Date: 07/08/23 Page no:

Software

System Software → All OS, linker, loader, compiler, interpreter

Application Software → All games and application

Application Software :- A set of computer programs that work together to solve a particular problem or used for a particular user defined application

☆ System Software are machine dependent

User
System Software
H/W

System programing :- It is used to define a collection of techniques used in the design of system program

each program in system software is called system program

The System Software is a collation of programs that bridges the gap b/w the level at which user wish to enteract with the computer and the level at which the copeter is capable of operating

Date: / / Page no:

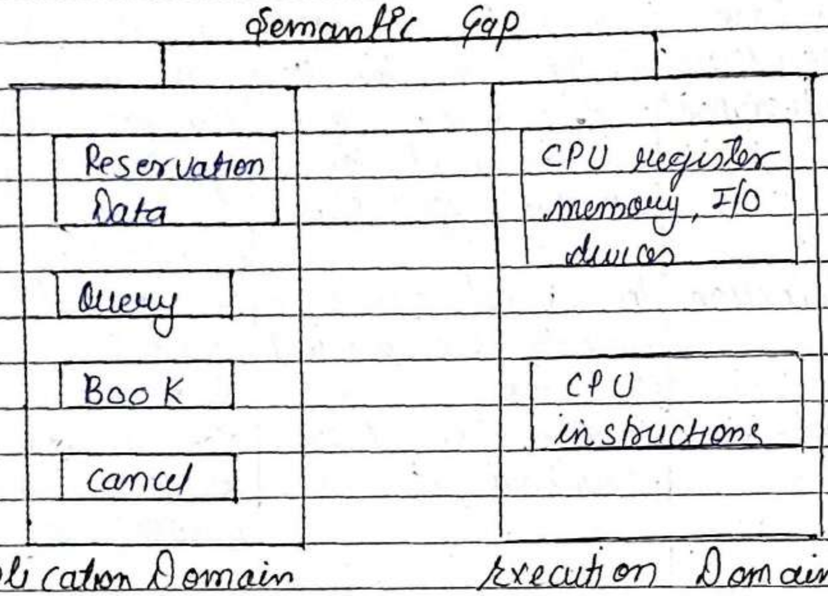
Difference b/w System Software & Appl^n software

Point of difference	System software	Application software
Definition	System s/w is designed to operate computer H/W and to provide a platform for running application software	Application software is designed to perform user specified task
Purpose	General purpose	Specific purpose
Execution environment	System s/w can run independently, provide environment for running appl^n S/w	Can't run independently, for execution of appl^n s/w System s/w need to be installed on coputers
Interaction	In general user doesn't directly interact with these s/w	In general user interacts with these S/W
Need	System cannot run without it	Not necessary to run system They are installed for specific need of users
Examples	OS, Compiler, linker etc	MS office, Tally, VS code, etc.

☆ Goals of System Software
↳ User convenience
↳ efficient use of computer resources
↳ Non interference

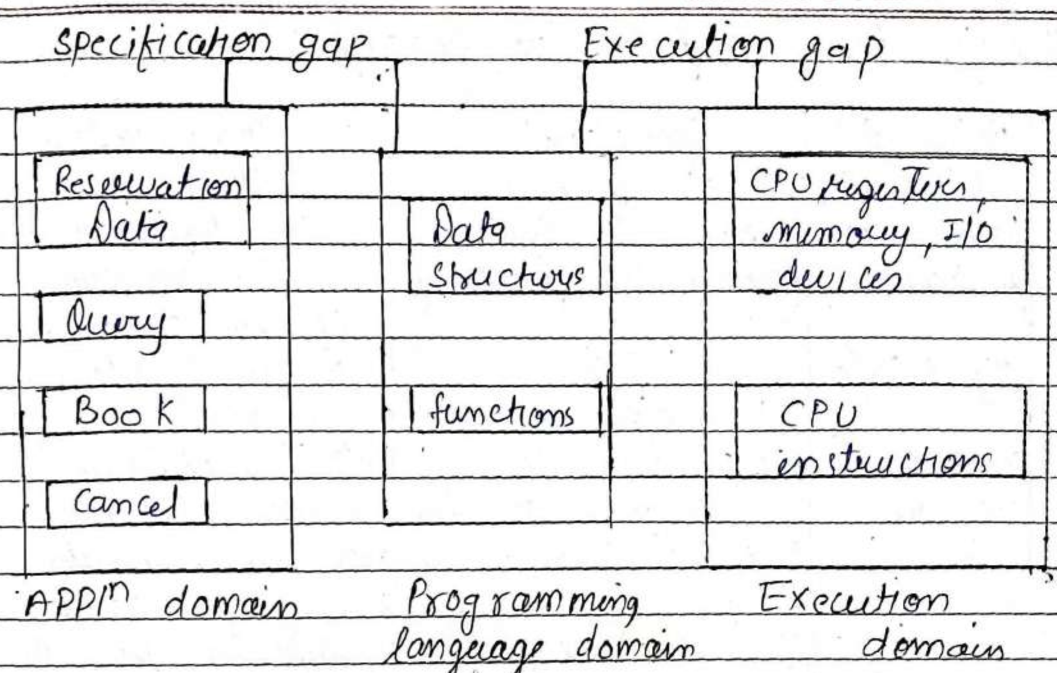
08/08/2023

Language Processor :- It is a system program that bridges the gap b/w how a user describe a computation also called specification and how a computer executes a program



⇒ To bridge this semantic Gap Language processor is used.

Date: / / Page no:



⇒ Specification Gap :- It is a semantic Gap b/w two specifications of the same task.

⇒ Execution Gap :- It is a semantic Gap b/w the semantics of the program that performs the same task that is written in different programming languages.

☆ Types of Language processor :-

↳ language translator (compiler, interpreter, assembler)
↳ detranslator ↳ preprocessor
↳ Language Migrator

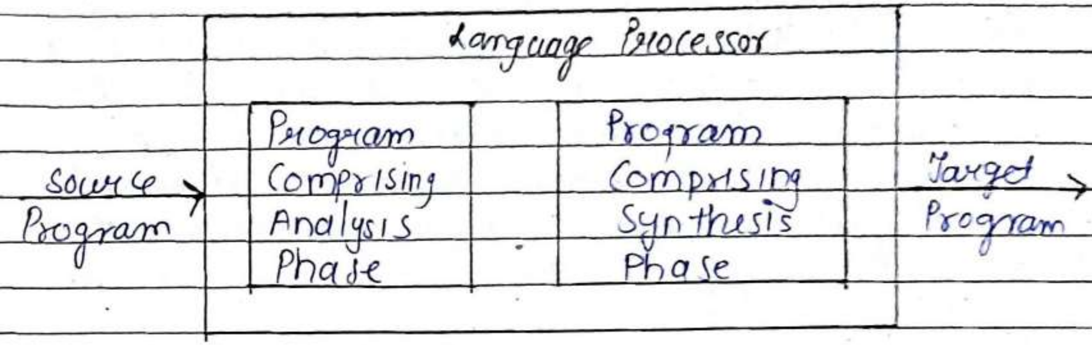
Diff. b/w Compiler & Interpreter

Point Of difference	Compiler	Interpreter
Translation process	Translates full Source code in one go	Translates and executes statements one by one
Object code	Object code is generated	No object code is generator
Memory required	If a program is successfully compiled once the object code is generated which can be executed any no. of times without the need to be translated it again on the same machine, so compiler need not be present in memory every time the program is executed	For executing a program it needs to be translated every time. So interpreter must be present in memory every time the program is executed
Time required	Comparitively less time is required	Takes more time as translation and execution occurs simultaneously
Error reporting	Errors are reported with line no. after reading whole Source Code	If an error occurs it stops immidiately at error point.

10/08/2023

* Fundamentals Of language Processor

⇒ Language processing :- Analyses of Source program, synthesis of Target



→ Analysis Phase
- → Lexical Analysis (Scanning)
- → Syntax Analysis
- → Semantic Analysis

→ Synthesis Phase
- → Memory allocation
- → Code generation

⇒ Assembly language code :- [Forward referencing]

```
MOVER  AREG  PROFIT
MULT  AREG, HUNDRED
DIV  AREG, COST_PRICE
MOVEM  AREG  PERCENT_PROFIT
PERCENT_PROFIT  DW  1
```

PROFIT DW 1
COST_PRICE DW 1
HUNDRED DC '100'

∵ DW = Data Word , DC = Data constant

MOVER = moving to Register
MOVEM = moving to memory

⇒ forward reference:- A forward reference of a program entity is a reference to the the entity in some statement of the program that occurs before the statement containing the defination or declaration of the entity.

⇒ Language processor pass:- It is a processing of every statement in a source program or in its equivalents representation to perform a language processing function

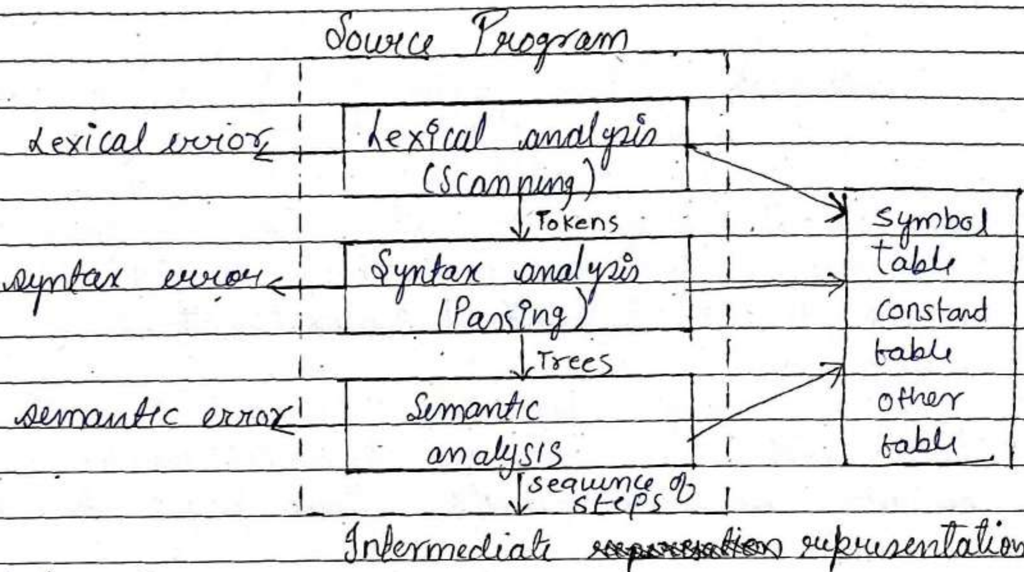
14/08/2023

Toy Compiler

↳ front end ↳ Back end

⇒ front end - The front end performs lexical, syntax and semantic analysis of the source program. Each kind of analyses involves the following functions

1. Check validity of a source statement from the viewpoint of analysis
2. Determine the 'Content' of a source statement
3. Construct a suitable representation of the source statement for use by subsequent analysis function, or by the synthesis phase of compiler



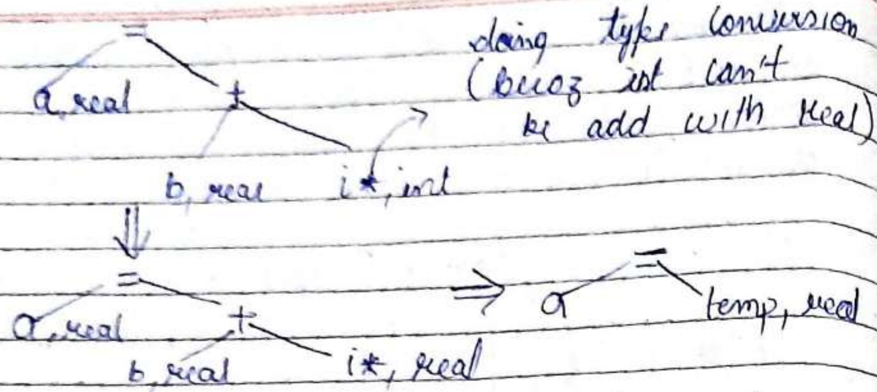
⇒ integer i;
real a, b;
a = b+i

Id #2 from Symbol table

OP#5 from operator table

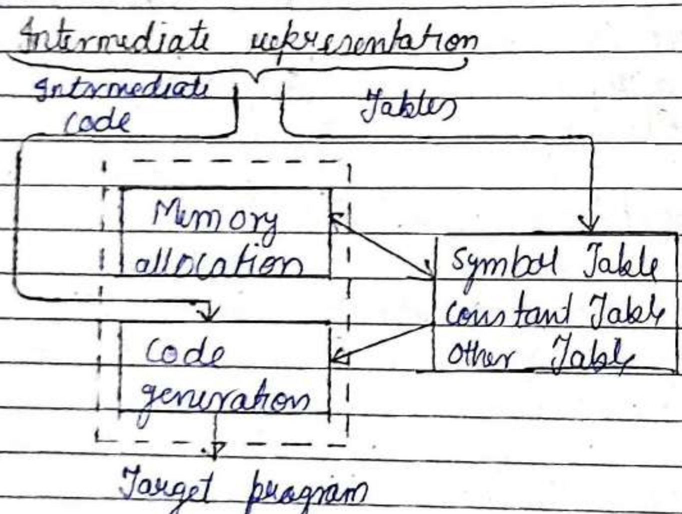
	Symbol	type	length	address
1	i	int		
2	a	real		
3	b	real		
4	i*	real		
5	temp	real		

Symbol table



⇒ Intermediat Code
↳ 1. Convert (Id #1) to real giving (Id #4)
↳ 2. add (Id #4) to (Id #3) giving (Id #5)
↳ 3. Store (Id #5) in (Id # 2)

⇒ Back end :- The back end performs memory allocation and code generation. Figure shows its schematic.



Unit II

Assembler

source code in assembly language → Assembler → target code in machine language

16/08/2023

Assembly language :-

⇒ Elements of Assembly language programing
Basic facilities
↳ Mnemonic operation code
↳ Symbolic operand ↳ Data declaration

⇒ Assembly language statement :-
general syntax -
[Label] <op code> <operand 1> [, <operand 2>...]
↳ symbolic name for memory allocation
{★ Those who are in square bracket are optional}

→ Types of statement
1. Imperative statement 2. Declaration statement
3. Assembler directive

1. Imperative statement :- An imperative statement indicates an action to be performed during the execution of the program or it is an instruction in assembly language program for ex. an arithmatic operation. example :- ADD, READ, MOV, SUB

2. Declaration Statement :-
[Label] <declaration> [<<constant>] | ex:- A DS 1
Used for reserving memory for variable | A DC 'S'

{ DS = ~~Data storage~~ Declare storage }
DC = Declare Constant

Date / / Page no.

3. Assembler directives :- Assembler directives instruct the assembler to perform certain action while assembling a program. It can't generate machine code

(i) START [<constant>] (ii) END

A simple assembly language

Length	Instruction opcode	Mnemonic opcode	Remarks
1	00	STOP	stops execution
1	01	ADD	Performs addition
1	02	SUB	Performs subtraction
1	03	MULT	Performs Multiplication
1	04	MOV	Move count of source to Destination
1	05	COMP	Compare and set condition code
1	06	BC	Branch on condition
1	07	DIV	Performs Division
1	08	READ	Reading Memory
1	09	PRINT	Writing to Memory / Print content of register

⇒ Registers

Register	Register no.
AX	01
BX	02
CX	03
DX	04

Date 17/08/23 Page no.

Assembler :-

⇒ Basic functions of assembler

↳ Translate mnemonic codes to their machine equivalent

↳ Assigns machine address to symbolic labels

↳ Converts symbolic operands to equivalent machine address

↳ Convert constants specified in source program into their machin representation

↳ Build the machine instruction in proper formate

↳ Write the object program to the memory and build the assembly listing (documentation of translation process)

⇒ Translation Process

↳ Analysis Phase ↳ Synthesis Phase

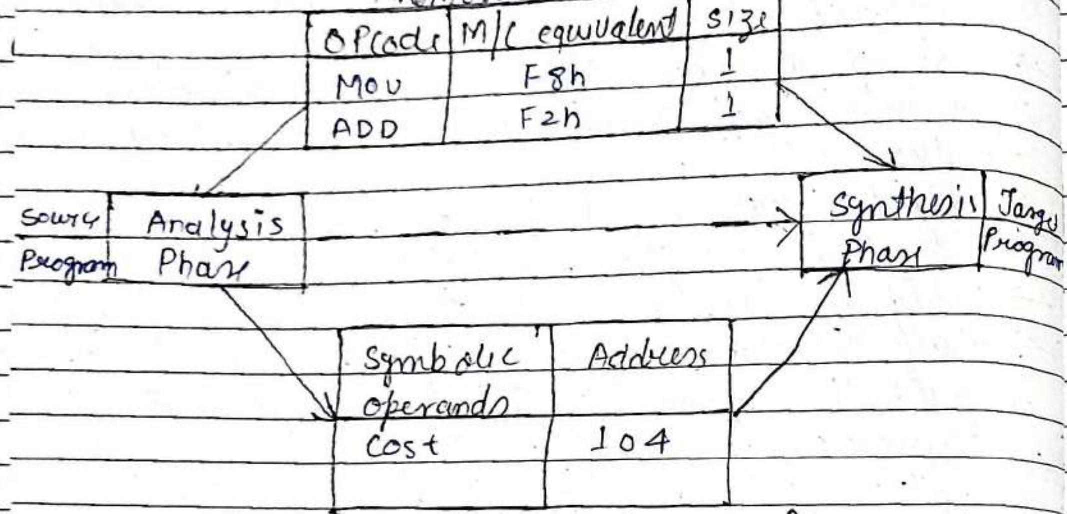
★ The tasks perform in analysis phase (Pass 1)

1. Separate contents of the label, mnemonic opcode and operands of a statements
2. Build a symbol table (SYMTAB) If a symbol is present in a label field enter the pair (Symbol, <LC>) (LC = Location Counter) in a new entry of a symbol table
3. Check validity of the mnemonic opcode through a lookup in the mnemonic table (OPTAB)
4. Perform location Counter (LC or LOCTR) processing that is update the value contained in

Date: / / Page no:

location counter by considering the opcode and operands of the statements.

Mnemonic Table (OPTAB)



Symbol Table (SYMTAB)

★ The Tasks performs in Synthesis Phase (Pass 2)

1. Obtain machine equivalent code corresponding to mnemonic opcode from mnemonic table
2. Obtain address of a memory operandans from symbol table
3. Synthesis a machine instruction or the machine form of constant as the case may be

⇒ Pass Structure of Assembler :-

↳ Two pass assembler :-

Data structures used in design of assembler:-

Date: / / Page no:

- Symbol Table (SYMTAB) • OPCode Table (OPTAB)
- Location Counter (LC or LOC):-

Location Counter is a variable that is used to hold the address of current instruction after analysis of each instruction. The instruction size is added to Location Counter

Location Counter Processing LC=0

START	100	Instruction address directive	LC=100
READ	N	100	LC=101
READ	M	101	LC=102
MOV	AX, N	102	LC=103
MOV	BX, M	103	LC=104
ADD	AX, BX	104	LC=105
MOV	R, AX	105	LC=106
PRINT	R	106	LC=106
N	WORD	107	LC=107
M	WORD	109	LC=109
R	WORD	111	LC=111
END		directive	LC=113

SYMTAB

SYM_name	Address
N	107
M	109
R	111

Progress Size = 113 − 100 = 13 byte

⇒ Pass I
Read first input line
if OPCODE = 'START' then
save #[OPERAND] as starting address
initialize LOCCTR to starting address
write line to intermediate file
read next input line
else
initialize LOCCTR to 0
while OPCODE ≠ 'END'
if this is not comment line then
if there is symbol in label field then
search SYMTAB for LABEL
if found then
set error flag (duplicate symbol)
else
Insert (LABEL, LOCCTR) INTO SYMTAB
//endif
search OPTAB for OPCODE
if found then add {instruction length} to LOCCTR
else if OPCODE = 'WORD' then
add {length of integer constant} to LOCCTR
else if OPCODE = 'BYTE' then
find length of constant in byte
add length to LOCCTR
else if OPCODE = 'RESB' then
add #[OPERAND] to LOCCTR
else if OPCODE = 'RESW' then
add 2 * #[OPERAND] to LOCCTR

else set error flag (invalid OPCODE)
write line to intermediate file
read next input line
//end while
write last line to intermediate file
save (LOCCTR - starting address) as program length
//end of pass 1

{★ RESB = Reserve Byte   RESW = Reserve Word}
★ 1 WORD = 2 BYTE
→ WORD :- Generates one word integer constant
→ BYTE :- Generates one Byte character constant
→ RESB :- It reserves the indicated no. of bytes for data area to be generated.
→ RESW :- It reserves the indicated no. of words for data area to be generated.

22/08/23

⇒ Algorithm for 2 pass assembler
Read first line from intermediate file
if OPCODE = 'START' then
write listing line
read next input line
while OPCODE ≠ END do
Search OPTAB for OPCODE
if found then store equivalent code as opcode value
if there is symbol in OPERAND field then
search SYMTAB for operand
if found then

store symbol value and operand address
else
store 0 as operand address and set error flag (undefined - Symbol)
else
store 0 as operand address
assemble object code instruction
elseif OPCODE = 'BYTE' or 'WORD' then
convert constant to object code
write listing file
read next input line
// End while
write last line to listing file
// End of Pass 2

EX:-

		START	100
100		READ	N
101		READ	M
102		mov	AX, N
103		mov	BX, M
104		ADD	AX, BX
105		Mov	R, AX
106		Print	R
107	N	WORD	2
109	M	WORD	5
111	R	RESB	2
113		END	

SYMTAB

Symbol	Address
N	107
M	109
R	111

OPTAB

OPCODE	m/c eg	size
READ	001	1
MOV	010	1
PRINT	011	1
ADD	101	1
MULT	111	1

* converting this example as defined in algo.

START
001 ~~107~~
001 109
010 01, 107
010 02, 109
101 01, 02
010 111, 01
011 111
107 — 010
109 — 101
111 —

Registers	Register NO
AX	01
BX	02
CX	03
DX	04

One Pass Assembler / Single Pass Assembler

It is also called Load and Go assembler because it doesn't save object code.
A ~~pa~~ one pass assembler makes single pass while translating the program that means all the phases of translation process are grouped as a single unit. The main problem associated with this approach is forward references in the source program

⇒ Load and Go assembler :- The assembler simply generates object code instructions as it scans the source program. Object program is not written in secondary storage therefore it doesn't require a loader to load the program in main memory

Date: / / Page no:

forward references are handled through Back Patching if it encounters an undefined symbol. the symbol address is omitted and symbol is inserted in symbol table. This entry is flaged to indicate that the symbol is undefined. The address of the operand field of that instruction is added to a list of forward references associated with the symbol table entry. When definition for symbol is encountered forward reference list for that symbol is scanned and proper address is inserted into the instructions. This process of filling the operand address back in the instructions is called Back Patching. It reports an error if there are still some entries in the symbol table indicating undefined symbol at the end of program.

23/08/23

⇒ Algorithm for 1 pass Assembler

```
Read first input line
if OPCODE = 'START' then
    Save #[OPERAND] as starting address
    initialize LOCCTR to starting address
else
    LOCCTR = 0
while OPCODE ≠ 'END' do
  if this is not a comment line then
```

Date: / / Page no:

```
// Processing of declaration statements if there
is symbol in LABEL field then search SYMTAB
for LABEL
if found then
  check if it is flagged
  if ~~flagged~~ FLAGGED then
    set symbol table address in SYMTAB as LOCCTR
    search forward references list with
    ~~if an entry~~          corresponding OPCODE
    if an entry in forward reference list is
    found then
      file operand address as corresponding symbol
          address in previously guaranted instruction
    unset the ~~flag~~ FLAG
  else
    generate error {duplicate symbol}
else
  insert (LABEL, LOCCTR) into SYMTAB
if OPCODE = 'BYTE' then
    find length of constant in bytes
    LOCCTR = LOCCTR + length of constant
else if OPCODE = 'WORD' then
    LOCCTR = LOCCTR + 2
else if OPCODE = 'RESB' than
    LOCCTR = LOCCTR # OPERAND
elseif OPCODE = 'RESW' then
    LOCCTR = LOCCTR + 2 * # [OPERAND]
else
  generate error {invalid OPCODE}
```

Date: / / Page no:

else // if there is symbol in field
// Processing of imperative statements Search
OPTAB for OPCODE
If found then generate equivalent machine code
while there are operands
If OPERAND is of register type
insert address of register
else if OPERAND is SYMBOL
search SYMTAB for OPERAND
If symbol is found, look for symbol address
If symbol address not null
store symbol address as operand address
else insert symbol and set flag
insert value of LOCCTR at end of forward
reference list
else ~~fa~~
insert value of LOCCTR at end of forward
reference list
// end of while
else
generate error {invalid OPCODE}
read next input line
// end while

ex:-	START	100			
100	READ	N	001	109	
101	READ	M	001	107	
102	MOV	AX, N	010	2CF	109
103	MOV	BX, M	010	2DF	107
104	ADD	AX, BX	011	2CF, 2DF	

Date: / / Page no:

105	MOV	R, AX	010 111	2CF
106	PRINT	R	110	111
107	M	WORD		
109	N	WORD		
111	R	RESB	2	
	END			

SYMTAB ⇒

flag	Symbol	Address
1	N	109
1	M	107
1	R	111

forward reference list

100 →	102
101 →	103
105 →	106

⇓

flag	Symbol	Address
0	N	109
0	M	107
0	R	111

24/08/23

Diff. b/w one pass and 2 Pass assembler.

S.NO.	Single pass Assembler	Two Pass Assembler
1.	It performs translation in one pass only	It performs translation in two pass
2.	Intermediate code not generated	Generation of Intermediate code
3.	forward referencing is handled by Back patching	After pass one, all symbol and literals are getting address

4	Back patching is handled by TII (Table of incomplete instruction)	NO need of Back patching
5	Default addresses are 0 for symbol and literals later on update to actual address	After pass one, all symbols and literals are getting address
6	More memory required compare to 2 pass assembler	Less memory required compare to single pass
7	Data structure used: Symbol Table, literal Table, Pool Table and TII	Data structure used: Symbol Table literal table, Pool Table
8	It is faster	It is slower

UNIT 3 :- Macro PreProcessor

Macro - #define PI 3.14

#define SQR(X) X*X

A macro is a unit of specification for program generation through expansion.

⇒ MACRO definition :-

MACRO

<macro name> [<formal Parameter specification>] — macro prototype statement — &AC (if we write this then it is called Actual Parameter)

mode statement / macro preprocessor statements

MEND

Type of statements in MACRO Definition
→ Macro Prototype
→ MODEL statement
→ Macro Preprocessor statement

▶ Macro prototype :- It declares name of the macro and names & type of its formal parameters '&' is prefixed to formal parameter names to distinguish them from actual parameter.

▶ Model Statement :- A statement from which an assembly language statement may be generated during macro expansion

▶ Macro Preprocessor statements :- It performs auxilary functions during macro expansion for example conditional statements like AIf (similar to if) to control the flow of program.

Macro to increment the value of Variable
Macro definition

```
Macro
    INCR   &MEM.VAL, &INCR_VAL, &REG
    MOV    &REG, &MEM_VAL
    ADD    &REG, &INCR_VAL
    MOV    &MEM_VAL, &REG
  MEND
```

{NOTE :- & is used to show formal parameters}

Macro call ⌒ Actual Parameters
INCR A, B, AX
⇒ After expansion
+ MOV AX, A
+ ADD AX, B
+ MOV A, AX

28/08/23

Macro Expansion
↳ Lexical substitution & ~~expansion~~.
↳ Semantic expansion

⇒ Lexical substitution :- Lexical substitution replaces occurences of formal parameter by corresponding actual parameter.

⇒ Semantic expansion:- It is based on control flow. The control flow in the definition determine the order in which model statement would be visited for expansion of macro call.
Expansion Time Control flow
★ MEC = Macro Expansion Counter.

⇒ Algorithm Macro Expansion.

1. MEC = Statement no. of ~~fost~~ first statement following prototype statement

2. while statement pointed by MEC is not MEND

statement
a) if a model statement then
 i) Expand the statement
 ii) MEC = MEC +1
b) Else (i.e a processor statement)
 (i) MEC = new value specified in statement
) Exit from Macro expansion

⇒ Altration of flow of control during expansion
i) Expansion time sequencing symbol (Label)

ii) Expansion time statements
↳ AIF ↳ AGO ↳ ANOP
(if) (Goto)

⇒ Sequencing symbol (Label)
A sequencing symbol is defined by putting it in the label field of statement in the macro body
<ordinary string>

★ AIF :- AIF (<expansion>) <sequencing symbol

★ AGO :- AGO <sequencing symbol>

29/08/2023

⇒ Macro to find greater number b/w 2 no.
Macro
COMP_GRT &A, &B, &R

```
    AIF (&A GT &B). NEXT
    MOV &R, &B
    AGO . OVER
.NEXT MOV &R, &A
.OVER MEND
```

Macro to evaluate algebraic expression A-B+C

```
~~Macro~~ MACRO
          EVAL &A &B, &C, &R
          MOV &R, &A
          SUB &R, &B
          ADD &R, &C
           MEND
```

```
MACRO
EVAL &A, &B, &C, &R
AIF (&B EQ &A). ONLYMOVEC
MOV &R, &A
SUB &R, &B
ADD &R, &C
AGO. OVER
.ONLYMOVEC MOV &R, &C
.OVER MEND
```

Expansion Time Variable (EV)

Expansion Time variables are the variables which can only be used during the expansion of macro calls.

There are two type of EV

• Local EV • Global EV

→ Local EV :- A Local EV is created for use only during a particular Macro Cal

→ Global EV:- A Global EV exist across all macro calls situated in a program and can be used in any macro which has a declaration for it

→ declaration of local EV :-

LCL <EV specification>[, <EV specification>]

eg:- LCL &A, &B

declaration of Global EV:- GBL &x

⇒ set statement :-

<EV spac> SET <SET expression>

&A SET 5

⇒ Macro that moves 8 number from first eight position of an array specified as first operand into first 8 position of an array specified as second operand.

```
MACRO
ARR_CPY &A, &B
LCL &I
&I SET 0
• AGAIN MOV &B+&I, &A+&I
&I SET &I+1
```

Date: / / Page no:

• Local EV • Global EV

↳ Local EV :- A Local EV is created for use only during a particular Macro cal

↳ Global EV:- A Global EV exist across all macro calls situated in a program and can be used in any macro which has a declaration for it

* declaration of local EV :-

LCL <EV specification>[, <EV specification>]

eg:- LCL &A, &B

* declaration of Global EV:- GBL &x

⇒ set statement :-

<EV spac> SET <SET expression>

&A SET 5

Macro that moves 8 number from first eight position of an array specified as first operand into first 8 position of an array specified as second operand.

```
MACRO
ARR_CPY &A, &B
LCL &I
&I SET 0
• AGAIN MOV &B + &I, &A + &I
&I SET &I+1
```

Not Equal

AIF (&I NE 8) .AGAIN

MEND

04/09/23

Expansion Time loops

rept statement :-

REPT <expression>

```
MACRO
CONST 10
LCL &M
&M SET 1
REPT 10
DC '&M'
&M SET &M+1
MEND
```

⇒ IRP Statement

IRP <formal Parameter> <argument list>

```
MACRO
CONSTS &M &N &Z
IRP &Z, &M, 7, &N
DC '&Z'
ENDM
MEND
```

⇒ formal parameter :- mentioned in the statement takes successive values from the argument list

⇒ Types of parameters :-

- Positional parameters
- Keyword Parameter

INCR A , B , AREG
↓ ↓
INCR 5 , 2 , AX

05/09/2023

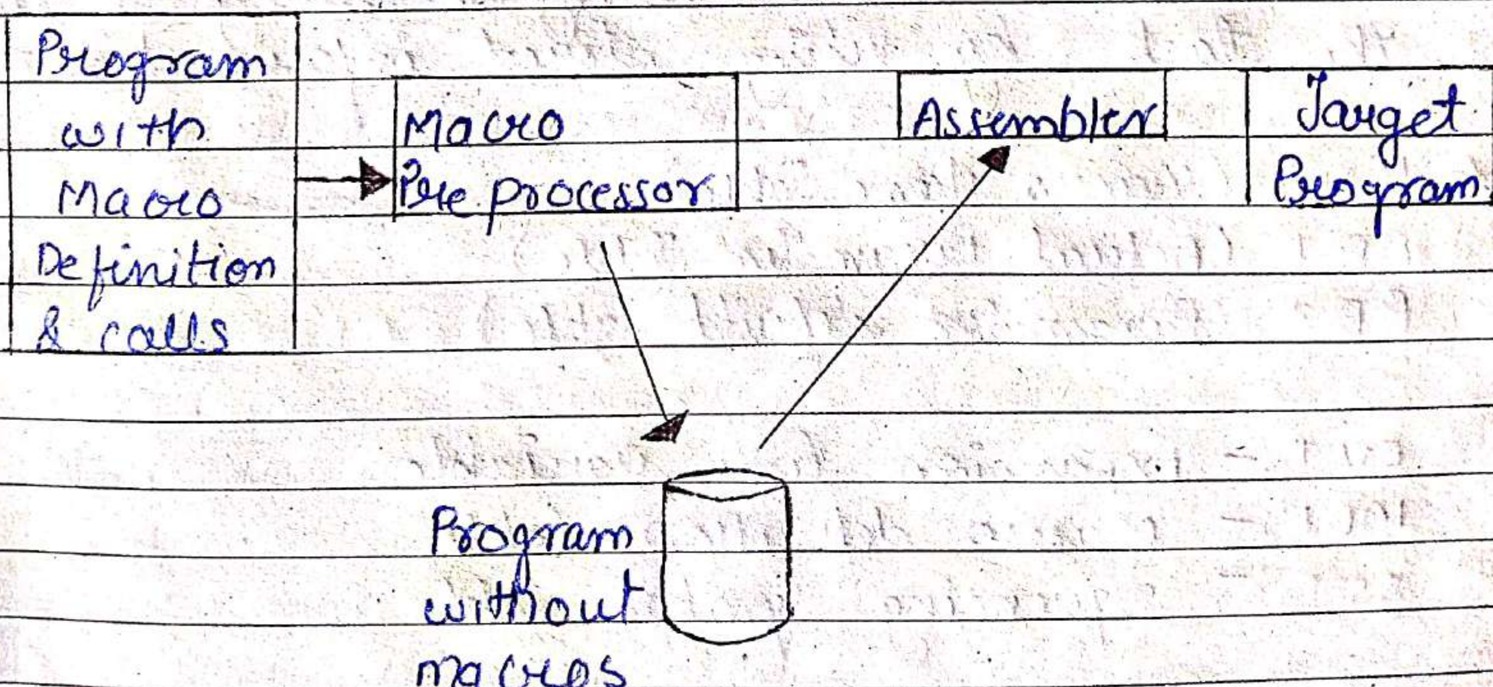
⇒ Type of parameters
→ Default value for parameter :-

⇒ Nested Macro :-

Nested Macro call

A model statement in macro may contain a call on another macro. Such calls are known as Nested macro calls. The macro containing the nested call is called Outer macro and the called macro is called Inner macro.

⇒ Design of MACRO PREPROCESSOR :-



Date: / / Page no:

⇒ Tasks in Macro Expansion :-
↳ Identify macro calls in program
↳ Determine value of formal paramter
↳ Maintain values of expansion time variables declared in program.
↳ Organise expansion time control flow
↳ Determine values of sequencing symbol
↳ Perform expansion of model statement

⇒ 4 step procedure to find design specification of each task :-
↳ Identify the information necessary to perform the Task.
↳ Design a suitable data structure to record / store the information.
↳ Determine the processing needed for obtaining & maintaing the information.
↳ Determine the processing necessary to perform the task by using stored information

★ MNT (Macro Name Table)
APT (Actual Parameter Table)
PDT (Parameter default Table)

EV.T :- Expansion time variable
MDT :- Macro definition table
SST :- sequencing symbol table

Date: 6/9/23 Page no:

Macro Preprocessor

⇒ Algorithm :-

→ Initializations

MNT _ Ptr = 0

APT _ Ptr = 0

MDT _ Ptr = 0

SST _ Ptr = 0

EVT _ Ptr = 0

Parameter = 0

EV = 0

⇒ Macro Prototyping Process :-

1 If this is Macro definition

a) Read next statement

b) MNT _ Ptr = MNT _ Ptr + 1

c) Enter Macro name in MNT

2 For each formal parameter

a) APT _ Ptr = APT _ Ptr + 1

b) Enter parameter name in APT

c) # Parameter = # Parameter + 1

3 Read next statement

4 While not a MEND statement

a) If an LCL or GBL statement then for each EV

(i) EVT _ ptr = EVT _ ptr + 1

(ii) Enter expansion time variable in EVT

(iii) #EV = # EV + 1

Read next statement
b) If label field contain sequencing symbol
If ANOP statement then
(i) SST_ptr = SST_ptr + 1
(ii) Enter sequencing symbol in SST
else search SST for seq - symbol
If found then
# MDT_entry = MDT_ptr + 1
enter statement in MDT
else
error Undefined seq_symbol

```
MACRO
ARR_CPY &A, &B
LCL &I
&I set 0
.AGAIN  ANOP
.AGAIN  MOV &B + &I ,&A+&I
        &I set &I+1
        AIF (&I NE 8) .AGAIN
MEND
```

MNT = Macro Name Table, APT = Actual Parameter Table
EV = Expansion time variable.

MNT (Macro Name Table)

1	ARR_CPY
2	

APT

formal Parameter	Actual Parameter
A	
B	

EVT

EV_name	EV_value
I	

21/09/23

⇒ Preprocessor Statement Processing

(c) If SET statement then ~~search EVT for given~~ ~~EV~~ if ~~EV~~ search EVT for given EV
if EV is found then
$MDT\_Ptr = MDT\_ptr + 1$ ~~set value in ev table~~
Enter statement in MDT
else
error {Undefined EV}
Read next statement

(d) else if AIF or AGO statement then
search SST for sequencing symbol present in statement
if Sequencing symbol is found
$MDT\_Ptr = MDT\_ptr + 1$
enter statement in MDT
else
Error {Undefined SS}
Read next statement

(e) if model statement then
$MDT\_ptr = MDT\_ptr + 1$
Enter statement in MDT

// After while MEND loop

if MEND statement
MDT_ptr = MDT_ptr+1
enter statement in MDT
// End of Macro definition processing algorithm

25/09/2023

Macro Expansion Algorithm

→ Perform initialization for the expansion of Macro
(i) MEC = first entry in MDT
(ii) Process formal parameter in actual parameter table (APT). Copy actual paramater in Actual -parameter value field

→ While statement pointed by MEC is not MEND statement
a) if a model statement then ~~by MEC is not MEND statement~~
(i) Replace operands of form (P #n) and (E #m) by value in APT [n] & EVT[m] respectively
(ii) Output generated by statement
(iii) MEC = MEC + 1

b) If SET statement with specification (F, #m) in label field then
(i) evaluate the expression in operand field and set an appropriate value in EVT [m]

(ii) MEC = MEC+1

c) If an AGO statement with (S, #S) in operand field than

(i) MEC = <MDT entry #> in SST for S

d) If an AIF statement with (S, #S) in operand field than
if condition in AIF is true then
MEC = <MDT entry #> in SST for S
else
MEC = MEC + 1

→ exit from Macro expansion

03/10/2023

```
MACRO
ARR_CPY &A, &B
LCL &I
&I SET 0
.AGAIN ANOP
.AGAIN MOV &B+&I, &A +&I
&I SET &I+1
AIF (&I NE 8).AGAIN
MEND
```

MEC (MACRO EXECU...

this get changes → $\boxed{1}$

Date: / / Page no:

MNT (MACRO NAME TABLE)

1	ARR-CPY

EVT

	Evname	Ev Value
1	I	O

APT

(ACTUAL PARAMETER TABLE)

	formal Parameter Name	Actual Parameter Value
#1	A	X
#2	B	Y

MDT (MACRO DEFINITION Table)

SST

SS-Name	#MNT-Entry
.AGAIN	2

#MDT Entry	statement
1	&I SET 0
2	.AGAIN MOV &B+&I, &A+&I
3	&I SET &I + 1
4	AIF(&I NE 8).AGAIN
5	MEND

* Initially these tables are empty

ARR-CPY X, Y

MNT-Ptr $\boxed{1}$

APT-Ptr $\boxed{2}$

MDT-Ptr $\boxed{1}$

SST-Ptr $\boxed{1}$

EVT-Ptr $\boxed{1}$

\# Parameter $\boxed{02}$

\# EV $\boxed{01}$

* Initially these pointers are 0

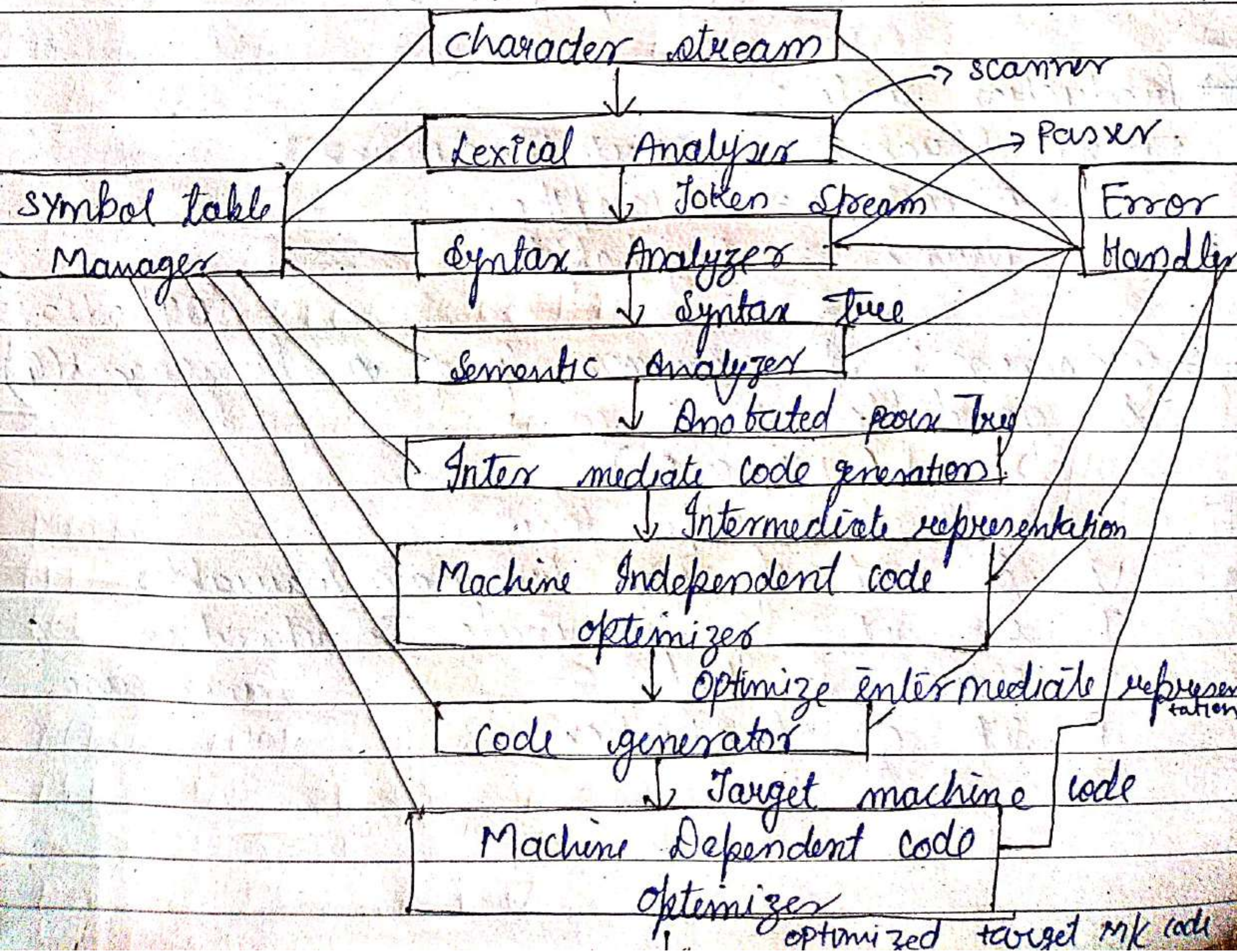
UNIT IV

Date 04/10/2023 Page no: ______

Compiler :-

Phases of Compiler :-

Lexical analysis
Syntax analysis
Semantic analysis
Intermediate code generation
Machine Independent code optimization
Code Generation
Machine dependent code optimization



Scanning & parsing

→ former language grammer :

- It is a set of lexical and syntatic rules which precisely specify the words and sentences of a language respectively.

★ Terminal Symbols / alphabets

→ $\Sigma = \{a, b, \ldots, z, 0, 1, \ldots 9\}$

★ Strings : $\Sigma \{a, b\}$

$\alpha = \{aab\}$, $\beta = \{bbaa\}$

→ Production rule :

<name phrase> := <Article> <Noun>

<Article> := a/an/the

<Noun> := boy/apple

09/10/2023

→ Grammer : A grammer G of a language L(G) is quadruple

(V, T, S, P)

→ {defined in upper case}

V is set of non terminals / variables

T is set of Terminals → {defined in lower case}

S start symbol

P set of production

$$P = \begin{Bmatrix} S \rightarrow \underline{\alpha} \, A \, \underline{\beta} \\ A \rightarrow \underline{a} \, A \, \underline{b} \, | \, \underline{\lambda} \end{Bmatrix} \qquad T = \{\alpha, \beta, a, b\}$$

$$V = \{S, A\} \qquad S = \{S\}$$

<sentence> → <Nounphrase> <Verb phrase>

<Noun Phrase> → <Article> <NOUN>

<Verb Phrase> → <verb> <nounphrase>

<Article> → a | an | the

<Noun> → boy | apple

<Verb> → ate

ex :- The boy ate an apple

<sentence> ⇒ <noun phrase> <verb phrase>
⇒ <Article> <noun> <verb phrase>
⇒ the boy <verb phrase>
⇒ the boy <verb> <noun phrase>
⇒ the boy ate <Article> <noun>
derivation ⇒ the boy ate an apple

⇒ Reduction

the boy ate an apple

↓

<Article> boy ate an apple

<Article><noun> ate an apple

<Nounphrase> ate an apple

<Noun phrase> <verb> an apple

<Noun phrase><verb><article><noun>

<Noun phrase><verb><noun phrase>

<Noun phrase> <verb phrase>

<sentence>

⇒ if the rules are given as

$$P \begin{cases} S \rightarrow a/b \\ S \rightarrow asa/bsb/\lambda \end{cases} \qquad T = \{a, b\}$$

$$V = \{S\} \qquad S = \{s\}$$

$\boxed{w = aabaa}$ → this is a string which we need to express

$s \Rightarrow asa \quad \{\because s \rightarrow asa\}$

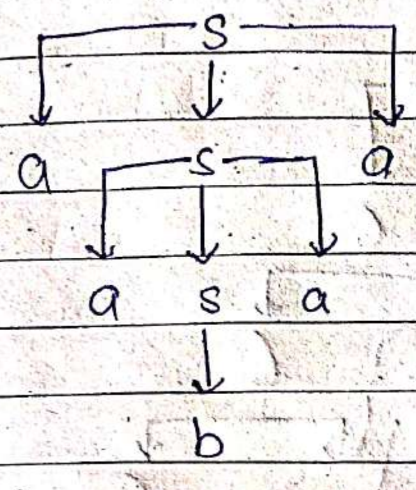
$\downarrow \quad \{\because Sa \Rightarrow asa\}$

$aasaa$

$\downarrow \quad \{s \rightarrow b\}$

$aabaa$

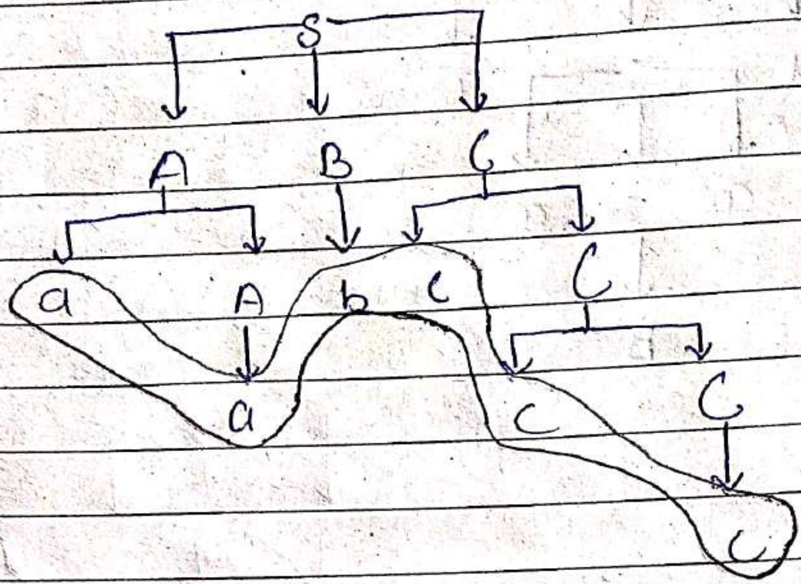
derivation tree



$S \to ABC$
$A \to aA \mid a$
$B \to bB \mid b$
$C \to cC \mid c$
$w = aabccc$

$S \Rightarrow ABC$ {∵ $A = aA$}
↓
$\Rightarrow aABC$ {∵ $A = a$}
↓
$\Rightarrow aaBC$ {∵ $B = b$}
↓
$\Rightarrow aabC$ {∵ $C = cC$}
↓
$\Rightarrow aabcC$ {∵ $C = cC$}
↓
$\Rightarrow aabccC$ {∵ $C = c$}
↓
$\Rightarrow aabccc$

⇒ Derivation trees:-



Reduction

⇒ w = aabccc

⇒ w = aAbccc

⇒ w = Abccc

⇒ w = ABccc

⇒ w = ABccC

⇒ w = ABcC

⇒ w = ABC

⇒ w = S

(1) $S \rightarrow aS \mid Sa \mid a$

$w = aaa$

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

$w_1 = id+id*id$

$w_2 = (id+id)*id$

(3) $S \to E$

$E \to T + E / T$

$T \to v * T / V$

$V \to$ <Id>

w = <id> + <id> * <id>

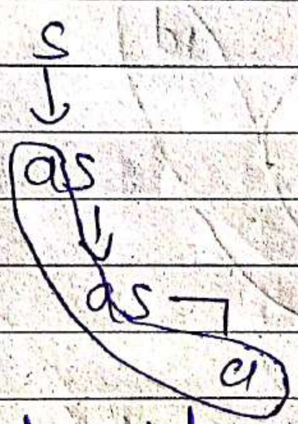
Soln 1) $w = aaa$

$S \Rightarrow as$

$\Rightarrow aas$

$\Rightarrow aaa$

Derivation tree



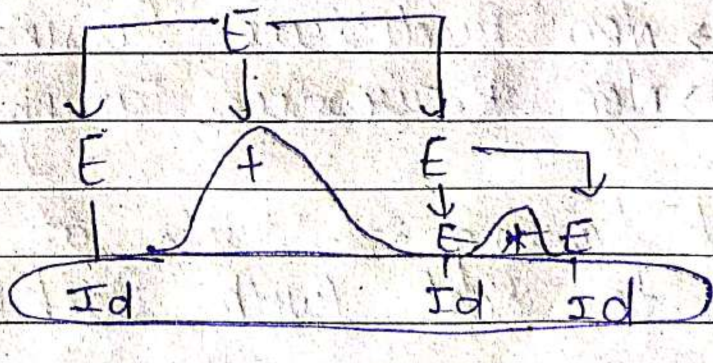
Soln 2) $w_1 = id + id * id$

$E \Rightarrow E + E$

$E \Rightarrow E + E * E$

$E \Rightarrow id + id * id$

Derivation tree



:= $w = (id + id) * id$

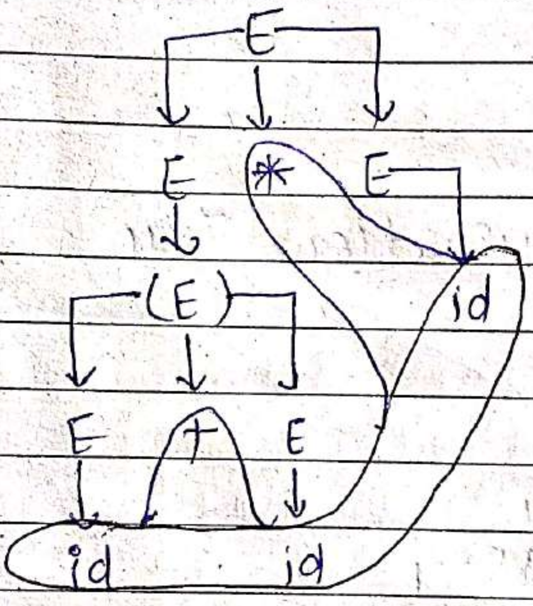
$\Rightarrow E = E * E$

$\Rightarrow E = (E) * E$

$\Rightarrow E = (E + E) * E$

$\Rightarrow E = (id + id) * id$

Derivation tree



12/10/23

Passing :- There are two types

→ Top down Passing :- Construction of pass tree starts at root and proceeds towards leaves

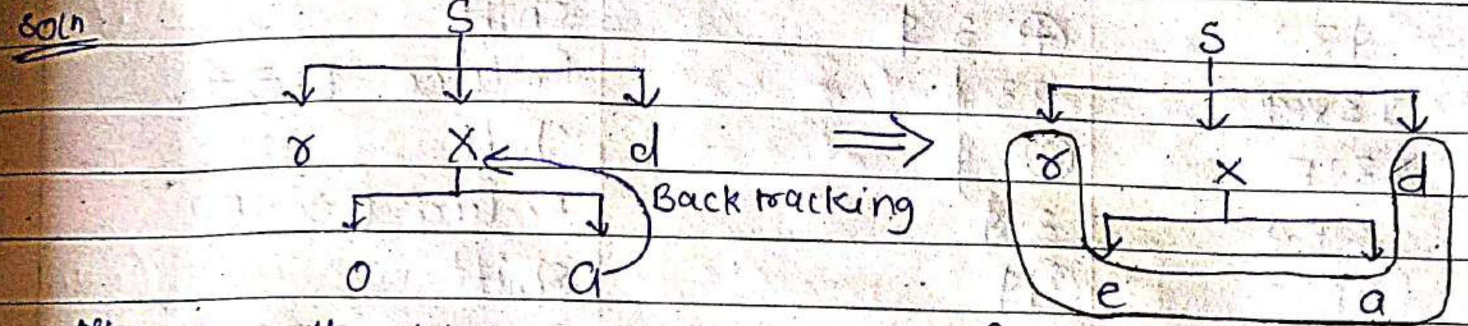
ex :-
- → Recursive Descendent Passing
- → Non Predictive Passing
- → Non Recursive Passing (LL(1) Passing)

→ Bottom up passing :- Starts at leaves and proceeds to Root (with Backtracking)

ex:- $S \rightarrow rxd / rzd$
$x \rightarrow oa / ea$
$z \rightarrow ai$

This is an example of Top down passing

soln



This method is known as Brute force Method this is also known as hit and trial method.

→ Bottom-Up Passing :-

Shift Reduce Technique
- → Shift = Push current I/p symbol to stack
- → Reduce = Symbol Replaced by Non Terminal
- → Accept
- → Error

Q1) $E \Leftarrow 2E2$, $E \leftarrow 4$
$E \leftarrow 3E3$, $w = 32423$

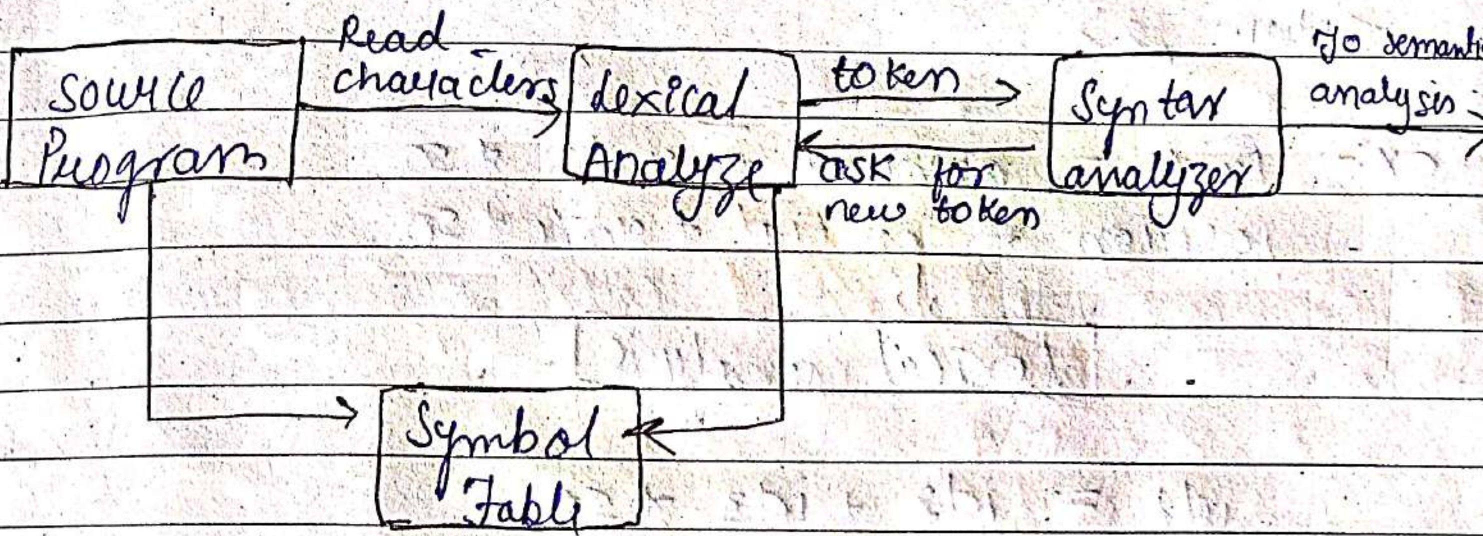
Date: / / Page no:

Stack	I/P Buffer	Action
$	32423 $	Shift
$3	2423 $	Shift
$32	423 $	Shift
$324	23 $	Reduce E → 4
$32E	23 $	Shift
$32E2	3 $	Reduce E → 2E2
$3E	3 $	Shift
$3E3	$	Reduce E → 3E3
$ E	$	Accept

Q2) S → S+S , S → S*S , S → id   ω = id+id+id

Stack	I/P Buffer	Action
$	id+ id+id $	Shift
$id	+id + id $	Reduce S → id
$S	+id + id $	Shift
$S +	id+id $	Shift
$S+id	+id $	Reduce S → id
$S+S	+id $	Reduce S → S+S
$ S	+id $	Shift
$S+	id $	Shift
$S+id	$	Reduce S → id
$ S + S	$	Reduce S+S → S+S
$ S	$	Accept

Lexical Analysis (Scanning)



⇒ Lexeme , Token (token name) , Pattern (token_value)

letter = a|b|..|z |A|B|..|Z
digit = 0|1 | 2|...|
int value = 100 ;

Lexeme	Pattern	Token
int	int	KEYWORD
value	letter(letter/digit)*	Identifier
=	=	ASSIGN_OP
100	digit (digit)*	NUM
;	;	;

ex:- E = M * C**2
<Id1 points to symbol for E >
<ASSIGN_OP>
<Id2 points to symbol for M>
<MULT_OP>

<Id3, pointer to symbol table for E>
<EXP, OP>
<Num, 2>

ex:- ~~bill = amount * rate * 50~~
position = initial + rate * 50

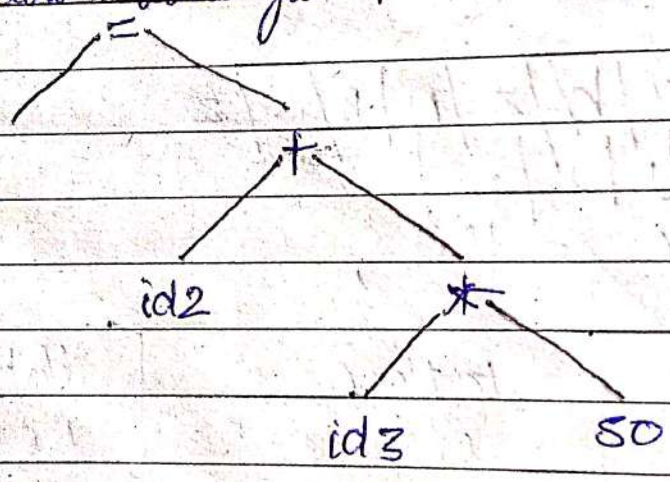
↓

$\boxed{\text{Lexical analysis}}$

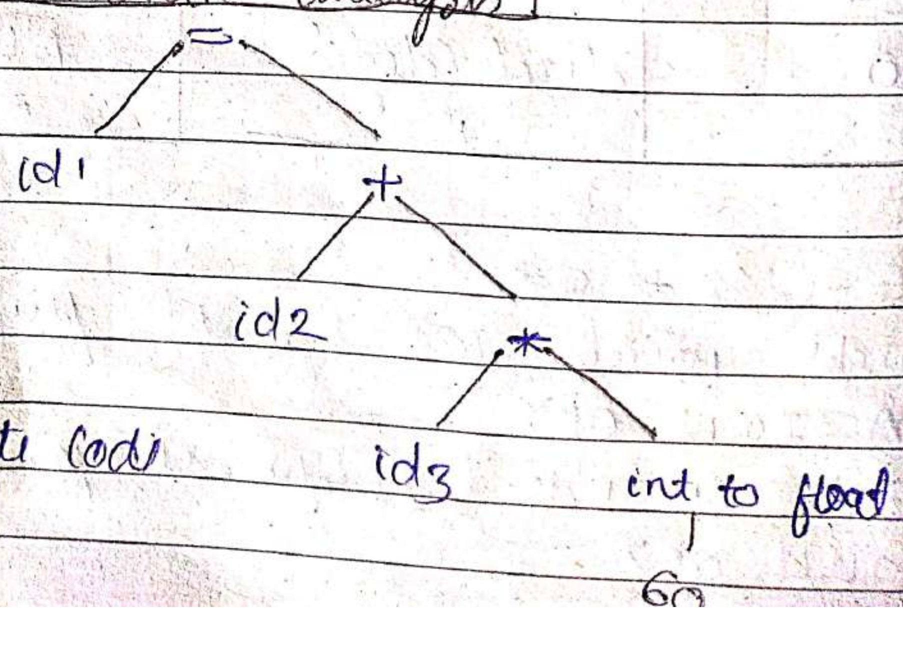
id1 = id2 + id3 * 50

↓

$\boxed{\text{Syntax analysis}}$



$\boxed{\text{Semantic analysis}}$



Intermediate code generation

⇒ There are two types of Intermediate code generation.

High level IR → Low level IR.

⇒ Three address code :- atmost three address location to calculate expression

Quadruple

OP	arg 1	arg 2	result
*	id3	60	t1
+	id2	t1	t2
=	id1		t3

18/10/2023

⇒ Intermediate Representation :-

a = b + c*d;

r1 = c*d;

r2 = b+r1;

a = r2;

Quadruple :-

OP	arg 1	arg 2	result
*	c	d	r1
+	b	r1	r2
=	r2		a

Date: / / Page no:

⇒ Triples :-

Op	arg 1	arg 2
*	c	d
+	b	(0)
=	(1)	a

Machine Independent code optimization

→ Compile time evaluation :-

$x = 12.4$

$y = x / 2.3$

$\Rightarrow y = \frac{12.4}{2.3}$

→ Variable propogation :-

$c = a * b;$

$x = a;$

$d = x * b + 4;$

$\Rightarrow d = a * b + 4; \quad \Rightarrow d = c + 4;$

→ Constant propogation :-

$x = 12.4$

$y = x / 2.3 \quad \Rightarrow y = 12.3 / 2.3$

const int n = 5
$c = n * 7 \Rightarrow c = 5 \times 7$

→ Constant folding :- Refer above example where we are doing $c = 5 \times 7$ so the answer of this will be $c = 35$ and this called constant folding. It simply means storing the answer in the constant

→ ~~Copy propagation~~ :- dead code elimination :-

C = a * b

(x = a) elimination of this code

d = a * b + 4 ;

C = a * b

d = a * b + 4 ;

here the code which is not performing any thing will be eliminated.

→ Strength reduction :-

C = a ** 2 } this exponent operator is
C = a * a } expensive so this will get
} replaced by multiplication

19/10/2023

Loop optimization :-

→ Code movement or frequency reduction :-

(i) frequency of expression ~~xx~~ is reduced

(ii) Loop invariant statement are brought out of loop

```
b = x+y;
a = 200;
while (a>0)
{ b = x+y;
  if (a%b == 0)
  a= printf("%d", a);
  a--;
}
```

Loop Jamming

```
for (int i=0; i<50; i++)   —— Loop 1
{ x = i*2; }
for(int i=0; i<50; i++)    —— Loop 2
{ y = i+3; }
for (int i=0; i<50; i++)   —— combination of both
{ x = i*2;
  y = i+3;
}
```

When we combine the two loops with same index

Loop Unrolling :-

```
for (int i=0; i<2; i++)
{ cout << "Hello";
```

instead of doing this we can directly print

```
cout << "Hello";
cout << "Hello";
```

Target Code generation :-

id1 id2 + id3

a = b + c

+	id2	id3	r1
=	r1		id1

Three address code.

target code :-

```
mov R1, id1
mov R2, id2
ADD R1 R2
mov Id1, R1
```

Machine dependent Code optimization

a = i++;

MOV AX, i

ADD AX, #1

we can replace these two statement by Incr i

Symbol Table

	Id name	data type	source line where declared	source line where performed	object sim add
int a,b,c	a	int	3	7	108
	b	int	3	7	110
a=b+c	c	int	3	7	112

Machine dependent Code optimization

a = i++;
MOV AX, i
ADD AX, #1

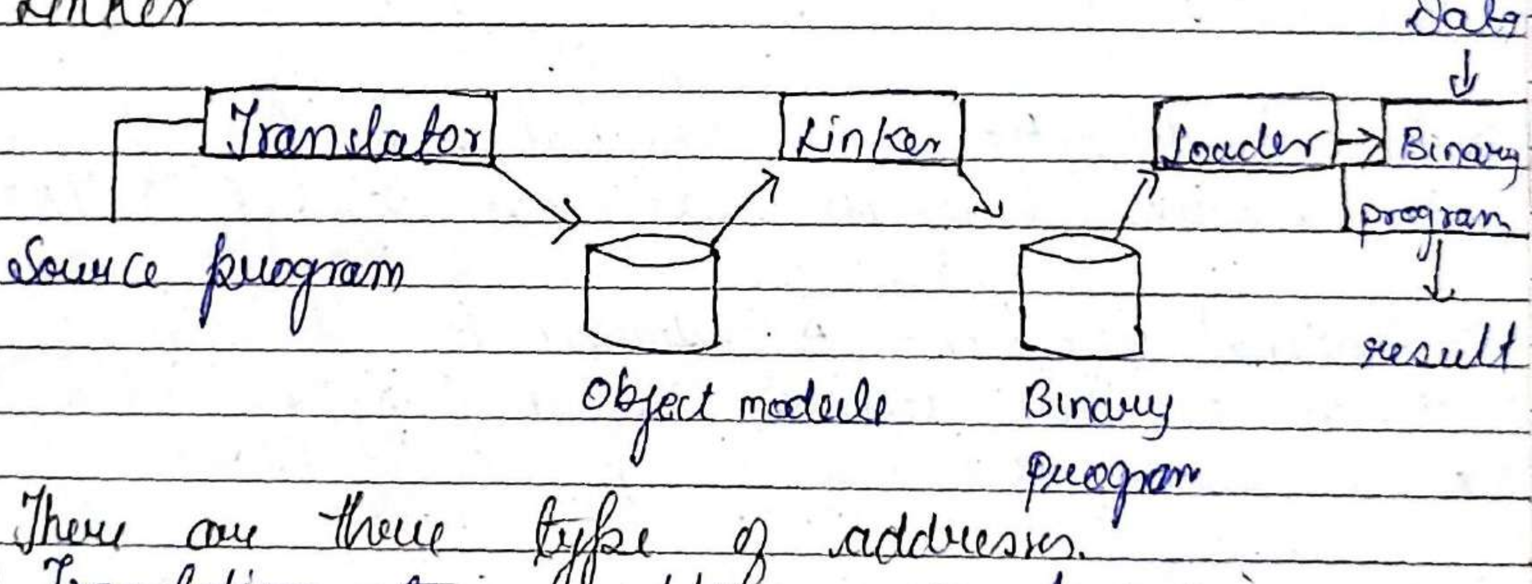
we can replace these two statement by Incr i

Symbol Table

	Id name	data type	source line where declared	Source line where performed	object tim add
int a,b,c	a	int	3	7	108
	b	int	3	7	110
a = b+c	c	int	3	7	112

26/10/2023

Linker



There are three type of addresses.
1. Translation time address or t origin
Linked address or l origin
Load time address or lo origin

Date: / / Page no:

They are two main task that linker does
-> Relocation - -> Linking

When t origin ≠ linked org then linker will do relocation
When linked origin ≠ load origin then loader will do relocation

=> Relocation :-
It is the process of modifying the addresses used in address sensitive instruction of a program such that a program can execute correctly from the designated area of memory.

=> Linking :- It is the process of binding an external reference to the correct link time address

=> Public definition :- A symbol may be referenced in another program. We use symbol 'ENTRY'

=> External reference :- A reference to symbol which is not defined in a program containing reference where it is used. We use symbol 'EXTRN'

=> Binary program :- It is a machine language program consisting a set of program units or modules such that each program unit Pi has been relocated to the memory area starting at its link origin and linking has been performed for external references.

Date: / / Page no:

SYNTAX :- linker <l origin> <object model name> [execution time address]

=> object module :-
-> Header -> Program -> Relocation Table (RELOCTAB)
-> Linking Table (LINKTAB)

EX :-
```
START 500
ENTRY TOTAL
EXTERN MAX, ALPHA
READ A
LOOP ANOP
MOV AX, ALPHA
ADD AX, MAX
AIF (&I LT #10).LOOP
STOP
A BYTE 1
TOTAL BYTE 1
END
```

-> Header :- Translated origin, size, execution start address of P

-> Program :- Machine language program

-> Relocation Table :- Translated address of address sensitive instruction.

Date: / / Page no:

→ LINKTAB :- public definition & exist

LINKTAB

Symbol	Type	Translated address
Total	PD	541
MAX	EXT	519
ALPHA	EXT	518

Translated address
500
538

⇒ Relocation factor :- R.F = lorigin − torigin
tsym, lsym, dsym

⇒ tsym = torigin + dsym —— (1)
⇒ lsym = lorigin + dsym —— (2)
⇒ R.F = lorigin + torigin —— (3)
⇒ lorigin = R.F + torigin —— (4)

lorigin = linked origin , torigin = translated origin
dsym = displacement symbol

Substitute (4) in (2)

lsym = RF + torigin + dsym

lsym = RF + tsym

ex:-

	START	100	100
100	READ N		101
101	READ M		102
102	MOV	AX, N	103
103	MOV	BX, M	104
104	ADD	AX, BX	105
105	MOV	R, AX	106
106	PRINT	R	107
107	N	WORD	109
	M	WORD	111
	R	WORD	113
	END		

Date: / / Page no:

tsym = 107
l-origin = 200
t-origin = 100
om-size = 113 − 100 = 13
RF = l-origin − torigin
Symbol = N = 200 − 100 = 100
lsym = R.F + tsym
= 100 + 107
= 207