Q1) a) Errors detected by assembler in :-

PASS 1 :

a) duplicate label
b) RESW or RESB has invalid operand.
c) location counter exceeds 1048575 (ie $2^{20}-1$)
d) Warning: extended addressing not allowed for this case. (ignored)
e) warning: Symbol too long - truncated to SYMBSIZE - 1
f) warning: BYTE directive has no close quote (added)
g) warning: X case BYTE directive has odd digit count (padded by 0)
h) warning: Assembler directive not implemented (ignored)
i) warning: unknown qualifier for operand (ignored)
j) warning: missing or invalid START statement (START assumed to be 0)
k) warning: missing END statement. (END value = START value is used)

PASS 2 :

a) out of range for PC-relative addressing & BASE is not in effect. ~~of range for~~
b) Out of range for PC-relative & BASE relative addressing.
c) Operand not found.
b) Bad Byte directive in X case.
e) invalid WORD operand.
f) invalid operand
g) invalid regester id
h) invalid register ID
i) invalid SHIFT amount
j) indexed/immediate not allowed

Q1)

b) Define Loader. What are different functions of a Loader?

→ A loader is a program which accepts the various objects program decks and prepares these decks for the execution by loading them into the main memory.

→ Functions of a Loader :

- Fundamentally, loader brings object program into memory and helps to start its execution.
- The object program contains translated instructions and data from the source program. It also specifies addresses in memory where these items are to be loaded.
- To perform above mentioned functions, the loader performs following subfunctions :

1. Allocation :

It is the process of allocation of space in memory for object program.

2. Linking :

It is the process of combining 2 or more object programs and resolve symbolic refrences between object decks. It also supplies the information needed to allow to refrence between

them.

3. Relocation :
It modifies the object program so that it can be loaded at an address different from the location originally specified and adjusts address dependant locations.

4. Loading :
It is the process of physically placing machine instructions in binary format into memory & loading pointers to that segment in memory for execution.

PARSERS

TOP-DOWN

- It creates parse tree from the root and expands it till all leaves are reached.
- This parsing technique uses Left Most Derivation (LMD)
- Main decision involves selection of production rules that construct the string.
- Examples are :
  a) Recursive Descent Parser.
  b) LL Parser
  c) Packrat Parser
  d) Unger Parser

BOTTOM-UP

- It creates parse tree by first identifying all leaves and then connecting them until it reaches the root.
- This parsing technique uses Right Most Derivation (RMD)
- Main decision involves selection of production rules that reduce string to starting symbols.
- Examples are :
  a) Precedence Parser
  b) Bounded Context Parsing
  c) LR parser
  d) SLR parser

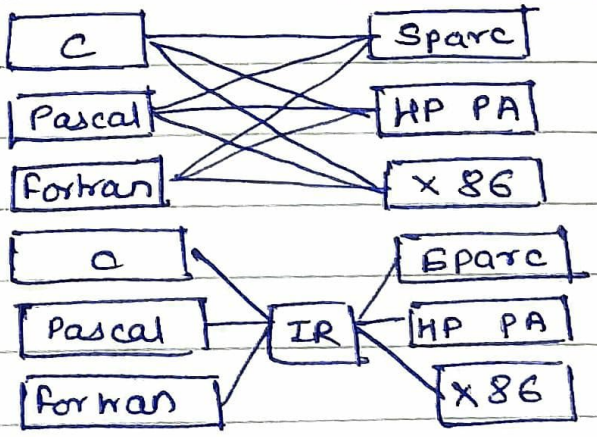
Q1) d) An intermediate representation (IR) is a data structure that is constructed from input data to a program, and from which all or part of the output data of the program is constructed in turn.

→ Need of Intermediate Code Generation:
Generation of IR from source code is called intermediate code generation. It is needed because:

a) Use of IR inbetween P1 and P2 of language Processor nullifies the rereading of data during P2 which results in no unnecessary repeated efforts, no wastage of memory and no wastage of processing time.

b) Use of IR lets the process of machine code generation more streamline because IR is architecture independent which provides clean & abstract machine language. which is independant of source language.

c) Diagramatically it reduces



To

which is simpler and less redundant.

→ Intermediate code generation forms are:

1) Three-address code:

- A statement involving no more than 3 refrences (2 for operand and 1 for result) is known as 3 address statement.
- 3 address statement is of the format $x = y$ op z where $x, y, \& z$ will have memory location.
- Sometimes a statement might contain less than 3 memory locations.
- Eg for expression $a+b * c+d$:

$T1 = b * c$

$T2 = a + T1$

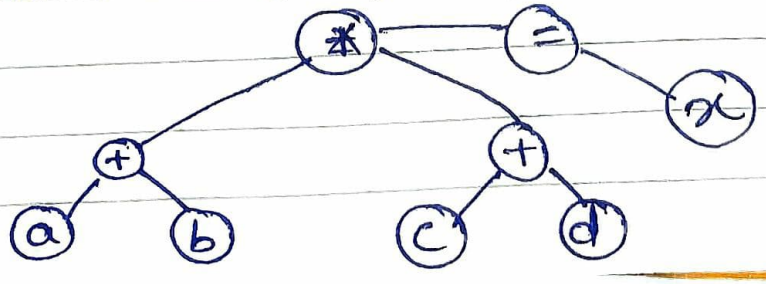
$T3 = T2 + d$

and T1, T2 and T3 are temp. variables.

2) Syntax Tree:

Syntax tree is a condensed form of parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are all operators while the ~~leave~~ leaf nodes are ~~operators~~ operands.

Example: $x = (a+b) * (c+d)$ is represented by



Q2) a)

→ • A grammar is said to be left recursive if it has a non-terminal say A such that there is a derivative $A \Rightarrow A\alpha$ for some string α.

• A predictive parser (a top-down parser without backtracking) insists the grammar must be left factored.

• Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for top-down parser.

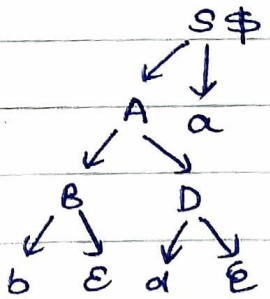
→ Let us consider the following grammar:

(1) $S \rightarrow Aa$

(2) $A \rightarrow BD$

(3) $B \rightarrow b \mid \varepsilon$

(4) $D \rightarrow d \mid \varepsilon$



First(S) = {b, d, a}

First(A) = ~~{b,d}~~ {b, d, ε}

First(B) = {b, ε}

First(D) = {d, ε}

Follow(S) = {$}

Follow(A) = {a}

Follow(B) = {d, a}

Follow(D) = {a}

Generated string in canonical form = $(b \mid \varepsilon) \cdot (d \mid \varepsilon) \cdot a\ \$$

Ans 6] The phases of compiler are given as follows

1] Lexical Analyzer

2] Syntax Analyzer

3] Semantic Analyzer

4] Intermediate Code Generator

5] Code optimizer

6] Code generator

Statements : int a, b, c = 1;

a = a * b - 5 * 3 / c

Source Program:

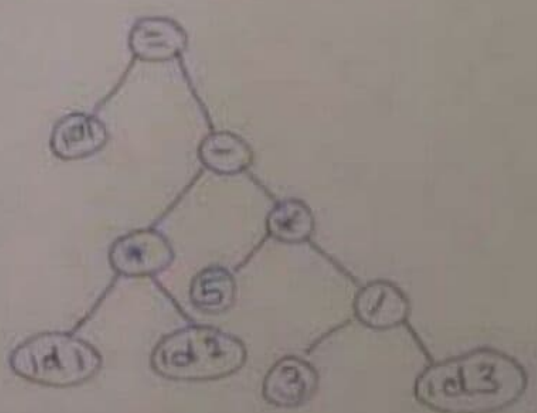
Lexical analyser : First the initialization of all variables will take place and the symbol table will be generated

	Attribute	Value
1 -	id	1 (a)
2	id	- (b)
3	id	- (c)

Now for a = a * b - 5 * 3 / c

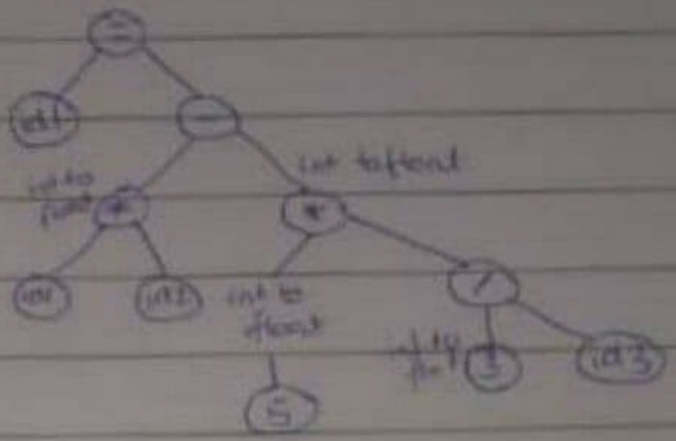
⇒ id1 = id1 * id2 - 5 * 3 / id3 ⇒ Stream of tokens

Syntax analysis : Here the parse tree would be generated.



P.T.O

Semantic analyser: Here the input is parse tree given by Syntax analyser and output is semantically correct parse tree



Intermediate code generator:

Using 3-Address code format:

$t1 = 3 / id3$

$t2 = 5 * t1$

$t3 = id1 * id2$

$t4 = t3 - t2$

$id1 = t4$

Code optimizer:

$t1 = 3 / id3$; $t2 = 5 * t1$

$t3 = id1 * id2$; $id1 = t3 - t2$

Code generator

LD R1, id3 — R1 → id3

DIV R2, #3 — R1 → id1

MUL R3, R2, 5 — R5 → id2

LD R4, id1 — R6 → t3

LD R5, id2 — R3 → t2

MUL R6, R4, R5

Q1] Explain YACC in detail.

→ A parse generator is a program that takes as input a specification of syntax and produces as output

YACC {Yet another Compiler-Compiler} is a LALR(1) (Look Ahead, Left to Right, Rightmost derivation producer with 1 look ahead token) parse generator. YACC was originally designed for being complemented by lex. It is the part of the compiler that tries to make syntatic sense of the source code, specifically ~~that tries to make syntatic sense of the~~ a LALR parser, based on an analytic grammar written in a certain notation similar to Backus-Naur Form (BNF)

The input to YACC is a grammar with snippets of C code (called "actions") attached to its rules. Its output is a shift reduce parser in C that executes the C snippet associated with each rule as soon as the rule is recognized. Typical actions involve the construction of parse tree. Using an example from Johnson, if the call node(label, left, right) constructions a binary parse tree node with the specification label and children then the rule.

expr : expr '+' expr { $$ = node('+', $1, $3); }

recognizes the summation expression and constructs nodes for them. The special identifiers $$, $1, $3 refers to items on the parser. YACC produces only a parser (phase analyser); for full syntactic analysis this requires an external lexical analyser to perform the first tokenization storage (word analysis) which is then followed by the stage proper.

Explain machine independent code optimization techniques.

code optimization in compiler design: The in the synthe phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources so that faster-running machine code will result compiler optimizing process should meet the following objectives:-

The optimization must be correct, it must not, in any way, change the meaning of program

optimization should increase the speed & performance

compilation time must be kept reasonable

process should not delay overall compiling process.

optimization process is of 2 types
machine dependent optimization
machine independent optimization
In machine independent optimization code optimizatio
phase attempts to improve the intermediate code to get
a better target code as o/p.
eg:-

```
do
{
item =10;
 value = value + item;
}
 while (value <100);
```

this code involves repeated assignment of identifier item, which if we put this way:

```
item =10;
do
{
value = value + item;
}
while (value <100);
```

should not only save CPU cycles, but can be used on any processor.
intermediate code generation process introduces many efficiencies, extra copies of variables, etc.
eg:-
copy propagation
loop unrolling
function inlining.

a)

Basis for comparison	Compiler	Interpreter
1) Input	It takes an entire program at a time	It takes a single line of code or instruction at a time
2) Output	It generates intermediate object code	It does not produce any intermediate object code
3) Working mechanism	The compilation is done before execution.	Compilation and execution take place simultaneously
4) Speed	Comparatively faster	Slower
5) Memory	Memory requirement is more due to	It requires less memory as it does not create intermediate object code
6) Errors	Display all errors after compilation, all at the same time	Display error of each line one by one.
7) Error detection	Difficult	Easier comparatively

c) Dynamic linker loader a special part of an operating system that loads external shared libraries into a running process and then binds those shared libraries dynamically to the running process. This approach is also called dynamic linking or late linking. It retrieves the address of function and variables contained in the library, execute those functions or access those variables, and unload the library from memory.

(i) It is provides the ability to load the routines only when they are needed so lot of time and memory is saved if subroutine are large with lots of external references

(ii) It helps in not loading the entire library for execution

(iii) In dynamic linking loader is used to load the main program.

Steps to acomplish the actual loading and linking of a called procedure.

1) The symbolic name of the routine in the program is used to make the load and call service request

to the operating system

(ii) The operating system checks its internal tables to determine whether or not the routine is already loaded

(iii) Control is then being passed from the operating system to the routine being called. When the called subroutine completes its processing, the operating system then returns the control of the program that issued this request.

Implementation of Dynamic linking loader.

1. Dynamic-Link library, or DLL, is Microsoft's implementation of the shared library concept in the Microsoft Windows and OS/2 operating systems. These libraries usually have the file extension DLL, OCX (for libraries containing Active X controls) or DRV (for legacy system drivers).
2. In Apple Darwin operating system, OS X and iOS operating systems the dynamically loaded shared libraries can be identified either by the filename suffix, dylib or by their placement inside the bundle for a framework.
3. In Unix-like operating systems using XCOFF, such as AIX, dynamically-loaded shared libraries

→ A grammar is said to be left recursive if it has a non-terminal, say A, such that there is a derivation $A \Rightarrow A\alpha$, for some string α. Presence of left recursion creates difficulties while designing the parser.

Left recursion maybe of 2 types

① immediate left recursion ② general left recursion

Top down parsing methods cannot handle left recursion of grammar from them by the following technique

First we group A-production as

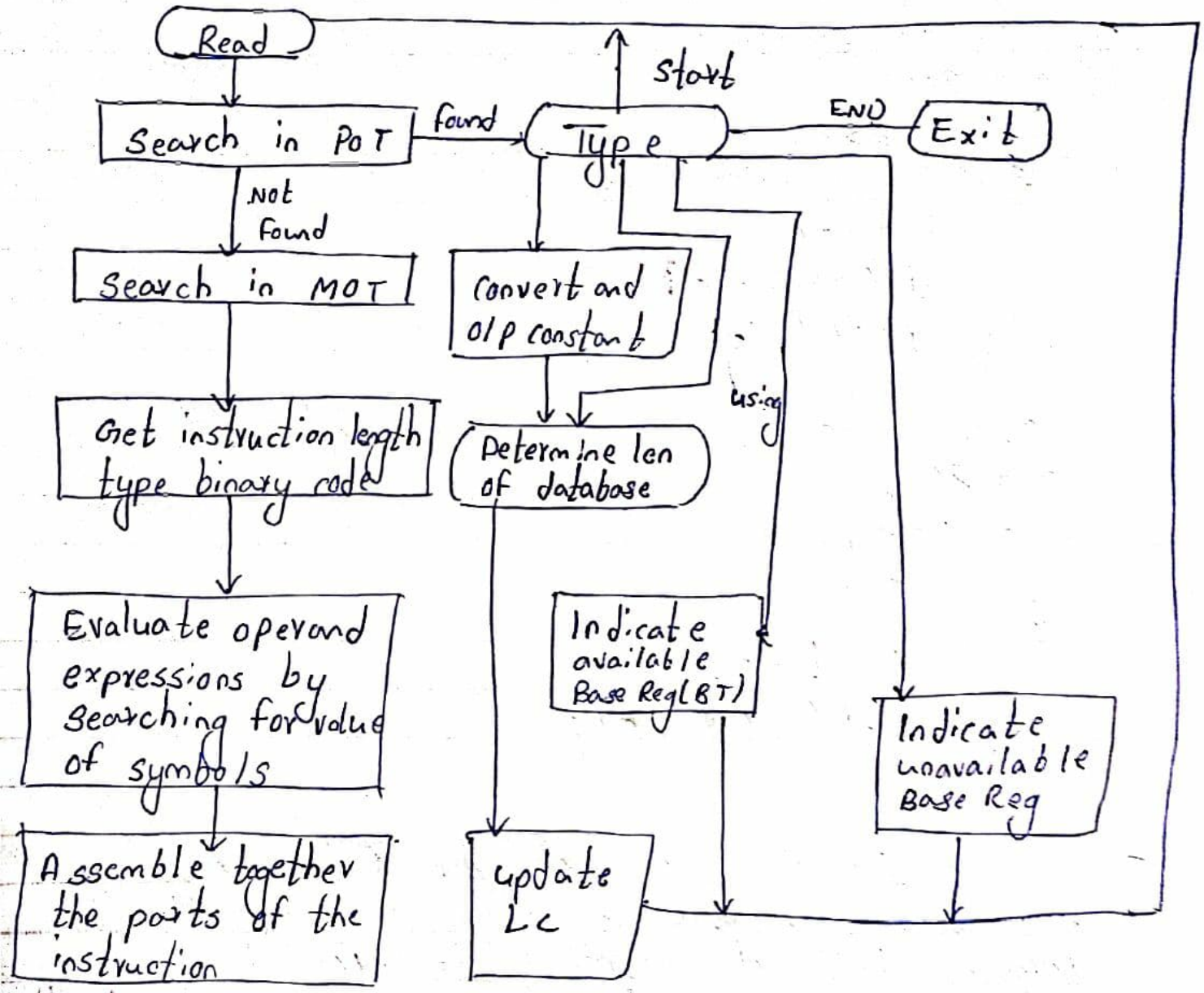
$$A \rightarrow A\alpha_1 | A\alpha_2 | \cdots | A\alpha_m | \beta_1 | \beta_2 | \cdots | \beta_m$$

where no β_i begins with an A. Then we replace the A-production by

$$A \rightarrow \beta_1 P | \beta_2 P | \beta_3 P | \cdots | \beta_m P$$

$$P \rightarrow \alpha_1 A | \alpha_2 A | \cdots | \alpha_m A | \varepsilon$$

a) Flowchart of Pass 2 assembler



Pass 2

It generate object code into the appropiate format for later processing by the loader

1) location counter is initialized

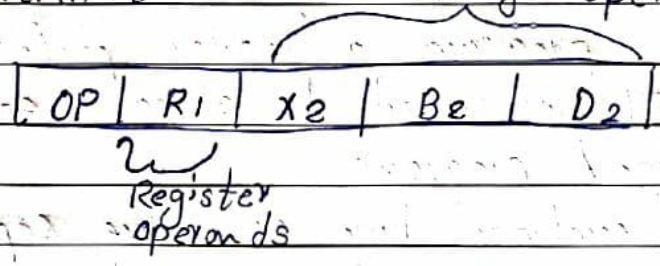
2) Statement is read from the copy file create by pass 1

3) opcode is checked to find whether it is pseudo-opcode or MOT to locate the match for source statements. if the entry found in MOT is matched then the MOT entry gives the length of the instruction

a) RR - format instruction
- It denotes register to register operation
- Each of the RR have 8-bit opcode and two 4-bit general register

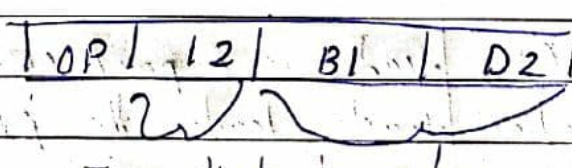
b) Rx - Register and Indexed storage operation

Rx - format

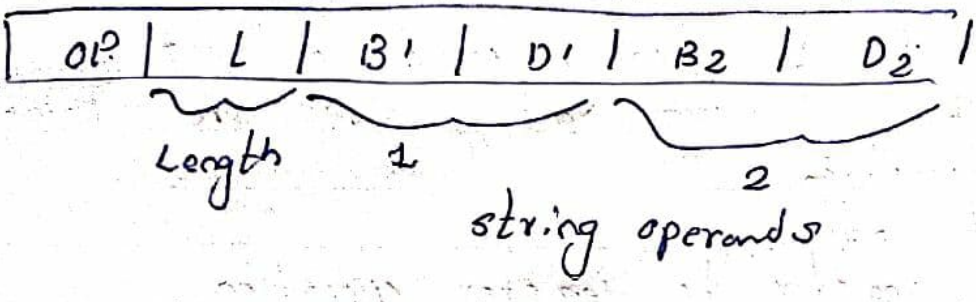


c) RS - Register and Storage operation
- This is five-byte instruction of the form OP R_1, R_3, D_2, B_2. The first byte contains the 8-bit instruction code
- The second byte contains two 4-bit field each of which encodes a register number

d) SI - storage and Immediate operation



e) SS - Storage and storage operation



3 b

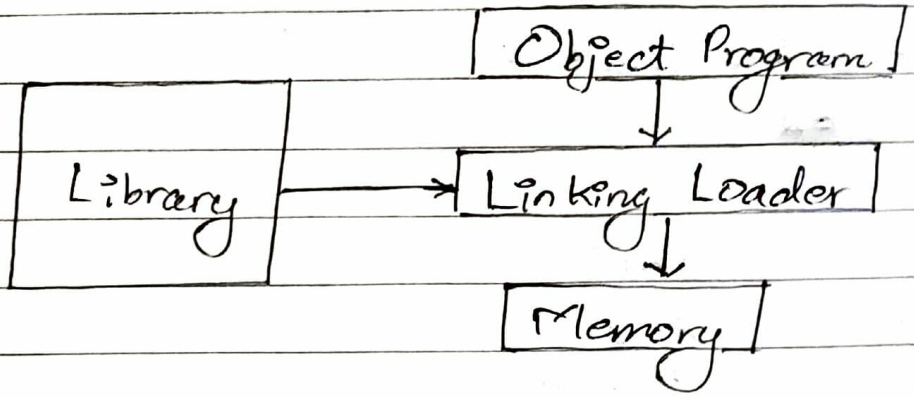
1 The fundamental task of a loader to
- Bring an object program into ~~memomor~~ memory
- Start it's execution

2 To execute a program a loader performs four function
- Allocation: It is used to allocate space in memory for the object programs
- Linking: It combines two or more separate program and resolve symbolic references between object decks
- Relocation: It modifies the object program so that it can be loaded at an different address
- Loading: Physically it places the machine instruction and data into the memory for the execution

Linker	loaders
• A linker combines file generated by compiler into a single executable file	• loader loads machine codes into system memory
• linker is a part of library files	• loader is a part of an operating sys
• linker perform the linking operation	• loader loads the program for execution
• It also links the user defined functions to the user defined libraries	• loading a program involves reading the contents of executable file into memory

Dynamic Linking Loader (DLL):

- DLL is a general re-locatable loader
- Allowing the programmer multiple procedure segments and multiple data segments and giving programmer complete freedom in referencing data or instruction contained in other segments.
- The assembler must give loader the following information with each procedure or data segments
- Dynamic linking defers much of the linking process until a program starts running. It provides a variety of benefits that are hard to get otherwise
- Dynamically linked shared libraries are easier to create than static ones
- The dynamically linked shared libraries are also easier to update than static ones
- The semantics of dynamically linked libraries can be much closer to those of unshared libraries
- Dynamic linking permits a program to load and unload routines at runtime, a facility otherwise difficult to provide



Length of segment

1. A list of all symbols in the segment that may be referenced by other segments.
2. List of all symbols not defined in the segment but referenced in the segment
3. Information where the address constant are loaded in the segment.

Format of Databases:

The assembler provides following types of record in object file as follows

1. External Symbol Dictionary (ESD)

- ESD record combine information about all the symbol that are defined in this program.
- But may be referenced in the program but defined elsewhere.

1. Text Cards (TXT)

- Text Card record control the actual object code translated version of the source program.

1. Relocation and Linkage Directory (RLD)

- The RLD records contain information about location in the program
- Whose contents depend on the address at which the program is placed.

1. END CARD:

- The END card records indicate the end of object file and specifies the start address for Execution.

Example:

```
RA
0       PG1         START
4       Entry       A, B
8       EXTRN       PG2, C
        |
20      A           DC
24      B           DC ...
        |
40      A+20
44      B-25
48      C-5
        |
52      END
```

ESD: It consists of three types of definitions

- SD - Segment Definition
- LD - Local Definition
- ER - External Reference

Symbol	Type	Id	RA	Length
PG1	SD	1	0	52
A	LD	1	20	
B	LD	1	24	
PG2	ER	2	-	
C	ER	3	-	

Relocating and Linking directory (RLD)

It includes different operations performed on it.

- Flag is important in this table

Esd id	Symbol	Flag	R.A
1	A	+	40
1	B	-	44
3	C	-	48

Note:

- Since all addresses cannot be resolved in the starting
- We required two pass Dynamic Linking Loader.
- In pass 1, it performs defining of a segment and local definition
- In pass 2, we work on with the text and RLD
  END
- Specifies the end of the program;
- And total address is generated by PLA + Slength.

conditional MACRO

Most macro processor can modify the sequence of statements generated from a macro expansion, depending on the argument supplied in the macro-expansion AIF and AGO are two conditional macro expansion pseudo code which permit conditional selection of the sequence of the machine instruction that appear in expansion of macro call.

```
Macro
Vary &count, &A1, &A2, &D3
AIF ( &Count EQ 1 ). FIN1 )
A2, &A2
AIF( &Count EQ 2 ). FIN1 )
A3, &A3
FIN1 MEND
Vary 3, d1, d2, d3
```

→ The important design issues of code generators are as follows :

① Input to code generator :-

The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front-end. Intermediate representation has the several choices :

ⓐ Postfix notation
ⓑ Syntax Tree
ⓒ Three address code.

We assume front-end produces low-level intermediate representation ie. values of names in it can directly manipulated by machine ~~input~~ instructions. The code generation phase needs complete error free intermediate code as an input requires.

⑪ Target Program :-

The target program is the output of the code generator. The output can be :

1) Assembly Language : it allows subprograms to be separately computed.
2) Relocatable Machine Language : It makes the process of code generation easier.
3) Absolute Machine Language : it can be placed in a fixed location in memory & can be executed immediately.

(III) Memory Management :—

→ during code generation process the symbol table entries have to be mapped to 'p' address and levels have to be mapped to instruction address. Mapping name ~~is~~ in the source program to address of data is co-operating done by the front-end & code-generator. Local variables are stack allocation in the activation record ~~with~~ while global variables are in stack area.

(IV) Instruction Selection :—

→ Nature of instruction set of the target machine should be complete and uniform. When you consider the effeciency of target machine then the instruction speed and machine idioms are imp. factors :

a := b + c

d := a + e

ineffecient assembly code

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

(V) Register allocation :—

Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand. The following subproblems arise when we use registers :

a) Register allocation: we select the set of variable that will reside in register.

b) Register Assignment: We pick the register that contains the variables certain machine requires even-odd pairs of registers for some operands & results.