Date: / /

Q.1 Define the following instruction.

a) STOP :- It is an impertine statement. It stop the execution of the Program op code is 00.

b) Sub :- It is an impertive statement. Its op code is 02. It substract the value with the register value.

eg :-

MOVER AREG, = '5'
SUB AREG = '1'

c) Add :- It is an impertive statement. Its op code is 01. It add the value with the register value.

d) MULT :- It is an impertive statement. Its op code is 03. It multiply the value with the register value.

Eg :-

MOVER AREG = '5'
MULT AREG = '1'

e) MOVER :- It is an impertutive statement. It's op code is 04. It is used to move the value from memory to the register.

Scanned with CamScanner

Page No. :
Date : / /

E.g. :- MOVER AREG, X

f) MOVER :- It is the impertive stutement. Its op code is 05. It is use to move data value from the register to the memory.

E.g :-

MOVEM AREG, X.

g) COMP :- It is an impertive statement. Its op code is 04. It is use to compare and set condition code.

E.g :-

comp AREG, X

h) BC :- It is an impertive stutement. Its co code is 07. It is used for the branch condition.

E.g. :-

comp AREG, X.

BC CE, AGAIN.

Here, it will check whether, the value of x and value of AREG should be less than or equal to. if condition is successful then the control will shift to the AGAIN pabel.

Scanned with CamScanner

Page No.:
Date: / /

(i) READ :- It is an imperative statement. Its op code is 09. It is used to read a value.

Eg:- READ x.

Here value of x will be read.

j) PRINT :- It is an imperative statement. Its op code is 10. It is used to print the contents of register.

Eg:- PRINT x.

k) ORIGIN :- This is assembler directive statement. The this directive instructs the assembler to set the address given by address specification in the location counter.

Eg:- ORIGIN LOOP+R.

Here, through this statement the value of the location counter will be the location value of loop label plus 2. i.e. if loop is at 2m then the location counter value will be 2014.

l) EQU :- It is Assembler directive statement. In EQU left side is label/symbol and right side is address.

Scanned with CamScanner

Eg:- <label> EQU <Address>

Eg:- X EQU Y

Here, x is set to the address of y.

m) PURGE :- It is an assembler directive statement. It is used undefined the symbol names which are define by the ORIGIN statement.

Eg:- ABC EQU xyz : ABC represent xyz

PURGE ABC : ABC No longer xyz.

n) ASSUME :- It is an assembler directive statement. It tells the assembler that it can assume the address of the indicated segment to be Present in registers.

Syntax :- ASSUME <register> <segment Name>

Eg. ASSUME DS : SAMPLEDATA.

Here, the ASSUME directive tells the assembler that the start address of SAMPLEDATA can be assumed to be in the DS register.

Scanned with CamScanner

Date: / /

a) SEGMENT :- It is the memory used to store three components of a program i.e. program code, data & stack.
So, the code, stack & Data segment register are used to condition the start address about of these three components. The extra segment register points to another memory area which can be used to store data.

b) PROC :- It is indicate that the it is a procedure.
e.g :- CALCULATE PROC.
Here CALCULATE is a procedure.

c) NEAR :- It indicate that the whether the call to the procedure it to be assembled as near call.
i.e. the procedure is called from the same segment.
e.g. : CALCULATE PROC FAR.

d) PUBLIC :- When a symbol is define as a public in a program that means it can be referred by the another program units also.

Scanned with CamScanner

t) EXTRN :- when an assembly module wishing to use a symbol declear in another assembly module is. The symbol should be declear as extrn.

syntax :-

EXTRN <symbolic name : <type>

v) OFFSET :- It is a displacement from the base address of segment.

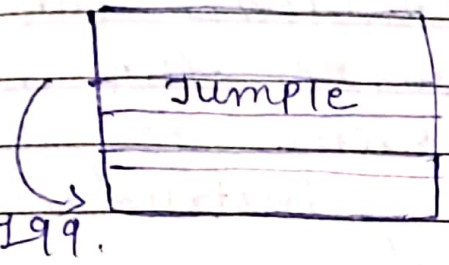
Q.2 Explain the meaning of

b) Base register :-

- The base register is used for Program Relocation.
- It holds the address of data in RAM to be excuted recently.

E.g.:- let say we have memory and in that we have store a Program P1 from 100 to 199

Scanned with CamScanner



P1 diagram 1

So according to diagram 1 their is an instruction Jmp 160. C using absulute address. so now control will be an address 160. Here, all works good.

But now this program is send to secondary memory for some times then another program i.e P2 will occupy address 100 of 199.

Then again Program P1 comes back to main memory then new address is allocated.

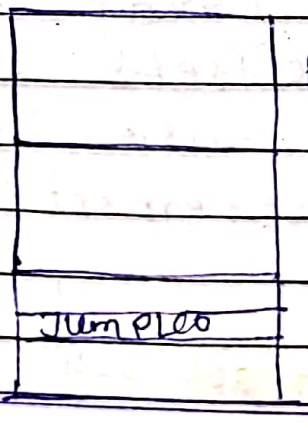


diagram 2

Page No. :
Date : / /

So, now the Problem will occur as we have change the memory address of program P1 but it is having an instruction Jump 160 so it create an illegal instruction. so the program P1 memory is from 1100 to 1199 so to solve this problem we have to add the displacement insteed of giving absolute value.

So the effective address
EA = Base register value + displacement.

So their will be no Problem if address is change.

c) Index register :-

The index register have the source index of array and destination Index i.e. offset. They are Provided with auto increment and auto-decrement facility.

E.g :-

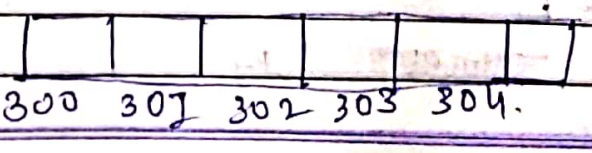


Diagram 1

Page No.:
Date: / /

Here, in diagram I the source index is 300. If we went to make at 303

Then,

Effective address = source index offset

= 300 + 3

= 303

This way we can move to 303 address.

d) Symbol table :-

→ An identifier used in the source program is called symbol. Thus names of variables function and producer are symbols.

→ Language Processor user the symbol table to maintain the information about the symbol used in source program.

→ Symbol table performs following things.

i) Add the symbol

ii) Add the location of symbol.

iii) Add the length of symbol

iv) Delete the symbol entry.

Scanned with CamScanner

Page No. :
Date : / /

(V) Access the symbol entry.
e.g.:-

SYMTAB

Symbol	Address	length
LOOP	202	4
NEXT	214	1

(E) literal table :- The use of literal table is to collect all literals used in a program. At any stage the current literal encountering an LTORG statement literals in the current POOL are allocated address starting with the current value in LC.

E.g. littab

literal	address
='=5'	211
'=1'	212

F) POOL table :-

This table contains the literal number of the starting literal of each literal POOL.

e.g.:- POOLTAB

literal no
#1
#3

Scanned with CamScanner

Page No. :
Date : / /

g) optab :-

optab contains the fields mnemonic code, opcode and size of the operator. It also contain the class field which tells us that the statement is imperative Assembler directive or declaration statement.

If an imperative statement the mnemonic info field contains the pure machine opcode, instruction (length) else it contain the id of a routine to handle the declaration or direction statement.

h) Declaration statment :-

- The syntax of declaration statment is follows :-

[label] DS <constants>
[label] DC <values>

- The DS statement reserves area of memory and assigntable names with them.
- The DS statement constructs a memory word containing constants.

e.g.:- A DS 200
ONE DS '1'

Date : / /

(i) Imperative statement :-

An imperative statement indicates an action to be performed during the execution of the assembled program each imperative statement typically translate into one machine instruction.

Eg. :-

MOVER AREA, X

(j) Assembler Directives :-

Assembler direction instruct the assembler the perform certain action during the assembly program.

Eg :-

START :n

This directive indicate that first word of machine should be placed in that memory word with address <constant>.

START <constant>

START 500.

Scanned with CamScanner

Page No.:
Date: / /

Here,
first word of the target program is stored from memory location 500 onwards.

Q.3 Diffrent between the following.

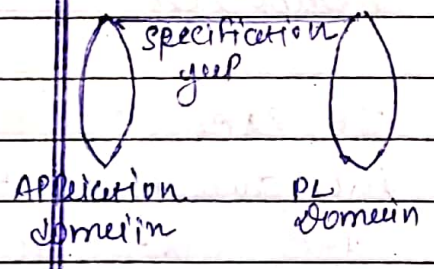
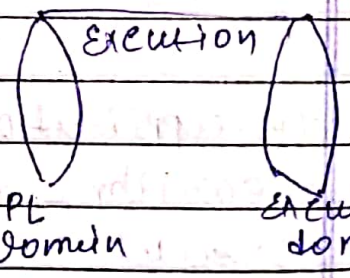
	Application Domain	Execution Domain
1)	In this domain designer expression their ideas.	In this the ideas express by designer are implemented
2)	The code of any program is written	on basis of code written output is generated.
3)	In application domain algorithm and syntax is checked	In Execution domain the run time problems are checked.
4)	In application domain algorithm and work of analysis thase is done	In execution domain work of the synthesis thase is done

Scanned with CamScanner

Page No. :
Date : / /

• E.g:- Printf ("Hello world"); Here we write the command and its syntex is checked	• E.g:- Hello world! By Processing the comm output is generated by excutation domain

b) Specification Gap and Excution gap.

• Specification Gap	Excution Gap.
• The gap between the application domain and the Programming domain is called specification gap.	The gap between the programming domain and execution domain is called Execution Gap.
	
• Specification gap is the semantic gap between task specification and same task.	Execution Gap is the gap between the semantic of program coriten in different Programming language

Scanned with CamScanner

Date: / /

4	The specifition gap is briged by the sofitwear development team	Excution gap is bridged by designer of Programming language Processor, translator or interpreter
5	e.g:- suppose a person who dont know programming language can use the application but no able to do Programming.	E.g:- Programmer unit Program in high level language and computer does not understand the high level language directly.

C) Source Program and target Program.

	Source Programs	Target Program.
1	A Program written in a programing language is called source Program	Target program is an output generated after compiling the source Program

Scanned with CamScanner

Page No. :
Date : / /

	Source Program is written is Programing language by the Programmer	Target Program is compiled and gurem by the compier.
5)	Source Program contains english words according Programming language	Target Program only contains iPbury code.
6)	Programmer can read the source Program.	Machine can read the target program
7)	Source Program is an input to compiler	Target Program is the out Put to compier.
	Compiler	Interpretor
8.	Compiler converts the entire Program	Interpreter translate code ensline at time.
9)	It Produce optimized code.	Code is not too much optimized.
10)	Analyses time is near.	Program analysis time is less.

Scanned with CamScanner

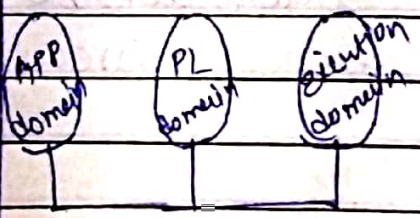
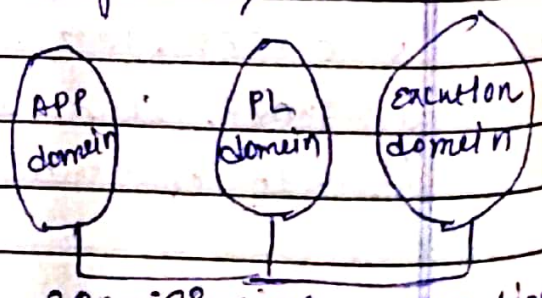
Page No. :
Date : / /

machine code is generated	No machine code is generated.
There is an execution gap	there is no execution gap.
language translator	Preprocessor
Language translate bridge an execution gap to the machine language of the computer system	It is a language processer which bridges an execution gap but is not a language translater
Assembler is an language translater whose source code is assembler language and convert it to machine lang...	Processor process highlevel language before language translator takes over.

Scanned with CamScanner

Date: / /

Q) Problem oriented and Procedure oriented.

Problem oriented	Procedure oriented
Problem oriented language are specific oriented language.	Procedure oriented language are common oriented language
Problem oriented language have large execution gap.	Procedure oriented language have large specification gap.
Execution gap is bridged by translator or interpreter.	specification gap is bridge by an application developer.
Eg:- KOBOL, FOTRAN	Eg:- C, C++
	
specification gap / Execution gap	specification gap / Execution gap.

Scanned with CamScanner

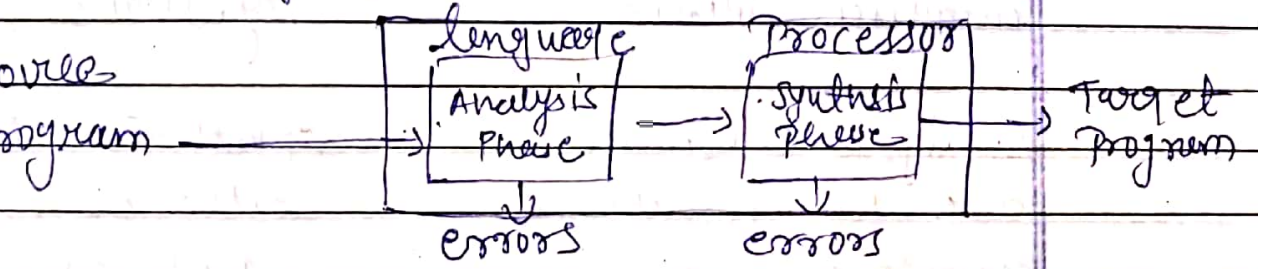
Page No. :
Date : / /

g) Lexical, syntax & semantic rules

Lexical :- It governs lexical rules in the source language. (eg identifier name is proper or not)

Syntax :- It checks validity of statement for syntax in the source language.

Symantic Analysis :- It check the semantic rules in the source language.



9 Analysis and syntasis Phase

Analysis Phase	syntasis phase
Analysis Phase uses each components of source Program to	syntasis Phase target language statement which have same

Scanned with CamScanner

determine relevant information out of it	meaning as source language statement
Analysis phase consists of lexical syntax and semantic analysis	synthesis phase work 2. Activities (i) creating data structure in target program. (ii) generation of target code.
Intermediate Representation (IR) it the output of analysis phase	Intermediate Representation (IR) is the input for synthesis phase
Analysis phase is also known as Pass I	synthesis phase is also action as Pass II.

Q.4 Explain the following

b) Difference between single pass assembler and two pass assembler.

Scanned with CamScanner

Single Pass assembler	Two Pass assembler
Scans entire source file only once.	Require two passes to scan source file. First pass :- responsible for label definition and introduce them in syntax table. Second pass :- translates the instruction into assembly language or generate machine code
Generally :- i) Deals with syntax. ii) Constructs symbol table. iii) Creates label list iv) Identifies the code segment, data segment stack segment etc.	Along with pass i pass two is also required which - i) Generates correct opcode. ii) Compute actual address of every label iii) Assign code address for storing the information iv) Translate operand

Scanned with CamScanner

Date : / /

	register memory. v) Immediate value is translator to binary string.
cannot resolve Forward reference of class symbol.	can resolve Forward references of data symbol.
No object Program is written hence no loader is required	loader is required as object code is genated.
Tends to be faster compared to two Pass	Two pass assembler requires recanning. Hence slow compared to one pass assembler
only create table with all symbols as address of symbol is calculated.	Address of symbols can be calculated
more memory required compare two pass assembler.	less memory required compare to single pass assembler.

Scanned with CamScanner

Intermediate code not generated	Generation of intermediate code.

- Explain the significance of location center.

- To implement memory allocation a data structure called location center (LC) is introduced.

- The location counter is always made to contain the address of the next memory word in the target program.

- It is initialized to the constant specified in the START statement.

- Whenever the analysis phase sees a lable in a assembly statement it enters the lebel and contents of LC in new entry of the symbol table. Then the value to the location counter is increment and it goes to the next statement.

- It ensure that LC points to the next memory word in the target program when a machine instruction have different length.

Scanned with CamScanner

Date: / /

4) To update the contents of LC analysis phase needs to know length of different instruction.

E.g. in Symbol table in

Symbol	address
AUTAIN	104
N	113

Here address is inserted by the value stored in the location counter.

e) Differentiate between two variants for generation of intermediate code in two pass assemblers.

	Variant I	Variant 2
4)	IS, DL, & AD all statement contain processed form.	DL & AD statements contain processed form while for IS statement, operand field is processed only to identify literal reference.
5)	Extra work in pass I	
6)	simplifies tasks in Pass 2	

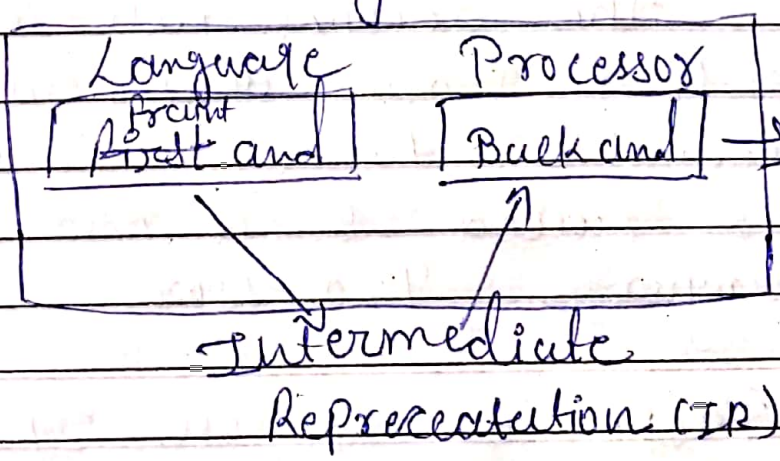
Scanned with CamScanner

Page No.:
Date: / /

Q.4) Explain the following in

(a) What is intermediate Representation in an Assembler?

→ An Intermediate Representation (IR) is a representation of a source Program which reflects the effects of source, but not all, analysis and synthesis tasks Performed during language processing.



Two Pass Language Processor

- The first pass performs analysis of the source program and reflects its result in the intermediate representation.

→ The second pass reads and analysis the IR, instead of source

Program, to perform synthesis of the target program.

- This avoids repeated processing of the source program.

- The first pass is concerned exclusively with source language issues, so it is called front-end of the language processor.

- The second pass is concerned with program synthesis for a specific target language so it is called back-end of the language processor.

- Properties of an IR :-

Ease of use :- IR should be easy to construct and analysis.

Processing efficiency :- efficient algorithms must exist for constructing and analising the IR

Scanned with CamScanner

Memory efficiency in IF must be compust.

(vi) Give the instruction format of 8088 microprocessor.

→ The intel 8088 microprocessor supports 8 and 16 bit arithmetic and also provides special instruction for string manipulation.

→ 8088 microprocessor supports three different instruction formats

[a] Register/Memory to register

[opcode d w | mod Reg r/m]

(b) Immediate to Register/memory

→ Lopcode d w | mod op r/m | data | d

(C) Immediate to Accumulator

| opcode w | data | data :

Date: / /

- The mod and r/m fields specify the first operand which can be in a register or in memory, while the reg field describe the second operand which is always a register.

- The instruction opcode indicates which instruction format is applicable.

- The direction field (d) in the instruction indicates which operand is the destination operand in the instruction.
  if d = 0, The register/memory operand is the destination.

- The width field (w) indicates whether 8 or 16 bit arithmetic is to be used.

Assembly statement	opcod	d	w	mod	reg	r/m	d
ADD AL, BL	000000	0	0	11	011	000	0
ADD AL, 12H[SI]	000000	1	0	01	000	100	0
ADD AX, 3456H	100000	0	1	11	000	000	
ADD AX, 3456H	000001	0	1	01	100	110	

Scanned with CamScanner

Page No. :
Date : / /

instruction format of microprocessor

Assembly statement	opcode	d	w	mod	reg	r/m	data/displacement
ADD. AL, BL	000000	0	0	11	011	000	00010010
ADD. AL, 12H(ST)	000000	1	0	01	000	100	01010110
ADD AX, 3456H	100000	0	1	11	000	000	00110100
ADD AX, 3456H	000001	0	1	01	010	110	00110100

Fig. 4.22 simple instruction of 8086

Adequate, The Third instruction 16 bits of immediate data note that the low byte of immediate data comes first followed by its high byte. The fourth assembly statement is identical to the third. however it has been encoded using the immediate.

Scanned with CamScanner

(C) Immediate to accumulator

r/m	opcode w mod=00	disp mod=01	disp mod=10	mod=11 w=0	mod=11 w=1
000	(BX)+(SI)	(BX)+(SI)+dx	Note 2	AL	AX
001	(BX)+(DI)	[BX]+(DI]+dx	Note 2	CL	CX
010	(BP)+(SI)	[BP]+([SI)+dx	Note 2	DL	DX
011	(BP)+(DI)	(BP]+(DI]+dx	Note 2	BL	BX
100	(SI)	(SI)+dx	Note 2	AH	SP
101	(DI)	(DI)+dx	Note 2	CH	BP
110	Note 1	(BP)+dx	Note 2	DH	SI
110	(BX]	(BX)+dx	Note 2	BH	DI

Note 1:- (BP)+ DISP for indirect addressing
d16 for direct.

Note 2 :- same as Previous column, except d16 instead of dx

Register

reg	8 bit (W=0)	16-Bit (W=1)
000	AL	Ax
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI

Page No. :
Date : / /

III BH DI

Instruction formats of Intel 8088.

Explain different registers of 8088 architecture.

The intel 8088 microprocessor supports 8 and 16 bit arithmatic and also provides special instruction for string manipulation.

Different register of 8086 architecture
- Data registers AX, BX, CX, and DX.
- Index registers SI and DI.
- Stack pointer registers BP and SP
- segment register code, stack, data and Extra.

Data register :-

Each data register is 16 bits in size split into the upper and lower halves either half can be used 8 bit arithmatic, while the two halves together constitute the data register for 16 bit arithmatic

Index registers :-

The index registers SI also DI are used to index the source and destination addresses in string manipulation instruction. They are provided with the auto-increment and auto-decrement facility.

Stack Pointer registers :-

Two stack Pointer registers called SP and BP are provided to address the stack.

SP Points into the stack implicity also by the arithmetic unit to store subroutine and interrupt return addresses.

BP can be used by the Programmer in any desired memory.

Segment registration

The code, stack and data segment registers are used to contain the start addresses of code, data, and stack compartment

Page No. :
Date : / /

- The extra segment register points to another memory area which can be used to store data.

(i) What is the meaning of segment overiding? Explain with example.

- For arithmatic and mov instruction the architecture uses the data segment by defult.

To overids this, an instruction can be preceded by a 1-byte segment overids prefix with the following format :-

001	seg	110

Fix value

where seg, reprented in 2 bits, has the meaning

seg	segment register
00	ES
01	CS
10	SS
11	DS

Scanned with CamScanner

Page No. :
Date : / /

Example = If the code segment is to be used instead of data segment, it can be rewritten as.

ADD AL, CS : 12H [SI]

The assembler would encode two as. this segment overids instruction.

001 01 110 0000 10 01 000 100 00100

f) Explain the significence of segment register.

y segmentation is the Process in which the cencein Memory of the computer is logally divided into different segments. and each segment has its own base address.

y) Need for segmentation :-

(BIU) contains four 16 bits special purpose register called the segment registers.

Scanned with CamScanner

Page No. :
Date : / /

4) Types :-

(i) code segment (CS) :- is used for addressing memory location in the code segment of the memory where the executable program is store.

(ii) Extra segment (ES) :- Also refers to a segment in the memory which is another data segment in the memory.

(iii) Data segment (DS) :- Points to the data segment in the memory where the data is stored.

(iv) stack segment (SS) :- It is used for addressing stack segment of the memory. The stack segment is that segment of memory which is used to store stack data.

Scanned with CamScanner

Assignment

Q.2) Define translated link and load time addresses.

→ While compiling program P a translator is given an origin specification for P. This is called translated origin of P.

→ The Translator uses the value of the transuted origin to perform memory allocation for the symbols declared in P. This result in the assignment of a translation time address to which each symbol in the program

→ Translated Address in The start address pecified by the translator is the translated start address of the Program.

→ Their are a reason for which the linker or loader change the address of Program.

Scanned with CamScanner

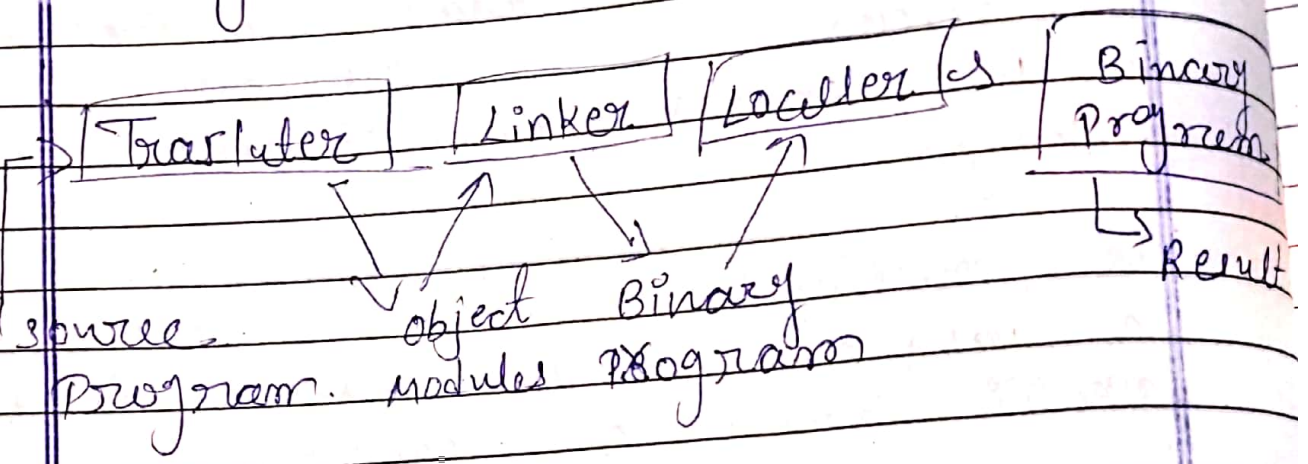
(1) The translated addresses may have been used in diffrent object modules constituting a Program. so memory allocation to such programs would conflict unless their origins are changed.

(2) operating system may required that a program should execute from a specific area of memory.

- linked address :- Address assigned by the linker.
- Local time Address -> Address assigned by the loader.
- Traslated origin :- Address of the assigned by the translater. This is the address specified by the programmer in an ORIGIN statement
- Linked orgin :- Address of the origin assigned by the linker while producing a binary program.

Scanned with CamScanner

4) Load origin :- Address of the origin assigned by the loader while loading the program for execution



PROGRAM EXCUTION

Q.3 Define

a) Program Relocation.

4) Address sensitive instruction :- An instruction which uses an address i.e AA.

Address constant :- a cluter word which contains an address i.e AA.

4) Program relocation is the process of modifying the address word

Page No. :
Date : / /

The address sensitive instruction of a Program such that the Program can excute correctly from the regi degnation area the of memory.

- If linked origin ≠ translated origin relocation must be performed by the linker.

- If load origin ≠ linked origin relocati must be performed by the loader.

- In general a linker always perform relocation.

- If the load origin = linked origin a loader are called absolute loaders

- loaders that origini = linked perform relocation i.e. relocating loaders.

b) Address sensitive Instruction or IPR.

- It's a set of instruction or IPR th perform relocation in Program P.

Scanned with CamScanner

Date :

4) steps :-

1) Calculate Relocation Factor (RF)

4) Add it to translation time address (a) for every instruction which is member of IIR

E.g :- Relocation Factor = 900 - 500 = 400
IIR contains translation address 540 & 558 (instruction : read 17)

Address is changed to 540 + 400 = 940 &
538 + 400 = 838

C) Relocation Factor :-

1) Let the translated & linked origins of program P be t-origin and l origin P respectively.

4) Consider a symbol symb in P (let its traslation time address be t symb and links time address symb.
The relution factor is :-

Scanned with CamScanner

relocation factor P = longing - t-origin

The relocation can be Pritive negative or zero.

d) Public definiction & extand References

i) Public definitions :- The ENTRY statement lists the Public definitions of a Program unit i.e it lists thoes symbole defined in the Program unit which may be referenced in other Program units.

ii) External Refernce :- The EXTRN statement lists the symbol to which External refernes are mute in the program unit i.e. the symbol use from the other program unit.

Q.4) Diffrent b/w non relocatable relocatable, self relotable module.

1. Non Relocatable module :- A Program is one that cannot be executed in any memory area other then the area starting on its translated origin.

Scanned with CamScanner

E.g:- A hand coded machine language Program.

4 Relocatable Module :- A Program is one that can be processed to relocate it to a desired area of memory.

E.g:- An object Module if can be relocated to the new memory address.

5 The difference b/w a relocatable and a non relocatable Program is the avelability of information concerning the address consitive instruction.

6 self relocatable Module in A Program is the one can perform the relocation of its own address sensitive instruction.

A self relocatable module can be executed in any area of memory.

E.g:- This is very important in time sharing oprating system where load

Scanned with CamScanner

address of a Program is likely to be different, for different executions.

Q1) Explain data structure used in nested macro calls.

→ Two Provision are required to implement the expension of nested macro calls :-

I) each macro under expension must have its own set of data structure i.e

⇒ MEC :- The flow of control during macro expersion is implemented using a macro expension counter (MEC).

→ APTAB :- This table contain the list actual parameter from the macro

→ EVTAB :- This table contain the expansi time variable of the macro.

→ APTAB ptr :- It points to the index value of APTAB and it is incremental when new parameter is encountered.

Scanned with CamScanner

Date : / /

4) EVTAB ptr :- It Points to the index value of EVTAB and it is incremented when new parameter is encountered.

5) extended stack model. A stack is used to accommodate the expansion time data structure.

- The stack consist of expansion records, each expression record, accommodating one set of expansion time data structure.
- The expression record of the top of the stack corresponds to the marco call currently being expanding to the main call. when nested marcro call is recognized a new expansion record is pushed on the stack to hold the data structure for the call.
- At MEND an expansion record is popped off the stack.

2) An expansion nesting counter (Nest cntr) is maintained to count the number of nested macro call.

- Nest cntr is incremented when a macro call is recognized and decremented

Page No. :
Date : / /

when a MEND statement is encounted Thus nest-enter > 1 indicates that a nested macro call is under expesion. while nest-entr = 0 implies that macro expansion is not in progress currently

Scanned with CamScanner

Q.6

Explain Basic object Module format

4 The object module of a program contains all information necessary to relocate and link the program with other program

4) The object module of a program P consists of 4 components

1.) HEADER :- The Header contains translated origin, size and execution start address of P.

2.) Program :- This component contains the machine language program corresponding to P.

3.) Relocation table :- (RELOCTAB). This table describe IRRP. Each Relotab Entry contains a single Field.

Translated address :- Translated address of an address sensitive instruction.

4.) Linking table (LINKTAB) :- This table

Scanned with CamScanner

Date : / /

information concering the public difinitions and external refrence in P

Each LINKTAB Entry contains there filds :n

symbol :- symbolic name.
TYPE :- PD / EXT indicating wherther Public definition or external refrecen
Translated address :- For a public definitution this is the address of the first memory word allocated to the symbol For an external refrence its is the address of the memory word which is required to contain the address of the symbol.

	Eg :n statement		Address	Code
	START	5000		
	Entry	Total		
	Extrn	MUL, PIPHU		
	RECD	A	5000	+04 0 5440
Loop			5001	
	:			
	MOVER	AREG, PIPHU	5100	+04 1 000

Scanned with CamScanner

	BC	Any, MUX	519)	to6	6	000
	:					
	BC	LT, LOOP	538)	to6	1	501
	stop		539)	too	0	000
A	DS	1	540)			
total	DS	1	541)			
	END					

⟹ object module of above program:-

1) translated origin = 500, size = 42, execution, start address = 500

2) machine language instruction

3) Relocation table

500
538

4) Linking table

Alpha	Ext	518
Mux	Ext	519
A	PD	540

Scanned with CamScanner

Date: / /

Q7 MSDOS LINKER.

Q Explain MSDOS Linker.

Ans Object Module Format (explain object module of the Program)

- An Intel 8086 object module is a sequence of object records, each object record describing specific aspects of the Program in the object module.
- There are 14 types of object records containing the following five basic categories of information.

- Basic Binary image (i.e. code generated)
- External references
- Public definitions
- Debugging information
- Miscellaneous information

- We only consider the object records corresponding to first three categories
- a total of eight object record types

- Each object record contains variable length information and may refer to the contents of previous object records
- Each name in an object record

Scanned with CamScanner

Date : / /

is represented in the following format

length (1 byte) name

THEADR, LNAMES and SEDEF records

THEDR Record

80H length T-Module name checksum

- The module name in the THEADR record is typically derived by the translator from the source file name.

- This name is used by the linker to report errors.
- An assembly Programmer can specify the module name the NAME directive

→ LNAMES record:

96H length name-list checksum

SEGDEF record.

98H length attributes segment name index
(1-4) length (2) index (1)

Scanned with CamScanner

- A SEGDEF record designates a segment name using an index into this list.
- The attributes filed of SEGDEF records indicates whether the segment is relocatable or absolute, whether (and in what manner) it can be combined with other segment as also the aligment requirment of its base address (e.g. byte, word or paragraph, i.e. 16 byte, aligment)
- Such segments with the same name are concatenated with each other, while common segment with the same name are overlapped one other.
- The attribute filed also contains the origin specification for an absolute segment.

→ EXTDEF and PUBDEF record:

8CH	length	External reference list		check-sum
90H	length	base (2-4)	name	offset - check-sum

- The EXTDEF record contains a list of External used by the Programs of this module.

Scanned with CamScanner

Date : / /

- A EXTDEF record designates an external symbol name by using an index into this list.
- A PUBDEF record contains a list of public defined in a segment of the this list.
- The base specification identifies the segment.
- Each (name, offset) pair in the record define one public name, specifying the name of the symbol and it's offset within the segment designated by the base specification.

→ LEDATA Records

ADH	length	segment index (1-2)	data offset (2)	data	check-sum

- An LEDATA record contains the binary image of the code generated by the language translator.
- segment index identifies the segment to which the code generated by the language translator

Scanned with CamScanner

Page No. :
Date : / /

→ FIXUPP record

gCH	length	locat (1)	fix dat (1)	frame datum (1)	target datum (1)	target offset (2)	...
						check sum	

- A FIXUPP record contains the binary information for the one or more relocation and linking fixup to be performed.
- The locat field contains a numeric code called loc code to indicate the type of a fixup.
- The meaning of these codes are given in table.

4) MODEND records :-

8AH	length	type (1)	start addr (5)	check sum

4) The MODEND record signifies the end of the modules with the types field indicating whether it is the main program.

Scanned with CamScanner

Date : / /

Eg

Sr no	statement		offset
001	NAME	DEMO	
002	TEST	SEGMENT	
003		EXTRN X, Y, Z	
004		public A, B, C	
007	A	---	0010
!		!	
012		MOV AX, SEGT, X	0025
!		!	
022		MOV AX, offset Y	0050
!		!	
030		MOV AX, SEGT, Z	0075
!			
035	B	---	0080
!			
040	C	---	0085
!			
045		RDB 25	0149
046	TEST	ENDS	
		END	

object Modules DEMO :

type	length	other Fields	checksum
80H	---	04 DEMO	--- THEADR
96H	---	04 TEST	--- LNAME
98H	---	60H 120 01	--- SEGDEF

Scanned with CamScanner

Page No. :
Date : / /

Type	Length	other Fields	checksum
90H	---	01 01 A 0010	--- PUBDEF
90H	---	01 01 B 0080	--- PUBDEF
90H	---	01 01 C 0085	--- PUBDEF
90H	---	01 X 01 Y 012	--- EXTDEF
8CH	---	01 0025 01 0000	-- LEDATA
ADH	---	8801 06 01 01	--- FIXUPP
9CH	---	01 0050 A1 00 00	--- LEDATA
ADH	---	8401 06 01 02	--- FIXUPP
9CH	---	01 007F A1 0000	-- LEDATA
ADH	---	8801 06 01 03	LEDATA
90H	---	80H	LEDATA
80H	---	00H	

7) FIXUPP codes and fix update field codes

4) The locat field contains a numbers code could for code for to indicate the type of a fixup.

Loc code	Meaning
0	Low order byte fix
1	offset fix
2	segment ~~fix~~ fix
3	Pointer fix

Scanned with CamScanner

FIXUPP codes

- Local also contains the offset of the FIXUP location in the previous LEDATA record.

- The Fix data field indicates the manner in which the target datum and target offset fields are to be interpreted.

Code	Context of target datum & offset fields
0	segment index and displacement
2	External index and target displacement
4	segment index (offset field not used)
6	External index (offset not used)

(6) Take a assembly language program and Explain the execution of - MSDOS linker.

- Linker performs relocation and linking of all named object modules to produce a binary program with the specified load origin.

Scanned with CamScanner

Page No. :
Date : / /

- Linker Excution terminates when all external reference have been resolved or when the unresolved external refrences cannot be found in any of the library files.

- Linker uses a two-pass strategy.

- In the first pass the object modules are processed to collect information concerning segment and public definition.

- The second pass perform relocation and linking.

- First pass :- In the first pass LINKER only processes the object records for building NTAB.

second pass :- In the second pass LINKER builds NAMETAB, SEGDEF, and EXTDEF.

Scanned with CamScanner

Sr no	statement	offset
001	NAME first	
002	Test 1 SEGMENT	
003	EXTRN, A; BYTE, Y: WORD	
004	PUBLIC A	
007	A ---	015
012	MOV AX, SEG X	028
022	MOV AX, OFFSET Y	056
035	TEST 1 ENDS	
036	END.	

001	Name second	
002	TEST 2 ~~END~~ SEGMENT	
003	EXTRN Z: BYTE	
004	PUBLIC X	
010	X ---	033
015	MOV AX, SEG Z	175
069	TEST 2 ENDS	
	END	

001	NAME THIRD.	
002	TEST 3 SEGMENT	
003	EXTRN A: WORD	
004	PUBLIC Y, Z,	
015	Y ---	043
033	Z ---	047

Scanned with CamScanner

Page No. :
Date : / /

037 MOV AX, SEG A
059 TEST 3 ENDS
END.

274

a) after Processing FIRST

NTAB

symbol	load address
TEST I	10000
A	10015

NAMFLIST

TEST I.

SEGTAB

segment	load address
TEST I	1000

EXTTAB

External symbol	load address
X	10153
Y	10219

b) after Processing SECOND

NTAB

symbol	load address
TEST 2	10120
X	10153

Scanned with CamScanner

SEGTAB	segment TEST 2	Load address 10120

EXTTAB	External Z	load address 10266

c) after processing THIRD

NTAB	symbol TEST 3 Y Z	load address 10176 10219 10266

NAMELIST	TEST 3

SEGDEF	segment name TEST 3	load address 10176

EXTAB	External symbol A	load address 10015

Scanned with CamScanner

Page No. :
Date : / /

SS Assignment :

1) Diffrent between screening and parsing.

* Scanning :

- Scanning is the process of recognizing the lexical components in a source string.

- The lexical features of a language can be specified using type-3 regular grammars. This facilitate automatic construction of efficient regcognizers for the lexical features of the language

- Finite state automatic (FSA) :- A finite state automation is a triple (S, F, T), where.
  S - is a finite set of state one of which is the initial state S0 and one or more of which are to final state.

  Σ - is the alphabet of source symbol

Scanned with CamScanner

T - is a finit set of state
ractione out of each $s_i \in S$,
on, encountering the symbol
of t.

- Deterministic finit state Automation (DFA) = It is an FSA such that $t_1 \in T$, T_1 $(s_i, symb, s_j)$ implies $A, t_1 \in T, t_1 = (s_i, symb, s_k)$. in other words It is a finit state automatic name of whose states has two on more transitions for the same source symbol The DFA has the Property that it reaches a unique state for every source string input to it.

- Regules Expression :- A regular expression is a sequence of chaterters that define a search pattern, mainly for use in pattern matching i.e. ried and replace " = like, oprations = example - r, s, r.s, etc.

* Parcing

Parsing is used to check the validity of a source string and is

Scanned with CamScanner

determine its syntatic structure.

b) For an invalid string the parser issues diagonstic messages reporting the cause and nature of errors in the string for valid string it builds a parse tree to reflect the sequence of derivations or reductions performed during passing.

c) Two fundamental approaches to parsing is.

- Top-down parsing :- It is the parser which operates according to a gram G which attempts to derive a string matching a source string through a sequence of derivatives. Future it is classified into -

(i) Top-down parsing with backtracking
(ii) Top-down parsing without backtracking
(iii) LL(1) parser.
(iv) Recursive descent parser

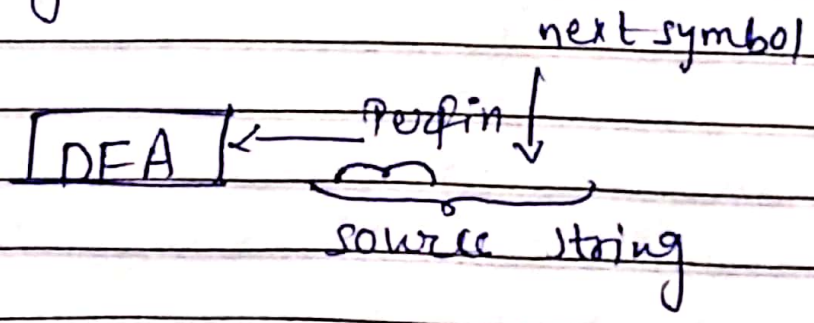
Scanned with CamScanner

Bottom-up parsing :- A bottom up parser constructs a passes tree for a source string through a sequance of reductions. It proceeds in a left-to-right manner. Further it is classified into

(i) operator precedence parser
(ii) LR parser.

Q.2 Define and give the usage of DFA

A It is a ~~def~~ finite state automaton none of whole states has two or more transitions for the same source symbol. The DFA has the property that reaches a unique state for every source string input to it.



A At any point, the DFA recognized some, perfix of the source string possibly the null string and would be posed to recognize the symbol

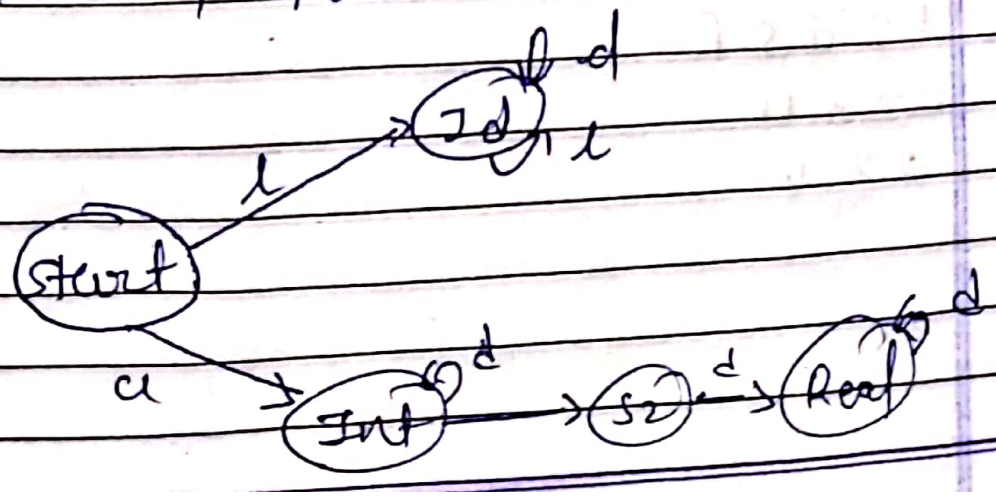
Pointed by the Pointed not symbol.

- The validity of a string is determined by going it at the input by of a DFA in its intial state

- The string is valid if and only if the DFA recognized every symbol in the string and find itself in a final state at the end of string.

Example of Building DFAs

State	Next symbol		
	l	d	
State	Id	Int	
Id	Id	Id	
Int		Int	S1
S2		Real	
Real		Real	



Scanned with CamScanner

Date: / /

A combined DFA for intergers real number & identifies.

→ Here, in this DFA is useing 3 final state - Id, int & Real corresponding to identities, unsigned integer and unsigned real respectivley, string like '25' in invalid because it leaves the DFA in state s_2 which is not a final state.

→ correct input would be like '1234-45' or 'asd12' etc.

Q.3) Take Example of any three valid string and constant DFA.

→ 0 valid string.

12dsF

234

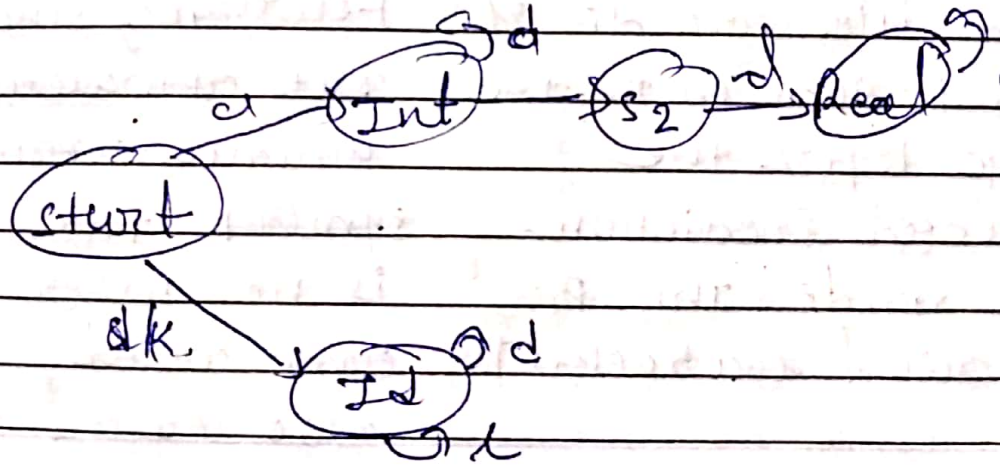
23.4

Scanned with CamScanner

Date: / /

start	Next symbol		
.	d	l	-
start	Int	Id	S_2
Int	Int		S_2
Id	Id	Int	S_2
S_2	Real		
Real	Real		

* Graphical Representation



Q.4) For Above same Example constant registers expression also.

Q.5) Different between ~~two~~ TOP-down Parsing with backtracking and without backtracking.

Scanned with CamScanner

Top-down parsing with Backtracking	Top-down parsing without Backtracking
1. The left-most nonterminal in CSF is identified i.e. E, so, then, it matches the input string its assigned grammar. If the check fails the predicities are discarded and it goes on to the state before the rejected prediction was made. This proud is called backtracking.	Here, excluded the use of Backtracking which makes the Predictive very easier. The grammer have to be modified so that it does not head left factoring since its ruts alternatives produce unique Terminal symbol. Thus, Parsing is no longer by trial and error. It is also called Predictive Parsers.
2) The backtracking results in problems like. (1) semantic actions cannot be performed while making a perdiction. (2) Precise error reporting is not possible.	when removed backtracking it resulted into several advantages like. (1) Parsing become more efficient and also be possible to perform semantic actions.

Scanned with CamScanner

Page No.:
Date: / /

③ also it consumes time as it goes to pervious stage for serching.	② Precise error reporting during passing. ③ It does not consume time compared to top-down passing with Backtracking
③ Grammer used Here, is, $E, = T + E / T$ $T : V * T / V$ $V := \langle id \rangle$	Grammar used Here is, $E := TE''$ $E'' := +E / \varepsilon$ $T := VT^*$ $T^* := *T / \varepsilon$ $V := \langle id \rangle$

Scanned with CamScanner

Date: / /

ii) Diffrent between top-down pussing and bottom up Pussing.

	Top-down pussing	Bottom up pussing
1)	It is a pussing sturgety that first looks at the lighter liked of Prosse that tree 'who' which down this pures tree by using the rules of grammer.	It is a pussing strategy that first wots of the lowest lived of the pres tree and wirts up the Prcle tree by using the rules of gramma
2)	Top-down pussing attempts to find the left most cleritions for an input storing.	Bottom up pussing can be define as an attemp to reduce this input string to sturts symbol of grammer.
3.	In this pussing technique, we stents pussing from top (start symbol of pules tree) to down (the lodd node apaces tree) in to a drawn mennner.	In the pussing technique we sturt pussing from bottom (and node of passes trees ho up (this start symbol of passes tree) in bottom up mennner.

Scanned with CamScanner

Date: / /

4.	This pussing techinge using left most derivation.	This pussing technique uses Right most derivation.
5.	It' mein decision is to bselect wheet production mule to use in order to constant the string	It's mein decision is to select when to use a production rule to reduce the string to get the sturing symbol.

Q. Define

(a) Grammer :-

→ The lenicel and syntactic Fourus of a programming lenguage are specified by its grammer. A grammer is a set of rules which Procidetyy speiPy the sentences of a lenguage I.

Example :-

< Noun phrase > → < Adtive > < Noun >

< Article > a / an / the

< Noun > boy / apple

Scanned with CamScanner

Page No. :
Date : / /

(b) Terminal symbol

→ A symbol in the alphbet is known as a terminal symbol.

(c) Non - Terminal symbol.

→ A non - Terminal symbol is the name of a symbon category of a language, p.g. now, New. etc. An non - terminal symbol is written as a single capital letter, or as name enclosed between <---->. eg. A or (NOWN)

→ During grammetic analysis, a non terminal symbol no present on instance of the category. Thus <NOWN> represntes a now.

(d) Production rule :-

→ A moduction also called a rewriting rule, is a rule of grammar, It has the from A non-terminal symbol → string of terminal and non-terminal symbol.

Scanned with CamScanner

Page No. :
Date : / /

where the notation '→' stands for 'is defined as' example < Noun phrase > → < Article > < Noun >

(C) Reduction Rule :

1) Let Production : P_1 of grammar of be of the from

$$P_1 : A ::= \alpha$$

2) and let σ be a string such that $\sigma = y \alpha \theta$, then replacement of α by in string σ constitutes a reduction according to Production P_1.

For Example :-

Grammar : < Noun phrase > → < Article > < Nouns >

< Articles > → a | an | the

< Nouns > → boy | apple

string = the boy

Step	string
0	the boy
1	< Article > boy
2	< Article > < Nouns >
3	< Noun phrases >

Scanned with CamScanner

Date: / /

(f) Derivation Rule :-

→ Let Production P_1 of Grammer G, be of the form

$$P_1 : A ::= \alpha$$

and let B be a string such that $B \equiv \gamma A \theta$,
then replacement of A by α in B string B constitues a derivation according to Production P_1.

Example :-

Grammar :- <Noun phrase> → <Article> <Noun>
<Article> → a | an | the
<Noun> → boy | apple.

String : the boy.
Derivation

<Noun phrase> = <Article> <Noun>
⇒ <the <Noun>>
⇒ the boy.

Scanned with CamScanner

Page No.:
Date: / /

(d) Abstruck syntan Tree (AST)

→ A Prose tree depicts the steps in prousing, hence it is, useful, for understanding the Process of Pussing, However, It consteins too much information. so, an abstrend syntan tree (AST) representes the structure of a source study in a meure economicou manner.

→ An AST is a tree reprentation of the obstanst syndatatic sturtiance of source code written in a Procegramming language. each node of the tree denotes a contulst occuring in the source code.

→ The word 'abstreet' implies that It is a representation designed by a complieds deginer for his own Purpose. Thus, the designed has total control, over the information represented in an AST.

(h) left-to-left Pussing :

→ TOP down Pussing trenading to a

attempts to derive a string meaning a source string through a sequence of derivations, starting with the distinguished symbol of G. For a valid source string α, a top down parser thus determines a derivation sequence:

$$S \Rightarrow \cdots \Rightarrow \cdots \Rightarrow \alpha$$

Since, we make the derivation for the left-to-left parsing (LL parsing)

(i) Left-to-right parsing :-

→ A bottom up parser constructs a parse tree for a source string throught a sequence of reductions the sources string is valid if it can be reduced so, h, the distinguished symbol of G. If, not, an error is to be detected, Bottom up parsing proceeds in a left-to-right, manner i.e in attempts, at reduction. start with the right theme, bottom up-parsing is also known as left-to-right parsing.

Scanned with CamScanner

Page No. :
Date : / /

(j) SSM :-
→ SSM stands for source starting marker. SSM points ths the first unmatched symbol is the source string.

(k) Prodition :-
→ when we select an alternative on the RHS of the production for the leftmost terminal in CSF, since, we, do not know wheter the derived string with satisfy the critiriation check, we call, this choice of prediction.

Q.8] Explain the following parsing with grammer algorithm and example

(a) Topdown parser with backtracking.

Grammar :- $F := T + E / T$
$T := V * T / V$
$V := \langle id \rangle$

→ source string → a+b+c to $\langle id \rangle + \langle id \rangle * \langle id \rangle$

Scanned with CamScanner

The Prediction making mechanism selects the RHS alternative of a Production in a left-to-right manner.

1.	E	(starts with E)
2.	$T + E$	(Applied rule 1 and replaced $E \to T + E$)
3.	$V * T + E$	(Applied rule 2 and replaced $\to V * T$)
4.	$\langle id \rangle * T + E$	(Applied rule 3 and replaced $V \to \langle id \rangle$ but it does not match with our and result use wort $\langle id \rangle +$ and not $\langle id \rangle$ so match go to step 3 and replace $T \to V$.
3.	$V + E$	(Applied rule 2 and replaced $T \to V$)
4.	$\langle id \rangle + E$	(Applied rule 3 and replaced $V \to \langle id \rangle$)
5.	$\langle id \rangle + T + E$	(Applied rule 1 & replaced $E \to T + E$)
6.	$\langle id \rangle + V * T + E$	(Applied rule 2 & replaced $T \to V * T$)
7)	$\langle id \rangle + \langle id \rangle * T + E$	(Applied rule 3 & replaced $V \to \langle id \rangle$)

Scanned with CamScanner

Date: / /

8. <id> + <id> * V * T * E (Applied rule 3 & replaced T → V * T)

9. <id> + <id> * <id> * T + E (Applied rule 3 & 2 replaced V → <Id> but it does not match so we have to backtrack and go back to step 5 and replace E by F).

5) <id> + T (Applied rule 1 & replaced E → T)

6) <id> + V * T (Applied rule 2 & replaced T → V * T)

7) <id> + <id> * T (Applied rule 3 & replaced V → <id>

8) <id> + <id> * V * T (Applied rule 2 & replaced T → V * T)

9) <id> + <id> * <id> * T (Applied rule 3 & replaced V → <id> but it does not match so we have to go back to step 8 and replaced T → V).

8. <id> + <id> * V (Applied rule 2 & replaced F → V)

9) <id> + <id> * <id> (Applied rule 3 & replaced V → <Id>)

Scanned with CamScanner

So at the end we got the string matching the source string.

(b.) Top-down parser without Backtracking.

Grammar:

$E := TE''$
$E'' := +E / \varepsilon$
$T := VT$
$T' := *T / \varepsilon$
$V := \langle id \rangle$

Source string → $a * b + c$ ⇒
$\langle id \rangle + \langle id \rangle + \langle id \rangle$

1. E (Starts with E)
2. TE (Applied rule 1 & replaced $E \to TE''$)
3. $VT''E''$ (Applied rule 3 & replaced $T \to VT''$)
4. $\langle id \rangle T''E''$ (Applied rule 5 & replaced $V \to \langle id \rangle$)
5. $\langle id \rangle * TE''$ (Applied rule 4 & replaced $T'' \to *T$)
6. $\langle id \rangle * VT''E''$ (Applied rule 3 & replaced $T \to VT''$)

Scanned with CamScanner

Page No. :
Date : / /

7) <id> * <id> T' E' (Applied rule 5 & replaced V → <id>)

8. <id> * <id> ε E (Applied rule 4 & replaced T with ε (epsilon) because we wanted * and not so it was the best prediction accumulated)

9) <id> * <id> ε E'' (Applied rule 4 & replaced ε'' → + ε)

10) <id> * <id> + ε E' (Applied rule 1 & replaced E → T E'')

11) <id> * <id> + VT'' E'' (Applied rule 3 & replaced T ε VT'')

12) <id> * <id> + <id> T'' E'' (Applied rule 5 & replaced V → <id>)

13. <id> * <id> + <id> ε ε (Applied rule 4 & 5 and replaced T'' → ε, E' → ε because we have found the matching string.

14) <id> * <id> + <id>

Scanned with CamScanner

Date: / /

Q) LL1 Parser.

A) An LL(1) parser is a table driven parser for left-to-right parsing the '1' in LL(1) indicates that the grammar uses a look ahead of one source symbol - that is, the prediction to be made is determined by the next source symbol.

A) Example:

$E ::= TE'$

$E' ::= +TE' \mid \varepsilon$

$T ::= VT'$

$T' ::= *VT' \mid \varepsilon$

$V ::= \langle id \rangle$

Source string:

$+\langle id\rangle * \langle id\rangle + \langle id\rangle -|$

Parser Table

Non Terminal	source symbol			
	$\langle id\rangle$	+	*	-|
E	$E \Rightarrow TE'$			
E'		$E' \Rightarrow TE'$		$E' \Rightarrow \delta$
T	$T \Rightarrow VT'$			
T'		$T \Rightarrow \varepsilon$	$T \Rightarrow *VT'$	$T \Rightarrow \varepsilon$
V	$V \Rightarrow \langle id\rangle$			

Scanned with CamScanner

Date: / /

LL Parsing

current sentential form	symbol	Tradition
$\vdash E \dashv$	<id>	$E \Rightarrow TE'$
$\vdash TE' \dashv$	<id>	$T \Rightarrow VT'$
$\vdash VT'E' \dashv$	<id>	$V \Rightarrow$ <id>
$\vdash$ <id> $T'E' \dashv$	*	$T' \Rightarrow *VT'$
$\vdash$ <id> $T'E' \dashv$	*	$V \Rightarrow$ <id>
$\vdash$ <id> * $VT'E' \dashv$	<id>	$V \Rightarrow$ <id>
$\vdash$ <id> * <id> $T'E' \dashv$	+	$T' \Rightarrow \varepsilon$
$\vdash$ <id> * <id> $E' \dashv$	+	$E' \Rightarrow +TE'$
$\vdash$ <id> * <id> + $TE' \dashv$	<id>	$T \Rightarrow VT'$
$\vdash$ <id> * <id> + $VT'E' \dashv$	<id>	$V \Rightarrow$ <id>
$\vdash$ <id> * <id> + <id> $T'E' \dashv$	+	$T' \Rightarrow \varepsilon$
$\vdash$ <id> * <id> + <id> $E' \dashv$	+	$E' \Rightarrow \varepsilon$
$\vdash$ <id> * <id> + <id> $\dashv$	—	—

Q. Recursive Descent parser :-

A recursive descent parser is a variant of top down parsing without back tracking.

- If uses a set of recursive procedures to perform parsing.
- To implement recursive decent parsing a left factored grammar is modified to remove to recursion

Date: / /

Offences of strings more explicit.

- Salient advantage of recursive descent passing are its simplicity and generalitive.
- It can be implemented in any lang age supporting recursive procedures.

Example :-

Recursive descent passes

```
procedure proc-G : (tree-root)

var
    a,b : pointer to a tree node.
begin
    proc-T (a)
    while (nextsymb) = '+') do
    {   match ('+');
        proc-T (b);
        a := tree build ('+'),a,b);

    tree-root := a;
    return,
end proc-E ;
```

Date: / /

```
Procedure PROC-T (tree-root)
var
  a,b : pointer to a tree node;
begin
  PROC-V('a')
(5)
(6) while (next-symb = 'K') do;
(7) { match ('*');
      PROC-V(b);
      a := treebuid ('*',a,b);

(8) tree-root := a;
    return;
    end PROC-T;

(9) { Procedure PROC-V (tree-root)
      var
        a: pointed to a tree-node;
      begin
          (nextsym = <id>) then
      tree-root := treebuid <id>,--)
      else Print "Error";
      return;
      End PROC-V;
```

string to be Persed → p+q*r.

Page No. :
Date : / /

call proc E

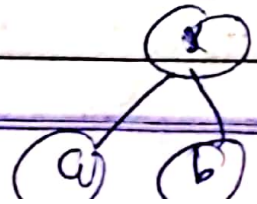
step - 3 call proc_T (a).
in that call proc V(a)
in that next symbol is id
so build a tree node → (8)

step 5 - return to step 5
next symbol is not '*'

step 7 :- so ~~then~~ return to step 5
must symbol is '+'
so match ('+')
call proc_T (b):
call proc_V (b)
not symbol is id
so build a tree node (9)

step 5 - return to step 5
then out symbol is '*'
so match '*'
and call proc V (c)
→ creti tree node → (10)

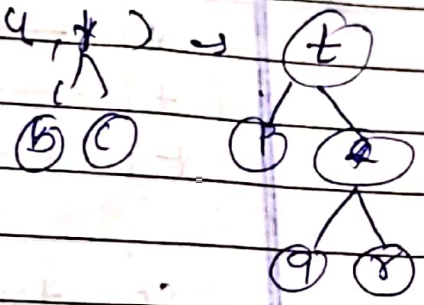
step (7) return to step (7)
and build the tree (*, a, b)



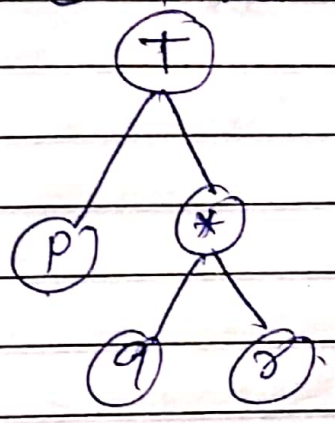
Scanned with CamScanner

Date: / /

step-③ Now, return, to step 3
and treebuild ('+', a, *) →



so, the final tree is



⑥ operator Precedence parser.

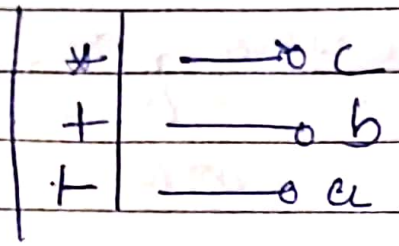
Precedence between operators a & b appering in a sentenital form a Pn where P is a null string is termed a operator precursor.

Example: The parsing string is
a + b * c

Date: / /

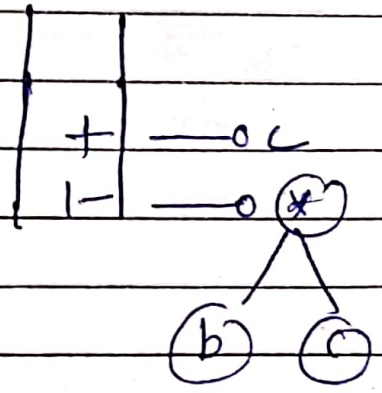
stack

1)



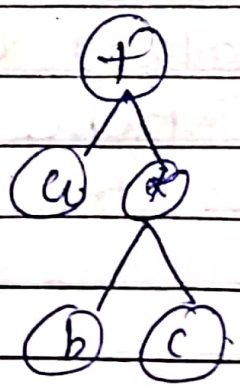
so the operator persidence of is higher so it we pop out

2.



Last remining operator will pop out

3)



Scanned with CamScanner

Page No. :
Date : / /

* Macros and Macro Prototype *

Q.1 Macro Prototype :-

- A Macro Prototype statement declares the name of a macro and the names and kinds of its parameters.

- The Macro prototype statment has the full syntax :-

<macro name> [<Formal parameter spec>[,...]]

- where <macro name> appears in the mnemonic field of an assembly statement and <Formal parameter spec> is of the form.

& <Parameter name> [<Parameter kind>]

Q.2 Macro Definition :-

- A Macro Definition is enclosed betwe a macro header statement and a macro end statement.

- Macro definitions are typically locate

Scanned with CamScanner

at the start of a Program A Macro definitetion consists of

1. Macro Prototypes statement
2. one or model statement.
3. Macro Preprocessor statement

→ The Macro Prototype statement declared the name of a assembly language statement may be generated during macro expansion.

→ A Preprocessor statement is used to Perform quility function during Macro expansion.

Exa :n

```
MACRO
INCR    &MEM_VAL, &INC_VAL, &REG
MOVER   &REG    : &MEM_VAL
ADD     &REG    &INC_VAL
MOVEM   &REG    &MEM_VAL
MEND
```

Q.3 Model statements :n

→ A model statement is a statement

Page No. :
Date : / /

from which assembly language statement may be generated during macro Expansion.

- The statements between MACRO and MEND directives define the model statements of the macro and can appear in the Expanded code.

Q.4 Preprocessor Statment :

A Preprocessor statement is used to perform auxiliary functions during macro expansion ~~AIF~~ and AGO are preprocessor statement.

Q.5 Positional Parameters

- For positional formal parameters, the specification & Parameterment kinds of synex rule is simply committed.

- Thus, a positional formal parameter is written as & <parameter name> ie, & SAMPLE, where SAMPLE, is the name of a parameter.

Scanned with CamScanner

Date : / /

- The a call on a macro using positional parameters in an ordinary string.

- The value of a positional formula parameter xyz is determined by the rule of positional association as follows:

* Find the orinal position of xyz in the list of formal parameters in the macro prototype statement.

* Find the actual parameter specification that occupies the same oridinal position in the list of actulll parameters in the macro call statement. If it is an oriamary string ABC, the value of formal parameter xyz would be ABC.

For Exa.

```
MACRO
N1   &P1, &P2, &P3
...
...
MEND
```

Scanned with CamScanner

Page No. :
Date : / /

Here, P1, P2, and P3, are positional parameters.

Q.6 Keyword Parameters

- Keyword parameters are symbolic parameters that can be specified in any order when the macro is called.

- The parameter will be replaced within the macro body by the value specified when the macro is called these parameters can be given a default value. If no default value is specified and if the parameters is not given a value when the macro is called, then the parameters will be replaced by a null string.

- Each keyword parameter will have an equal sing (=) as the last character of the parameter name.

For example:

```
MACRO
M1      &P1 = , &P2 =, &P3 =
---
---
MEND
```

Here, KP1 = , KP2, = KP3 = are keyword parameter.

Q.7 Defult parameters:

- A defult is a standard assumption in the absence of an explicit specification by the programmer.

- when the desired value is different from the defult value, the desired value can be specified explicity in a macro call, This specification overside the defult value of the parameter for the duration of the call.

- If a parameter has the same value in most calls on a macro, this value can be specified as its defults value in the macro definition.

- Defult values of keyword parameters can be specified by extending the syntax of formal parameter specification, as follows:-

Scanned with CamScanner

& < parameter name > [< parameter kind >

[< defult value >]

For exe.

MACRO

MI &P1 = A , &P2 = B.

MEND

Here, &P1 = A and &P2 = B are defult parameters.

Q.8 Give two Examples of nested macro calls.

- A modes statement in a macro may constitute a call on another macro such calls are known as nested macro calls.

- The macro containing the nested is called as the outer macro.

- The macro which is called in the outer macro, is called as a inner macro.

Scanned with CamScanner

Expansion of nested macro calls follows the last-in-first-out (LIFO) rule.

Ex:

1)
```
MACRO
MAC X &PAR1, &PAR2
MOVER REG1, &PAR1
MACRO                    //Nested macro
MAC = X, &PAR2, REG1 - REG3
ADD REG1, &PAR2
MOVEM REG1, &PAR2
MEND
PRINT &PAR1
MEND
```

2)
```
MACRO
MAC y &PAR2, REG1, REG3
ADD REG1, &PAR2
MOVEM REG1, &PAR2
MAND
MACRO
MAC-X &PAR1, &PAR2
MOVER
MAC Y REG1, PAR2
PRINT &PAR2, REG1 - REG3,
MEND &PAR1
```

Page No.:
Date: / /

Q. Expression Time variable

i) Expression time variables (EV's) are variable which can only be used during the expansion of macro calls.

ii) A local E.V. is created for use only during a paramicular macro calls. A global E.V. exists across all macro calls situated in a program and can be used in any macro which has a declaration for it.

iii) Local and global EV's are created through declaration statement with the following syntax:

LCL < E.V. specification > [, < EV specification > ...]
GBL < E.V. specification > [... < EV. specification > ...]

and < up Pu. specification > has the syntax P < E.V. name where < EV name > is an ordinary string

Page No. :
Date : / /

→ values of EV's can be manipulated through the preprocessor statement SET. A SET statement is written as

<PV specification> SET <SET. expression>

→ where <PV specification> appears in the label field and SET in the mnemonic field A SET statement assign the value of <SET. expression> to the EV specified>

~~to~~ Example :-

```
        MACRO
        CONSTANTS
        LCL     &A
&A      SET     1
        DB      &A
&A      SET     &A + 1
        DB      &A
        MEND
```

Q.10 Parameter Attributes :-

→ An attribute is written using the syntax <attribute name> <Parameter

parameter specs

c) and represents information about the value of the format parameter i.e. about the corresponding actual parameter. The type length, and size attributes have the names T, L, and S.

Example :

```
        MACRO
        CONSTANTS
&A      LCL     &A
        SET     1
        DB      &A
&A      SET     &A+1
        DB      &A
        MEND
```

Q.10 AIF

An AIF statement has the syntax

AIF (&Expression) &sequencing-symbo

where &expression is a relational expression involving ordinary strings

Page No. :
Date : / /

parameter and their attributes, and expression time variable.

If the relational expression evaluates to be true, expression time control is transferred to the statement containing <sequencing symbol> in its label field.

Q.12 AIF

An - AIF statement has the syntax :-

AIF <sequencing symbol>

It unconditionally transfer expression time control to the statement containing <sequencing symbol> in its labes fileds.

Example : MACRO
PVAL &X, &X, &Z
AIF (PY, EQ, &X), ONE

MOVER ARFET, &X
SUB ARGE, &Y
ADD ARGE, &Z
MOVEY ARGE, &X

Scanned with CamScanner

AUTO , OVER
ONY MOVER ARGUT , &2
MOVER ARGUT , &X
ONLY MEND

Q.13 ANOP

→ An ANOP statement has the syntax
< sequencing symbol > ANOP.

→ An significance of ANOP statement is to define sequencing symbol.

Q.14 REPT

→ REPT statement has the syntax :

REPT < Expression >

→ < Expression > should Evaluate in to a numerical value during macro expression the statement between REPT, and an ENDM statement would be processed for Expansion < expression > number of times.

Scanned with CamScanner

Following example illustrates the use of this facility to declared to constants with the values 1, 2 10

Exa.

```
        MACRO
        CONSTIO
        LCL     &M
&M      SET     1
        REPT    10
        DC      'AH'
&M      SETA    &M+1
        ENDM
        MEND
```

Q.15) IRP

The IRP statement

IRP <formal parameter>, <argument-list>

The formal parameter mentioned in the statements takes successive value from the argument list. For each value the statement between the IRP and ENDM statement are expanded once.

Example :- MACRO

CONSTAN &M, &N, &P

Page No. :
Date : / /

IRP
DC
ENDH
N END

P2, 2M, 7, 8N
'82'

y A MACRO call CONSTS 4, 10, leads to declarations of 3 constants with the value 4, 7, and 7.

Q.16. Show combined working of MACRO and assemble.

y A MACRO PROCESSOR is functionally independent of the assembler and the output of the MACRO PROCESSOR will be part of the input into the assembler

y A MACRO EXPRESSER similar any other assembler scans, and processor statements

y Often the use of a seperate MACRO PROCESSOR for sending MACRO instructions leads to less effient program translation because many

Scanned with CamScanner

Date: / /

Function are duplicated by the assembler and macro processor.

To overcome efficiency issues and avoid duplicate work by the assembler the macro processor is generally implemented within passes of the assembler.

The intergration of macro processor and assembler is often referred as macro assembler. such implementation will help in elimination the overheads of creating intermediate files.

thus improving the performance of integration by combining similar functions

To design the pass structure of a macro-assembler we identify the function of a macro preprocessor and be merged to advantage. After merging the function assemble this processes leads to the following pass structures:-

Scanned with CamScanner

Page No. :
Date : / /

Pass I.
1. Macro definitation processing
2. SYNTAB construction

Pass 2:
1. MACRO expression
2. allocation cam LC processing
3. Processing of literals
4. Intermediate code generation.

Pass III
1. Target code generation

Q.17 Macro Name Table :-

Macro Name table (MNT) has Entries for all macro defined in a Program.

1) each MNT Entry contains three Pointers MDTP, KPDTP and SSTP, which are pointed to MDT, KPDTAB and SSNATB, for the macro representively

2) MNT also contains number of Positional Parameters (#PP), number of keyword Parameter (#KP) and

Date: / /

of Expansion time variable (EV)

Q.18 parameter Name Table.

→ parameter Name table (PNTAB) contains all the parameter name used in macro.

→ It contains all keyword, default and positional parameters names.

Q.19 EV Name Table:

→ EV Name Table [EVNTAB] contains all the expansion time variable names.

→ Expansion time variable are local variable and global variable which are declared in macro.

Q.20 SS NAME Table

→ SS Name table (SSNTAB) contains the sequencing symbols.

Scanned with CamScanner

Page No. :
Date : / /

Q.2 Keyword Parameters Default Table.

→ Keyword Parameters Default table (KPDTAB) contains the name of keyword parameters and it's value.

Q.22) Macro Definition Table in

Macro Definition table (MDT) Entries are constructed while expressing the model statement and preprocessor statements in the macro body.

Scanned with CamScanner

Page No. :
Date : / /

Assignment - 5 Compiler.

Q1) Diffrent between static and dynamic memory allocations

Static memory Allocation	Dynamic memory Allocation.
→ In this memory is allocated to a variable before the execution of a program begin.	The memory binding are established and destroyed during the exception of program
→ No, memory allocation and deallocation action are perfomed during the execution of a program.	Memory is allocated and deallocated during the execution of program
→ variable remain permanently allocated allocation of variable exists over if the program unit in which is is define is not active.	The variable get allocated only if the program unit get active. - e.g PL/I, pascal, Ada etc.

Scanned with CamScanner

Date : / /

(b) explain scope rule with example

- For data declaration it is possible to use same name in memory blocks program.
- This would be established memory bindings of the form (name, vari for different value of a variable.
- scope rule determines which of these binding so, it effective at a specific place in the program
- e.g.:- consider the block structured program. The variable accessible within the variable blocks are as follows :-

```
A [ x, y, z : integer ;
    B [ q : r ;
        C [ h, z : real ;
    A [ i, j, : integer ;
```

Scanned with CamScanner

Page No. :
Date : / /

Block	Accessible variables	
	local	non local
A	xA, yA, zA	–
B	qB	xA, yA, zA
C	hC, zC	xA, yA, qB
D	iD, jD	xA, yA, zA

→ variable zA is not accessible inside Block C since C, contains a declaration using the name z, Thus, zA, and zC, are two distinent variable this would be true even if they had identical attributes. i.e. even if z of way declared to be an integer.

Q. Explain memory allocation in Recursion.

→ Recursion Procedure (or function) ture characterized by the fact that many invocations of a Procedure coexist during the execution of a Program.

Scanned with CamScanner

Page No.:
Date: / /

a) A copy of the local variable of the procedure must be allocated for each invocation.

b) This does not pose any problem when a stack model of memory allocations is used because an activation record is created for every introduction of a procedure or function.

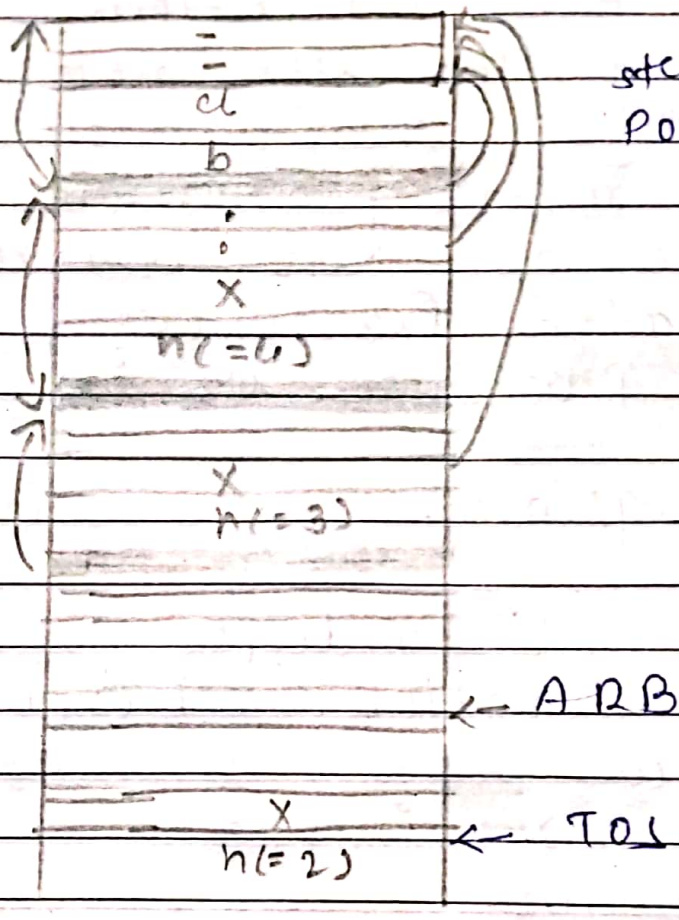
4) Program sample (input, output);
var
a, b : integer;
function fib (n) : integer;
var
x : integer
begin
if n > 2 then
x := fib (n-1) + fib (n-2);
else
x := 1;
return (x);
end fib;
begin
fib (4);
end.

For the call fib (4) the function makes recursive calls.

Scanned with CamScanner

Page No. :
Date : / /

- pictorial representation of recursive call in fibonacci series program



- define Dope vector with Example.

- If array dimension bounds are not known during compilation the dop vector has to exist during program execution.

- The number of dimention of an array determines the format and

Scanned with CamScanner

size of its DV.

The dop vector is allocated in the A.R of a block and dov, it's displacement in AR, is noted in the symbol table entry of the Array.

```
MOVER     AREG I
COMP      AREG = "S"
BC        GT, I, ERROR, RTN, Error if i≥
COMP      AREG = '1'
BC        LT, ERROR, RTN, Error if i<1
MOVER     AREG, 'I',
COMP      ARGT = "10",
BC        LT, ERROR, RTN  Error if j>10
OLT       AREG, '-5", Mutiply by range
ADD       AREG, I.
MULT      AREG, = '2'  Mutiply by
                       element size
ADD       AREG, ADDR, BASE, Add,
                       Address of a[0,0].
```

The array is allocated dynamic by repeating step for the array.

Date: / /

- Its start address value of dj, l, j, and range ij are entered in the DV.

- The generated code used a dov to acess the complete of DV to check the validity of subistruction and compute the address of an array element.

Define with Example

1) operand descripter

- It has the following fields.
- 1) Attributes :- contains the subfield types length and miscellaneous information.

2) Addressability :- specifies where the opprands is located and between it can be assessed. It has 2 subfields.
   1) Addresbility code
   2) Address

Scanned with CamScanner

Page No. :
Date : / /

- An operand descriptor is built for every operand participating in an expression.

- An operand descriptor is built for an id when the id reduced during parsing

- A partial result pri is the result of evaluating same opprator opj... A descriptor is built for pri prime ~~eliately~~ immidiatly after code is generated for oprator opj.

- for simplicity assume that all operand descriptor are stands in an array called operand descriptor.

Eg: MOVER AREG, A
MOLT AREG, B

operand descriptor :-

1	(int, 1)	M, addr(a)	Pesmip for a
2	(int, 1)	M, addr(b)	" b
3	(int, 1)	R, (addr(AREG)	" a x b

Scanned with CamScanner

Page No. :
Date : / /

(ii) Register Descriptor :-

A Register descriptor has two fields

1) status :-
contains the code free or occupied. For indicate register status.

2) operand descriptor # :-

If status = occupied, this field contains the descripter # for the operand contained in the register.

- Register descriptor are stored in an array called Register discriptor.

- one register discripter exists for each (PU) register.

E.g :-
The register descriptor ~~regist~~ for ARCH after generating code for up, b.

occupied	#3

Scanned with CamScanner

Page No. :
Date : / /

This indicates that register AREG (contains the operand descriptor i

Q.1 write the steps for Assembly lang code generation from an expression specified in high level language.

A) Expression :- $x + y * p + q$

step no.	Parsing action	code generation
1	$\langle id \rangle x \rightarrow E^1$	Build discriptor #1
2	$E^1 \rightarrow T^1$	-
3	$\langle id \rangle y \rightarrow E^2$	Build descriptor #2
4	$T^1 + E^2 \rightarrow T^3$	generate MOVER, AREG, x, ADD, AREG, y, Build descriptor #3.
5.	$T^3 \rightarrow E^3$	Build descriptor #
6.	$\langle id \rangle p \rightarrow E^4$	Build descript #
7	$E^4 \rightarrow T^4$	-
8.	$\langle id \rangle q \rightarrow E^5$	Build descriptor
9.	$T^4 * E^5 \rightarrow T^6$	generate MOVER

Scanned with CamScanner

Date : / /

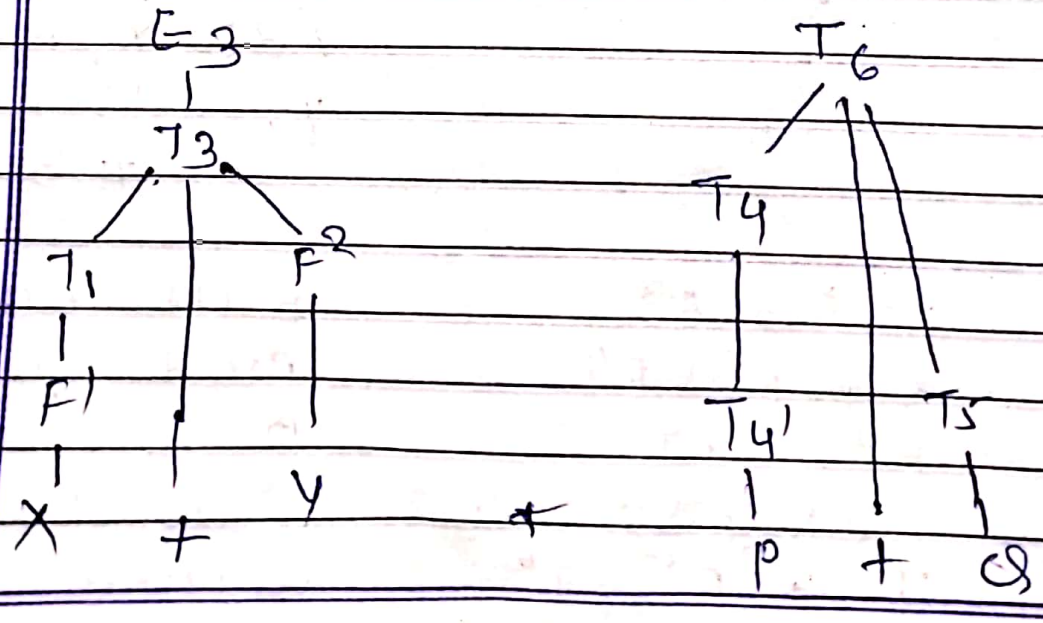
TEM, MOVER, AREG, P
ADD, AREG, Q
Build descriptor

10 $E^3 + T^6 \rightarrow E^6$ generate MULT
AREG TEM

⇒ operand descriptor is

1	(int, I)	m, addr (x)
2	(int, I)	m, addr (y)
3	(int, I)	m, addr[temp (I)]
4	(int I)	m, addr (P)
5	(int, I)	m, addr (Q)
6	(int, I)	m addr (AREG)

2)



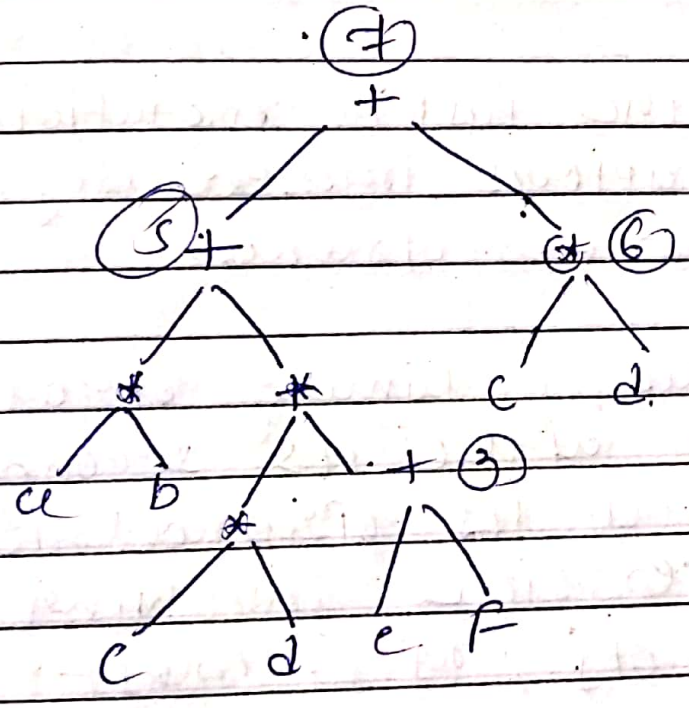
Scanned with CamScanner

Register descripter

=> | C X C | , C |

with an Example, Explain the alternative of assembly language code generation for an Expression

X + Y + P*Q + (R + S) + P * Q



MOVER	AREG, X
MOLT	AREG, Y
MOVERM	AREG, TEMP 1
MOVER R	AREG, P
MULT	AREG, Q
MOVEM	AREG, TEMP-2
MOVERM	AREG, R

Scanned with CamScanner

MOVER	AREG, s.
MULT	AREG, TEMP-2
ADD.	AREG, TEMP-3
MOVER	AREG, P.
MULT	AREG, Q
ADD	AREG, TEMP-3

Q.) now Prefix string can be used for generating the intermediate code for expression.

→ In the postfix notation each operator appears immediately after its last ~~de~~ operands.

→ Thus, a binary operator opp. appears after its second operand If any of its operands is itself an expression involving an operator opj, rop j, must appear, before opi,

e.g :- 1- a + b * c + d * e T F - 1
source string.

Postfix string is a popular intermedi code in non-optiminimum compiler due to ease of generation as we

- The code generation from the postfix string using a stack of operand descriptor.

- operand description are pused on the stack as oprand appear in the string.

- when an operater with arity x, appears in the string k descriptor are popped of R the stack.

- A conversion of fix to postfix is modified i.e. operends appearing in the source string are copied into the postfix string straightway.

- The TOS oprator is popped into the postfix string is to c operators → current oprator.

- How triples and quadraples can be used for generating the intermediate crude for expression.

- <u>Triples</u> :- A triples is a representation of an elementary operation in

Scanned with CamScanner

Page No. :
Date : / /

the form of a pseudo machine instruction.

Format :-

operator	operand 1	operand 2

⇒ each operand of a triple is either a variable / constant or the result of some evaluation represented by another triple.

⇒ conversion of infix, string into triples can be achieved by a slight modification i.e.-

infix string :- 1 - a + b * c * d * e f - 1

⇒ [I] in the operand field of triples ∝ indicate that the operanded i.e. the value of b * e, represented by triple number 1.

	operator	operands	operand 2
1	*	b	c
2	+	[1]	a
3	*	e	f
4	*	d	[3]
5	+	[2]	[4]

Scanned with CamScanner

Page No. :
Date : / /

quadruples

A quadruples represents an elementary evaluation in the following format.

operator	operand 1	operand 2	result name

here, result name designates the result of the evaluation. It can be used as the operand of another quadruples.

E.g :-	operator	operand 1	operand 2	Result name
1	-	b	c	t1
2	+	t1	a	t2
3	↑	e	f	t3
4	*	d	t3	t4
5	+	F2	t4	t5

List the step taken by compiler at function call & produce call.

while implementing a function call the compiler must ensure the following.

Scanned with CamScanner

Date : / /

(i) Actual parameter are accessible call the compiler ~~and~~ call function
(ii) The called function is able to produce side effects according to the rules of the PL
(iii) control is transfered to end is referenced from the called function.
(iv) The function value is returned to the calling program.
(v) All other aspects of execution of a calling programs are unaffected by the function calls.

→ compiler used a set of factor to implement function.
(i) parameter list
(ii) calling conventions.
(iii) save area,

→ Diffrent function calls :-
→ Diffrent type of functions calls are :-

(i) call by value :- In this mechanism value of actual

Page No. :
Date : / /

Parameters are passed to the called function.

c) These value are assigned to the corresponding formal parameters the passing as value only taken place in one direction i.e. from calling program to the called function.

d) In this value changed in the formal parameter does not affect the value of actual parameter.

(ii) call by value → Result → In this mechanism the value of formal parameter is copied back into actual parameter.

→ This mechanism indicates the simplicity of the call the by the value mechanism but increase higer overhead.

(ii) call by reference

→ In this the address of actual

Scanned with CamScanner

Page No. :
Date : / /

parameter is passing to the called function.

- The parameter list is thus a list of address of actual parameter At every access of a formal parameter in the function the address of the corresponding actual parameter is obtain.

- If the value is change in the formal parameter it will after the value of actual parameter.

(iii) <u>call by name</u> :- In this also the same machinism is follow of the call by reference.

- But the only diffrence is that if every occurence of a formal parameter in the body of called function is replaced by the name of the corresponding actual parameter.

Scanned with CamScanner

=> Here, also change in formal parameter effect the value of actual parameter.

Scanned with CamScanner