + t1

where t1 and t2 are compiler-generated temporary names. This unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization. The use of names for the intermediate values computed by a program allow- three-address code to be easily rearranged – unlike postfix notation. three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag in above Figs are represented by the three-address code sequences in below Fig. Variable names can appear directly in three-address statements, so fig has no statements corresponding to the leaves of above Fig.

Code for syntax tree

```
t1 := -c
t2 :=  b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5
```

Code for DAG

```
t1 := -c
t2 := b * t1
t5 := t2 + t2
a := t5
```

The reason for the term ”three-address code” is that each statement usually contains three addresses, two for the operands and one for the result. In the implementations of three-address code given later in this section, a programmer-defined name is replaced by a pointer tc a symbol-table entry for that name.

Types Of Three-Address Statements

Three-address statements are machine-independent code. 3-address Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of a three-address statement in the array holding inter- mediate code. Actual indices can be substituted for the labels either by making a separate pass, or by using ”back patching,”. Here are the commonly used three-address statements:

1. Assignment statements of the form x: = y op z, where op is a binary arithmetic or logical operation.
2. Assignment instructions of the form x:= op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
3. Copy statements of the form x: = y where the value of y is assigned to x.
4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.
5. Conditional jumps such as if x relop y goto L. This instruction applies a relational operator (<, =, >=, etc.) to x and y, and executes the statement with label L next if x stands in relation relop to y. If not, the three-address statement following if x relop y goto L is executed next, as in the usual sequence.
6. param x and call p, n for procedure calls and return y, where y representing a returned value is optional. Their typical use is as the sequence of three-address statements

param x1

param x2
param xn
call p, n
generated as part of a call of the procedure p(x,, x~,..., x”). The integer n indicating the number of actual parameters in ”call p, n” is not redundant because calls can be nested. The implementation of procedure calls is outline d in Section 8.7.

7. Indexed assignments of the form x: = y[i] and x [i]: = y. The first of these sets x to the value in the location i memory units beyond location y. The statement x[i]:=y sets the contents of the location i units beyond x to the value of y. In both these instructions, x, y, and i refer to data objects.
8. Address and pointer assignments of the form x:= &y, x:= *y and *x: = y. The first of these sets the value of x to be the location of y. Presumably y is a name, perhaps a temporary, that denotes an expression with an I-value such as A[i, j], and x is a pointer name or temporary. That is, the r-value of x is the l-value (location) of some object!. In the statement x: = ~y, presumably y is a pointer or a temporary whose r- value is a location. The r-value of x is made equal to the contents of that location. Finally, +x: = y sets the r-value of the object pointed to by x to the r-value of y.

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set must clearly be rich enough to implement the operations in the source language. A small operator set is easier to implement on a new target machine. However, a restricted instruction set may force the front end to generate long sequences of statements for some source, language operations. The optimizer and code generator may then have to work harder if good code is to be generated.

8.7 Syntax-Directed Translation into Three-Address Code

When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. The value of non-terminal E on the left side of E -> E1 + E will be computed into a new temporary t. In general, the three-address code for id: = E consists of code to evaluate E into some temporary t, followed by the assignment id.place: = t. If an expression is a single identifier, say y, then y itself holds the value of the expression. The S-attributed definition generates three-address code for assignment statements given in above fig. Given input a: = b+ – c + b+ – c, it produces the code shown in above fig. The synthesized attribute S.code represents the three- address code for the assignment S. The non-terminal E has two attributes:

1. E.place, the name that will hold the value of E, and
2. E.code, the sequence of three-address statements evaluating E.

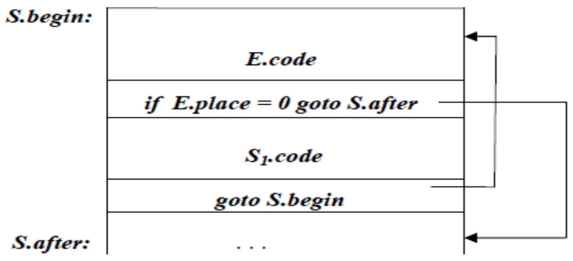
The function newtemp returns a sequence of distinct names t1, t2,... in response to successive calls. For convenience, we use the notation gen(x ’: =’ y ’+’ z) in Fig. shown to represent the three-address statement x: = y + z. Expressions appearing instead of variables like x, y, and z are evaluated when passed to gen, and quoted operators or operands, like ’+’, are taken literally. In practice, three- address statements might be sent to an output file, rather than built up into the code attributes.

Syntax-directed definition to produce three-address code for assignments

PRODUCTION	SEMANTIC RULES
$S \rightarrow id := E$	*S.code : = E.code || gen(id.place ':=' E.place)*
$E \rightarrow E_1 + E_2$	*E.place := newtemp;* *E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)*
$E \rightarrow E_1 * E_2$	*E.place := newtemp;* *E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)*
$E \rightarrow - E_1$	*E.place := newtemp;* *E.code := E_1.code || gen(E.place ':=' 'uminus' E_1.place)*
$E \rightarrow (E_1)$	*E.place : = E_1.place;* *E.code : = E_1.code*
$E \rightarrow id$	*E.place : = id.place;* *E.code : = ' '*

Flow-of-control statements can be added to the language of assignments by productions and semantic rules like the ones for while statements. In the figure, the code for S -> while E do S, is generated using' new attributes S.begin and S.after to mark the first statement in the code for E and the statement following the code for S, respectively.

Semantic rules generating code for a while statement



PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **while** E **do** S_1	*S.begin := newlabel;* *S.after := newlabel;* *S.code := gen(S.begin ':') ||* *E.code ||* *gen ('if' E.place '=' '0' 'goto' S.after)||* *S_1.code ||* *gen ('goto' S.begin) ||* *gen (S.after ':')*

These attributes represent labels created by a function new label that returns a new label every time it is called. Note that S.after becomes the label of the statement that comes after the code for the while statement. We assume that a non-zero expression represents true; that is, when the value of F becomes zero, control leaves the while statement. f:expressions that govern the flow of control may in general be Boolean expressions containing relational and logical operators. 1he postfix notation for an identifier is the identifier itself. The rules for the other productions concatenate only the operator after the code for the operands. For example, associated with the production E – E, is the semantic rule

E.code:= E1.code || 'uminus'

1n general, the intermediate form produced by the syntax-directed translations in this chapter can he changed by making similar modifications to the semantic rules.

8.8 Implementations of three-Address Statements

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are quadruples, triples, and indirect triples.

Quadruples

A quadruple is a record structure with four fields, which we call op, arg l, arg 2, and result. The op field contains an internal code for the operator. The three-address statement x:= y op z is represented by placing y in arg 1. z in arg 2. and x in result. Statements with unary operators like x: = – y or x: = y do not use arg 2. Operators like param use neither arg2 nor result. Conditional and unconditional jumps put the target label in result. The quadruples in Fig. are for the assignment a: = b+ – c + b i – c. They are obtained from the three-address code of above fig. The contents of fields arg 1, arg 2, and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples

	op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Fig QUDRAPLES

	op	Arg1	Arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Fig TRIPLES

To avoid entering temporary names into the symbol table. we might refer to a temporary value bi the position of the statement that computes it. If we do so, three-address statements can be represented by records with only three fields: op, arg 1 and arg2. The fields arg l and arg2, for the arguments of op, are either pointers to the symbol table (for programmer- defined names or constants) or pointers into the triple structure (for temporary values). Since three fields are used, this intermediate code format is known as triples. Except for the treatment of programmer-defined names, triples correspond to the representation of a syntax tree or dag by an array of nodes.

8.9 Summary

We have learnt about intermediate code generation.

8.10 Terminal Questions

What are different forms of intermediate code?

UNIT - 9 : COMPILER-OPTIMIZATION

Structure

9.0 Introduction

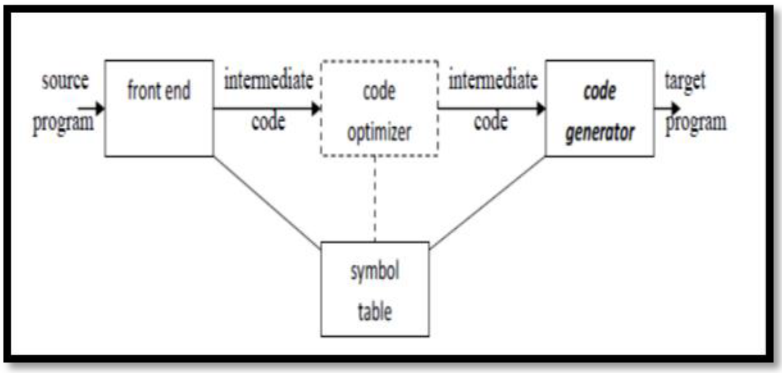
In this unit we will learn about code optimization techniques.

9.1 Objective

- Machine independent code optimization
- Code generation
- Translation of expression, control structures and procedure calls

9.2 CODE GENERATION

The final phase in compiler is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The techniques of code generation discussed in this unit can be used whether or not an optimizing phase occurs before code generation.



DESIGN ISSUES OF CODE GENERATOR : While the details are dependent on the target language and the operating system, issues such as (i) memory management, (ii) instruction selection, (iii) register allocation, and (iv) evaluation order are inherent in almost all code generation problems.

INPUT TO THE CODE GENERATOR : The input to the code generator is the intermediate code of source program along with symbol table information. This determines the run time addresses of the data objects denoted by the names in the intermediate representation. The output of the code generator is the target program of variety of forms: absolute machine language, relocatable machine language, or assembly language.

Producing an absolute machine language program as output has the advantage that it can be placed in a location in memory and immediately executed. A small program can be compiled and executed quickly.

Producing a relocatable machine language program as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

For relocatable object modules, a great deal of flexibility is achieved in being able to compile subroutines separately and to call other previously compiled programs from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program segments.

Producing an assembly language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help generate code.

MEMORY MANAGEMENT : Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator. We assume that a name in a three-address statement refers to a symbol table entry for the name. If machine code is being generated, labels in three address statements have to be converted to addresses of instructions.

Suppose that labels refer to quadruple numbers in a quadruple array. As we scan each quadruple in turn we can deduce the location of the first machine instruction generated for that quadruple, simply by maintaining a count of the number of words used for the instructions generated so far. This count can be kept in the quadruple array (in an extra field), so if a reference such as j: goto i is encountered, and i is less than j, the current quadruple number, simply generate a jump instruction with the target address equal to the machine location of the first instruction in the code for quadruple i. If, however, the jump is forward, so i exceeds j, we must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. Then we process quadruple i, we fill in the proper machine location for all instructions that are forward jumps to i.

INSTRUCTION SELECTION : The nature of the instruction set of the target machine determines the difficulty of instruction selection. The uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling.

Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three-address statement we can design a code skeleton that outlines the target code to be generated for that construct.

For example, every three address statement of the form x := y + z, where x, y, and z are statically allocated, can be translated into the code sequence

```
MOV y, R0   /* load y into register R0  */
ADD z, R0   /* add z to R0 */
MOV R0, x   /* store R0 into x */
```

Unfortunately, this kind of statement–by-statement code generation often produces poor code. For example, the sequence of statements

```
a := b + c
d := a + e
```

would be translated into

```
MOV   b, R0
ADD   c, R0
MOV   R0, a
MOV   a, R0
ADD   e, R0
MOV   R0, d
```

Here the fourth statement is redundant, and so is the third if ‘a’ is not subsequently used. The quality of the generated code is determined by its speed and size.

A target machine with a rich instruction set may provide several ways of implementing a given operation. Since the cost differences between different implementations may be significant, a naive translation of the intermediate code may lead to correct, but unacceptably inefficient target code. For example if the target machine has an “increment” instruction (INC), then the three address statement a := a+1 may be implemented more efficiently by the single instruction INC a, rather than by a more obvious sequence that loads a into a register, add one to the register, and then stores the result back into a.

```
MOV   a, R0
ADD   #1,R0
MOV   R0, a
```

Instruction speeds are needed to design good code sequence but unfortunately, accurate timing information is often difficult to obtain. Deciding which machine code sequence is best for a given three address construct may also require knowledge about the context in which that construct appears.

REGISTER ALLOCATION : Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. The use of registers is often subdivided into two sub problems:

1. During register allocation, we select the set of variables that will reside in registers at a point in the program.
2. During a subsequent register assignment phase, we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single register values. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register usage conventions be observed.

Now consider the two three address code sequences (a) and (b) in which the only difference is the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in(c). R_i stands for register i. L, ST and A stand for load, store and add respectively. The optimal choice for the register into which 'a' is to be loaded depends on what will ultimately happen to e.

```
  t  :=  a + b                           t := a + b
  t  :=  t * c                            t  :=  t + c
  t  :=  t / d                           t  := t / d
```

Two three address code sequences

```
L     R1, a                        L           R0, a
A     R1, b                        A           R0, b
M     R0, c                        A           R0, c
D     R0, d                        SRDA    R0, 32
ST    R1, t                        D           R0, d
                                   ST          R1, t
   (a)                                  (b)
```

Optimal machine code sequence

CHOICE OF EVALUATION ORDER : The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. Picking a best order is another difficult, NP-complete problem.

BASIC BLOCKS AND FLOW GRAPHS

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control. Flow graph of a program can be used as a vehicle to collect information about the intermediate program. Some register-assignment algorithms use flow graphs to find the inner loops where a program is expected to spend most of its time.

BASIC BLOCKS

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The following sequence of three-address statements forms a basic block:

```
t1 := a*a

t2 := a*b

t3 := 2*t2

t4 := t1+t3

t5 := b*b

t6 := t4+t5
```

A three-address statement x := y+z is said to define x and to use y or z. A name in a basic block is said to live at a given point if its value is used after that point in the program, perhaps in another basic block.

The following algorithm can be used to partition a sequence of three-address statements into basic blocks.

Algorithm 1: Partition into basic blocks.

Input: A sequence of three-address statements.

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

1.We first determine the set of leaders, the first statements of basic blocks.

The rules we use are the following:

I) The first statement is a leader.

II) Any statement that is the target of a conditional or unconditional goto is a leader.

III) Any statement that immediately follows a goto or conditional goto statement is a leader.

2.For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example : Consider the fragment of source code shown below; it computes the dot product of two vectors a and b of length 20. A list of three-address statements performing this computation on our target machine is shown below.

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i := i+1;
     end
      while  i<= 20
  end
```

Program to compute dot product

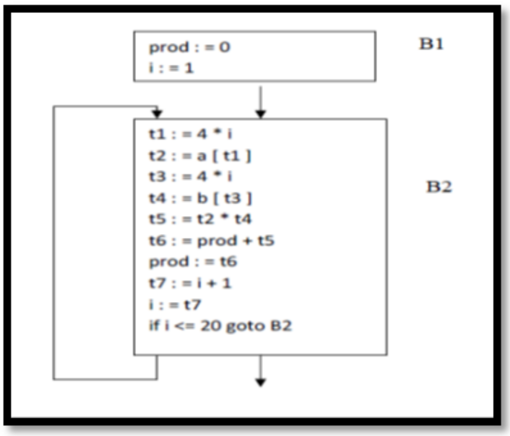
```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4*i
(4)   t2 := a [ t1 ]
(5)   t3 := 4*i
(6)   t4 :=b [ t3 ]
(7)   t5 := t2*t4
(8)   t6 := prod +t5
(9)   prod := t6
(10)  t7 := i+1
(11)  i := t7
(12)  if  i<=20 goto (3)
```

Three-address code computing dot product

A basic block computes a set of expressions. These expressions are the values of the names live on exit from block. Two basic blocks are said to be equivalent if they compute the same set of expressions. A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Many of these transformations are useful for improving the quality of code that will be ultimately generated from a basic block. There are two important classes of local transformations that can be applied to basic blocks; these are the structure-preserving transformations and the algebraic transformations.

Flow Graphs

Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program. The nodes of the flow graph are basic blocks. It has a distinguished initial node. In the example flow-graph below, B1 is the initial node. B2 immediately follows B1, so there is an edge from B1 to B2. The target of jump from last statement of B1 is the first statement B2, so there is an edge from B1 (last statement) to B2 (first statement). B1 is the predecessor of B2, and B2 is a successor of B1.



Loops : A loop is a collection of nodes in a flow graph such that

i. All nodes in the collection are strongly connected.

ii. The collection of nodes has a unique entry.

A loop that contains no other loops is called an inner loop.

NEXT-USE INFORMATION : If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.

9.4 Sources of optimization

The code produced by the compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers. Optimizations are classified into two categories. They are (i) Machine independent optimizations, and (ii) Machine dependent optimizations. Machine independent optimizations: A program transformations that improve the target code without taking into consideration any properties of the target machine. Machine dependent optimizations: Based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations: The program transformations that yield the most benefit for the least effort. The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does. A transformation must, on the average, speed up programs by a measurable amount. Also reducing the size of the compiled code although the size of the code.

Flow analysis is a fundamental prerequisite for many important types of code improvement. Generally control flow analysis precedes data flow analysis. Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as (i)control flow graph (ii) Call graph.

Data flow analysis (DFA) is the process of ascerting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

PRINCIPAL SOURCES OF OPTIMISATION : A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first. Function-Preserving Transformations: There are a number of ways in which a compiler can improve a program without changing the function it computes.

The transformations a) Common sub expression elimination, b) Copy propagation, c) Dead-code elimination, and d) Constant folding are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed. Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

1. **Common Sub expressions elimination:** An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

For example

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

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

The above code can be optimized using the common sub-expression elimination as shown in right side.

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

2. **Copy Propagation :** Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example:

```
x=Pi;

……

A=x*r*r;
```

The optimization using copy propagation can be done as follows:

A=Pi*r*r; Here the variable x is eliminated

3. **Dead-Code Eliminations :** A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programme is unlikely to introduce any dead code Intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example :

```
i=0;

if(i=1)

{

a=b+5;

}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

4. Constant folding: We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,

a=3.14157/2 can be replaced by

a=1.570 thereby eliminating a division operation.

5. **Loop Optimizations:** We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop. Three techniques are important for loop optimization:

   i. code motion, which moves code outside a loop;
   ii. Induction-variable elimination, which we apply to replace variables from inner loop.

iii. Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion : An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2) /* statement does not change limit*/
```

Code motion will result in the equivalent of

```
t= limit-2;

while (i<=t) /* statement does not change limit or t */
```

6. **Induction Variables :** Loops are usually processed inside out.

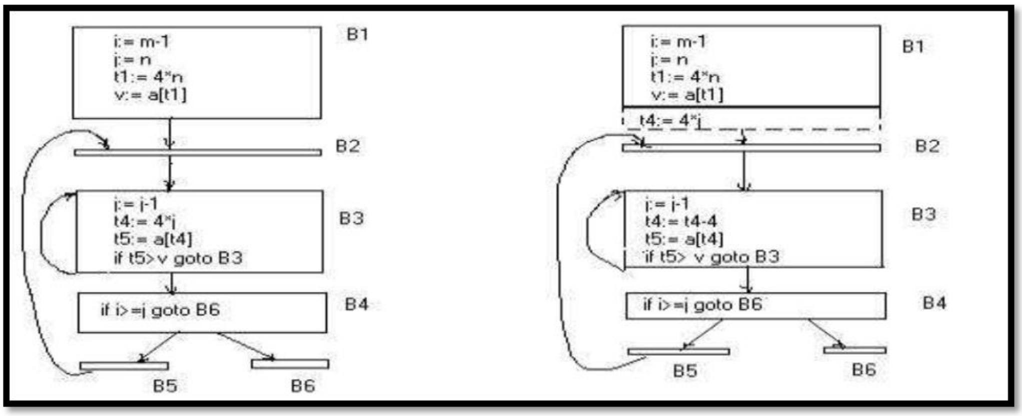
For example(shown below) consider the loop around B3.

Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

Example: As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:=4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in second Fig



The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

7. **Reduction In Strength:** Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

OPTIMIZATION OF BASIC BLOCKS

There are two types of basic block optimizations: Structure-preserving transformations and Algebraic transformations.

1. STRUCTURE-PRESERVING TRANSFORMATIONS : The primary structure-preserving transformations on basic blocks are :

i. common sub-expression elimination

ii. dead-code elimination

iii. renaming of temporary variables

iv. interchange of two independent adjacent statements

a) Common sub-expression elimination

Consider the basic block

```
a:= b+c
b:= a-d
c:= b+c
d:= a-d
```

```
a:= b+c
b:= a-d
c:= b+c
d:= b
```

The second and fourth statements compute the same expression, namely b+c-d, and hence this basic block may be transformed into the equivalent block although the 1st and 3rd statements in both cases appear to have the same expression on the right, the second statement redefines b. Therefore, the value of b in the 3rd statement is different from the value of b in the 1st, and the 1st and 3rd statements do not compute the same expression.

b) Dead-code elimination

Suppose x is dead, that is, never subsequently used, at the point where the statement x:= y+z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

c) Renaming temporary variables

Suppose we have a statement t:= b+c, where t is a temporary. If we change this statement to u:= b+c, where u is a new temporary variable, and change all uses of this instance of t to u, then the value of the basic block is not changed. In fact, we can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary. We call such a basic block a normal-form block.

d) Interchange of statements

Suppose we have a block with the two adjacent statements

t1:= b+c

t2:= x+y

Then we can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2. A normal-form basic block permits all statement interchanges that are possible.

2. Algebraic transformations :

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Examples:

i) x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.

ii) The exponential statement x : = y ** 2 can be replaced by x : = y * y.

9.5 Summary

In this unit we have learnt about concept of code optimizations, sources of code optimization.

9.6 Terminal Questions

1. What are different code optimization techniques?
2. Mention the properties that a code generator should possess.
3. What is a flow graph?
4. Define peephole optimization.
5. Mention the issues to be considered while applying the techniques for code optimization.
6. List the different storage allocation strategies.
7. What are the properties of optimizing compiler?

UNIT-10 : INTERPRETERS

10.0 INTRODUCTION

Interpreter is a computer language processor that translates a program line-by-line (statement-by-statement) and carries out the specified actions in sequence.

RECALL : In contrast, an assembler or compiler completely translates a program written in a high-level language (the source program) into a machine-language program (the object program) for later execution.

An interpreter allows examination and modification of the program while it is running (executing).

10.1 Objectives

In this unit you will learn about interpreter strategies for program execution:

a. parse the source code and perform its behavior directly.

b. translate source code into some efficient intermediate representation and immediately execute this.

10.2 Use and overview of 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.

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.

10.3 Pure and Impure interpreters

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.

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.

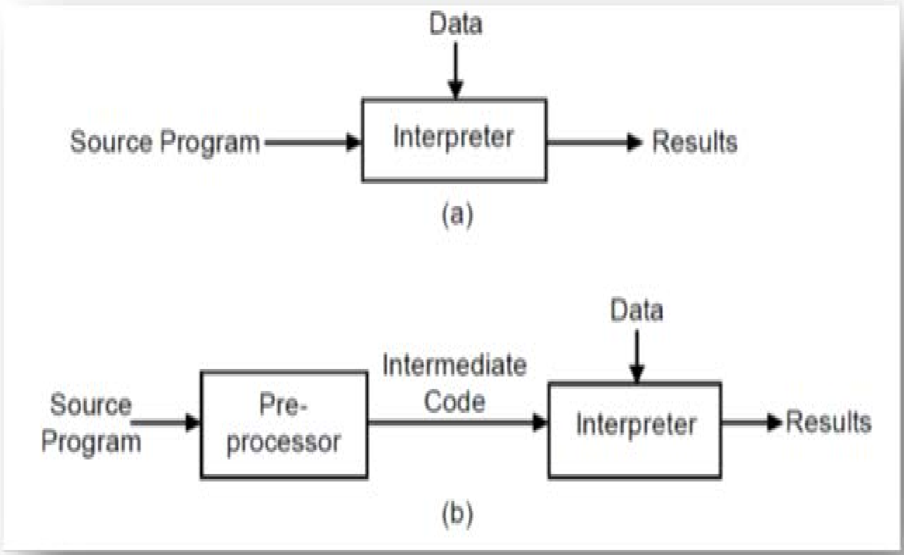


Fig (a) Pure Interpreter (b) Impure Interpreter

10.4 Summary

In this unit we have studied about interpreters and its types such as pure and impure interpreters. The main components of interpreters have been explained.

10.5 Terminal Questions

1. What is difference between interpreter and compiler?
2. What are different types of interpreter?
3. What are main components of interpreter?

॥ सरस्वती नः सुभगा मयस्करत् ॥

Uttar Pradesh Rajarshi Tandon Open University

BCA-1.18

Bachelor in Computer Applications

Block

3

System Software

Linker, Loaders and device Drivers

Printed By : Chandrakal Universal Pvt Ltd. 42/7 Jawahar Lal Neharu Road Prayagraj

UNIT - 11 : LOADERS AND LINKERS

Structure

11.0 INTRODUCTION

In this unit we will understand the concept of linking and loading. As discussed earlier the source program is converted to object program by assembler. Linking combines various pieces of code, module or data together to form a single executable unit that can be loaded in memory. A loader is a program that takes object program and prepares it for execution and loads this executable code of the source into memory for execution.

11.1 Objective

- o learn the basics of loader and linker
- o learn about two pass linking
- o learn about loading schemes
- o learn about the concept of relocation

11.2 Basic loader functions: Design of an Absolute Loader – A Simple Bootstrap Loader

Loading is the process of bringing a program into main memory so that it can run. Each program is loaded into a fresh known address space and the executable is linked to that address. Loader is takes object code as input prepares it for execution and loads the executable code into the memory. Thus loader is actually responsible for initiating the execution process.

The loader is responsible for the activities such as allocation, linking, relocation and loading. Functions of Loader:

1) Allocation: Calculate the size of the program and allocates the space for program in the memory. This activity is called allocation.
2) Linking: Resolves the symbolic references across the object modules by assigning address to all the subroutine and library references. This activity is called linking.
3) Relocation: Based on allocated space, the address dependent locations in the program- ex address constants, etc is adjusted according to allocated space. This activity is called relocation.
4) Loading: Placing program's machine instructions, data and the subroutines into the memory to make program ready for execution. This activity is called loading.

Loader Schemes :

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

1) Compile-and-go loader : The instruction is read line by line, its machine code is obtained and directly place in the main memory at a known address. After completion of assembly process, assign starting address of the program to the location counter. In this scheme some portion of memory is occupied by assembler which is simply wastage of memory. The source code is directly converted to executable form without creating object file. Hence even though there is no modification in the source program it needs to be assembled and executed each time and cannot handle multiple source programs or multiple programs written in different languages.

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 : The execution time will be more in this scheme as every time program is assembled and then executed.

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

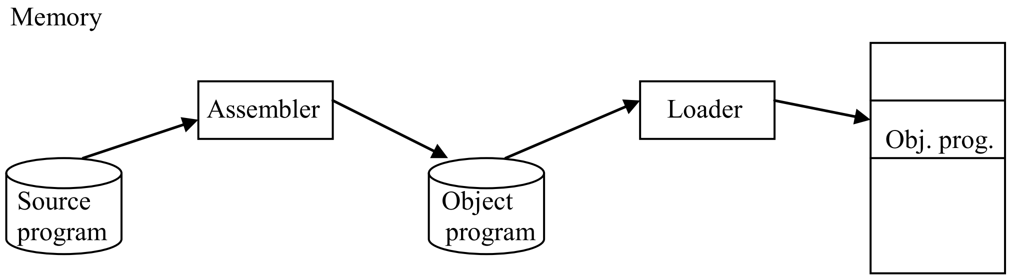
Advantages :

The program need not be retranslated each time while running it, as object program gets generated loader can make use of this object program to convert it to executable form. It is possible to write source program with multiple programs and multiple languages, because of the object programs, loader accepts these object modules to convert it to executable form.

3) Absolute Loader : Absolute loader creates relocated object files and places them at specified locations in the memory. This type of loader is called absolute because no relocation information is needed, as it is obtained from the programmer or assembler. The starting address of every module is stored in the object file. The task of loader becomes very simple 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 is handled by the programmer. The programmer should take care of two things: (i) starting address of each module. If some modification is done in some module then the length of that module may vary. (ii) branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction.

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

1) Allocation is done by either programmer or assembler
2) Linking is done by the programmer or assembler
3) Resolution is done by assembler
4) Simply loading is done by the loader



Loading an object code into memory

Bootstrap loaders

When a computer is first turned on or restarted, bootstrap loader is executed. Bootstrap loader is a simple absolute loader. Its function is to load the first system program to be run by the computer, which is the operating system or a more complex loader that loads the rest of the system.

Bootstrap loader is coded as a fixed-length record and added to the beginning of the system programs that are to be loaded into an empty system. A built-in hardware or a very simple program in ROM reads this record into the memory and transfers control to it. When it is executed, it loads the following program, which is either the operating system itself or other system programs to be run without the operating system.

11.3 Machine dependent loader features Relocation

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

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

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

Similarly, the typical example of absolute address is the constants which are generated by assembler are absolute. The assembler calculates which addresses are absolute and which addresses are relative during the assembly process. During the assembly process the assembler calculates the address with the help of simple expressions.

For example

LOADA(X)+5

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

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

11.4 Program Linking

The output of translator is a program called object module. The linker processes these object modules binds with necessary library routines and prepares a ready to execute program. Such a program is called binary program. The "binary program also contains some necessary information about allocation and relocation. The loader then load s this program into memory for execution purpose.

Various tasks of linker are -

1. Prepare a single load module and adjust all the addresses and subroutine references with respect to the offset location.
2. To prepare a load module concatenate all the object modules and adjust all the operand address references as well as external references to the offset location.
3. At correct locations in the load module, copy the binary machine instructions and constant data in order to prepare ready to execute module.

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

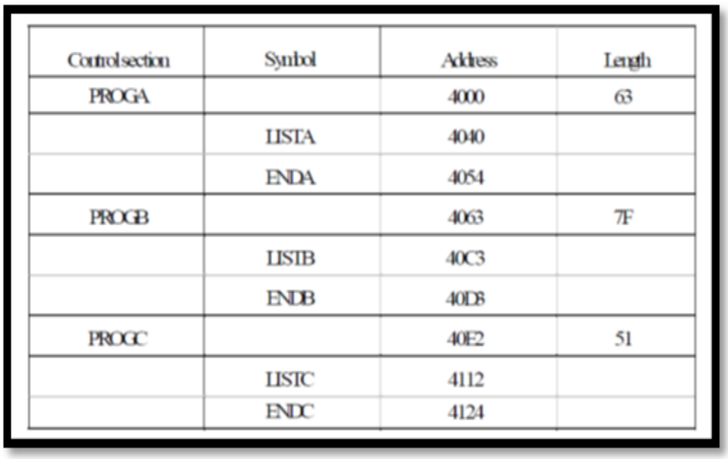
11.5 Algorithm and Data Structures for Linking Loader

The algorithm for a linking loader is considerably more complicated than the absolute loader program. The concept of program linking is used for developing the algorithm for linking loader. The modification records are used for relocation so that the linking and relocation functions are performed using the same mechanism. Linking Loader uses two-passes. ESTAB (external symbol table) is the main data structure for a linking loader.

Pass 1: Assign addresses to all external symbols

Pass 2: Perform the actual loading, relocation, and linking ESTAB

- ESTAB for the example (refer three programs PROGA PROGB and PROGC) given is as shown below. The ESTAB has four entries in it; they are name of the control section, the symbol appearing in the control section, its address and length of the control section.



Control section	Symbol	Address	Length
PROGA		4000	63
	LISTA	4040	
	ENDA	4054	
PROGB		4063	7F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	51
	LISTC	4112	
	ENDC	4124	

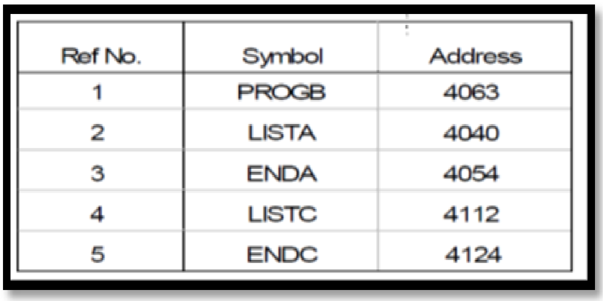
Program Logic for Pass 1: Pass 1 assign addresses to all external symbols. The variables & Data structures used during pass 1 are, PROGADDR (program load address) from OS, CSADDR (control section address), CSLTH (control section length) and ESTAB. The pass 1 processes the Define Record.

Program Logic for Pass 2: Pass 2 of linking loader perform the actual loading, relocation, and linking. It uses modification record and lookup the symbol in ESTAB to obtain its address. Finally it uses end record of a main program to obtain transfer address, which is a starting address needed for the execution of the program. The pass 2 process Text record and Modification record of the object programs.

The efficiency can be improved by the use of local searching instead of multiple searches of ESTAB for the same symbol. For implementing this we assign a reference number to each external symbol in the Refer record. Then this reference number is used in Modification records instead of external symbols.

01 is assigned to control section name, and other numbers for external reference symbols. The object programs for PROGA, PROGB and PROGC are shown below, with above modification to Refer record (Observe R records).

Symbol and Addresses in PROGA, PROGB and PROGC are as shown below. These are the entries of ESTAB. The main advantage of reference number mechanism is that it avoids multiple searches of ESTAB for the same symbol during the loading of a control section



Ref No.	Symbol	Address
1	PROGB	4063
2	LISTA	4040
3	ENDA	4054
4	LISTC	4112
5	ENDC	4124

11.6 Machine-independent loader features

Here we discuss some loader features that are not directly related to machine architecture and design. Automatic Library Search and Loader Options are such Machine-independent Loader Features. Automatic Library Search This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded. The routines are automatically retrieved from a library as they are needed during linking. This allows programmer to use subroutines from one or more libraries. The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program and loaded. The loader searches the library or libraries specified for routines that contain the definitions of these symbols in the main program. Loader Options Loader options allow the user to specify options that modify the standard processing. The options may be specified in three different ways. They are, specified using a command language, specified as a part of job control language that is processed by the operating system, and an be specified using loader control statements in the source program. Here are the some examples of how option can be specified. INCLUDE program-name (library-name) - read the designated object program from a library DELETE csect-name – delete the named control section from the set pf programs being loaded CHANGE name1, name2 - external symbol name1 to be changed to name2 wherever it appears in the object programs

LIBRARY MYLIB – search MYLIB library before standard libraries NOCALL STDDEV, PLOT, CORREL – no loading and linking of unneeded routines Here is one more example giving, how commands can be specified as a part of object file, and the respective changes are carried out by the loader.

LIBRARY UTLIB INCLUDE READ (UTLIB) INCLUDE WRITE (UTLIB) DELETE RDREC, WRREC CHANGE RDREC, READ CHANGE WRREC, WRITE NOCALL SQRT, PLOT The commands are, use UTLIB (say utility library), include READ and WRITE control sections from the library, delete the control sections RDREC and WRREC from the load, the change command causes all external references to the symbol RDREC to be changed to the symbol READ, similarly references to

WRREC is changed to WRITE, finally, no call to the functions SQRT, PLOT, if they are used in the program.

11.7 Automatic Library Search

Previously, the library routines were available in absolute code but now the library routines are provided in relocated form that ultimately reduces their size on the disk, which in turn increases the memory utilization. At execution time certain library routines may be needed. Keeping track of which library routines are required and how much storage is required by these routines, if at all is done by an assembler itself then the activity of automatic library search becomes simpler and effective. The library routines can also make an external call to other routines. The idea is to make a list of such calls made by the routines. And if such list is made available to the linker then linker can efficiently find the set of required routines and can link the references accordingly.

For an efficient search of library routines it desirable to store all the calling routines first and then the called routines. This avoids wastage of time due to winding and rewinding. For efficient automated search of library routines even the dictionary of such routines can be maintained. A table containing the names of library routines and the addresses where they are actually located in relocatable form is prepared with the help of translator and such table is submitted to the linker. Such a table is called subroutine directory. Even if these routines have made any external calls the -information about it is also given in subroutine directory. The linker searches the subroutine directory, finds the address of desired library routine (the address where the routine is stored in relocated form).Then linker prepares aload module appending the user program and necessary library routines by doing the necessary relocation. If the library routine contains the external calls then the linker searches the subroutine directory finds the address of such external calls, prepares the load module by resolving the external references.

11.8 Loader Options Loader design options

Direct Linking Loaders

The direct linking loader is the most common type of loader. This type of loader is a 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

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

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

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

Overlay Structures and Dynamic Loading :

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

11.9 Linkage Editors

The execution of any program needs four basic functionalities and those are allocation, relocation, linking and loading. As we have also seen in direct linking loader for execution of any program each time these four functionalities need to be performed. But performing all these functionalities each time is time and space consuming task. Moreover if the program contains many subroutines or functions and the program needs to be executed repeatedly then this activity becomes annoyingly complex .Each time for execution of a program, the allocation, relocation linking and -loading needs to be done. Now doing these activities each time increases the time and space complexity. Actually, there is no need to redo all these four activities each time. Instead, if the results of some of these activities are stored in a file then that file can be used by other activities. And performing allocation, relocation, linking and loading can be avoided each time. The idea is to separate out these activities in separate groups. Thus dividing the essential four functions in groups reduces the overall time complexity of loading process. The program which performs allocation, relocation and linking is called binder. The binder performs relocation, creates linked executable text and stores this text in a file in some systematic manner. Such kind of module prepared by the binder execution is called load module. This load module can then be actually loaded in the main memory by the loader. This loader is also called as module loader. If the binder can produce the exact replica of executable code in the load module then the module loader simply loads this file into the main memory which ultimately reduces the overall time complexity. But in this process the binder should knew the current positions of the main memory. Even though the binder knew the main memory locations this is not the only thing which is sufficient. In multiprogramming environment, the region of main memory available for loading the program is decided by the host operating system. The binder should also know which memory area is allocated to the loading program and it should modify the relocation information accordingly. The binder which performs the linking function and produces adequate information about allocation and relocation and writes this information along with the program code in the file is called linkage editor. The module loader then accepts this rile as input, reads the information stored in and based on this information about allocation and relocation it performs the task of loading in the main memory. Even though the program is repeatedly executed the linking is done only once. Moreover, the flexibility of allocation and relocation helps efficient utilization of the main memory.

Check your progress

1. Which of the following functions is/ are performed by the loader?

A. Allocate space in memory for the programs and resolve symbolic references between object decks
B. Physically place the machine instructions and data into memory
C. Adjust all address dependent locations, such as address constants, to correspond to the allocated space

11.10 Dynamic Linking

In dynamic linking, a subroutine is loaded and linked to the other programs when it is first called during the execution. Dynamic linking allows several executing programs to share the same copy of a subroutine.

Dynamic linking provides the ability to load the routines only when they are needed. Dynamic linking also avoids the necessity of loading the entire library for each execution. For example, a program may be calling a different routine depending on the input data.

Overlay structure certain selective subroutines can be resident in the memory. It is not necessary to make resident all the subroutines in the memory for all the time. Only necessary routines can be present in the main memory and during execution the required subroutines can be loaded in the memory. This process of postponing linking and loading of external reference until execution is called dynamic linking.

For example suppose the subroutine main calls A,B,C,D then it is not desirable to load A,B,C and D along with the main in the memory. Whether A, B, C or D is called by the main or not will be known only at the time of execution. Hence keeping these routines already before are really not needed. As the subroutines get executed when the program runs. Also the linking of all the subroutines has to be performed. And the code of all the subroutines remains resident in the main memory. As a result of all this is that memory gets occupied unnecessarily. Typically 'error routines' are such routines which can be invoked rarely. Then one can postpone the loading of these routines during the execution. If linking and loading of such rarely invoked external references could be postponed until the execution time when it was found to be absolutely necessary, then it increases the efficiency of overhead of the loader. In dynamic linking, the binder first prepares a load module in which along with program code the allocation and relocation information is stored. The loader simply loads the main module in the main memory. If any external reference to a subroutine comes, then the execution is suspended for a while, the loader brings the required subroutine in the main memory and then the execution process is resumed. Thus dynamic linking both the loading and linking is done dynamically.

Advantages

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

Disadvantages

The linking and loading need to be postponed until the execution. During the execution if at all any subroutine is needed then the process of execution needs to be suspended until the required subroutine gets loaded in the main memory

11.11 Bootstrap Loaders.

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.

11.12 Implementation examples : MSDOS linker.

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

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

Nothing is wrong in it. But from assembler's point of view while generating the code for B, as B is not defined in the segment A, the assembler can not find the value of this symbolic reference and hence it will declare it as an error. To overcome problem, there should be some mechanism by which the assembler should be explicitly informed that segment B is really defined in some other segment C. Therefore whenever segment B is used in segment A and if at all B is defined in C, then B must -be declared as an external routine in A. To declare such subroutine asexternal, we can use the assembler directive EXT. Thus the statement such as EXT B should be added at the beginning of the segment A. This actually helps to inform assembler that B is defined somewhere else. Similarly, if one subroutine or a variable is defined in the current segment and can be referred by other segments then those should be declared by using pseudo-ops INT. Thereby the assembler could inform loader that these are the subroutines or variables used by other segments. This overall process of establishing the relations between the subroutines can be conceptually called a_ subroutine linkage.

For example

```
MAIN START
EXT B
.
.
.
CALL B
.
.
END
B START
.
. RET
END
```

At the beginning of the MAIN the subroutine B is declared as external.

When a call to subroutine B is made, before making the unconditional jump, the current content of the program counter should be stored in the system stack maintained internally. Similarly while returning from the subroutine B (at RET) the pop is performed to restore the program counter of caller routine with the address of next instruction to be executed.

11.13 Summary

In this unit we have learnt linker and loader. Linking is the process of combining various pieces of code and data together to form a single executable. Loading is the process of bringing a program into main memory so that it can run. We have learnt the concept of relocation, symbol resolution and dynamic linking.

11.14 Terminal Questions

1. What is a loader? What are the functions of loader?
2. What is bootstrap loader?
3. Define relocating loader?
4. What is linker?
5. Why origin of a program may have to be changed by the linker or loader?
6. Write meaning of translation time, linked address and load time.
7. What is static linking?
8. What is dynamic linking?
9. Write one limitation of absolute loader.
10. Which information are stored in binary program of absolute loader?
11 Write one function of bootstrap loader.
12. Which information are stored in binary program of relocating loader.
13. What is object file?
14. List components of object module.
15. List data structure needed for linking loader.
16. Which tasks are performed in first pass of algorithm for linking loader?
17. Which tasks are performed in second pass of algorithm for linking loader?
18. Write one advantage of reference number mechanism in linking loader algorithm.
19. When it is better to use linkage editor?
20. What is a dis-advantage of linkage editor?

UNIT - 12 : DEVICE DRIVERS

Structure

12.0 INTRODUCTION

A device driver is a computer program that operates or controls a particular type of device that is attached to a computer. A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

Device Control

Almost every system eventually maps to a physical device. With the exception of the processor, memory, and a few other entities, any and all device control operations are performed by code that is specific to the device being addressed . That code is code device driver.

The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape drive.

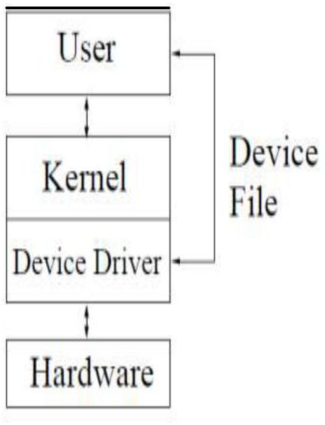
12.1 Objective

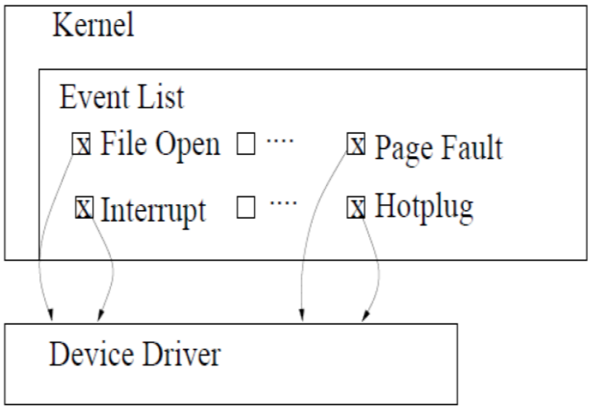
To learn about device drivers, its types-character and block. To know the details of Unix device drivers.

12.2 Design and anatomy of UNIX device driver

Anatomy of a Device Driver

A device driver has three sides :one side talks to the rest of the kernel, one talks the hardware, and one talks to the user.-





Kernel Interface of a Device Driver

In order to talk to the kernel, the driver registers with subsystems to respond to events. Such an event might be the opening of a file, a page fault, the plugging in of a new USB device, etc.

User Interface of a Device driver

Since Linux follows the UNIX model, and in UNIX everything is a file, users talk with device drivers through device files. Device files are a mechanism, supplied by the kernel, precisely for this direct User - Driver interface.

Characteristics of a device driver

There are many different device drivers in the Linux kernel (that is one of Linux's strengths) but they all share some common attributes:

kernel code

Device drivers are part of the kernel and, like other code within the kernel, if they go wrong they can seriously damage the system. A badly written driver may even crash the system, possibly corrupting file systems and losing data,

Kernel interfaces

Device drivers must provide a standard interface to the Linux kernel or to the subsystem that they are part of. For example, the terminal driver provides a file I/O interface to the Linux kernel and a SCSI device driver provides a SCSI device interface to the SCSI subsystem which, in turn, provides both file I/O and buffer cache interfaces to the kernel.

Kernel mechanisms and services

Device drivers make use of standard kernel services such as memory allocation, interrupt delivery and wait queues to operate,

Loadable

Most of the Linux device drivers can be loaded on demand as kernel modules when they are needed and unloaded when they are no longer being used. This makes the kernel very adaptable and efficient with the system's resources,

Configurable

Linux device drivers can be built into the kernel. Which devices are built is configurable when the kernel is compiled,

Dynamic

As the system boots and each device driver is initialized it looks for the hardware devices that it is controlling. It does not matter if the device being controlled by a particular device driver does not exist.

In this case the device driver is simply redundant and causes no harm apart from occupying a little of the system's memory.

Major and Minor Numbers

Char devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the */dev*directory. Special files for char drivers are identified by a "c" in the first column of the output of *ls -l*. Block devices appear in */dev* as well, but they are identified by a "b." The focus of this chapter is on char devices, but much of the following information applies to block devices as well.

If you issue the *ls -l* command, you'll see two numbers (separated by a comma) in the device file entries before the date of the last modification, where the file length normally appears. These numbers are the major and minor device number for the particular device. The following listing shows a few devices as they appear on a typical system. Their major numbers are 1, 4, 7, and 10, while the minors are 1, 3, 5, 64, 65, and 129.

crw-rw-rw-	1 root root 1, 3 Apr 11 2002 null
crw-------	1 root root 10, 1 Apr 11 2002 psaux
crw-------	1 root root 4, 1 Oct 28 03:04 tty1
crw-rw-rw-	1 root tty 4, 64 Apr 11 2002 ttys0
crw-rw----	1 root uucp 4, 65 Apr 11 2002 ttyS1
crw--w----	1 vcsa tty 7, 1 Apr 11 2002 vcs1
crw--w----	1 vcsa tty 7, 129 Apr 11 2002 vcsa1
crw-rw-rw-	1 root root 1, 5 Apr 11 2002 zero

Traditionally, the major number identifies the driver associated with the device. For example, */dev/null* and */dev/zero* are both managed by driver 1, whereas virtual consoles and serial terminals are managed by driver 4; similarly, both *vcs1* and *vcsa1* devices are managed by driver Modern Linux kernels allow multiple drivers to share major numbers, but most devices that you will see are still organized on the onemajor-one-driver principle.

The minor number is used by the kernel to determine exactly which device is being referred to. Depending on how your driver is written (as we will see below), you can either get a direct pointer to your device from the kernel, or you can use the minor number yourself as an index into a local array of devices. Either way, the kernel itself knows almost nothing about minor numbers beyond the fact that they refer to devices implemented by your driver.

The Internal Representation of Device Numbers

Within the kernel, the dev_t type (defined in *<linux/types.h>*) is used to hold device numbers—both the major and minor parts. As of Version 2.6.0 of the kernel, dev_t is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number. Your code should, of course, never make any assumptions about the internal organization of device numbers; it should, instead, make use of a set of macros found in <*linux/kdev_t.h*>.

To obtain the major or minor parts of a dev_t, use:

MAJOR(dev_t dev);

MINOR(dev_t dev);

If, instead, you have the major and minor numbers and need to turn them into a dev_t, use:

MKDEV(int major, int minor);

Note that the 2.6 kernel can accommodate a vast number of devices, while previous kernel versions were limited to 255 major and 255 minor numbers. One assumes that the wider range will be sufficient for quite some time, but the computing field is littered with erroneous assumptions of that nature. So you should expect that the format of dev_t could change again in the future; if you write your drivers carefully, however, these changes will not be a problem.

12.3 Types of device driver

Classes of Devices and Modules (Types of Device Drivers)

The Unix way of looking at devices distinguishes between three device types. Each module usually implements one of these types, and thus is classifiable as a *char module*, a *block module*, or a *network module*. This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code. Good programmers, nonetheless, usually create a different module for each new functionality they implement, because decomposition is a key element of scalability and extendability.
The three classes are the following:

Character devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the *open*, *close*,*read*, and *write* system calls. The text console (*/dev/console*) and the serial ports (*/dev/ttyS0* and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as */dev/tty1* and */dev/lp0*. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using *mmap* or *lseek*.

Block devices

Like char devices, block devices are accessed by filesystem nodes in the */dev* directory. A block device is something that can host a filesystem, such as a disk. In most Unix systems, a block device can be accessed only as multiples of a block, where a block is usually one kilobyte of data or another power of 2. Linux allows the application to read and write a block device like a char device -- it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node and the difference between them is transparent to the user. A block driver offers the kernel the same interface as a char driver, as well as an additional block-oriented interface that is invisible to the user or applications opening the */dev* entry points. That block interface, though, is essential to be able to *mount* a filesystem.

Network interfaces

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Though both Telnet and FTP connections are stream oriented, they transmit using the same device; the device doesn't see the individual streams, but only the data packets.

Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as */dev/tty1*is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the filesystem.

Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of *read* and *write*, the kernel calls functions related to packet transmission.

12.4 General design of UNIX character device driver

Character Device Drivers

One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with. The necessary function for this task is register_chrdev_region, which is declared in
<linux/fs.h>: int register_chrdev_region(dev_t first, unsigned int count,char *name);

Here, first is the beginning device number of the range you would like to allocate. The minor number portion of first is often 0, but there is no requirement to that effect. count is the total number of contiguous device numbers you are requesting. Note that, if count is large, the range you request could spill over to the next major number; but everything will still work properly as long as the number range

you request is available. Finally, name is the name of the device that should be associated with this number range; it will appear in /proc/devices and sysfs.
As with most kernel functions, the return value from register_chrdev_region will be 0 if the allocation was successfully performed. In case of error, a negative error code will be returned, and you will not have access to the requested region. register_chrdev_region works well if you know ahead of time exactly which device numbers you want. Often, however, you will not know which major numbers your device will use; there is a constant effort within the
Linux kernel development community to move over to the use of dynamically-allocated device numbers. The kernel will happily allocate a major number for you on the fly, but you must request this allocation by using a different function:

int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);

With this function, dev is an output-only parameter that will, on successful completion, hold the first number in your allocated range. firstminor should be the requested first minor number to use; it is usually 0. The count and name parameters work like those given to request_chrdev_region. Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with:

void unregister_chrdev_region(dev_t first, unsigned int count);

The open Method

The *open* method is provided for a driver to do any initialization in preparation for later operations. In most drivers, *open* should perform the following tasks:

i. Check for device-specific errors (such as device-not-ready or similar hardware problems)
ii. Initialize the device if it is being opened for the first time
iii. Update the f_op pointer, if necessary
iv. Allocate and fill any data structure to be put in filp->private_data

The first order of business, however, is usually to identify which device is being opened. Remember that the prototype for the *open* method is:

int (*open)(struct inode *inode, struct file *filp);

The release Method

The role of the *release* method is the reverse of *open*. Sometimes you'll find that the method implementation is called *device*_close instead of *device*_release. Either way, the device method should perform the following tasks:

i. Deallocate anything that *open* allocated in filp->private_data
ii. Shut down the device on last close

The basic form of *scull* has no hardware to shut down, so the code required is minimal:The other flavors of the device are closed by different functions because *scull_open* substituted a different filp->f_op for each device.

```
int scull_release(struct inode *inode, struct file *filp)
{
return 0;
```

Check your progress

1. How the kernel identifies the driver? With its -major number?
2. Does the kernel uses the minor numbers to represent the sub-functionalities of the driver?
3. In Linux kernel do we have 256 character drivers and 256 block drivers at the same time?
4. Does the major number identifies the driver associated with the device?

12.5 General design of UNIX block device driver

The first step taken by most block drivers is to register themselves with the kernel. The function for this task is register_blkdev (which is declared in <linux/fs.h>):

```
int register_blkdev(unsigned int major, const char *name);
```

The arguments are the major number that your device will be using and the associated name (which the kernel will display in /proc/devices). If major is passed as 0, the kernel allocates a new major number and returns it to the caller. As always, a negative return value from register_blkdev indicates that an error has occurred. The corresponding function for canceling a block driver registration is:

```
int unregister_blkdev(unsigned int major, const char *name);
```

Here, the arguments must match those passed to register_blkdev, or the function returns -EINVAL and not unregister anything.

Read/ Write Operations 2 methods: (1) Polling, and (2) Interrupt based

i. Direct I/O with polling – the device management software polls the device controller status register to detect completion of the operation; device management is implemented wholly in the *device driver*, if interrupts are not used
ii. Interrupt driven direct I/O – interrupts simplify the software's responsibility for detecting operation completion; device management is implemented through the interaction of a *device driver* and interrupt routine
iii. Polling: CPU is used in a program code loop, to continuously check a device to see if it is ready to accept data or commands, or produce output

Loop (until device ready)

Check device

End loop

CPU is tied up in communication with device until I/O operation is done. No other useful work can be accomplished by the CPU.

Interrupt based

I/O via software polling can be inefficient

Too much time spent by CPU waiting for devices. ie., amount of time between I/O operations is large

Only one CPU; polling ties up this resources

Better to design differently

Since CPU is rare resource, we need to keep it busy processing jobs

Handle I/O as different type of event

CPU has wire called an interrupt request line

Instead of I/o controller setting status [busy] =0 control[command –ready]=0, I/O controller sets the interrupt request line to 1

CPU, as part of instruction processing cycle, automatically checks interrupt request line.

If it is set to 1, this generates an interrupt.

12.6 Summary

In this unit, we have learnt details of Unix device drivers.

12.7 Terminal Questions

1. What are types of device drivers supported in Unix? How many of them are supported?
2. Describe block device driver in Unix.
3. What is a major number and a minor number of a device driver?
4. The kernel identifies the driver with its major number or minor number?
5. What is the minor number range should be ?