Unit-4

Loader

Loader schemes, Compile & go, General loading Scheme, absolute loaders, Subroutine Languages, Relocating loaders, Direct linking loaders, other loading Schemes – Binders, linking loaders, Overlays, Dynamic binders. Design of absolute loader. Design of a direct linking loader Specification of problem, Specification of data structure, format of data bases algorithm.

Defn: Loader is a program that places programs into memory and prepares them for execution.

4.1. Functions of Loader

The loader is responsible for the activities such as allocation, linking, relocation and loading
Allocation: allocating the space for program in the memory, by calculating the size of the program.
Linking: It resolves the symbolic references (code/data) between the object
Relocation: Address dependent locations in the program, such address constants must be adjusted according to allocated space
Loading: Physically places all the machine instructions and data into the memory

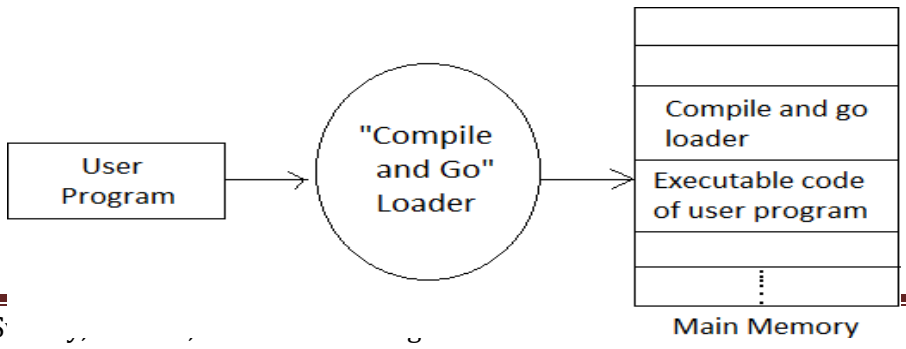
4.2. Loaders Scheme or types of Loader:

Based on the above four functions the loader is divided into different types, they are

i. Compile and go loader or Assemble and go loader
ii. General loader scheme
iii. Absolute loader
iv. Direct linking loader
v. Relocating loader
vi. Dynamic linking loader

4.2.1 Compile and go loader or Assemble and go loader

In this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address. That means the assembler runs in one part of memory and the assembled machine instructions and data is directly put into their assigned memory locations. After completion of assembly process, assign starting address of the program to the location counter.
Ex: WATFOR-77



Advantages

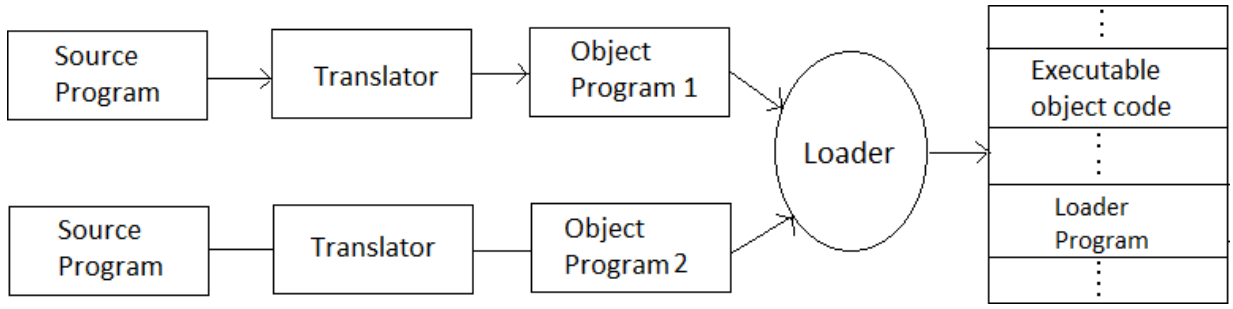
- Easy to implement,

Disadvantages

- Portion of memory is wasted because combination of assembler and loader activities, this combination program occupies large block of memory
- There is no production of .obj file
- It cannot handle multiple source programs or multiple programs written in different languages
- The execution time will be more in this scheme as every time program is assembled and then executed

4.2.2. 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
- There is no wastage of memory, because assembler is not placed in the memory
- It is possible to write source program with multiple programs and multiple languages

4.2.3 Absolute Loader

Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in the memory. This type of loader is called absolute because no relocation information is needed; rather it is obtained from the programmer or assembler.

The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file. In this scheme the programmer or assembler should have knowledge of memory management. The resolution of external references or linking of different subroutines are the issues which need to be handled by the programmer.

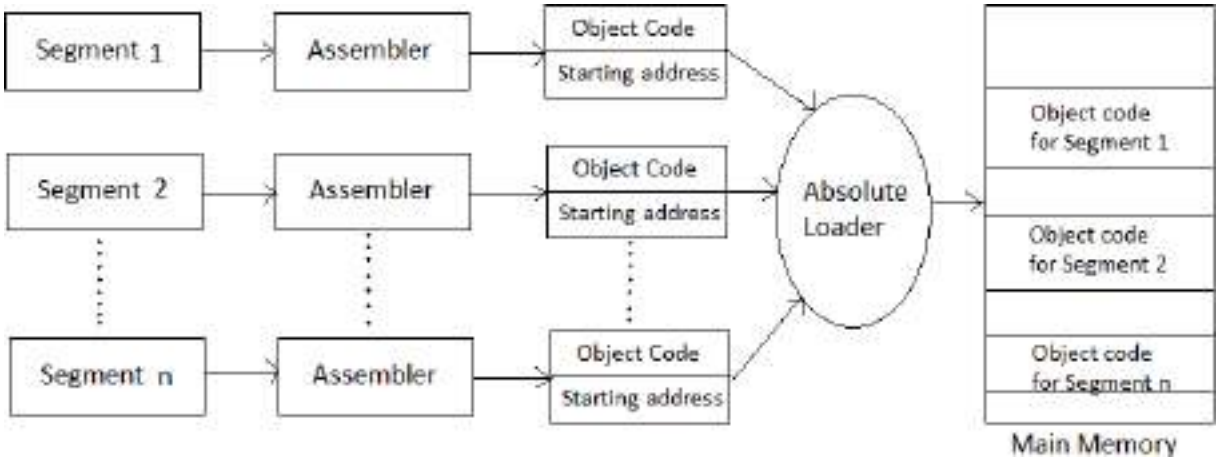
The programmer should take care of two things: first thing is:

i. Specification of starting address of each module to be used. If some modification is done in

some module then the length of that module may vary. This causes a change in the starting

address of immediate next modules, its then the programmer's duty to make necessary changes in the starting addresses of respective modules.

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



Thus the absolute loader is simple to implement in this scheme

i. Allocation is done by either programmer or assembler
ii. Linking is done by the programmer or assembler
iii. Resolution is done by assembler
iv. Simply loading is done by the loader

4.2.4 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 cannot find the value of this symbolic reference and hence it will declare it as an error.

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

To declare such subroutine as external, we can use the assembler directive EXT. Thus the statement such as EXT B should be added at the beginning of the segment A. This actually helps to inform assembler that B is defined somewhere else. This overall process of establishing the relations between the subroutines can be conceptually called a *subroutine linkage*.

4.2.5 Direct Linking Loaders

The direct linking loader is the most common type of loader. The loader cannot have the direct access to the source code. The assembler should give the following information to the loader

i. The length of the object code segment
ii. The list of all the symbols which are not defined in the current segment but can be used in the current segment.
iii. 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 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.

There are 4 types of cards available in the direct linking loader. They are

i. ESD-External symbol dictionary
ii. TXT-card
iii. RLD-Relocation and linking dictionary
iv. END-card

i. *ESD card:* It contains information about all symbols that are defined in the program but reference somewhere, It contains:

- Reference number
- Symbol name
- Type Id
- Relative location
- Length

There are again ESD cards classified into 3 types of mnemonics. They are:

i. SD [Segment Definition]: It refers to the segment definition
ii. LD; It refers to the local definition
iii. ER: it refers to the external reference they are used in the [EXTRN] pseudo op code

***ii. TXT Card*:** It contains the actual information are text which are already translated.

iii. RLD Card: This card contains information about location in the program whose contexts depends on the address at which the program is placed.

In this we are used '+' and '–'sign, when we are using the '+' sign then no need of relocation, when we are using '-'sign relocation is necessary.

The format of RLD contains:

i. Reference number
ii. Symbol
iii. Flag
iv. Length
v. Relative location

iv. END Card: It indicates end of the object program.

4.2.6 Design of direct linking loader: Here we are taking PG1 and PG2 are two programs. The relative address and secure code of above two programs is written in the below

ESD Cards: In an ESD card table contains information necessary to build the external symbol dictionary or symbols table. In the above source code the symbols are PG1, PG1ENT2, PG2, and PG2ENT1

Source card reference	*Name*	*Type*	*Id*	*Relative address*	*length*
1	PG1	SD	01	0	60
2	PG1ENT1	LD	-	20	-
2	PG1ENT2	LD	-	30	-
3	PG2	ER	-	-	-
3	PG2ENT1	ER	-	-	-

Here, the PG1 is the segment definition it means, the header of program. PG1ENT1 and PG1ENT2 those are the local definition of program1, so that we are using the type LD. PG2 and PG2ENT1 those are using the EXTRN pseudo op code, so that we are using the type ER.

Text cards: The format of card will be

Source card reference	*Relative address*	*Content*	*Comments*
6	40-43	20	
7	44-47	45	=30+15
8	48-51	7	=30-20-3
9	52-55	0	Unknown to PG1
10	56-60	-16	-20+4

RLD Card:

Source card reference address	*ESD ID*	*Length [bytes]*	*Flag + or -*	*relative*
6	02	4	+	40
7	02	4	+	44
9	03	4	+	52
10	02	4	+	56
10	03	4	+	56
10	02	4	-	56

Specification of data structure: ***Pass1 database:***

i. Input object decks
ii. *Initial Program Load Addresses [IPLA]:* The IPLA supplied by the programmer or operating system that specifies the address to load the first segment.
iii. *Program Load Address counter [PLA]:* It is used to keep track of each segments assigned location
iv. *Global External Symbol Table [GEST]:* It is used to store each external symbol and its corresponding assigned core address

v. A copy of the input to be used later by pass2

vi. A printed listing that specifies each external symbol and its assigned value

Pass2 database:

i. A copy of object program is input to pass2
ii. The Initial Program Load Address [IPLA]
iii. The Program Load Address counter [PLA]
iv. A table the Global External Symbol Table [GEST]
v. The Local External Symbol Array [LESA]: which is used to establish a correspondence between the ESD ID numbers used on ESD and RLD cards and the corresponding External symbols , Absolute address value

Format of data bases:

i. *Object deck:* The object deck contains 4 types of cards
ii. ESD Card format:

Source card reference	Name	Type	ID	Relative address	length

iii. TEXT Card:

Source card reference address	Relative address	content

iv. RLD Card:

Source card references	ESD ID	Length	Flag + or -	Relative address

v. *Global External Symbol Table (GEST):* It is used to store each external symbol and its corresponding core address.

External symbol (8 bytes) character	Assigned core (4 bytes) address decimal
"PG1bbbbb"	104
"PG1ENT1b"	124

vi. *Local external symbol array (LESA):* The external symbol is used for relocation and linking purpose. This is used to identify the RLD card by means of an ID number rather than the symbols name. The ID number must match an SD or ER entry on the ESD card

Assigned core address of corresponding symbol [4 bytes]
104
124
134
….
….

This technique saves space and also increases the processing speed.

Other loading segments:

Binders:

In order to avoid the disadvantages of direct linking divides the loading process into two separate programs:

i. A binder
ii. A module loader

Binder is a program that performs the function as direct linking loader in binding together. It outputs the text as a file or card deck, rather than placing the relocated and linked text directly into memory. The output files are in format ready to be loaded and are called a load module. The module loader loads the module into memory. The binder performs the function of the allocation, relocation and linking

The modules loader performs the function of loading. There are 2 major classes of binders:

i. *Core image builder:* A specific memory allocation of the program is performed at a time that the subroutines are bound together. It is called a core image module and the corresponding binder is called a core image builder

   Advantages: Simple to implement and Fast to execution

   Disadvantages: Difficult to allocate and load the program and Linkage editor

ii. *Linkage editors:* The linkage editor can keep track of relocation information so that the resulting load module can be further relocated and their care the module loader must performs additional allocation and relocation as well as loading but it does not worry about the problem of linking.

   Advantages: More flexible allocation and loading scheme

   Disadvantages: Implementation is so complex

Difference between macro and subroutine

Macro	Subroutine
Macro can be called only in the program it is defined	Subroutine can be called from other programs also.
Macro can have maximum 9 parameters.	Can have any number of parameters.
Macro can be called only after its definition.	This is not true for Subroutine.
A macro is defined inside: DEFINE … …. END-OF-DEFINITION.	Subroutine is defined inside: FORM ….. ….. ENDFORM.
Macro is used when same thing is to be done in a program a number of times.	Subroutine is used for modularization.
Macro doesn't have any return statement...	Subroutine can have return statement
Execution time needed for a macro is much lesser than subroutine	Execution time is high

Difference between BLAR and USING

BALR	USING
BALR is an machine op code	USING is pseudo op
BALR: Branch and Link Register.	USING itself USING
BALR loading the address of next instruction	USING indicates to the assembler which general register to use as a base register and what contents will be
Sets the register with the next address	Only provides information to the assembler

VVIMP: Construct the parse tree\ Decision tree for the following arithmetic expression

COST=RATE*(START-FINISH)+2*RATE*(START-FINISH-100);

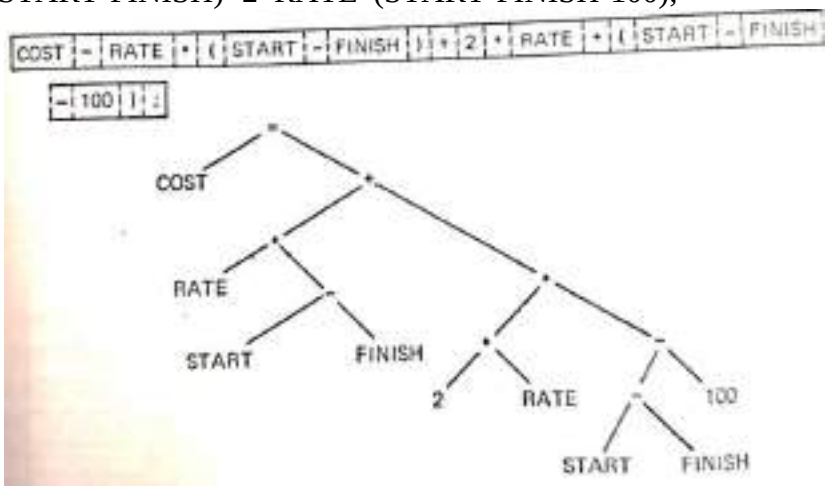


Fig. Decision tree/ Parse tree / intermediate form of arithmetic statement

Expected Question from Unit -4 for the Examination

ONE Marks Questions

1. What is loader?
2. What is nibble? *4 bit memory is called nibble*
3. What is the tool used in lexical analysis phase? *Tool is LEX*
4. What is Binder?
   The program which performs allocation, relocation, and linking called binder.
5. What is overlays?
   The inter dependency of the segments can be specified by a tree like structure called static overlay structure.
6. What is dynamic loading?
   Dynamic loading is the process in which one can attach a shared library to the address space of the process during execution
7. What is direct linking loader?
   A Direct linking loader is a general relocating loader it allows the programmer to use multiple procedure and multiple data segments.
8. What is relocating loader?
   The relocating loader will load the program anywhere in memory, altering the various addresses as required to ensure correct referencing.
9. Expand IPLA, PLA, GEST, LESA,

THREE Marks Questions

1. What are the function of loader?
2. Explain subroutine linkage.
3. Explain binders?
4. What are the cards used in direct linking loader.
5. What is general loader scheme? Mention its advantage and disadvantage

FIVE Marks Questions

1. Explain general loader scheme?
2. With neat diagram explain compile and go loader scheme.
3. Explain databases used in pass1 and pass2 loader scheme?
4. Explain database format of loader.
5. Differentiate subroutine and macro
6. Explain ESD, RLD, TXT and END cards

SEVEN Marks Questions

1. With neat diagram explain general loader scheme. Mention its advantages and disdav.
2. With neat diagram explain compile and go loader scheme.
3. Explain Absolute loader scheme
4. Describe the four cards used in direct linking loader
5. Design direct linking loader.

Unit-5

Compiler

General model of compiler. Simple Structure of Compiler, 7 Phases of Compilers: Lexical analysis, Syntax analysis, Semantic analysis, Intermediate (machine-independent) code generation, Intermediate code optimization, Target (machine-dependent) code generation, Target code optimization

Compiler: A compiler is a magic box that converts the high level language program into machine language program.

OR

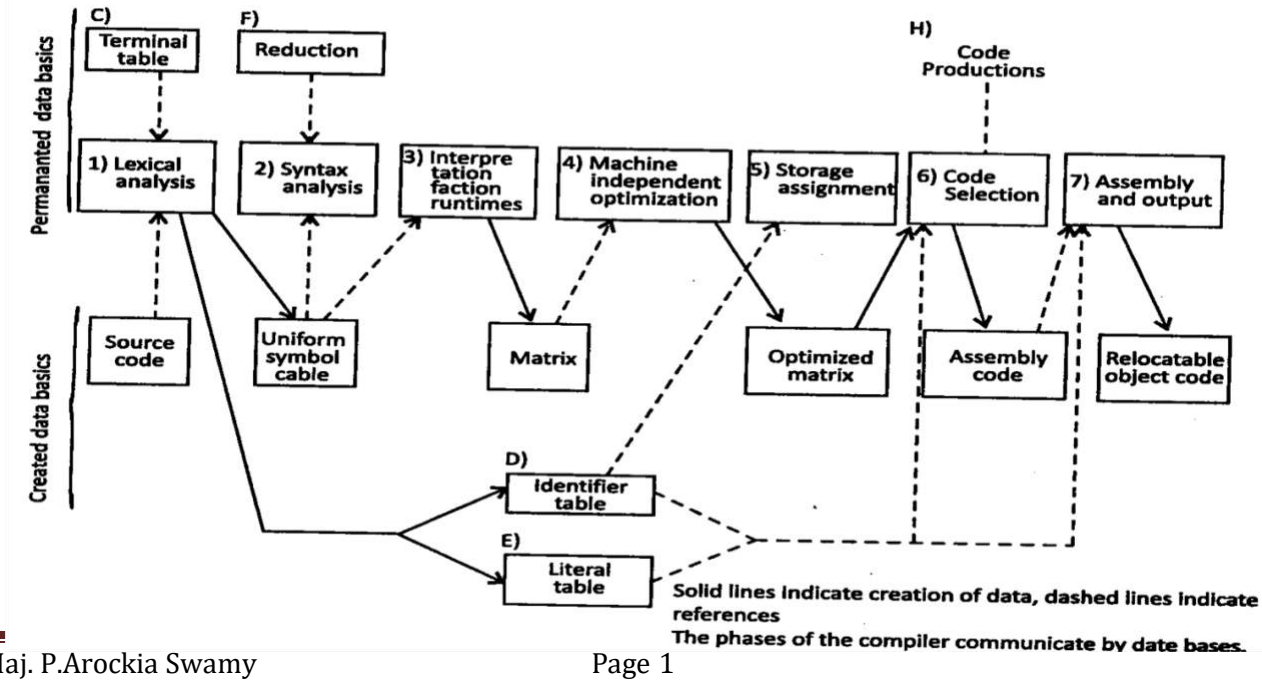
A compiler is a software program that converts high-level language into a machine language, which can be executed by a computer.

High Level Language → Compiler → Machine Level Language

5.1 General Model of Complier or Simple Structure of Compiler

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. The general model of a compiler consists of 7 distinct phases:

1. *Lexical analysis*
2. *Syntax analysis*
3. *Integration phase*
4. *Machine independent optimization*
5. *Storage assignment*
6. *Code generation*
7. *Assembly and output*



i. *Lexical analysis:* Recognition of basics element or tokens and creation of uniform symbols.
ii. *Syntax analyses*: Recognition of basics syntactic construct through reduction table.
iii. *Interpretation phases:* It describes the definition of exact meaning, creation of matrix and tables by action routines.
iv. *Machine independent optimization:* Creation of more optimal matrix by removing the duplicate entries in the matrix table.
v. *Storage assignment:* It makes entries in the matrix that allow code generation to create code that allocates dynamic storage and also the assembly phase to reserve the proper amount of storage.
vi. *Code generation:* A macro processor is used to produce more optimal assembly code.
vii. *Assembly and Output:* It resolving symbolic address and generating the machine language.

5.2 The database used

i. *Source code:* The program written by user or the user program
ii. *Uniform symbol table:* It consist list of all the tokens or basic elements as they appear in the program created by lexical analysis phase and given as input syntax analysis and interpretation phase
iii. *Terminal table:* This table is created by lexical analysis phase and contains all variable in the program
iv. *Identifier table:* It contains all variable in the program and temporary storage and information needed to reference allocate storage for the variables. This table is created by lexical analysis.
v. L*iteral tables:* It contains all contents in the program
vi. *Reductions:* It is a permanent table of decision rules in the form of pattern for matching with the uniform symbols table to discover synthetic structure
vii. *Matrix:* Matrix is created by the intermediate form of the program which is created by the action routine. It is optimized and then used for code generation
viii. *Code productions:* It is permanent table of definition. There is one entry defining code for each matrix operator
ix. *Assembly code:* The assembly language variation of the program which is created by the code generation phase and it is input to the assembly phase
x. *Re-locatable object codes:* The final output of the assembly phase ready to be used as input to loader

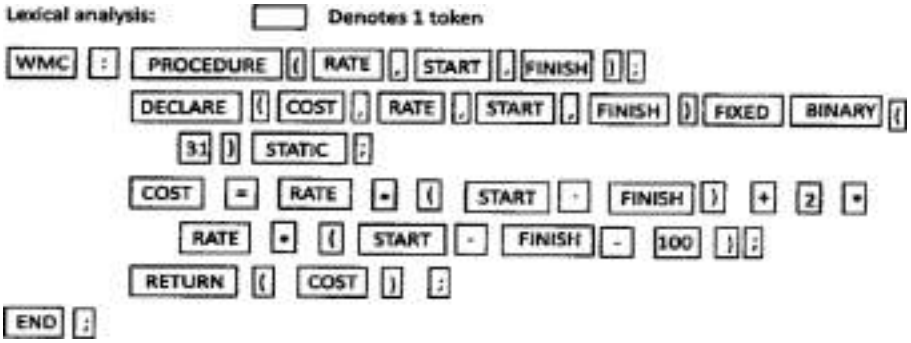
Consider a simple example

```
WCM: procedure (Rate, Start, finish);
Declare (Cost, Rate, Start, Finish) fixed binary (31) static;
Cost=Rate *(Start- Finish) +2*Rate*(Start-Finish-100);
Return (Cost);
End;
```

5.3.1 Lexical Analysis Phase

The lexical phase performs the following three tasks:

1. Recognize basic elements are tokens present in the source code
2. Build literal and an identifier table
3. Build a uniform symbol table



Recognizing the basic elements- Tokens of example program

Database: Lexical phase involves the manipulation of 5 databases

i. Source program
ii. Terminal table
iii. Literal table
iv. Identifier table
v. Uniform symbol table

i. Source program: The original form of the program created by the user

ii. Terminal Table: It is a permanent database it consist of 3 fields

- Symbol: operators, keywords and separators [(,;,:]
- Indicators: values are YES or NO
  Yes=> operators, separators
  No=> Keywords
- Precedence: Used in later phase

Step	Symbol	Indicator	Precedence
1	:	Yes	
2	;	Yes	
3	(	Yes	
4	)	Yes	
5	,	Yes	
6	*	Yes	
7	Declare	No	
8	Procedure	No	
9	+	Yes	
10	*	Yes	
11	Rate	No	
12	Start	No	

iii. Literal table: It describes all literals constants used in the source program. It consists of 6 fields:

Literals	*Base*	*Scale*	*Precision*	*Other information*	*Address*
31	Decimal	Fixed	2		
2	Decimal	Fixed	1		
100	decimal	fixed	3		

iv. Identifier Table: It describes all identifiers used in the source program. It consists of three fields

Name	*Data attribute*	*Address*
WCM		
RATE		
START		
FINISH		
COST		

v. Uniform symbol tables: It consist list of all the tokens or basic elements as they appear in the program created by lexical analysis phase. There is one uniform symbol for every token in the program. It consists of 2 fields:

Table class	**Index**	**Token**
IDN	1	WCM
TRM	1	:
TRM	8	Procedure
TRM	3	(
IDN	2	Rate
TRM	5	,
IDN	3	Start
TRM	5	,
IDN	4	Finish
TRM	4	)
TRM	2	;

Algorithm:

Step1: Parse the input character string into tokens

Step2: Make appropriate entries in to the table

Implementation:

i. The input string is separated into tokens by break character. Brake characters are denoted by the contents of a special field in the terminal table

ii. Lexical analysis 3 types of tokens: Terminal symbols[TRM], Identifiers [IDN],Literals [LIT]

iii. if symbol== TERMINAL table then

Create Uniform Symbol Table of type TRM

else if symbol==IDENTIFIER table then

Create Uniform Symbol Table of type IDN

else End if

Create
Uniform
Symbol
Table of
type LIT

5.3.2 Syntax Phase:

The functions of the syntax phase are

1. To recognize the major construct of the language
2. To call the appropriate action routines that will generate the intermediate form or matrix form the constructs

Databases: The Syntax analysis phase involves the manipulation of 3 databases

i. Uniform symbol table: The table created a by lexical phase. The uniform symbols are the source of input to the stack which is used by syntax and interpretation phase

Table classes	**Index**

ii. Stack: The stack is a collection of uniform symbol i.e., currently being worked on the stack is organized in LIFO technique.

*iii. Reduction table***:** The syntax rules of the source language are contained in the reduction table The general form of the reduction or rules is:-

Label: old top stack/ action routine/ new top stack/ next reduction

5.3.3 Interpretation Phase:

Interpretation phase is a collection of routines that are called when a constructs recognized. The purpose of action routines is to create an intermediate form of the source program and add the information to the identifier. The interpretation phase interprets the precise meaning into the matrix or identifier table while syntax phase recognize the syntactic constructs.

Databases:

i. *Uniform symbol table*
ii. *Identifier table*
iii. *Stack*
iv. *Matrix:* it is primary intermediate form of the program. A matrix entry consists of a triplet entry where the first element is a uniform symbol denoting the terminal symbol of operator and other two element are uniform symbols denoting the arguments.

Operator	***Operand 1***	***Operand 2***

For ex:

B=A

*A=C*D*(C*D+B)*

	Operator	***Operand 1***	***Operand 2***
M1	=	*B*	*A*
M2	*	*C*	*D*
M3	+	*M2*	*B*
M4	*	*C*	*D*
M5	*	*M4*	*M3*
M6	=	*M5*	*A*

5.3.4 Optimization Phase:

Removing or deleting the duplicate entries in the matrix and modifying all references to the deleted entries is called optimization. Optimization dependent by a compiler are of two types. They are

i. *Machine dependent optimization* is related to the instructions that get generated. So it is incorporated into the code generation phase.
ii. *Machine independent optimization* is done at separated phase

Databases

i. Matrix: This is the major database in the optimization phase

Operator	***Operand 1***	***Operand 2***	***Backward pointer***	***Forward pointer***

ii. Identifier table

iii. Literal table

Algorithm:

Step 1: place the matrix in a form so that common sub expression can be recognized

Step 2: Recognize two sub expression as being equivalent

Step 3: Eliminate one of them

Step 4: Alter the rest of the matrix to reflect the elimination of this entry

For ex:

B=A

*A=C*D*(C*D+B)*

Step1:

	Operator	***Operand 1***	***Operand 2***	***Backward pointer***	***Forward pointer***
M1	=	*B*	*A*	*0*	*2*
M2	*	*C*	*D*	*1*	*3*
M3	+	*M2*	*B*	*2*	*4*
M4	*	*C*	*D*	*3*	*5*
M5	*	*M4*	*M3*	*4*	*6*
M6	=	*M5*	*A*	*5*	*?*

Step 2:

	Opr	***Op1***	***Op2***	***Bk. ptr***	***Fr. Ptr***
M1	=	*B*	*A*	*0*	*2*
M2	*****	***C***	***D***	***1***	***3***
M3	+	*M2*	*B*	*2*	*4*
M4	*****	***C***	***D***	***3***	***5***
M5	*	*M4*	*M3*	*4*	*6*
M6	=	*M5*	*A*	*5*	*?*

Step 3 & 4:

	Opr	***Op1***	***Op2***	***Bk. ptr***	***Fr. Ptr***
M1	=	*B*	*A*	*0*	*2*
M2	*****	***C***	***D***	***1***	***3***
M3	+	*M2*	*B*	*2*	*4*
M4	*	***M2***	*M3*	*3*	*5*
M5	=	*M5*	*A*	*4*	*?*

5.3.5 Storage Assignment Phase:

The purpose of this phase is to

i. *Assign storage to all variables referenced in the source program*
ii. *Assign storage to all literals*
iii. *Assign storage to all temporary locations for intermediate results*
iv. *Ensure that the storage is allocated and appropriate locations are initialized*

The storage assignment phase first scan the identifier table assigns locations to entry with a storage class of static or automatic. Initialize the location counter to zero and also keep track of how much storage it has assigned. For each scanning this phase do the following steps:

i. *Updates the location counter with boundary alignment*
ii. *Assigns the current value of location counter to the address field*
iii. *Calculate the length of storage required by the variable*
iv. *Updates the location counter by adding this length to it.*

The storage allocation creates a matrix entry for varibles as shown below

Storage class	*Size*	*Operand*

Where, Storage classes are: Static, Automatic, Controlled, Base

For each variable that required initialization, the storage allocation phase generates matrix entry as shown below

Initialize	*variable*	*Operand*

The literal table similarly scanned and locations are assigned to each literal and a matrix entry

LIT	*Size*	*Operand*

5.3.6 Code Generation Phase:

The Purpose of the code generation is to produce appropriate code in the form of either assembly or machine language. In this phase Matrix is the input data base and uses the code production table which defines the operators that may appeared in the matrix to produce code.

Data bases:

i. *Matrix*
ii. *Identifier table*
iii. *Literal table*
iv. *Code productions:* it is a permanent database defining all possible matrix operators. The standard code for operators is:

+	*L 1, &operand1* *A 1, &operand2* *ST 1, &N*
-	*L 1, &operand1* *S 1, &operand2* *ST 1, &N*

*	*L 1, &operand1* *M 1, &operand2* *ST 1, &N*
=	*L 1, &operand2* *ST 1, &Operand 1*

For ex: A = B + C - D

Matrix				*Original Code*	*Better Code*
M1	+	B	C	L 1, B A 1, C ST 1, M1	L 1, B A 1, C
M2	-	M1	D	L 1, M1 S 1, D ST 1, M2	S 1, D
M3	=	M2	A	L 1, M2 ST 1, A	ST 1, A

5.3.7 Assembly Phase:

The task of assembly phase depends on how much has been done in the code generation phase. The assembly phase must do

i. *Resolve label reference in the object program*
ii. *Calculate address*
iii. *Generate machine language instructions*
iv. *Generate storage and literals*
v. *Format the appropriate information for the loader*

Databases:

i. Identifier Table
ii. Literal table
iii. Object code

Algorithm: The assembly phase

Step 1: Scans the object code to resolving all label references and producing TXT cards
Step2: Then scans the identifier table to create ESD (External Symbol Directory) cards
Step 3: Using TXT cards and ESD cards create RLD (ReLocation Directory) cards.

5. 4 Passes of a Compiler

The following diagram depicts a flowchart of a compiler.

Pass1: It corresponds to the lexical analysis of a compiler. It scans the source program and creates the identifiers, literals and uniform symbol tables.

Pass2: It corresponds to syntax and interpretation phases. Pass2 scans the uniform symbol table produces the matrix.

Pass3 through Pass N-3 means Pass4: They correspond to the optimization phase.

Pass N-2: Pass 5: It corresponds to the storage assignment phase.

Pass N-1: Pass 6: It corresponds to code generation phase. It scans the matrix.

Pass N: Pass 7: It corresponds to Assembly and output phase.

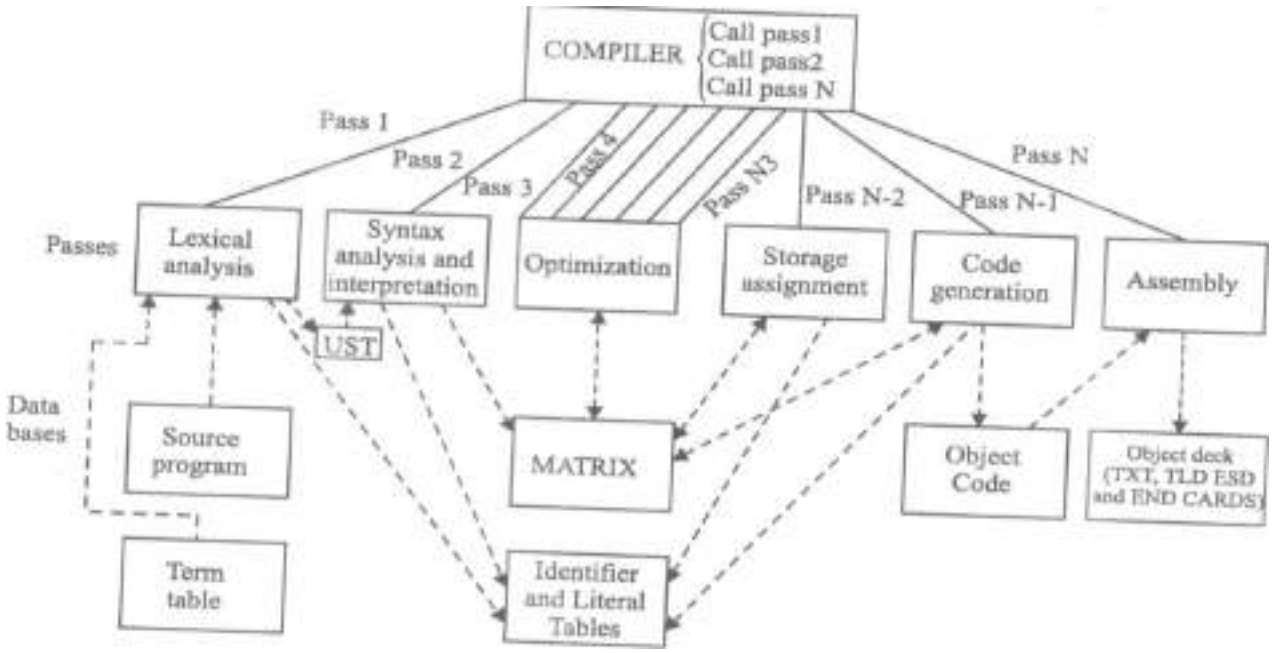


Fig: Passes of compiler

LIST OF COMPILERS

i. *Ada compilers*
ii. *ALGOL compilers*
iii. *BASIC compilers*
iv. *C# compilers*
v. *C compilers*
vi. *C++ compilers*
vii. *COBOL compilers*
viii. *Common Lisp compilers*
ix. *ECMAScript interpreters*
x. *Fortran compilers*
xi. *Java compilers*
xii. *Pascal compilers*
xiii. *PL/I compilers*
xiv. *Python compilers*
xv. *Smalltalk compilers*

Expected Question from Unit -5 for the Examination

ONE Marks Questions

1. What is compiler?
2. What is lexical analysis
3. Define source program
4. What is optimization
5. Mention three tasks of lexical analysis phase

THREE Marks Questions

1. What are tokens? Give an example
2. Explain interpretation phase
3. Explain storage assignment phase

FIVE Marks Questions

1. Explain code generation phase with an example
2. With an example explain optimization phase

SEVEN Marks Questions

1. With neat diagram explain General model (Structure or Block diagram) of compiler
2. Explain the databases used in compiler design
3. With an example explain lexical analysis phase