easy – solutions

Strictly as per the new revised syllabus of
Mumbai University w.e.f. academic year 2014-2015

System Programming & Compiler Construction

Semester VI
Computer Engineering

Marketed By

EME928

Scanned by CamScanner

System Programming & Compiler Construction

Semester VI – Computer Engineering (MU)

Copyright © with easy-solutions. All rights reserved. No part of this publication may be reproduced, copied, or stored in a retrieval system, distributed or transmitted in any form or by any means, including photocopy, recording, or other electronic or mechanical methods, without the prior written permission of the publisher.

This book is sold subject to the condition that it shall not, by the way of trade or otherwise, be lent, resold, hired out, or otherwise circulated without the publisher's prior written consent in any form of binding or cover other than which it is published and without a similar condition including this condition being imposed on the subsequent purchaser and without limiting the rights under copyright reserved above.

Edition 2017

This edition is for sale in India, Bangladesh, Bhutan, Maldives, Nepal, Pakistan, Sri Lanka and designated countries in South-East Asia. Sale and purchase of this book outside of these countries is unauthorized by the publisher.

Printed at : Image Offset, Dugane Ind. Area Survey No. 28/25, Dhayari Near Pari Company, Pune - 41, Maharashtra State, India. E-mail : rahulshahimage@gmail.com

Published by

easy-solutions

Head Office : B/5, First floor, Maniratna Complex, Taware Colony, Aranyeshwar Corner, Pune - 411 009. Maharashtra State, India. Ph : 91-20-24225065, 91-20-24217965. Fax 020-24228978. Email : info@techmaxbooks.com, Website : www.techmaxbooks.com

(Book Code : EME92B)

Scanned by CamScanner

INDEX

Chapter 1	:	System Software
Chapter 2	:	Assemblers
Chapter 3	:	Macro & Macro Processors
Chapter 4	:	Loaders and Linkers
Chapter 5	:	Software Development Tools
Chapter 6	:	Compilers
Chapter 7	:	Lexical Analysis
Chapter 8	:	Syntax Analysis
Chapter 9	:	Syntax Directed Translation
Chapter 10	:	Intermediate Code Generation
Chapter 11	:	Code Generation
Chapter 12	:	Code Optimization
Chapter 13	:	Run Time Storage
Chapter 14	:	Compiler-Compilers

Table of Contents

- **Index**
- **Syllabus**
- **May 2015** — **M(15)-1 to M(15)-20**
- **Dec. 2015** — **D(15)-1 to D(15)-21**
- **May 2016** — **M(16)-1 to M(16)-07**
- **Dec. 2016** — **D(16)-1 to D(16)-15**
- **University Question Papers** — **Q-1 to Q-04**

❑❑❑

Scanned by CamScanner

Syllabus

1. **System Software :** Concept, introduction to various system programs such as assemblers, loaders, linkers, macro processors, compilers, interpreters, operating systems, device drivers.

2. **Assemblers :** General Design Procedure , Design of Assembler (Single Pass - Assembler IBM PC, multi pass Assembler IBM 360/370 Processor), Statement of Problem, Data Structure, format of Databases, Algorithm, Look for modularity.

3. **Macros & Macro processors :** Macro instructions, Features of Macro facility, Design of 2 pass macroprocessor.

4. **Loaders and Linkers :** loader schemes, Design of Absolute loader, Design of Direct linking loader.

5. **Software Tools :** Software Tools for Program development, Editors: Types of Editors, Design of Editor, Debug Monitors.

6. **Compilers :** Introduction to Compilers, Phases of a compiler, comparison of compilers and interpreters.

7. **Lexical Analysis :** Role of a Lexical analyzer, input buffering, specification and recognition of tokens, Designing a lexical analyzer generator, Pattern matching based on NFA's.

8. **Syntax Analysis :** Role of Parser, Topdown parsing, Recursive descent and predictive parsers (LL), BottomUp parsing, Operator precedence parsing, LR, SLR and LALR parsers.

9. **Syntax Directed Translation :** Syntax directed definitions, Inherited and Synthesized attributes, Evaluation order for SDDs , S attributed Definitions, L attributed Definitions.

10. **Intermediate Code Generation :** Intermediate languages: declarations, Assignment statements, Boolean expression, case statements, back patching, procedure calls.

11. **Code Generation :** Issues in the design of Code Generator, Basic Blocks and Flow graphs, code generation algorithm, DAG representation of Basic Block.

12. **Code Optimization :** Principal sources of Optimization, Optimization of Basic Blocks, Loops in Flow graph, Peephole Optimization.

13. **Run Time Storage :** Storage Organization, storage allocation strategies, parameter passing, Symbol table, introduction to garbage collection and compaction.

14. **Compiler-Compilers :** JAVA compiler environment, YACC compiler-compiler.

❑❑❑

Scanned by CamScanner

System Programming & Compiler Construction

Statistical Analysis

Chapter No.	May 2015	Dec. 2015	May 2016	Dec. 2016
Chapter 1	05 Marks	10 Marks	05 Marks	10 Marks
Chapter 2	15 Marks	10 Marks	10 Marks	08 Marks
Chapter 3	-	10 Marks	-	10 Marks
Chapter 4	05 Marks	10 Marks	10 Marks	-
Chapter 5	05 Marks	05 Marks	05 Marks	05 Marks
Chapter 6	-	10 Marks	10 Marks	-
Chapter 7	03 Marks	-	05 Marks	05 Marks
Chapter 8	25 Marks	30 Marks	10 Marks	32 Marks
Chapter 9	-	05 Marks	05 Marks	10 Marks
Chapter 10	20 Marks	10 Marks	10 Marks	15 Marks
Chapter 11	20 Marks	-	20 Marks	-
Chapter 12	15 Marks	10 Marks	10 Marks	10 Marks
Chapter 13	10 Marks	05 Marks	05 Marks	10 Marks
Chapter 14	03 Marks	05 Marks	05 Marks	05 Marks
Repeated Questions	-	14 Marks	74 Marks	53 Marks

May 2015

Chapter 1 : System Software [Total Marks : 05]

Q. 1(a) Differentiate between application program and system program. **(5 Marks)**

Ans. :

Difference between application program and system program :

Sr. No.	System Program	Application Program
1.	System software is any computer software which manages and controls computer hardware so that application software can perform a task.	Application software, are programs that enable the end-user to perform specific, productive tasks, such as word processing or image manipulation.
2.	Example: Operating systems, such as Microsoft Windows, Mac OS X or Linux, Compilers, Linkers etc.	Example: Microsoft Word, Oracle, Paint brush, spread sheets, anti-virus tools etc.
3.	The function of the systems software is to manage the resources of the computer, automate its operation and facilitate program development. It is generally provided by the computer manufacturer or a specialized programming firm	Application software's are designed to perform specific data processing or computational tasks for the user. These programs are specifically designed to meet end-user requirements.

Scanned by CamScanner

Sr. No.	System Program	Application Program
4.	System software is the Operating System which handles devices, etc.	Application Software is third party software that seeks help from the system software
5.	System software or Operating System is the software that allows the computer to boot.	Application software are programs that run inside of or on top of the Operating System and allows you to do things like email or word processing.
6.	System software performs tasks like transferring data from memory to disk, or rendering text onto a display device. Specific kinds of system software include loading programs, operating systems, device drivers, programming tools, compilers, assemblers, linkers, and utility software.	Application software are used to meet the user specific needs.

Chapter 2 : Assemblers [Total Marks : 15]

Q. 1(b) State the reason for assembler to be multi pass program. **(5 Marks)**

Ans. : Reasons of multi pass program of assembler : A multi pass assembler means more than one pass is used by assembler. Examples of the multi pass assemblers are IBM 360 / 370 processor. The advantages of multi pass assemble over single pass assembler are as follows:

1. It is used to eliminate forward reference problem.
2. Numbers of passes are created in multi pass assembler to process the definition of symbols.
3. Multi pass assembler does the work in two pass
   1. First Pass :
      - Scans the code.
      - Validate the tokens.
      - Create Symbol Table.
   2. Second Pass :
      - Solves forward reference problem.
      - Converts the code to the machine code.

A two pass assembler does two passes over the source file. Below mentioned is the detail of task accomplished by each pass. The forward reference problem is solved by making two passes over the assembly code.

Machine Independent: Since the multiple passes include a modular structure, and the code generation decoupled from the other steps of the compiler, the passes can be reused for different hardware/machines.

Q. 2(a) For following code what will be output generated by Pass-I and Pass-II for two pass assembler. Explain with database. **(10 Marks)**

```
ABC     Start   0
        USING   *,15
        L       1, FIVE
        A       1, FOUR
        ST      1, TEMP
FOUR    DC      F'4'
FIVE    DC      F'5'
TEMP    DS      1F
        END
```


Scanned by CamScanner

Ans. :

PRG	START	0
	USING	*, 15
	L	1, FIVE
	A	1, FOUR
	ST	1, TEMP
FOUR	DC	F '4'
FIVE	DC	F '5'
TEMP	DS	IF
	END	

The above mentioned program was assumed to be called by another program that left the address of the first instruction in register 15.

Pass 1 :

Sample Program			Relative Address	Mnemonic Instruction
PRG	START	0		
	USING	*,15		
	L	1, FIVE	0	L 1, – (0, 15)
	A	1, FOUR	4	A 1, – (0, 15)
	ST	1, TEMP	8	ST 1, – (0, 15)
FOUR	DC	F '4'	12	4
FIVE	DC	F '5'	16	5
TEMP	DS	1F	20	–
	END			

Look up bit configuration for the mnemonic in a table and put the bit configuration for the L in the appropriate place. Now we need the address of five. At this point we don't know where FIVE is, so we cannot supply its address. And no index register is used. It put in 0 for the index register and we know that the register 15 is being using as the base register, but we cannot calculate offset so we write 1, – (0, 15). It maintain a location counter indicating the relative address of the instruction being processed this counter is incremented by 4. The next instruction is an Add instruction. We look up the opcode but we do not know the offset for four. The same thing happens with store (ST) instruction. The DC instruction is a pseudo-opcode directing us to define some data.

Pass 2 :

For 'four' that have to produce a '4'. This word has been stored at relative location 12, from the pass 1. So give the value 12 to the Load (L) instruction in pass 2. Same case is with Add and store instruction and their location counter value will be given as 12 and 20.

Sample program :

```
PRG  START
     USING *, 15
     L    1, FOUR
     A    1, FIVE
```


Scanned by CamScanner

```
        A   1, = F' 2'
        ST  1, TEMP
FOUR    DC  F'4'
FIVE    DC  F'5'
TEMP    DS  1F
        END
```

First pass :

Relative Address	Mnemonic Instruction
0	L 1, –(0,15)
4	A 1, – (0, 15)
8	ST 1 – (0,15)
12	4
16	5
20	
24	–

Second Pass :

Relative Address	Mnemonic Instruction
0	L 1, 12 (0,15)
4	A 1, 16 (0, 15)
8	ST 1, 20 (0,15)
12	4
16	5
20	
24	–

Symbol Table :

Symbol (8 Bytes)	Value (4 Bytes)	Length (1 Byte)	Relocation (1 Byte)
PRG	0	01	R
FOUR	12	04	R
FIVE	16	04	R
TEMP	20	04	R

Literal Table :

Literal	Value	Length	Relocatable
= F'2'	28	04	R

Scanned by CamScanner

Base Table :

	Availability Indicator	Contents Of Base Reg
0	N	
1	N	
. . .	. . .	
15	Y	00

Chapter 4 : Loaders and Linkers [Total Marks : 05]

Q. 1(c) Explain Functions of loader. **(5 Marks)**

Ans. :

Functions of loader :

The fundamental task of a loader is to :

1. Bringing an object program into memory and 2. Starting its execution

An object program contains translated instructions and data from the source program. It also specifies addresses in memory where these items are to be loaded.

To execute a program a loader performs the following four functions :

1. **Allocation :** It is used to allocate space in memory for the object programs. Translators cannot allocate space since overlap may occur or large wastage of memory takes place.
2. **Linking :** It combines two or more separate object programs and resolve symbolic references between object decks. It also supplies the information needed to allow to reference between them.
3. **Relocation**
   It modifies the object program so that it can be loaded at an address different from the location originally specified and adjusts all address dependent location.
4. **Loading**
   Physically it places the machine instructions and data into the memory for the execution.

Chapter 5 : Software Development Tools [Total Marks : 05]

Q. 6(c) Write short note on : Software tools. **(5 Marks)**

Ans. :

Software development tools :

Software development tools can be roughly divided into following categories.

(i) Debugging tools
(ii) Static analysis and formal verification tools
(iii) Performance analysis tools.
(iv) Application build tools
(v) Correctness checking tools
(vi) Integrated development environment
(vii) Memory usage tools.

Scanned by CamScanner

1. Debugging Tools

Debugger or debugging program allows us to monitor control of a program during program execution. Debugger informs the program to execute until a specific statement is reached upon reaching statement the debugger allows us to look at the change value of different variables in the program and also the registers used by the PC. Debugger debug the program in four basic steps they are

1. Discovering the Bug.
2. Isolating the Bug
3. Finding the Bug
4. Fixing the bug

Some widely used debuggers are :

(i) GNU debugger (GDB)
(ii) Intel Debugger (IDB)
(iii) Microsoft visual studio Debugger.
(iv) Eclipse Debugger.

2. Static Analysis and Formal Verification Tools

These type of tools are generally used by the developers as part of the development and component testing process. In this source code is not executed or run, but the tool itself is executed and the source code is the input data to the tool.

3. Performance Analysis Tools

These types of tools are used for the parallel programming. There are important tools to understand the behaviour of the program.

4. Application Build Tools

An application build tool is a utility that is used when building a new version of a program.

5. Correctness Checking Tools

These type of tools are used to check the correctness of the program. This tools automatically verify the correctness proportion of the program.

6. Integrated Development Environment

It is also known as interactive development environment is a software application that provides comprehensive facilities to computer programmers for software development.

7. Memory Usage Tools

These type of tools are used to understand how to solve commonly memory issues.

Chapter 7 : Lexical Analysis [Total Marks : 03]

Q. 6(b) Write short note on : LEX. **(3 Marks)**

Ans. :

LEX

LEX is commonly used with the YACC parser generator. It was originally written by Eric Schmidt and Mike Lesk. LEX is a program generator designed for lexical processing of character input streams. The code written in LEX recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. LEX helps in writing the programs whose control flow is directed by instances of regular expressions in the input stream.

The regular expressions are given to the LEX and it associates the regular expressions and the program fragments. LEX is not a complete language, but rather a generation representing a new language feature which can be added to different programming languages, called "host languages." LEX can write code in different host languages. The host language is used for the output code generated by LEX and also for the program fragments added by the user.

Scanned by CamScanner

As LEX is adaptable to various environments and users so it provides the compatible run-time libraries for the different host languages. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions.

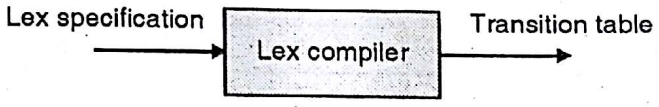


Fig. 1 : LEX compiler

Chapter 8 : Syntax Analysis [Total Marks : 25]

Q. 2(b) Explain operator precedence parser along with example. **(10 Marks)**

Ans. :

Operator precedence parser

It is a bottom-up parser that interprets an operator-precedence grammar. For example, most calculators use operator precedence parsers to convert from the human-readable infix notation. The operator precedence parsing technique can be applied to operator grammars.

Operator Grammar are defined as the grammars with the following properties :

1. **No epsilon** in the right hand side of any production
2. **No adjacent non-terminals** in the right hand side of any production.

This property enables the implementation of efficient operator-precedence parsers. These parsers rely on the following three precedence relations :

Relation	Meaning
a <· b	a yields precedence to b
a =· b	a has the same precedence as b
a ·> b	a takes precedence over b

Fig. 2 : Precedence relations

These operator precedence relations allow delimiting the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end. Also, a <· b need not imply b ·> a

Let us assume that between the symbols ai and ai + 1 there is exactly one precedence relation. Suppose that $ is the end of the string.

Then for all terminals we can write : $ <· b and b ·> $. If we remove all nonterminals and place the correct precedence relation : <·, =· , ·> between the remaining terminals, there remain strings that can be analyzed by developed parser.

For example, the following operator precedence relations can be introduced for simple expressions :

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

Scanned by CamScanner

Example :

Consider the grammar
E → E + E | E * E | id

Precedence Table

	+	*	id	$
+	·>	<·	<·	·>
*	·>	·>	<·	·>
Id	·>	·>	·>	·>
$	<·	<·	<·	<·

Consider the input string "id1 + id2 * id3"
The string with the precedence relations inserted from the above table is

$ <· id1 ·> + <· id2 ·> * <·id3 ·> $

Having precedence relations allows identifying handles as follows :
1.Scan the string from left until seeing ·>
2.Scan backwards the string from right to left until seeing <·
3.Everything between the two relations <· a ·> forms the handle
The handles can be found as
If <· Push in stack
If ·> Pop till a <· is found and reduce
$ <· id1 ·> + <·id2 ·> * <·id3·> $
$ <· E + <·id2 ·> * <·id3·> $
$ <· E + <·id2 ·> * <·id3·> $
$ <· E + <·id2 ·> * <·id3·> $
$ <· E + <· E * <·id3·> $
$ <· E + <· E * E ·> $
$ <· E + E· > $
S <· E ·> $

Q. 4(a) Construct predictive parsing table for following grammar. **(10 Marks)**

S → A
A → aB | Ad
B → bBC | f
C → g

Ans.:

S → A
A → aB|Ad
B → bBC|f
C → g
C → g

S′ → S $
S → A
A → aB|Ad
B → bBC|f

FIRST (S) → { a }
FIRST (A) → { a }

FOLLOW (S) → { $ }
FOLLOW (A) → { d }

Scanned by CamScanner

FIRST (B) → { b, f } FOLLOW (B) → { g }
FIRST (C) → { g, f } FOLLOW (C) → { $ }
FIRST (S′) → { a }

Predictive Parser table or LL(1) table :

	a	b	g	f	$
S′	S → S$				
S	S → A				
A	A → aB|Ad				
B		B → bBC		B → f	
C			c → g	B → bBC	

Q. 6(d) Write short note on : Left recursion and left factoring removal technique. **(5 Marks)**

Ans.:

Left Recursion

A grammar is said to be left recursive if it has a non-terminal, say A, such that there is a derivation A => Aα, for some string α. Presence of left recursion creates difficulties while designing the parsers. Left recursion may be of two types :

1. Immediate left recursion
2. General left recursion

Top-down parsing methods cannot handle left recursive grammars, so a transformation that eliminates left-recursion is needed. It means driving the input in one or more steps. Here A is non terminal and α denotes some input string. This expansion of A causes further expansion of A only and due to generation of A, A α, Aαα, Aααα the input pointer will not be advanced. This causes major problem in top down parsing and therefore elimination of left recursion is a must.

Removing direct left recursion

The general algorithm to remove direct left recursion follows. Several improvements to this method have been made. For a left-recursive nontermina A, discard any rules of the form A → A and consider those that remain:

$$A \rightarrow A\alpha_1 | \ldots | A\alpha_n | \beta_1 | \ldots | \beta_m$$

where:

1. Each α is a nonempty sequence of nonterminals and terminals, and
2. Each β is a sequence of nonterminals and terminals that does not start with A.

Replace these with two sets of productions, one set for A:

$$A \rightarrow \beta_1 A' | \ldots | \beta_m A'$$

and another set for the fresh nonterminal A′ (often called the "tail" or the "rest"):

$$A' \rightarrow \alpha_1 A' | \ldots | \alpha_n A' | \varepsilon$$

Repeat this process until no direct left recursion remains.As an example, consider the rule set

Expression → Expression + Expression | Integer | String

This could be rewritten to avoid left recursion as

Expression → Integer Expression' | String Expression'

Expression' → + Expression Expression' | ε

Scanned by CamScanner

Removing all left recursion :

By establishing a topological ordering on nonterminals, the above process can be extended to also eliminate indirect left recursion:

Inputs *A grammar: a set of nonterminals* A_1.....A_n *and their productions*

Output *A modified grammar generating the same language but without left recursion*

For each nonterminal A_i :

Repeat until an iteration leaves the grammar unchanged :

For each rule $A_1 \rightarrow \alpha_i \alpha_l$ being a sequence of terminals and nonterminals :

If α_i begins with a nonterminal A_j and $j < i$;

Let β_i be α_i without its leading A_j.

Remove the rule $A_i \rightarrow \alpha_r$.

For each rule $A_j \rightarrow \alpha_j$

Add the rule $A_i \rightarrow \alpha_j \beta_i$.

Remove direct left recursion for A_i as described above.

Note that this algorithm is highly sensitive to the nonterminal ordering; optimizations often focus on choosing this ordering well.

Example :

Consider the following left-recursive grammar for arithmetic expressions.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

Elimination of immediate left recursion from the rules modifies the grammar as,

$$E \rightarrow TE'$$
$$E' \rightarrow + TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow * FT' \mid \varepsilon$$
$$F \rightarrow (F) \mid id$$

Left Factoring

A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*. The basic idea behind left factoring is that when we are not clear which production rule is to be used for the expansion of any non-terminal.

For example, if we have the two productions

$$S \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

On seeing this we cannot identify which productions are used to expand A whether $\alpha\beta_1$ or $\alpha\beta_2$ this will lead us to the infinite parse tree. However, we can make the grammar left factored by expanding A to $\alpha A'$. Then, after seeing the input derived fromα, we may expand S' to $\beta 1$ or $\beta 2$. So the expression changes to

$$S \rightarrow \alpha S'$$
$$S' \rightarrow \beta_1 \mid \beta_2$$

Algorithm :

Construction of Left factoring grammar

Input : Grammar

Scanned by CamScanner

Output: Left factored grammar

Steps :

For each non-terminal S with two or more alternatives (production rules) with a common non-empty prefix, let say

$$S \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

convert it into

$$S \rightarrow \alpha S' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
$$S' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

Here S′ is a new non terminal. Repeat this transformation until no two alternatives for a non terminal have a common prefix.

$$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$$
$$\Downarrow$$
$$A \rightarrow aA' \mid \underline{cdg} \mid \underline{cde}B \mid \underline{cdf}B$$
$$A' \rightarrow bB \mid B$$
$$\Downarrow$$
$$A \rightarrow aA' \mid cdA''$$
$$A' \rightarrow bB \mid B$$
$$A'' \rightarrow g \mid eB \mid fB$$

Chapter 10 : Intermediate Code Generation [Total Marks : 20]

Q. 3(a) Generate three address code for following code.

```
While (a < b) do
if (c < d) then
x = y + 2
else
x = y – 2
```

(10 Marks)

Ans. :

Three address code for a given expressions

```
while (a < b) do
if (c < d) then
x = y + 2
else
x = y - 2
L1 :  while (a < b) goto L2
      goto last
L2 :  if (c < d) goto L3
      Goto L4:
L3 :  t1 = y
      t2 = 2
      t3 = t1 + t2
      x = t3
L4:   t1 = y
```


Scanned by CamScanner

$t_2 = 2$
$t_3 = t_1 - t_2$
$x = t_3$
last.

Q. 3(b) Discuss with example quadruple, triple and indirect triple. **(10 Marks)**

Ans. :

Qaudraple :

A quadruples is a record structure having four fields as follows :

Operator	Operand 1	Operand 2	Result

Fig. 3 : Structure of Quadruple

Where operand1 and operand2 are two operands for the operator and result field contains the result of the operation on operand1 and operand2. For example **a:=b+c** will be represented as

+ b c a

The fields operand1, operand2 and result are pointers to the symbol table entries, therefore temporary names must be entered into the symbol table. The three-address statement

x: = a op b

is represented by placing a in operand1, b in operand 2, and x in result.

Unary operators like **arg1:= -arg2 or arg1:=arg2** does not use operand2. Procedure call like ***param*** use neither operand2 nor result. A conditional and unconditional jump puts the target label in result. Operand1, Operand2 and result are usually pointers to symbol entries. Quadraples may need to use many temporary names.

Example :

a: = b*-c + b*-c.

Three-Address code :

t_1: = - c
t_3: = - c
t_5: = $t_2 + t_4$
t_2: = b * t_1
t_4: = b * t_3
a: = t_5

	Operator	Operand1	Operand2	Result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Fig. 4 : Quadruples representation for the three-address statement

Triple :

A triple is a record structure having three fields as follows

Opewrator	Operand 1	Operand2

Fig. 5 : Structure of Triples

To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.

Scanned by CamScanner

If we do so the three-address code can be represented with only three fields operator, operand1 and operand2 are either pointers to the symbol table or pointers into the triple structure. It refers to the symbol table for user defined names or constants and to the triple structure for temporary values.

The triple A * B represents the infix notation *A B.

Example : a:= b*-c + b*-c.

Three-Address Code

t_1: = c
t_2: = b * t_1
t_3: = c
t_4: = b * t_3
t_5: = t_2 + t_4
a: = t_5

	Op	arg1	arg2
(0)	Uminus	c	
(1)	*	b	(0)
(2)	Uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	Assign	a	(4)

Fig. 6 : Triples representation for Three-Address statements

Parenthesized numbers represent pointers into the triple structure, while symbol-table pointers are represented by the name themselves. In practice the information needed to interpret the different kinds of entries in the operand1 and operand2 fields can be encoded into the operator field or some additional fields. The triples in Fig. 6 correspond to the quadruples in Fig. 4. Note that the copy statement a:= t5 is encoded in triple representation by placing a in the operand field by using the operator assign.

Indirect Triples : Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called indirect triples. In indirect triples we use an array statement to list pointers to triples in the desired order. Then the triples in Fig. 6 might be represented as in Fig. 7.

	Statements
0	100
1	101
2	102
3	103
4	104
5	105

→

#	Operator	Argument 1	
100	uminus	c	
101	*	b	100
102	uminus	c	
103	*	b	102
104	+	101	103
105	=	a	104

Fig. 7 : Indirect Triples Representation of Three-Address Statements

Advantages of Indirect Triple :

1. In indirect triples the temporaries are implicit and easier to rearrange code.
2. A statement can be moved by reordering the statement list.

Scanned by CamScanner

Chapter 11 : Code Generation [Total Marks : 20]

Q. 1(d) What is flow graph ? State its signification in code generation. **(5 Marks)**

Ans. : Flow graph :

A flow graph is a graphical representation of a sequence of instructions with control flow edges. A flow graph can be determined at the intermediate code level or target level.

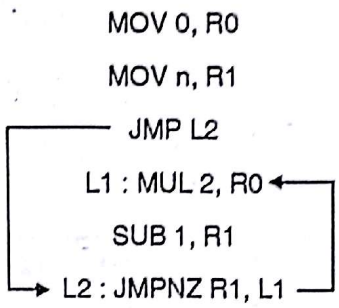


Fig. 8 : Flow Graph Representation

To build the graph, first divide the code into basic blocks. A basic block is a sequence of consecutive intermediate language statements in which flow of control can only enter at the beginning and leave at the end.

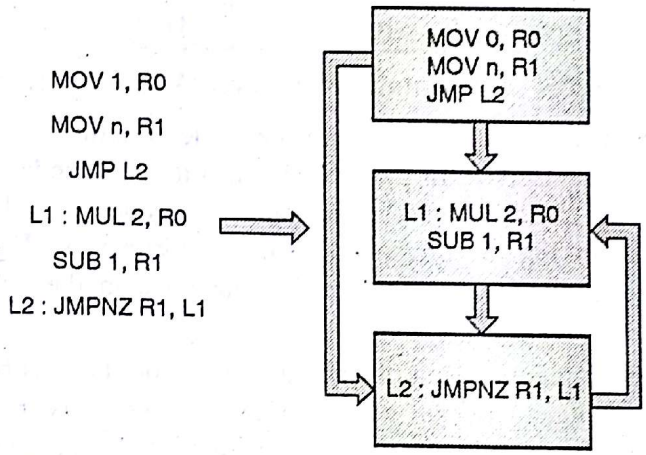


Fig. 9 : Basic Block and Control Flow Graph Representation

A basic block has exactly one entry and one exit point. It states that in basic block only the first statement can be reached from outside the block and all statements are executed consecutively one after another.

Significance of flow graph in code generation :

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.

Q. 5(a) What are different issues in code generation, explain in detail. **(10 Marks)**

Ans. :

1. Input data for the Code Generator :

(a) Below mentioned are the following formats used as input for code generator

(b) Three-Address Code (quadruples, triples, indirect triples)

Scanned by CamScanner

(c) Virtual machine presentations (byte-code, stack-machine)
(d) Linear presentation (postfix, infix, prefix)
(e) Graphical presentation (syntax trees, DAGs, parse tree, syntax directed translation)

2. Target Program :

Knowledge of machine architecture and instruction set are the pre-requisite for the design of good code generator. Proper selection of machine architecture helps in producing absolute machine-language program and re-locatable machine-language program.

1. Absolute machine language :

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

2. Relocatable machine language :

   Producing a relocatable machine language program as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader, to link the separately compiled program segments.

3. Assembly language :

   Producing an assembly language program as output makes the process of code generation some what easier.

3. Memory Management :

Mapping names in the source program to addresses of data objects in run-time memory is done cooperatively by the front end and the code generator. A name in a three- address statement refers to a symbol-table entry for the name. From the symbol-table information, a relative address can be determined for the name in a data area for the procedure.

4. Instruction Selection :

The complexity of mapping IR program into code sequence depends on :

1. Level of Intermediate Representation (high-level or low-level)
2. Type and nature of instruction set.

5. Register Allocation and Assignment :

Selection of set of variables that will reside in registers at each point and choosing specific register helps in faster execution of data.

6. Evaluation Order :

Selecting the order in which computations are performed effects the efficiency of the target code.

Chapter 12 : Code Optimization [Total Marks : 15]

Q. 4(b) Explain loop optimization with example. **(10 Marks)**

Ans. :

Loop Optimizations :

Loops plays a very important role in code optimizations. Since the body of the loop is executed again and again, and moreover the loops are nested so small changes in the execution of the loop body can results in a remarkable improvement in the performance of the program.

Scanned by CamScanner

Loop optimization can be viewed as the application of a sequence of specific **loop transformations** (listed below) to the source code or intermediate representation, with each transformation having an associated test for legality. A transformation (or sequence of transformations) generally must preserve the temporal sequence of all dependencies if it is to preserve the result of the program (i.e., be a legal transformation). It is most important set of optimizations in this programs are likely to spend more time in loops.

Loop optimization techniques :

1. Code Motion :

Code motion (also called *code hoisting*) unifies sequences of code common to one or more basic blocks to reduce the code size and potentially avoid expensive re-evaluation. The most common form of code motion is *loop-invariant* code motion that moves statements that evaluate to the same value every iteration of the loop to somewhere outside the loop. In code motion **invariant expressions should be executed only once.**

In Code Motion the code moves from one part of the program to other without modifying the algorithm. The two advantages of using it are as :

1. It reduces the size of the program.
2. It reduces the execution frequency of the code subjected to the movement.

2. Strength Reduction :

Operator strength reduction replaces an operator by a "less expensive" one. Given each group of identities below, which operations are the most and least expensive, assuming f is a float and i is an int ? (Trick question : it all depends on the architectures we need to know the target machine to optimize well!)

3 Induction Variable Strength Reduction :

Basic induction variables are created from derived induction variables. The rules required for creating the basic induction variables are as :

x := y op z; It states that

op can be any *, <<, +, or –

y is an induction variable and z is an invariant

x is a register and no other statement modifies x

x is not y or z

4. Loop Induction Variable :

Induction variables values are either incremented or decremented whenever the value of the variable is changed. There are three types of induction variable :

Basic Induction Variable : These type of variables are assigns within a loop. There are two types of induction variable. For example

K = K +/- D where D is a constant.

Primary Induction Variable : The loop execution is controlled in it.

(for j=0; j<100; j++)

i is the primary induction variable.

Derived Induction Variable : Derived variable is a linear function of a basic induction variable.

5. Loop Fission :

Loop Fission tries to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. It improves the locality of reference.

Scanned by CamScanner

6. **Loop Fusion or Loop Combining :**

The transformation opposite to loop fission is loop fusion. When two adjacent loops would iterate the same number of times, their bodies can be combined as long as they make no reference to each other's data.

7. **Loop Interchange :**

These optimizations exchange inner loops with outer loops.

8. **Loop Unrolling**

It duplicates the body of the loop multiple times, in order to decrease the number of times the loop conditions needs to be tested. It also reduces the number of jumps from the end of the loop to the beginning.

9. **Loop Splitting :**

Loop Splitting simplifies a loop or eliminate dependencies by breaking the loop into multiple loops which have the same bodies but iterate over different portions.

Q. 6(a) Write short note on: Code motion. **(5 Marks)**

Ans.:

Code Motion :

Code motion (also called code hoisting) unifies sequences of code common to one or more basic blocks to reduce the code size and potentially avoid expensive re-evaluation. The most common form of code motion is loop-invariant code motion that moves statements that evaluate to the same value every iteration of the loop to somewhere outside the loop. In code motion invariant expressions should be executed only once.

In Code Motion the code moves from one part of the program to other without modifying the algorithm. The two advantages of using it are as :

1. It reduces the size of the program.
2. It reduces the execution frequency of the code subjected to the movement.

Example :

```
for (int i = 0; i < x.length; i++)
        x[i] *= Math.PI * Math.cos(y);
```

```
double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i++)
 x[i] *= picosy
```

A loop is a set of basic blocks which satisfies two conditions :

1. All blocks are strongly connected, i.e. there is a path between any two blocks.
2. The set has a unique entry point, i.e. every path from outside the loop that reaches a block inside the loop enters through a single node.

Code Motion is of two types :

1. **Code Space Reduction :** Similar to common sub-expression elimination but with the objec to reduce code size.

Scanned by CamScanner

2. **Execution Frequency Reduction :** It reduces execution frequency of partially available expressions (expressions available at least in one path).

Chapter 13 : Run Time Storage [Total Marks : 10]

Q. 5(b) Explain run time storage organization in details. **(10 Marks)**

Ans. :

Run time storage organization :

In computer science it is not enough if generate machine code from the intermediate code. When execute a program only small amount of data is available. Hence only those actions are used which is necessary at the runtime of the program for this need to for this, we need to understand the relationship between names and data objects i.e. **address** and **value**. Most languages permit memory allocation and de-allocation of its data and is taken care by the run time environments which are loaded by the routines obtained by the target code. Whenever a program is executed an activation of procedure take place and if at all procedure is recursive then a number of its activation may be active at the same time. Interface between the program and the computer resource system is needed. There is a need to manage memory when program is running.

The memory management must relate to the data objects of the program. It should deal with activations of two procedures whose lifetimes are either non-overlapping or nested. For example, if x and y are two procedure activations records and y is entered before x is left, then control action must leave y before leaves a. A procedure is recursive if a new activation can begin before an earlier activation of the same procedure has ended. Programs request for the creating and the releasing of the memory block. Passing parameters to the function also need the run time resources. A procedure is a specified series of actions or operations which have to be executed in the same manner in order to always obtain the same result under the same circumstances. Procedure is associated with an identifier and a statement. The identifier represents the procedure name and the body of the procedure is given by the statement.

In an executable statement when a procedure name appears, the body of the procedure is invoked at that time. Then the procedure call executes the procedure body and the identifiers appearing in the procedure definition are known as the formal arguments or actual arguments of the function. Arguments, known as actual arguments may be passed to a called procedure, they are substituted by the formal parameters in the procedure body.

Information required during an execution of a program is kept in a block of storage called an activation record. The semantics of procedures in a language determines how names are bound to storage during allocation. Abstract machines usually offer the following type of storage areas:

1. Code : Instructions for the program. (Usually static: does not change after the program is loaded into memory).
2. Static Area : Space for variables with global scope (lifetime is same as that of the program).
3. Stack : Space for variables/data structures that are created at run-time (whose lifetime is same as the procedure where they were created).
4. Heap : Space for data structures that can be dynamically allocated or deallocated (whose lifetime is different from the procedure where they were created).

Storage Organization :

Storage organization of a program is divided into four parts :

1. Target code
2. Data objects

Scanned by CamScanner

3. Procedure activation record for stack
4. All other details are kept by the Heap area

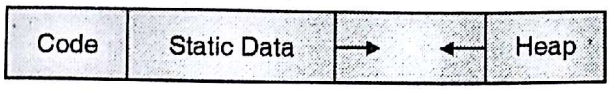


Fig. 10 : Runtime memory is divided into code and data areas

This type of runtime storage is used by the high level languages such as **FORTRAN, C and Java.** In this storage:

1. Memory is used to store the size of the generated code and data objects statically.
2. The heap may be used to store dynamically allocated data objects, and the activation records of other objects.
3. Dynamically both the stack and the heap may modify there size during program execution, so they cannot be allocated a fixed amount of space.

Chapter 14 : Compiler-Compilers [Total Marks : 03]

Q. 6(b) Write short note on : YACC. **(3 Marks)**

Ans. :

YACC-Yet Another Compiler-Compiler :

YACC stands for Yet another Compiler Compiler, first version of YACC is created by S.C. Johnson. It is a tool that translates any grammar that describes a language into a parser for that language. It generates a power (the part of a compiler that tries to make syntactic sense of the source code based on an analytic grammar written in a notation similar to BNF. YACC generates the code for the parser in the C programming language. YACC provides a general tool for imposing structure on the input to a computer program. The YACC user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. YACC then generates a function to control the input process. This function, called a parser. As lex is associated with scanners similarly YACC is linked with the parsers. We provide the input of a grammar specification and it generates an LALR (1) parser to recognize sentences in that grammar. YACC is probably the most common of the LALR tools.

It is written in Backus Naur Form (BNF), by convention, a YACC file has the suffix **.y**. The YACC compiler is invoked from the command line as:

$yacc<options>
<filename ending with .y>

The steps involved in creating a compiler in YACC are as :

1. When YACC is run over the programming language grammar a parser is generated.
2. Identify the grammar.
   The grammar is written with .y file extension.
   Lex is used to process input and pass tokens to the parser.
   yyparse() function starts parsing.
   Write error handling routines like yyerror().
3. Compile the code produced by YACC as well as from any other relevant source files.
4. Link the object files to appropriate libraries for obtaining the executable parser.

Scanned by CamScanner

Yacc generates C code for a LR(1) syntax analyzer (parser) and it also generate BNF rules for the grammar. It is written in C language which is portable and efficient.

YACC provides a tool for describing the input to a machine, specifies the structures of the input used by the machine, and together with code to be invoked for each structure is recognized. It accepts LALR (1) grammars with disambiguating rules.

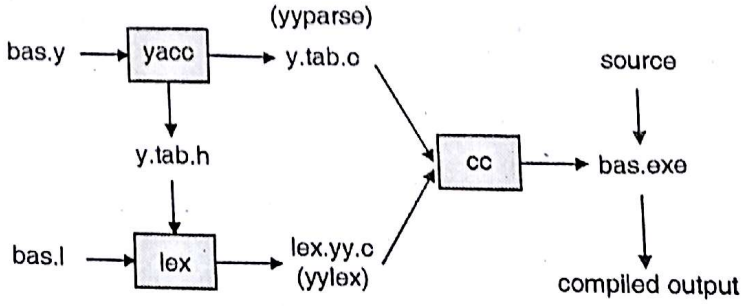


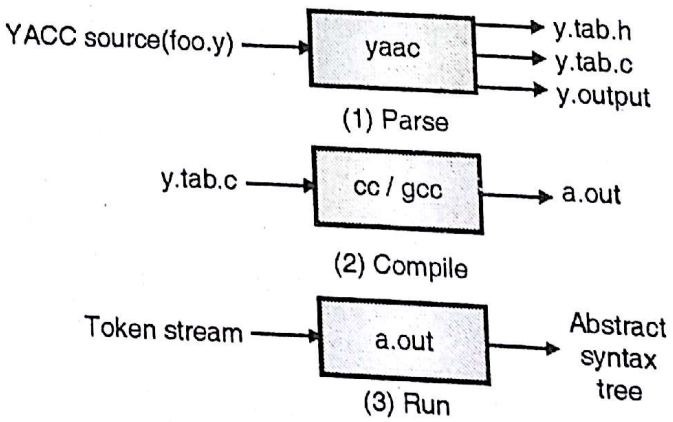
Fig. 11

Working of YACC

YACC is designed for use with C code and generates a parser written in C and it is used in combination with Lex-generated scanners. YACC parser generator is invoked .y file and it creates the y.tab.h and y.tab.c files containing a thousand or so lines of intense C code that implements an efficient LALR(1) parser for our grammar, including the code for the actions we specified.

Then the file provides an extern function yyparse.y that will attempt to successfully parse a valid sentence and compiled using C file and linked with the rest of the code, and the parser is generated. By default, the parser reads from stdin and writes to stdout, just like a lex-generated scanner does.

% yacc myFile.y	creates y.tab.c of C code for parser
% gcc -c y.tab.c	compiles parser code
% gcc –o parse y.tab.o lex.yy.o –ll -ly	link parser, scanner, libraries
% parse	invokes parser, reads from stdin



Scanned by CamScanner

Dec. 2015

Chapter 1 : System Software [Total Marks : 10]

Q. 1(a) Differentiate between Application program and system program. Indicate the order in which following system programs are used, from developing program upto its execution. Assemblers, Loaders, Linker, Macro processor, compiler, Editor. **(5 Marks)**

Ans. :

Difference between application program and system program :

Please refer Q. 1(a) of May 2015.

The order in which following system program are used from developing program upto its execution are as :

Editor → Macro processor → Assembler → Linker → Compiler → Loader

Developing---------- → Execution

Q. 1(d) What are the different functions of loader. **(5 Marks)**

Ans. : Functions of a Loader

The loader mainly performs the following four functions. Fig. 1-Q. 1(d) shows the four functions of the loader.

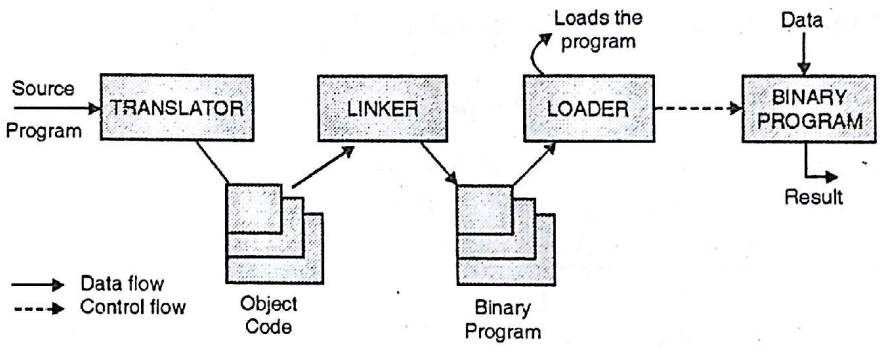


Fig. 1-Q. 1(d) : Basic Structure of Program Execution

ALLOCATION : It uses to allocate space in the memory for the programs.

LINKING : It links two or more separate object codes and provides the information needed to allow reference between them.

RELOCATION : It modifies the object program by changing certain instruction, so that it can be loaded at an address different from the location originally specified.

LOADING : It brings the object program into the memory for execution.

Chapter 2 : Assemblers [Total Marks : 10]

Q. 6(a) With reference to Assembler explain following tables with suitable example.
(i) POT (ii) MOT (iii) ST (iv) LT **(10 Marks)**

Ans. : (i) POT : Pseudo Opcode Table

POT is fixed length table i.e. the contents of these tables are not filled in or altered during the assembly process.

Scanned by CamScanner

Table 1-Q. 6(a) : Pseudo-Op Table (POT) for pass 1 (similar table for pass 2)

← 8-bytes per entry →

Pseudo-Opcode (5-bytes) (character)	Address of routine to process Pseudo-Opcode (3-bytes = 24 bit address)
"DROPb"	P1DROP
"ENDbb"	P1END
"EQUbb"	P1EQU
"START"	P1START
"USING"	P1USING

These are presumably lables of routines in pass 1; the table will actually contains the physical addresses. P1→PASS1

(ii) MOT : Machine Opcode Table

MOT is a fixed length table i.e. the contents of these tables are not filled in or altered during the assembly process.

Table 2-Q. 6(a) : Machine-Op Table (MOT) for pass 1 and pass 2

← 6-bytes per entry →

Mnemonic op-code (4-bytes) (characters)	Binaryop-code (1-byte) (hexadecimal)	Instruction length(2-bits) (binary)	Instruction format(3-bits) (binary)	Not used in this design (3-bits)
"Abbb"	5A	10	001	
"AHbb"	4A	10	001	
"ALbb"	5E	10	001	
"ALRb"	1E	01	000	
"ARbb"	1A	01	000	
...	...	...	...	
"MVCb"	D2	11	100	

b~represents the character "blank"

Codes

Instruction length

01 = 1 Half-Words = 2 bytes
10 = 2 Half-Words = 4 bytes
11 = 3 Half-Words = 6 bytes

Instruction format

000 = RR – Register to Register
001 = RX – Register and Indexed Storage Operation
010 = RS – Register and Storage Operation
011 = SI – Storage and Immediate Operation
100 = SS – Operation using all implied operand and storage

Scanned by CamScanner

(iii) ST : Symbol Table

Table 3-Q. 6(a) : Symbol Table (ST) for pass 1 and pass 2

← 14-bytes per entry →

Symbol (8-bytes) (characters)	Value (4-bytes) (hexadecimal)	Length (1-bytes) (hexadecimal)	Relocation (1-byte) (character)
"JOHNbbbb"	0000	01	"R"
"FOURbbbb"	000C	04	"R"
"FIVEbbbb"	0010	04	"R"
"TEMPbbbb"	0014	04	"R"

Symbol Table includes for each entry :

i) Name of the symbol
ii) Symbols assembly time value
iii) Length (in bytes) and
iv) Relative Location indicator (R/A)

Operand can be either immediate (typically one byte values coded in the instruction itself/)or the addresses of the data located elsewhere

(Relative/Absolute value)

R - Relative value

A - Absolute value

(iv) LT : Literal Table

Literal is equivalent to define a constant explicitly and assign an address label for it literal table is often organized as a hash table, using literal name or value as the key.

Literal Table includes for each entry

(i) Name of the literal
(ii) Literal assembly time value
(iii) Length (in bytes) and
(iv) Relative Location Indicator (R/A)

(The format is same as that of symbol Table - with required changes) based on the definition of the Literal table.

Name of Literal (1-byte)	Value of Literal (4-byte)	Length of Literal (1-byte)	Relative Location Indicator (1-byte)

Chapter 3 : Macro & Macro Processors [Total Marks : 10]

Q. 5(a) Explain different pseudo-ops used for conditional macro expansion along with an example.

(10 Marks)

Ans. : Conditional MACRO Expansion

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

Scanned by CamScanner

In conditional macro expansion only part of the macro is copied out into the code. Which part is copied out will be under the control of the parameters in the macro call.

Example

```
    ⋮
MACRO
START 1     D     1,     VALUE 1
            D     2,     VALUE 2
                         ⋮
START 2     D     1,     VALUE 1
                  2,     VALUE 2
            ⋮
VALUE 1     DC    F'0'
VALUE 2     DC    F'1'
            ⋮
      MEND
```

In the above example the labels, the operands and the number of instruction generated change in each sequence. This example can be written as :

```
Source
   ⋮
MACRO
& AG        DATA    &C,&A2, &A3
& AG        D     2, & A2
                AIF    (&C EQ 1).FINI
                                    (TEST
      if C = 1)
                D          3, &A3
                AIF        (&C EQ 2).FINI
                                    (TEST
      if C = 2)

MEND
      ⋮
      ⋮

START 1       DATA     2, VALUE2,    VALUE3
      ⋮
START 2       DATA     1, VALUE 2
                ⋮
VALUE 1       DC          F'0'
VALUE 2       DC          F'1'
⋮
MEND
```

```
Expanded source
      ⋮
      ⋮
      ⋮
      ⋮
      ⋮
      ⋮
      ⋮

START 1     D       2, VALUE 2
            D       3, VALUE 3

START 2     D       2, VALUE 2

      ⋮
```

Fig. 1-Q. 5(a) : Example of CONDITIONAL MACROS

Scanned by CamScanner

AIF : Arithmetic operations are performed by AIF and if nested condition is true then only they are branched. It is conditional opcode.

AGO : It is similar to 'go-to' statement in C/C++. It is an unconditional pseudo-opcode. It specifies a label appearing on some other statements in the macro definition. The flow of program during macro expansion is controlled by AIF and AGO. Other than AIF and AGO we also use IF-ELSE-ENDIF and WHILE-ENDW structure for the conditional branching. Symbol table is maintained to store the values of all macro variables. This table is used to look up the current value of a macro. Labels with a period (·) such as · FINI do not appear in the output of the macro processor. AIF (&C EQ 1)· FINI is a statement which guides the macro processor not to execute the statement labelled· FINI if the value corresponding to C = 1.

The macro time IF-ELSE-ENDIF structure

It provides the mechanism for either generating or skipping selected statements in the macro body. Whenever an IF statement is found during the expansion of macro the value is computed in terms of Boolean expression. If the executed value is true IF block is executed otherwise Else is evaluated.

The macro time looping WHILE-ENDW

The **While** conditional statement is useful because it allows us to repeat a block of code as long as a conditional test is **True,** rather than just a simple number of times. It is useful in situations where we need to loop through code, but do not know how many times we need to run the portion of code.

e.g. of WHILE - ENDW Structure

```
MACRO
WHILE (N! = 0)
N = N % 10
N = N / 10
SUM = SUM + N
ENDW
⋮
MEND
```

Explanation for the above example

1. The expression is evaluated.
2. If expression is initially false, the body of the while statement is never executed, and control passes from the while statement to the next statement in the program.
3. If expression is true (nonzero), the body of the statement is executed and the process is repeated beginning at step 1.
4. As soon as ENDW statement is found, the macro processor checks the Boolean expression by returning the call to the WHILE, and action is taken based on the new value of the expression.

Examples on conditional macro expansion

Example 1

The following macro called DOIO has three dummy parameters, namely &DEV, &BUF and &CODE. If the actual parameter corresponding to &CODE is R, then the expansion of the macro results in statements that read one character from the device specified by &DEV and store the character in the memory location (byte) specified by &BUF.

Scanned by CamScanner

If the actual parameter corresponding to &CODE is not R, then the expansion of the macro results in statements that write the character stored in the memory location (byte) specified by &BUF to the device specified by &DEV.

```
DOIO MACRO          &DEV, &BUF, &CODE
Initialize register A depending upon &CODE.
        IF          (&CODE EQ 'R')
        CLEAR A
        ELSE
        LDCH        &BUF
        ENDIF
Generate RD or WD instruction depending on &CODE.
$CHECK  TD          =X' &DEV'
        JEQ         $CHECK
        IF          (&CODE EQ 'R')
        RD          =X' &DEV'
        STCH        &BUF
        ELSE
        WD          =X' &DEV'
        ENDIF
        MEND
```

Notes :

1. When the DOIO macro is invoked by the statement

   DOIO A5, LOC, R

   The expanded version of the macro is as follows :

```
                CLEAR       A
AACHECK  TD     =X' A5'
                JEQ         AACHECK
                RD          =X'A5'
                STCH        LOC
```

2. When the DOIO macro is invoked a second time by the statement

   DOIO A7, LOC, W

   The expanded version of the macro is as follows :

```
                LDCH        LOC
ABCHECK         TD          =X'A7'
                JEQ         ABCHECK
                WD          = X' A7'
```

3. When the above macro is invoked using the statement

   DOIO A7, LOC

The expansion would be similar to that in (2) above. (In this case, the SIC/XE assembler treats the value of the parameter &CODE as the null character.)

Scanned by CamScanner

Chapter 4 : Loaders and Linkers [Total Marks : 10]

Q. 4(a) Explain Design of Dynamic Linking Loader along with example. **(10 Marks)**

Ans. : Dynamic Linking Loader

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

1. It is often used to allow several executing programs to share one copy of a subroutine or library. For e.g., run time support routines for C could be stored in Dynamic Link Library. A single copy of the routines in this library could be loaded into the memory of the computer. All C programs currently in execution could be linked to this one copy, instead of linking a separate copy into each object program.
2. It provides the ability to load the routines only when they are needed so lot of time and memory is saved if subroutines are large with lots of external references.
3. It helps in not loading the entire library for execution.
4. In dynamic linking loader is used to load the main program.

Implementation of Dynamic Linking Loader

1. Dynamic-link library, or DLL, is Microsoft's implementation of the shared library concept in the Microsoft Windows and OS/2 operating systems. These libraries usually have the file extension DLL, OCX (for libraries containing ActiveX controls), or DRV (for legacy system drivers).
2. In Unix operating systems, the dynamic linker is actually an external executable that the operating system kernel loads and executes first in a process address space newly constructed as a result of calling exec or posix_spawn functions. When an executable file is loaded, the operating system kernel reads the path of the dynamic linker from it and then attempts to load and execute this other executable binary.
3. In Apple Darwin operating system, OS X and iOS operating systems the dynamically loaded shared libraries can be identified either by the filename suffix .dylib or by their placement inside the bundle for a framework.
4. In Unix-like operating systems using XCOFF, such as AIX, dynamically-loaded shared libraries use the filename suffix .a.
5. Dynamic linking from Assembler language programs in IBM OS/360 and its successors is done typically using a LINK macro instruction containing a Supervisor Call instruction that activates the operating system routines that makes the library module to be linked available to the program.

Advantage

1. No over head is incurred unless the procedure to be called or referenced is actually used.
2. A further advantage is that the system can be dynamically configured.

Disadvantage

1. Considerable overhead and complexity incurved, due to the fact that we have postponed most of the binding process until execution time.

Scanned by CamScanner

Chapter 5 : Software Development Tools [Total Marks : 05]

Q. 3(b) Write a brief note on Design of an Editor. **(5 Marks)**

Ans. : Design of Editor

The structure of a text editor depends largely on the types of editing features and displaying capabilities that are to be supported.

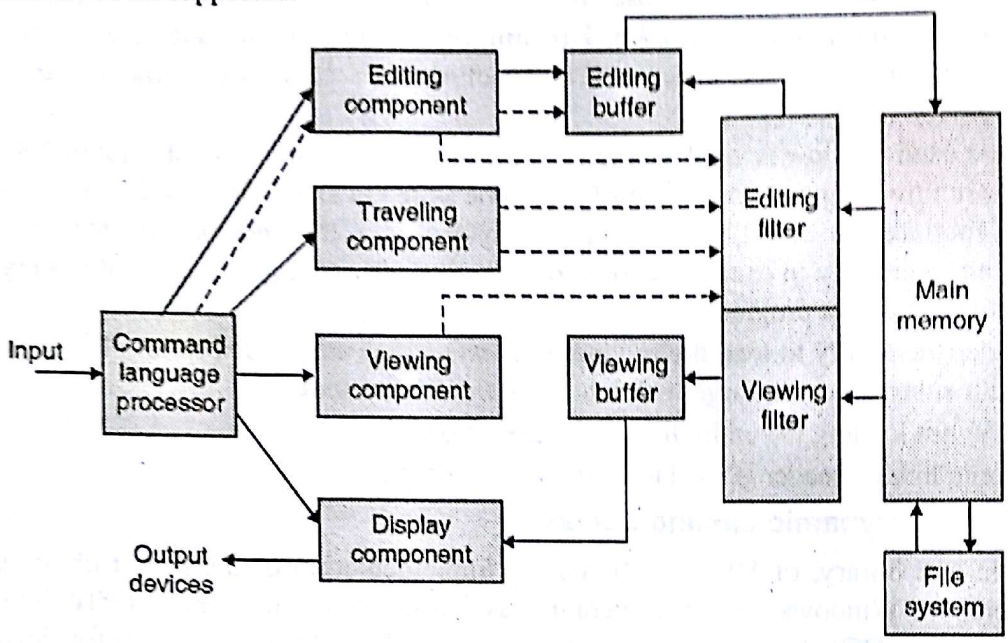


Fig. 1-Q. 3(b) : Design of an editor

Most text editors have structure similar to shown in the Fig. 1-Q. 3(b) regardless of features and the computers.

Explanation of the components used in the structure of editor are as

1. **Command Language Processor :** It accepts commands and uses semantic routines to performs various functions such as editing and viewing of text.
2. **Semantic Routine :** It involves travelling, editing, viewing and display functions, Editing operations are taken care by the user and the display operations are specified implicitly by the editor.
3. **Editing Document :** The start of the area to be edited is determined by the current editing pointer maintained by editing component. It is a collection of modules dealing with editing tasks. Current editing pointer can be set or reset due to next paragraph, next screen, cut paragraph, paste etc.
4. **Editing Command :** When editing command is issued editing component involves the editing filler and generates a new editing buffer. It contains part of the document to be edited from the current editing pointer. Filtering and editing may be interleaved with no explicit editor buffer being created.
5. **Viewing a Document :** It is a collection of modules responsible for determining the next view. The current viewing pointer determined the start of the area to be viewed. Present viewing pointer can be set or reset as a result of previous editing operation.
6. **Viewing Component :** It involves the viewing filter and generates a new viewing buffer. It contains the part of the document from the viewing buffer and also contains the part of the document to be viewed from current viewing pointer.

Scanned by CamScanner

7. **Display Component :** Display component of the editor is used to display the data. The data of the viewing buffer is passed to the display component which produces a display by mapping the buffer to a screen called a window. The editing and viewing buffers may be identical or may be completely disjoint. Identical user edits the text direct on the screen and disjoint user do the find and replace operations. Windows typically cover entire screen or a rectangular portion of it.

8. The component of the editor deal with an user document on two levels.
   (i) **Main Memory :** Instead of loading full program in the memory only part program is loaded based on the page demanding technique.
   (ii) **Disk File System :** The full program is loaded on the disk and used wherever it is needed.

   Documents are not stored sequentially as a string of characters. Editors functions in three basic types of computing environments they are :
   1. **Time Sharing :** Editor must function quickly with the computer's processor, memory and I/O devices.
   2. **Stand-alone Environment:** Stand- alone environment is used to carry out editing and viewing operations.
   3. **Distributed Environment :** In this editor functions as a combination of both stand-alone editor and time sharing environment.

Chapter 6 : Compilers [Total Marks : 10]

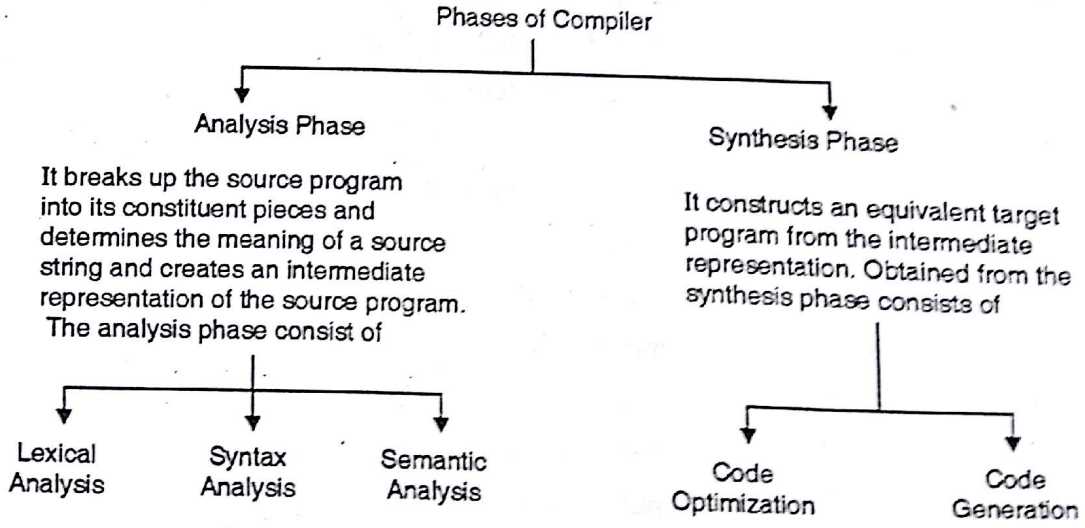
Q. 5(b) What are the different phases of Compiler? Illustrate compilers internal representation of source program for following statement after each phase. Position = initial + rate * 60

(10 Marks)

Ans. : Phases of Compiler

A Compiler operates in various phases, each of which transforms the source program from one representation to another. A phase is an independent task in the compilation process which transforms the source program from one representation to another. The fundamental language processing model for compilation consists of two steps for processing of a source program.

1. Analysis of Source Program
2. Synthesis of Source Program



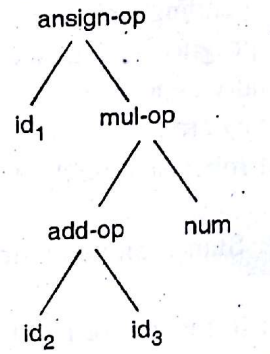
There are two other activities or phases which interact with all the other phases of the compiler at the same line.

Scanned by CamScanner

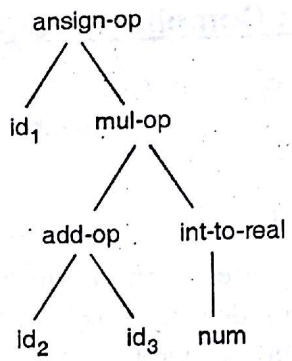
These phases are as :
Symbol Table Management
Error Detection and Handling.

Position = initial + rate * 60

⇓ Lexical analysis



⇓ Semantic analysis



⇓ Intermediate code generation

temp 1 = $id_2 + id_3$

temp 2 = int to real (num)

temp 3 = temp 1 * temp 2

id1 = temp 3

⇓ Code optimization

t1 = $id_2 + id_3$

$i\,d_1$ = t1 + num

⇓ Code generation

mov f id_3, r_2

add f id_2, r_2

mov f r_{num}, r_1

mult f r_2, r_1

mov f $r_1, i\,d_1$

f - floating point arithmetic

r_1, r_2 – two registers to store the values.

Scanned by CamScanner

Chapter 8 : Syntax Analysis [Total Marks : 30]

Q. 1(b) Eliminate Left recursion in the following grammar (Remove Direct and Indirect recursion)

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$ **(5 Marks)**

Ans. : Algorithm : Indirect left-recursion

Arrange the nonterminals in some order $A_1,\ldots,A_n$.

for (i in 1..n) {

for (j in 1..i-1) {

replace each production of the form $A_i ::= A_j g$ by the productions $A_i ::= d_1 g \mid d_2 g \mid \ldots \mid d_k g$ where

$A_j ::= d_1 \mid d_2 \mid \ldots \mid d_k$

}

eliminate the immediate left recursion among A_i productions }

For the above grammar, Let the order of nonterminals be S, A. For i=1, the rule $S \rightarrow Aa$ is through, since there is no immediate left recursion. For i=2, $A \rightarrow Sb \mid c$ is modified as, $A \rightarrow Aab \mid c$, which has immediate left recursion and hence is eliminated by modifying the rules as,

$$S \rightarrow Aa$$
$$A \rightarrow Sb|c$$

After removing immediate left recursion :

$$S \rightarrow Aa$$
$$A \rightarrow cA'$$
$$A' \rightarrow abA'|\varepsilon$$

Q. 2(a) For a given grammar below, construct an operator precedence relation matrix, assuming *, + are binary operators and id as terminal Symbol and E as non-terminal.

$E \rightarrow E + E$ $\quad E \rightarrow E * E$ $\quad E \rightarrow id$

Apply operator precedence parsing algorithm for the statement

id + id * id **(10 Marks)**

Ans. : **Please refer Q. 2(b) of May 2015.**

Q. 3(d) Find FIRST and FOLLOW Set for given grammar below

$E \rightarrow T E'$ $\quad E' \rightarrow + T E' \mid \varepsilon$

$T \rightarrow F T'$ $\quad T' \rightarrow * F T' \mid \varepsilon$

$F \rightarrow (E)$ $\quad F \rightarrow id$ **(5 Marks)**

Ans. : FIRST and Follow for the following Grammar :

$E \rightarrow TE'$

$E' \rightarrow + TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Now FIRST Set's

FIRST (F) = {(, id} $\qquad$ FIRST (TE′) = {(, id}

FIRST (T′) = {*, ε} $\qquad$ FIRST (+ TE′) = {+}

Scanned by CamScanner

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

FIRST (ε) = {ε}
= {(,id}
= {*}

FOLLOW Set's

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

Q. 4(b) For the following grammar construct LL(1) parser table.

S → F S → (S – F) F → a

And Parse the string (a – a). Show contents of stack and i/p buffer and action taken after each step.

(10 Marks)

Ans. :

S → f
S → (S – f)
f → a

construct LL(1) parser table

Step 1 : Find the first and follow set of the given expression.

FIRST (S) = First (P) = {a,(}
FIRST (f) = {a}
FOLLOW (S) = { $ }
FOLLOW (f) = { – }

LL (1) Table :

Non Terminals	Input symbols			
	a	–	$	(
S	S → f	S → (S – f)		S → (S – f)
F	F → a			

Parse the string (a – a)

Stack	Input	Output
$ S	(a – a)	S → (S – f)
$) S – f (	(a – a)	S → (S – f)
$) S – f	a – a)	f → a
$) S – a	a – a)	f → a
$) S –	– a)	
$) S	a)	S → f
$) f	a)	f → a
$) a	a)	f – a
$)	)	
$	accept	

Scanned by CamScanner

Chapter 9 : Syntax Directed Translation [Total Marks : 05]

Q. 3(c) Explain synthesized and Inherited attributes used in Syntax Directed Definition. **(5 Marks)**

Ans. : Synthesized Attributes

An attribute is synthesized if its value at a parent node can be determined from attributes of its children. In general, given a CFG production rule of the form 'A=a B g' then an associated semantic rule of the form "A.attribute=f(a.attribute, B.attribute, g.attribute)" is said to specify a synthesized attribute of A.

Synthesized attribute of the node is defined in terms of

Attribute values at children of the node.

Attribute value at node itself.

We have productions of the form : $A \rightarrow X_1 X_2 X_3 \ldots X_n$

with semantic rules of the form :b:= $f(c_1, c_2, c_3, \ldots, c_n)$

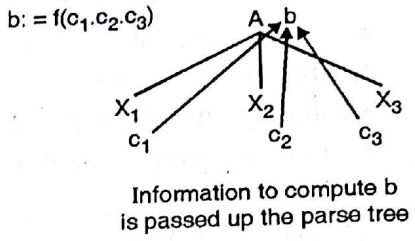


Fig. 1-Q. 3(c) : Synthesized Attributes

where b and the c's are attributes of the grammar symbols

b is called a synthesized attribute if :

b is an attribute of A (i.e. the LHS), and

The c's are all attributes of the X's (symbols on the RHS).

Synthesized attributes can be evaluated by a single bottom-up traversal of the parse tree.

Example 1

Consider the following context tree grammar for computing the synthesized attributes

		Production Rule	Semantic Actions
$E \rightarrow EN$		$E \rightarrow EN$	type $(E \cdot val)$
$E \rightarrow E + T$		$E \rightarrow E + T$	$E \cdot val = E \cdot val + T \cdot val$
$E \rightarrow T$		$E \rightarrow T$	$E \cdot val = T \cdot val$
$T \rightarrow T / F$	$\Rightarrow$	$T \rightarrow T * F$	$T \cdot val = T \cdot val / F \cdot val$
$T \rightarrow F$		$T \rightarrow F$	$T \cdot val = F \cdot val$
$F \rightarrow (E)$		$F \rightarrow (E)$	$F \cdot val = E \cdot val$
$F \rightarrow id$		$F \rightarrow id$	$F \cdot val = id \cdot lexval$
$N \rightarrow ;$		$N \rightarrow ;$	; is terminating symbol so it can be ignored by lexical analyzer

Syntax directed definitions for the above grammar.

E, T and F are non-terminals and their values can be found using the attribute "val".

"lexval" is a synthesized attribute whose value can be obtained from the lexical analyzer. This attribute is associated with token digit.

Following steps are followed to compute S-attributed definition:

1. Proper semantic rules are required to write the syntax directed definition for corresponding production rule of the given grammar.
2. The annotated parse tree is generated in bottom up fashion and attribute values are computed.
3. The value generated at the root node is supposed to be the final output.

Scanned by CamScanner

Inherited Attribute

An attribute is inherited if the attribute value of a parse-tree node is determined from attribute values of its parent and siblings. In general, given a CFG production rule of the form 'A=a B g' then an associated semantic rule of the form "*B.attribute=f(a.attribute, A.attribute, g.attribute)*" is said to specify an inherited attribute of B. The inherited attributes are passed down from parent nodes to the children nodes of the abstract syntax tree during the semantic analysis of the parsing process.

Inherited attribute of the node is defined in term of

Attributes values at parent node.
Attribute values at sibling.
Attributes value of node itself.
We have the production of the form :
$A \rightarrow X_1 X_2 X_3 \ldots X_n$
Semantic rules of the form
$b := f(c_1, c_2, c_3, \ldots, c_n)$
b is an inherited attribute if
b is an attribute of one of the X's (i.e. RHS).
The c's are the attributes of A and/or one or more of the other X's.
Which means they are beside or above where b is needed.

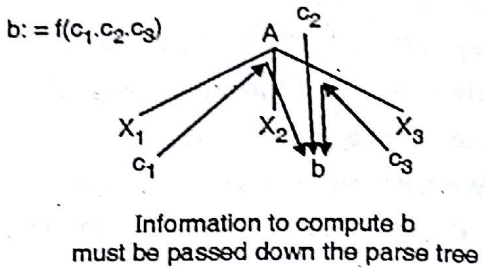


Fig. 2-Q. 3(c) : Inherited Attributes

Example 2

Below mentioned context free grammar consists of the keyword int and real, followed by a list of identifiers.

D:: = TL
T::= int | real
L::= L, id | id

Production	Semantic Rules
$D \rightarrow T\,L$	L.in = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ real	T.type = real
$L \rightarrow L1$, id	L1.in := L.in symtab(id.entry, L.in)
$L \rightarrow$ id	symtab(id.entry, L.in)

Fig. 3-Q. 3(c) : Inherited Attribute L.in.

Declaration consists of type T followed by a list of variables.
Type of T is synthesized and inherited to L :
"type" synthesized attribute.
"in" inherited attribute.

Procedure "symtab" adds type of each identifier to its entry in the symbol table pointed to by attribute entry. The Fig. 4-Q. 3(c) shows an annotated parse tree for the sentence real id_1, id_2, id_3.

Scanned by CamScanner

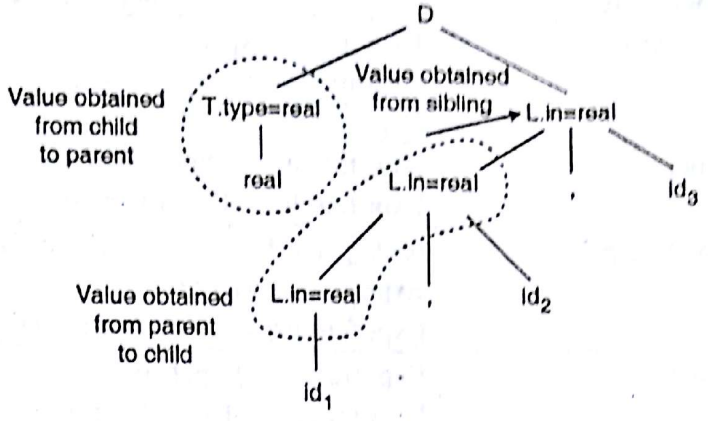


Fig. 4-Q. 3(c) : Annotated Parse tree with inherited attributed

Chapter 10 : Intermediate Code Generation [Total Marks : 10]

Q. 6(b) Explain Backpatching with an example. **(10 Marks)**

Ans. : Backpatching

The main problem in generating three address codes in a single pass is that in a single pass we may not know the labels where the control must go to at the time jump statements are created. So to get rid of this problem a sequence of branching statements with the targets of the jumps temporarily left unspecified are generated.

Each such branching statement will be put on a list of goto statements whose labels will be filled in when the proper label can be identified. This subsequent filling of addresses for the specified labels is called BACPATCHING. In other words Backpatching is the activity of filling up unspecified information of labels using appropriate semantic actions during the code generation process. For implementing Backpatching, we generate quadruples into a quadruple array. Three functions are used to manipulate the list of labels. Backpatching Algorithms perform three types of operations

make_address_list(i) – Generates a new list having only i, an index to the array of quadruples and returns pointer to the list.

merge_list(n1,n2) – It joins two list pointed by two pointers.

back_patch(n,i) – Fills i as the target label for each of the statements on the list pointed by n.

Translation of Boolean Expressions using Backpatching

Let's now try to construct the translation scheme for Boolean expression.

Let the grammar be :

Expr → Expr1 || N Expr2

Expr → Expr1 & N Expr2

Expr → ! Expr1

Expr → (Expr1)

Expr → id1 relop id2

Expr → false

Expr → true

M → ε

Scanned by CamScanner

The Translation scheme is as follows :

	Production Rule	Semantic Rule
1.	Expr → Expr1 or M E2	back_patch(Expr1.falselist, N.val) Expr.truelist=merge_list(Expr1.truelist, Expr2.truelist) Expr.falselist = Expr2.falselist
2.	Expr ----> ! Expr1	Expr.truelist = Expr1.falselist Expr.falselist = Expr1.truelist
3.	Expr→ Expr1 & N Expr2	back_patch(Expr1.truelist, N.eval) Expr.truelist = Expr2.truelist Expr.falselist = merge_list(Expr1.falselist, Expr2.falselist)
4.	Expr -----> (Expr1)	Expr.truelist = Expr1.truelist Expr.falselist = Expr1.falselist
5.	Expr----> num1 relop num1	Expr.truelist = make_address_list(nextval) Expr.falselist = make_address_list(nextval +1) generate('if'num1.place relop num2.place 'goto __') generate('goto ___')
6.	Expr----> true	Expr.truelist = make_address_list(nextval) generate('goto ___')
7.	Expr ---> false	Expr.falselist = make_address_list(nextval) generate('goto ___')
8.	N -----> epsilon	N.val = nextval

Fig. 1-Q. 6(b) : Translation schema for Boolean Expression using Backpatching

As we see in Fig. 1-Q. 6(b) we have insert a marker non-terminal N into the grammar to cause a semantic action to pick up, at appropriate times the index of the next value to be generated.

This is done by the semantic action :

{N.val = nextval} for the rule

N → epsilon

Truelist and falselist are two synthesized attributes which are used to generate jumping code for boolean expressions.

Expr. truelist : Contains the list of all the jump statements left incomplete to be filled by the label for the start of the code for Expr → true.

Expr. falselist : Contains the list of all the jump statements left incomplete to be filled by the label for the start of the code for Expr → false.

The variable nextval holds the key of the next value to follow by. For the expression Expr → Expr1 and N Expr2 the value will be backpatched onto $Expr_1$.truelist where it has the address of the first statement of $Expr_2$.code. For the expression Expr → Expr1 or N Expr2 the value will be backpatched onto $Expr_1$.falselist where it has the address of the first statement of $Expr_2$.code.

Translation of Control Flow Constructs using Backpatching

Now we try to construct a translation scheme for control flow constructs like if-then-else and while-do statements. Most programming languages have a common set of statements that define the control flow of a program written in that language.

Scanned by CamScanner

These statements are :

1. **Assignment**

   A single statement is assigning some expression to a variable. After executing this statement, the control flows to the immediate next statement in the sequence.

2. **if<condition>then<then-part>else<else-part>**

   Control flows to either the then-part or the else-part depending on whether the condition is true or false.

3. **while<condition>do <loop>**

   Control remains within the loop until the condition becomes false.

4. **<block of statements>**

   A group of statements put within a begin ... end block marker.

The grammar for these statements is given by :

```
S  → if B then S
     | if B then S else S
     | while B do S
     | begin L end
     | A /*assignment*/
L  →   L S
     | S
```

To aid the generation of the code, two new marker nonterminals M and N are introduced. The modified grammar becomes :

```
S  →if B then M S
     | if B then M S N else M S
     | while M B do M S
     | begin L end
     | A /*assignment*/
L →    L M S
     | S
M→     'epsilon'
N →    'epsilon'
```

B is a Boolean expression.

Attributes are defined as follows :

S.nextlist : List of quadruples with jumps to the quadruple succeeding S.

L.nextlist : The same, where L is a group of statements.

The nonterminal N iss used as it permits the generation of a jump after the 'then' in if-then-else. The atribute N.nextlist holds the quadruple number for this statement.

Actions

```
S  → if B then M S1
     {
     back_patch (B.truelist,M.val)
     S.nextlist = merge_list(B.falselist,S1.nextlist)
     }
S  → if B then M1S1N else M2S2
     {
```

Scanned by CamScanner

```
        back_patch (B.truelist,M1.val)
        back_patch (B.falselist,M2.val)
        S.nextlist = merge_list(S1 .nextlist,merge_list(N.nextlist,S2.nextlist))
        }
S  → while M1B do M2S1
        {
        back_patch(S1.nextlist,M1.val)
        back_patch (B.truelist,M2.val)
        S.nextlist = B.falselist
        generate('goto' M1.val)
        }
S  → begin L end
        {
        S.nextlist = L.nextlist
        }
S  → A
        {
        S.nextlist = <nil>
        }
L  → L1 M S
        {
        back_patch(L1.nextlist,M.val)
        L.nextlist = S.nextlist
        }
L  → S
        {
        L.nextlist = S.nextlist
        }
```

Chapter 12 : Code Optimization [Total Marks : 10]

Q. 2(b) Explain the role of code optimization in compiler designing? Explain Peephole optimization along with an example. **(10 Marks)**

Ans. :

Role of code optimization :

Optimization is the procedure of converting a piece of code into more efficient code in terms of time and space complexity. The transformed optimized code should have the following characteristics in it :

The output of the program should remain same without having any side-effects. The optimized code must execute faster and it should consume less memory.

Code Improvement (or Optimization)

The basic aim of code improvement is as :

It analyses IR and rewrites (or transforms) IR. Its primary goal is to reduce running time of the compiled code. It may also improve space, power consumption. It must preserve "meaning" of the code. It measures the name by the values.

Scanned by CamScanner

Peephole Optimization :

Peephole optimizations examine at most a few instructions, transforming instructions into other less expensive ones, such as turning a multiplication of x by two into an addition of x with itself. It works on a small set of instructions. The small set is called a "peephole" or a "window". It works by recognizing sets of instructions that don't actually do anything, or that can be replaced by a leaner set of instructions.

Peephole optimization is a pass that operates on the target assembly and only considers a few instructions at a time (through a "peephole") and attempts to do simple, machine dependent code improvements. For example, peephole optimizations might include elimination of multiplication by 1, elimination to load a value into a register when the previous instruction stored that value from the register to a memory location, or replacing a sequence of instructions by a single instruction with the same effect. Because of its myopic view, a peephole optimizer does not have the potential payoff of a full-scale optimizer, but it can significantly improve code at a very local level and can be useful for cleaning up the final code that resulted from more complex optimizations.

Common techniques used in peephole optimizations are as :

Constant folding : It finds out the constant sub-expressions in advance.
Strength reduction : Slow operations are replaced with faster equivalents.
Null sequences : Useless operations are removed
Combine Operations : Similar operations are replaced with one equivalent.
Algebraic Laws : Algebraic laws are used to simplify or to reorder instructions.

Chapter 13 : Run Time Storage [Total Marks : 05]

Q. 1(c) What is an activation record? Draw diagram of General Activation record and explain the purpose of different fields of an activation record. **(5 Marks)**

Ans. : Activation Records

An *activation record*, is a chunk of computer memory which holds the arguments and ``normal'' local variables of the function. An **activation record** for a procedure has fields to hold parameters, results, machine-status information, local data and temporaries. We keep track of below mentioned address statements during procedure call. They are as :

Call for the routine
Space for the local variables of the routine,
The return address back to the routine's caller, and
The parameter values passed into the routine.
Halt
Action

Activation record or frame is used to store the information needed by a single execution of a procedure. It consists the collection of the fields shown in the Fig. 1-Q. 1(c).

When we call a function F, we push a new activation record on the runtime stack, which is particular to the function F. The activation record is used to store the information required by a single procedure call.

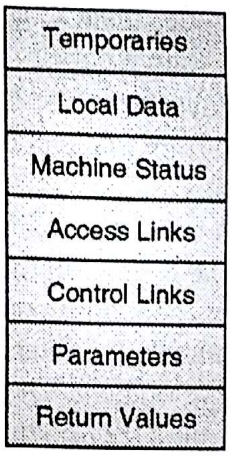


Fig. 1-Q. 1(c) : A General Activation Record

Scanned by CamScanner

When the callee function returns to the caller, the activation record of the callee is popped out. For example, if the main program calls function P, P calls E, and E calls P, then at the time of the second call to P, there will be four activation records in the stack in the following order : ***main,P,E,P.***

The two points to be remembered while executing a function call are as

The first that we should be able to pop-out the current activation record of the callee. Activation record of caller should restore. This can be done with the help of a pointer in the current activation record, called the **dynamic link**. Dynamic link links all the activation records together. This is called a **dynamic chain**. If we need to access the variables stored in previous activation records in the stack, this is done with the help of the **static link** form a **static chain.** Not all the fields shown in the Fig. 1-Q. 1(c) may be needed for all languages. The record structure can be modified as per the language/compiler requirements. The access and control links are optional. Also, actual parameters and return values are often stored in registers instead of the activation record, for greater efficiency and the activation record for a procedure call is generated by the compiler. The strategies used for storage allocation in such cases will be discussed in the next section. The purpose of the fields of an activation record is as follows, starting from the field for temporaries.

Temporaries : Expression is evaluated in it.

Local Data : It stores the local data.

Saved Machine Status : It holds the information about machine status before the procedure call.

Access Link : In this non local data is accessed.

Control Link : It points to the activation record of the caller.

Actual Parameters : it holds the value of actual parameters.

Returned Value : Value to be returned is stored in this.

An activation record is created based on three actions such as:

Function Definition

Variable Definition

Function Application (call)

Chapter 14 : Compiler-Compilers [Total Marks : 05]

Q. 3(a) Write a note on JAVA compiler environment. **(5 Marks)**

Ans. : Java Compiler Environment : JavaCC is similar to YACC compiler and it generates a parser for the EBNF (Extended Backus Naur form) notation grammar and accepts the output in JAVA source code. JavaCC generates top-down parsers, which limits it to the LL(k) class of grammar and it is an open source parser.

Java Compiler is the most popular parser i.e. it reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. It is used with most Java applications. JavaCC run on any platforms, we think this is a great testimonial to the "***Write Once Run Anywhere***" aspect of the Java programming language. It has been tested on various platforms. A Java compiler is a compiler for the Java programming language. The most common form of output from a Java compiler are Java class files containing platform-neutral Java bytecode. The Java compiler, converts Java source code into Java bytecode (an intermediate language for the Java Virtual Machine (JVM)), is provided as part of the Java Development Kit (JDK). The Java Runtime Environment (JRE), complementing the JVM with a just-in-time (JIT) compiler, converts intermediate bytecode into native machine code on the fly. Byte code executes faster than the interpretation of the source code, because the key portion of the analysis and verification of the source code is done during the compilation step. Byte code is machine-independent as machine-dependent part is implemented in the virtual machine.

Scanned by CamScanner

JAVA Virtual Machine

Java uses a combination of compilation and interpretation. Programs written in Java are compiled into machine language, but it is a machine language for a computer that does not really exist. This so called "virtual" computer is known as the 'Java Virtual Machine'. Java Virtual Machine or JVM as its name suggest is a "virtual" computer that resides in the "real" computer as a software process. JVM gives Java the flexibility of platform independence. Java code once written is saved with .java file extension. While compiling it generates the .class file and it is also known as **"byte code"**. The byte code is given as an input to Java Virtual Machine. JVM read this code and interpret it and executes the program.

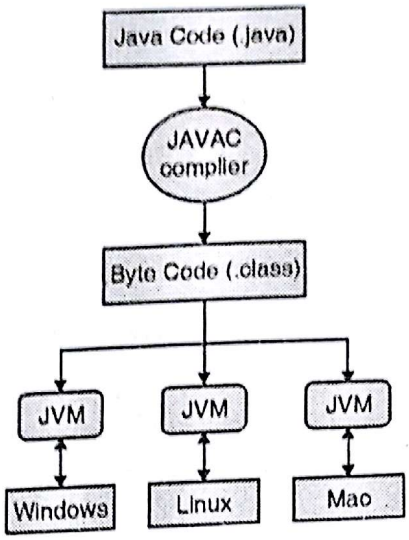


Fig. 1-Q. 3(a) : Compilation and Execution of Java program

JVM uses different components like stack, the garbage-collected heap, the registers and the method area as shown in Fig. 2-Q. 3(a) to execute the program and generates output.

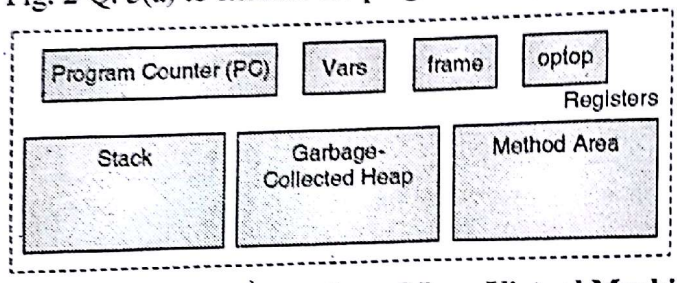


Fig. 2-Q. 3(a) : Representation of Java Virtual Machine

1. **The Stack**

   Stack in Java Virtual Machine is used to store various method arguments, local variables of any method and registers for stack manipulation. Stack keeps track of each and every method invocation. There are three registers vars, frame, optop which is used to point to different parts of the current stack.

   Java stack frame is divided into three parts :

   a. **Local Variables :** It comprises of all the local variables being used by the current method. It is pointed by the **vars** register.

   b. **Execution Environment: The** operations of the stack is maintained in this. It is pointed to by the **frame** register.

   c. **Operand Stack :** It is used as a work space by bytecode instructions. The top of the operand stack is pointed to by the **optop** register.

2. **Method Area**

   Bytecodes resides in the method area. It has a program counter which keeps track of the current instruction which is being executed. After execution of an instruction, the JVM sets the PC to next instruction. Method area is shared among all the threads of a process. Synchronization is needed if more than one threads are accessing any specific method or any instructions. Synchronization in JVM is achieved through Monitors.

3. **Garbage Collected Heap**

   The garbage collected Heap is where the objects in Java programs are stored.

❑❑❑

Scanned by CamScanner

May 2016

Chapter 1 : System Software [Total Marks : 05]

Q. 1 (d) What is the difference between compiler and interpreter. **(5 Marks)**

Ans. : Difference between Interpreter and Compiler

Sr. No.	Interpreter	Compiler
1.	It translates the high level language into intermediate form line-by line.	It compiles the whole program at one time directly into machine language.
2.	Interpreters are slow in comparison to compiler.	Compiled programs run faster than interpreted program.
3.	Less compact.	More compact.
4.	Interpreted code is good for simple applications.	Compiled code is good for complex application.
5.	The advantage of line by line interpretation is that, it can interrupt in between can change the code and start again.	Translation of the whole program is done at a one time resulting in faster execution of program.
6.	The disadvantage is that every line has to be translated every time it is executed.	The disadvantages are that we cannot change the program without going back to the original source code.
7.	e.g. of Interpreters are JAVA SCRPIT, LISP, FORTH, BASIC	e.g. of compilers are VISUAL BASIC, C, C++, C#, COBOL etc.

Chapter 2 : Assemblers [Total Marks : 10]

Q. 6(a) With reference to Assembler explain following tables with suitable examples.
(i) POT (ii) MOT (iii) ST (iv) LT **(10 Marks)**

Ans. : **Please refer Q. 6(a) of Dec. 2015.**

Chapter 4 : Loaders and Linkers [Total Marks : 10]

Q. 2(b) What is the difference between Dynamic Loading and Dynamic Linking explain with an example. **(10 Marks)**

Ans. : Difference between Dynamic Loading and Dynamic Linking

Sr. No	Dynamic Loading	Dynamic Linking
1.	It refers to mapping (or less often copying) an executable or library into a process's memory after is has started.	Dynamic linking refers to resolving symbols - associating their names with addresses or offsets - after compile time.
2.	It is similar to plugins that is an exe can actually execute before the dynamic loading happens.	The linker while creating the exe does minimal work. For the dynamic linker to work it actually has to load the libraries.

Scanned by CamScanner

Sr. No	Dynamic Loading	Dynamic Linking
3.	Dynamic loading is concerned with loading the subroutines of a program as and when required during run-time, instead of loading all the subroutines at once before the program execution starts.	Dynamic linking is concerned with linking library routines at run-time instead of combining them with the executable program code before the program execution starts i.e. static linking. To achieve this, a small code call 'stub' is inserted in the program code wherever library routine is required. The stub contains information about where to find the routine if it is not already in the main memory.
4.	Efficient use of memory.	Efficient use of memory.
5.	It does not require special support from Operating system, it is the responsibility of the programmer to check whether the routine that is to be loaded does not exist in main memory.	Dynamic Linking requires special support from operating system, the routine loaded through dynamic linking can be shared across various processes.

Chapter 5 : Software Development Tools [Total Marks : 05]

Q. 3(b) Write a brief note on Design of an Editor. **(5 Marks)**

Ans. : **Please refer Q. 3(b) of Dec. 2015**

Chapter 6 : Compilers [Total Marks : 10]

Q. 5(b) What are the different phases of compiler ? Illustrate compilers internal representation of source program or following statement after each phase.Amount = P + P * N * R / 100

(10 Marks)

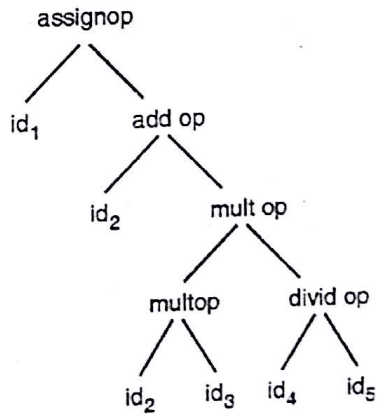
Ans. : **Phases of Compiler : Please refer Q. 5(b) of Dec. 2015.**

Amount = P + P * N * R | 100

⇓ **Lexical Analysis**

id, assign-op id_2 addop id_2 multop id_3 multop id_y divide op num

⇓ **Syntax Analysis**



Scanned by CamScanner

⇓ **semantic analysis**

Convert all the variables into one form either into int or float by type caste int to float or float to int

⇓ **Intermediate code Generation.**

t0 = P
t1 = P * N
t2 = R | 100
t3 = t1 * t2
t4 = t 0 + t3
Amount = t4

⇓ code optimization

t0 = P
t1 = t0 * N
t2 = R / 100
t3 = t1 * t2
Amount = t 0 + t3

⇓ code generation

R1 ← P
R2 ← N
R3 ← R
R4 ← R1 * R2
R5 ← R| 100
R6 ← R4* R5
R7 ← R1 + R6

Chapter 7 : Lexical Analysis [Total Marks : 5]

Q. 1(a) What is the role of an automata in compiler design. **(5 Marks)**

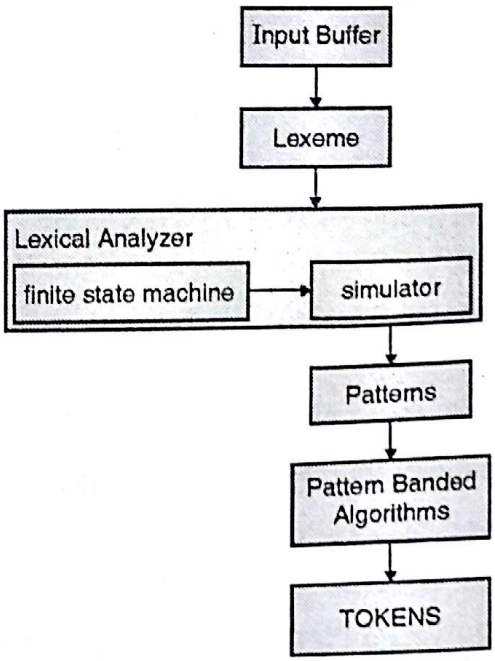
Ans. :

Role of an Automata in Compiler Design

Lexical analysis is the process where the stream of characters making up the source program are scanned from left-to-right and grouped together into tokens. Thus, lexical analyzer stores the input in a buffer. It builds the regular expressions for corresponding tokens. From these regular expressions, finite automata's are build. A finite automata is supposed to run on some given sequence or string of inputs in discrete time steps. An automaton contains a finite set of stales. At each instance in time of some run automaton is in one of its states. The automaton reads the input word one symbol after another in the sequence and transits from state to state according to the transition function, until the word is read completely. When lexeme matches with the pattern generated by finite automata, the specific token get recognized.

Scanned by CamScanner

The schematic diagram far this process is shown below :



Chapter 8 : Syntax Analysis [Total Marks : 10]

Q. 1(b) Eliminate Left recursion in the following grammar (Remove Direct and Indirect recursion)

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$ **(5 Marks)**

Ans. : **Please refer Q. 1(b) of Dec. 2015.**

Q. 3(d) Find FIRST and FOLLOW set for given grammar below

$E \rightarrow TE'$ $E' \rightarrow + TE' \mid \varepsilon$

$E \rightarrow FT'$ $T' \rightarrow * FT' \mid \varepsilon$

$E \rightarrow (E)$ $F \rightarrow id$ **(5 Marks)**

Ans. : **Please refer Q. 3(d) of Dec. 2015.**

Chapter 9 : Syntax Directed Translation [Total Marks : 05]

Q. 3(c) Explain synthesized and Inherited attributed used in Syntax Directed Definition. **(5 Marks)**

Ans. : **Please refer Q. 3(c) of Dec. 2015.**

Chapter 10 : Intermediate Code Generation [Total Marks : 10]

Q. 2(a) Explain with an example Quadruples, Triples, Indirect triples. **(10 Marks)**

Ans. : **Please refer Q. 3(b) of May 2015.**

Chapter 11 : Code Generation [Total Marks : 20]

Q. 5(a) Draw and explain DAG and represent the following example with it.

(a / b) + (a / b) * (c + d) **(10 Marks)**

Scanned by CamScanner

Ans. : DAG is formed by grouping vertices and directed edges. DAG is a directed graph with no directed cycles. In DAG each edge is used to connect one vertex to another. In DAG there is no way to start at some vertex p and follow a sequence of edges that eventually loops back to p again. DAG representation of (a | b) + (a | b) * (c + d)

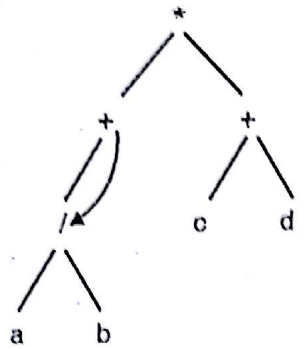


Fig. 1-Q. 5(a) : DAG Representation

Q. 6(b) What are the different issues in design of Code Generator ? Explain with an example.

(10 Marks)

Ans. : Please refer Q. 5(a) of May 2015.

Chapter 12 : Code Optimization [Total Marks : 10]

Q. 4(a) Explain different Code Optimization techniques along with an example. **(10 Marks)**

Ans. : The major principle source of optimization are as :

1. Function Preserving Transformations

These transformation improves the code without changing the function. They are majorly four type of transformation they are as :

1. Compile Time Evaluation :

In this technique expressions whose values can be pre-computed at the compilation time and used, perhaps repeatedly, in the execution of the program are identified. Then the computations of such expressions are used during compilation rather than during execution. There are two possible ways in which compile time evaluation can be carried out :

i. Constant folding ii. Constant propagation

2. Common Sub-Expression Elimination :

Two operations are *common* if they produce the same result. In such case, it is likely more efficient to compute the result once and reference it to the second time rather than re-evaluate it. An expression is *alive* if the operands used to compute the expression have not been changed. An expression that is no longer alive is *dead*. Identify common sub-expression present in different expression, compute once, and use the result in all the places. The *definition* of the variables involved in sub-expression elimination should not change.

Example

```
a := b * c                    temp := b * c
...                           a  := temp
...                           ...
x := b * c + 5                x := temp + 5
```


Scanned by CamScanner

3. **Dead Code Elimination :**

Dead Code is a portion of the program which will not be executed in any path of the program. If an instruction's result is never used in the program, the instruction is considered to be "dead" and can be removed from the instruction stream. So if we have

tmp1 = tmp2 + tmp3;

If **tmp1** is never used again, we can eliminate this instruction altogether. However, we need to be careful and check whether tmp1 holds the result of a function call :

tmp1 = swap( );

Even if **tmp1** is never used again, we cannot eliminate the instruction because we can't be sure that called function has no side-effects. Dead code can occur in the original source program but is more likely.to have resulted from some of the optimization techniques run previously.

4. **Copy Propagation :**

This optimization is similar to constant propagation, but generalized to non-constant values. Copy propagation is the process of removing them and replacing the copies generate by the compilers in an intermediate form. It often reveals dead-code.

2. **Loop Optimizations**

Loops plays a very important role in code optimizations. Since the body of the loop is executed again and again, and moreover the loops are nested so small changes in the execution of the loop body can results in a remarkable improvement in the performance of the program. Loop optimization can be viewed as the application of a sequence of specific **loop transformations** (listed below) to the source code or intermediate representation, with each transformation having an associated test for legality. A transformation (or sequence of transformations) generally must preserve the temporal sequence of all dependencies if it is to preserve the result of the program (i.e., be a legal transformation). It is most important set of optimizations in this programs are likely to spend more time in loops. The commonly used loop optimization techniques are :

i. Code Motion
ii. Strength Reduction
iii Induction Variables and Strength Reduction
iv. Loop Induction Variable
v. Loop Fission
vi. Loop Fusion or Loop Combining
vii. Loop Interchange
viii. Loop Unrolling
ix. Loop Splitting

3. **Optimization for Basic Blocks**

A basic block is a linear piece of code in which flow of control moves in at the beginning and exit at the end without halt or possibility of branching except at the end. The analysis and optimization of the basic blocks are easier.

The following sequence of three-address statements forms a basic block :

t1 := a*a
t2 := a*b
t3 := 2*t2
t4 := t1+t3
t5 := b*b

Scanned by CamScanner

t6 := t4+t5

A name in a basic block is said to live at a given point if its value is used after that point in the program, perhaps in another basic block. Optimization in the basic blocks are done in four ways they are as :

1. Common Sub-Expression Elimination.
2. Eliminate Dead Code.
3. Algebraic Optimizations.

4. **Reducible Flow Graphs**

A flow graph is reducible when the edges can be partitioned into forward edges and back edges. The forward edges must form an acyclic graph in which every node can be reached from the initial node. Structured control-flow statements such as if-then-else, while and break produces reducible control-flow. Irreducible flow graphs cannot be optimized as they create loops. With the help of the duplication code irreducible flow graphs can always be converted into reducible flow graph.

5. **Global Optimizations**

The scope of local optimization is limited to short code sequences, whereas the scope of global optimization is a program unit like a procedure or a function body including entire loops. This required considerable analysis of the program structure, but its gains are much higher than those of local optimizations. In order to perform a global optimization, the program is to be represented in the form of a program flow graph. Two types of analysis are performed on the program flow graph. Control flow analysis and data flow analysis.

6. **Machine Dependent / Machine Independent Optimization**

The code optimization phase attempts to improve the intermediate code so that faster running machine code will result. **An optimization is classified as machine dependent or machine independent.**

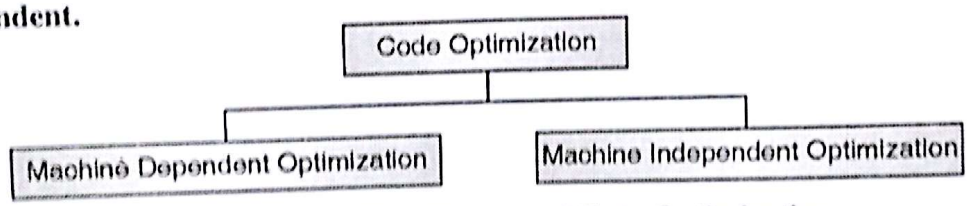


Fig. 1-Q. 4(a) : Classification of Code Optimization

Chapter 13 : Run Time Storage [Total Marks : 05]

Q. 1(c) What is an activation record ? Draw diagram of General Activation record and explain the purpose of different fields of an activation record. **(5 Marks)**

Ans. : **Please refer Q. 1(c) of Dec. 2015.**

Chapter 14 : Compiler-Compilers [Total Marks : 05]

Q. 3(a) Write a note on JAVA compiler environment. **(5 Marks)**

Ans. : **Please refer Q. 3(a) of Dec. 2015.**

□□□

Scanned by CamScanner

Dec. 2016

Chapter 1 : System Software [Total Marks : 10]

Q. 4(b) What is Loader ? Explain functions of loader with examples. **(10 Marks)**

Ans. :

For function of loader : Please refer Q. 1(c) of May 2015.

Loader : A loader is a program that accepts the object program and prepares these programs for execution by loading them into the memory.

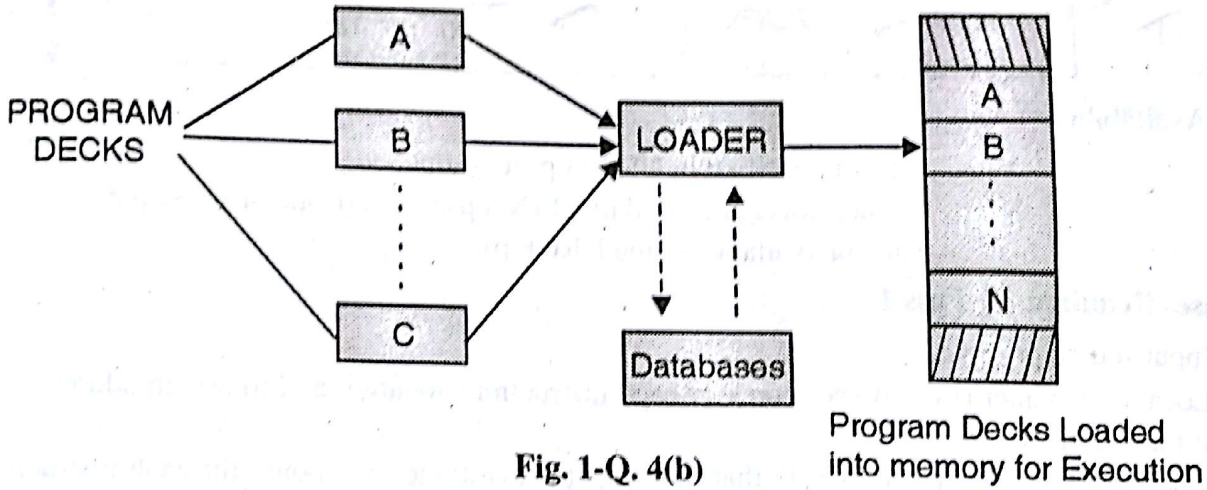


Fig. 1-Q. 4(b)

In general Programmers usually write a program which has to be loaded at some predefined location in main memory. But the loading addresses specified by programmer are not coordinated with operating system. Loader coordinates these addresses with operating system and initial loading address for .exe file of corresponding program.

Chapter 2 : Assemblers [Total Marks : 08]

Q. 3(b) What are various databases used in two pass assembler design. Explain with Example. **(8 Marks)**

Ans. :

Various databases used in two pass assembler design are as follows :

Machine Opcode Table (MOT) : Please refer Q. 6(a) of Dec. 2015.

Pseudo Opcode Table (POT) : Please refer Q. 6(a) of Dec. 2015.

Symbol Table (ST) : Please refer Q. 6(a) of Dec. 2015.

Literal Table (LT) : Please refer Q. 6(a) of Dec. 2015.

Base Table (BT)

It is used by the assembler to generate the proper base register reference in machine instructions and its correct offsets.

Scanned by CamScanner

Table 4-Q.3(b) : Base Table (BT) for pass 2

4-bytes per entry

	Availability indicator (1-byte) (character)	Designated relative-address Contents of base register (3-bytes = 24 bit address) (hexadecimal)
1	"N"	–
2	"N"	–
:	:	
14	"N"	–
15	"Y"	00 00 00

15 entries

Availability

Y ~ register specified in USING pseudo-opcode
N ~ register never specified in USING pseudo-opcode or subsequent made unavailable by the DROP pseudo-opcode

Databases Required for Pass 1

1. Input source program.
2. Location Counter (LC) to keep track of each instructions location and to assign address for each symbol defined.
3. Machine Operation Table (MOT) that indicates the symbolic mnemonic for each instruction and its length.
4. Pseudo Operation Table (POT) that indicates the pseudo-opcode and the action to be taken in pass 1.
5. Symbol Table (ST) that is used to store each label and its corresponding value.
6. Literal Table (LT) that is used to store each literal encountered and its corresponding assigned location.
7. A copy of the input to be used later by pass 2.

Databases Required for Pass 2

1. Copy of source program input to pass 1.
2. Location Counter (LC)
3. MOT that indicates for each instruction
   (a) Symbolic mnemonic (b) length
   (c) Binary machine op-code (d) format (RR, RX, ...)
4. POT that indicates for each pseudo-opcode the symbolic mnemonic and the action to be taken in pass 2.
5. Symbol Table (ST) prepared by pass 1, containing each label and its corresponding value.
6. Literal Table (LT) prepared by pass 1, containing each literal encountered and its assigned location.
7. Base Table (BT) that indicated which registers are specified as base registers by USING pseudo-opcodes and what the specified contents of these registers are.
8. INSTRUCTION workspace which is used to hold each instruction as its various parts is being assembled together.

Scanned by CamScanner

9. PRINT LINE workspace used to produce printed listing.
10. PUNCH CARD workspace used to punch the assembled instructions in the format needed by the loader.
11. An output deck of assembled instructions in the format needed by the loader.

Chapter 3 : Macro & Macro Processors [Total Marks : 10]

Q. 2(a) Explain two pass macro processor with flowchart and databases. **(10 Marks)**

Ans. :

Databases used in two pass macro processor are as follows :

1. **Argument List Array (ALA)**

ALA is used in both pass 1 and 2. Dummy arguments are substituted with the positional indicators in ALA during pass 1. Also the dummy arguments on the memory are represented by #. Where # is used by the macros. For example consider the Fig. 1-Q. 2(a) The store macro definition would be

```
LOOP1    LAB      &A1, &A2, &A3
& STRG      D      3, #3
# 0         D      2, #2
            D      1, #1
          MEND
            :
```

Fig. 1-Q. 2(a) : MACRO DEFINITION TABLE FOR PASS 1

The index markers # are replaced with macro call arguments in PASS2.

LOOP1 LAB DATA1, DATA2, DATA3

Generated ALA is

Argument list Array	
Index	Arguments (2 bytes)
0	"XYZ bbb"
1	"RECORD1 bbb"
2	"RECORD2 bbb"
3	"RECORD 3 bbb"

b – blank character

ALA during PASS 2

2. **Macro Definition Table (MDT)**

The MDT consists of 80 bytes string per entry. MDT stores each line of every macro definitions except the word MACRO. The MEND indicates the end of the MACRO.

Scanned by CamScanner

Macro Definition Table (80 bytes per entry)			
Index			Card
:			:
10	& STRG	LAB	&A3, &A1, &A2
11	#0	A	3, #3
12		A	1, #1
13		A	2, #2
14		MEND	
:			:

Macro Definition Table

3. **Macro Name Table (MNT)**

MDT table indicates the beginning of the macro definition consists of a character string and a pointer to the entry in the MDT. MNT is almost similar to MOT and POT

	8 bytes	4 bytes
Index	Name	MDT Index
:	:	:
4	"LAB bbb"	10
:	:	:

Macro Name Table

DATABASES Required for PASS 1

1. Macro source code for input.
2. Output of source code is used by pass 2.
3. Macro definitions are stored in MDT.
4. Names of defined macros are store in MNT.
5. The next entry in MDT is indicated by the Macro Definition Table Counter (MDTC).
6. The next entry in MNT is indicated by the Macro Name Table Counter (MNTC.
7. To replace the index marker entry for dummy arguments before storing in MNT is stored in ALA.

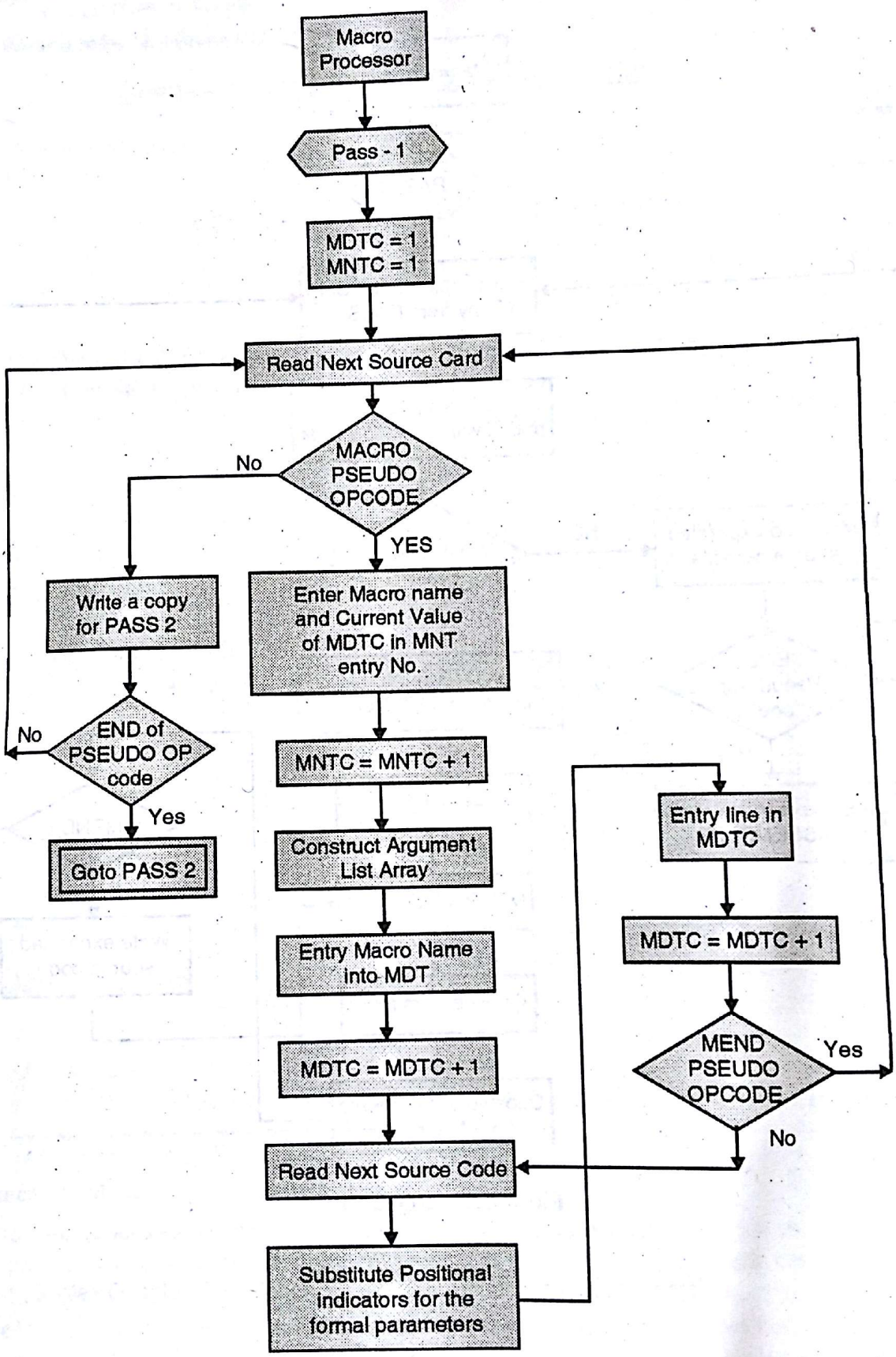
DATABASES Required for PASS 2

1. Output from PASS1 will be input to PASS2.
2. Output from PASS2 is used by assembler as input.
3. PASS2 uses the MDT and MNT made by PASS1.
4. The next line of data to be used during macro expansion is given by the Macro Definition Table Pointer (MDTP.
5. To replace the macro call arguments for the index markers in the stored macro definitions are given by ALA.

Scanned by CamScanner

Flowchart of two pass macro processor Is as follow

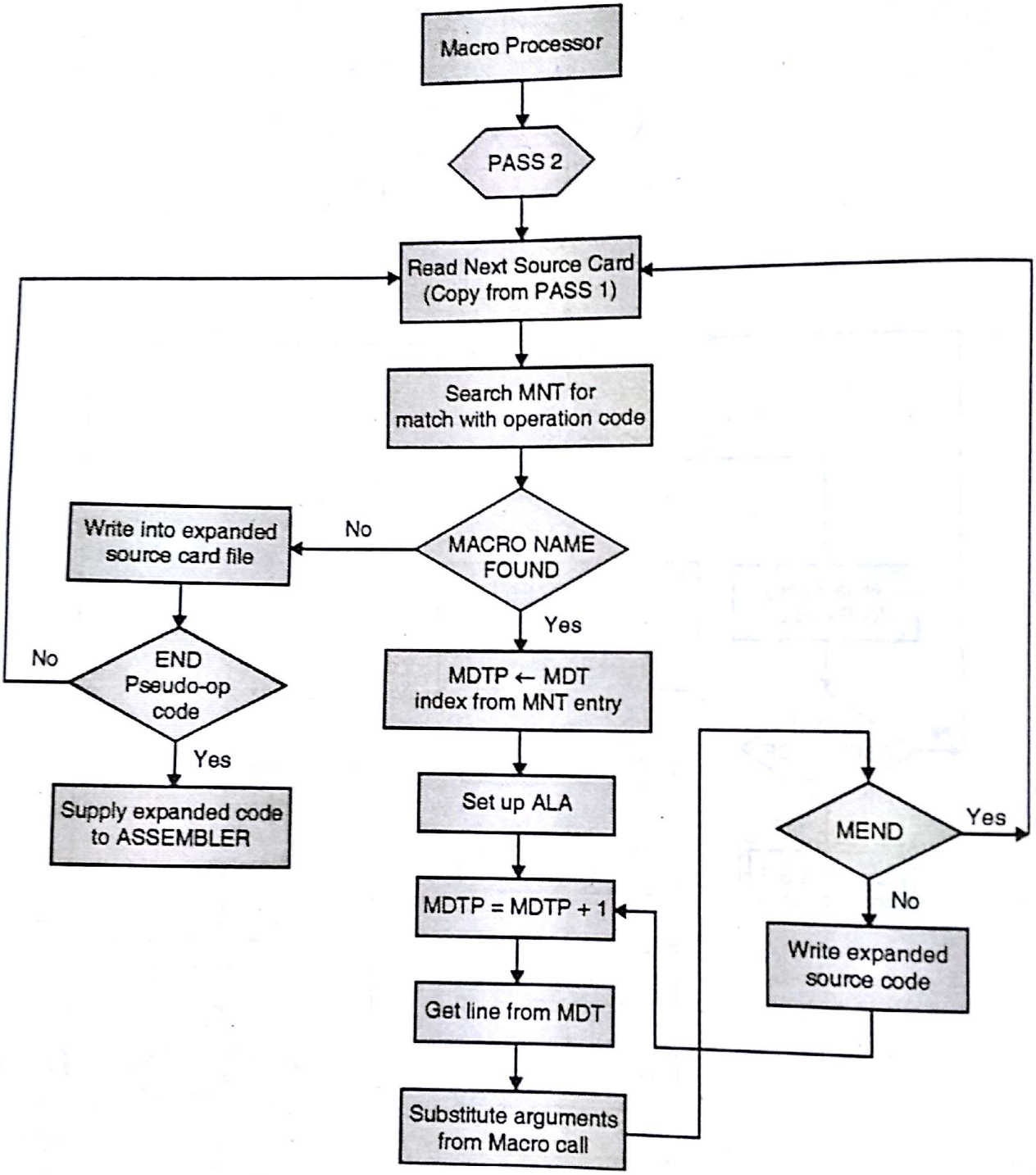
Pass1 flowchart



Flowchart 1-Q. 2(a)

Scanned by CamScanner

Pass 2 flowchart :



Flowchart 2-Q. 2(a)

Scanned by CamScanner

Chapter 5 : Software Development Tools [Total Marks : 05]

Q. 6(b) Write short note on Design of an editor **(5 Marks)**

Ans. : **Please refer Q. 3(b) of Dec. 2015.**

Chapter 7 : Lexical Analysis [Total Marks : 05]

Q. 1(b) What is role of finite automata in compiler theory ? **(5 Marks)**

Ans. : **Please refer Q. 1(a) of May 2016.**

Chapter 8 : Syntax Analysis [Total Marks : 32]

Q. 1(a) What is Handle pruning? **(5 Marks)**

Ans. :

Handle Pruning : The process of discovering a handle & reducing it to the appropriate left-hand side is called handle pruning. Handle pruning forms the basis for a bottom-up parsing method.

A handle is a right sentential sequence which is the rightmost derivation within the string being parsed.

For Example

Consider the grammar mentioned in

$E \rightarrow E+T \mid T$

$T \rightarrow T+F \mid F$

$F \rightarrow (E) \mid id$

and the input string is "id1+id2+id3"

Right-Most Sentential Form	Reducing Production
id+id+id	$F \rightarrow id$
$\underline{F}$+id+id	$T \rightarrow F$
$\underline{T}$+id+id	$E \rightarrow T$
E+$\underline{id}$+id	$F \rightarrow id$
E+$\underline{F}$+id	$T \rightarrow F$
E+T+$\underline{id}$	$F \rightarrow id$
E+$\underline{T+F}$	$T \rightarrow T*F$
$\underline{E+T}$	$E \rightarrow E+T$
E	

Handles are underlined in the right-sentential forms.

Here it can be noted that string appearing to the right of a handle contains only terminal symbols.

Q. 6(d) Write short note on Recursive Descent parsing **(5 Marks)**

Ans. : Recursive Descent parsing :

A recursive descent parser is a top-down method of syntax analysis which uses a collection of recursive procedures to process the input. A procedure is associated with each non-terminal of the grammar. In recursive descent parsing :

We backtrack for other production rules if a choice of production rule does not work.

It tries to find the left most derivations and it is a general parsing technique but not widely used.

Scanned by CamScanner

Example :

Consider the grammar

(1) $E \rightarrow T + E \mid T$ (2) $T \rightarrow int \mid int * T \mid (E)$

Input string is "$int_5 * int_2$"

Start with top-level non-terminal E

(i) Using given production rule we get

$E_0 \rightarrow T_1 + E_2$ (using (1))

(using (2))

But It does not match. Input token int_5

(ii) Try $T_1 \rightarrow int$ – (Token matches)

But + after T_1 does not match input token +

(iii) Try another rule

$T_1 \rightarrow int * T_2$ (using rule (2))

This will match but + after T_1 will be unmatched.

$E_0 \rightarrow int * T_2 + E_2$

The above are the exhausted choices for T, we will backtrack again to have the right solution for input string "$int_5 * int_2$"

$E_0 \rightarrow T_1$

Follow same steps as before for T_1

$T_1 \rightarrow int * T_2$

$T_2 \rightarrow int$

With the help following parse tree is

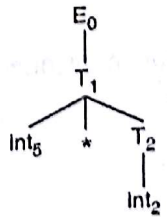


Fig. 1-Q. 6(d)

Q. 3(a) Construct SLR parser for following grammar Show how parsing actions are done for the input string ()()$. Show stacks content, I/p buffer, action buffer.

$S \rightarrow (S)\ S$

$S \rightarrow \varepsilon$

(12 Marks)

Ans. :

Augmenting grammar

1. $S' \rightarrow S$
2. $S \rightarrow (S)\ S$
3. $S \rightarrow \varepsilon$

Find the item sets

I0 →	$S' \rightarrow \cdot S$	I3 →	$S \rightarrow (S \cdot)\ S$
	$S \rightarrow \cdot\ (S)\ S$		
	$S \rightarrow \cdot \varepsilon$	I4→	$S \rightarrow (S) \cdot S$
	$S \rightarrow \cdot\ (S)\ S$		
	$S \rightarrow \cdot$		
I1 →	$S' \rightarrow S \cdot$		
I2 →	$S \rightarrow (\cdot\ S)\ S$	I5 →	$S \rightarrow (S)\ S \cdot$
	$S \rightarrow \cdot\ (S)\ S$		
	$S \rightarrow \cdot$		

Scanned by CamScanner

And now finally the parsing table is as :

State	Input/Action			Go To
	(	)	$	S
0	S2	R3	R3	G1
1			Accept	
2	S2	R3	R3	G3
3		S4		
4	S2	R3	R3	G5
5		R2	R2	

Q. 5(a) For the given grammar below, construct operator precedence relations matrix, assuming*, + are binary operators and id as terminal symbol and E as non terminal symbol.

E ⟶ E + E

E ⟶ E * E

E ⟶ id

Apply operator precedence parsing algorithm to obtain skeletal syntax tree for the statement id. + id * id **(10 Marks)**

Ans. : **Please refer Q. 2(a) of Dec. 2015.**

Chapter 9 : Syntax Directed Translation [Total Marks : 10]

Q. 1(c) What are different type of attributes in SDD? Explain with examples. **(5 Marks)**

Ans. : **Please refer Q. 3(c) of Dec. 2015.**

Q. 6(c) Write short note on Syntax Directed Translation **(5 Marks)**

Ans. :

Syntax-directed translation refers to a method of complier implementation where the source language translation is completely driven by the parser or syntax analysis. In other words, the parsing process and parse trees are used for direct semantic analysis and the translation of the source program. The semantic routines perform interpretation based on the syntax structure. It also attaches the attributes to the grammar symbol and values for these attributes are computed by semantic rules associated with the grammar productions. There are two notations for associating semantic rules with productions, they are as :

(1) Syntax-Directed Definitions.

(2) Translation Schemes.

(1) **Syntax-Directed Definitions**

In compiler construction syntax-directed translation is used for

Constructing abstract syntax tree

Type Checking

Intermediate code generation

It is a method of translating a string into a sequence of actions by attaching one such action to each rule of a grammar. In other words, the parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Thus, parsing a string of the grammar produces a sequence of rule applications. Such grammars are called attribute grammars.

Scanned by CamScanner

(2) Translation Schemes

Translation scheme refers to the method of compiler implementation where the semantic rules are to be processed based on the production rules. According to the definitions mentioned above the token is parsed to build the parse tree and then dependency graph is traversed to process the semantic rules at parse tree nodes (see Fig. 1-Q.6(c)).

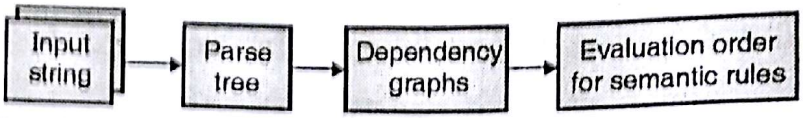


Fig. 1-Q.6(c) : Conceptual view of Syntax-Directed Translation

Each production rule in a grammar is associated with semantic rules which give the information how to calculate the attribute values associated with each grammar symbol in a production. These attributes are evaluated by the semantic rule. The evaluation of these semantic rules gives the following information.

Intermediate code is generated.

Symbol table is used to enter the information

Type checking is performed and error message is issued.

Syntax directed translations can also be implemented through one pass by evaluating semantic rules during parsing without construction of parse tree or a graph showing dependencies between attributes. The translation of the token stream is the result obtained by evaluating the semantic rules.

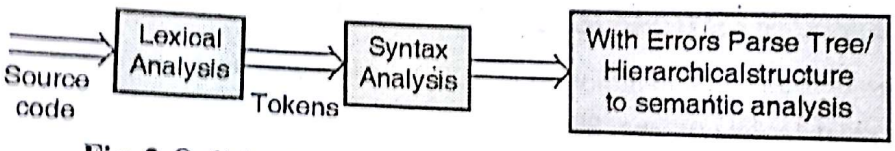


Fig. 2-Q.6(c) : Overview of analysis phase of compiler

Chapter 10 : Intermediate Code Generation [Total Marks : 15]

Q. 1(d) Backpatching with example.

Ans. : **Please refer Q.6(b) of Dec. 2015.** **(5 Marks)**

Q. 4(a) Discuss various intermediate code forms in detail.

Ans. : **(10 Marks)**

The various intermediate code forms :

Intermediate representations of source program are machine independent, but they are close to machine instructions. The intermediate code of any source language is given by the intermediate code generator. Various types of intermediate forms used by the designer are as :

Graphical representations forms are as:

Syntax trees, AST (Abstract Syntax Trees), and DAG (Direct Acyclic Graph)

Postfix notation

Three-address code (Triples, Indirect triples and Quadruples) can be used as an intermediate language.

Scanned by CamScanner

SSA (Static Single Assignment) form

IR should have individual components that describe simple things

Programming languages have well defined intermediate languages like JAVA, PROLOG etc. Intermediate languages are designed to use by the compilers. They are the language of an abstract machine designed to aid in the analysis of computer programs. In this compiler first translates the source code into an intermediate code as an intermediate step before generating machine code for a target machine. The design of an intermediate language typically differs from that of a practical machine language in three ways :

Each instruction represents exactly one operation. Instruction set does not include control flow information. The number of registers available may be large, even limitless. Syntax trees and postfix notations are two kinds of intermediate representation the third kind of IR which is commonly used is three address codes. Postfix notation is a notation in which a mathematical notation wherein every operator follows all of its operands.

The PN of expression :

1. 9* (5+2) is 952+*
2. (a+b)/(c-d) is ab+cd-/

The Rules for Postfix Notation are as :

1. Operands appear before their operators. If E is an expression of the form E1 op E2, the PN of E is E1'E2 'op.
2. If E represents variable/constant, the PN of E is E itself.
3. If E is a parenthesized expression of form (E1), the PN of E is the same as the PN of E1.
4. The expression is evaluated by scanning from left to right.

Graphical Representation

Syntax tree is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. A DAG (Directed Acyclic Graph) gives the same information but in more compact way because common sub expressions are identified.

Syntax Trees

A parse tree is an example of a very high-level intermediate representation. As parse tree contains all the information about the program that can be used to build the source code. A syntax tree is ph structure. Nodes in a syntax tree correspond to statements. Edges in a syntax tree represent s between statements.

Abstarct syntax tree is used to drive the semantic processing and code generation. An abstract ax tree (AST), or just syntax tree, is a tree representation of the syntactic structure of the source ode written in any programming language. Each node of the tree denotes a construct occurring in the source code. Each node has reference to its parent and child node.

Consider the following example of a programming language grammar :

program –> function_list

function_list –> function_list function | function

function –> PROCEDURE ident (params) body

params –> ...

A sample program for this language

PROCEDURE main()

BEGIN

Scanned by CamScanner

```
statement...
END
PROCEDURE factorial(n:INTEGER)
BEGIN
statement...
END
```

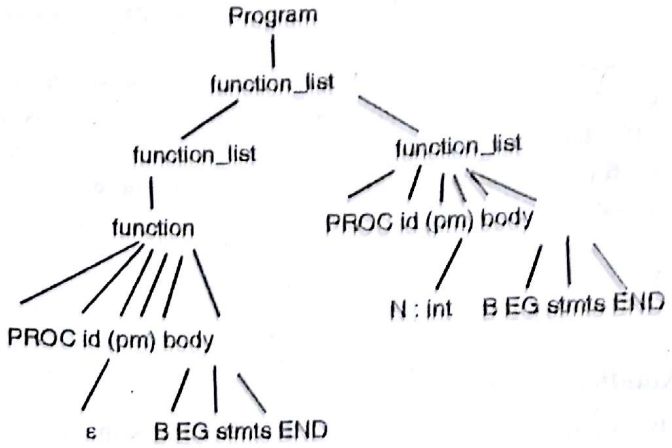


Fig. 1-Q.4(a) : Parse tree for the sample example

Consider again the syntax tree for the arithmetic expressions (a + b * c)

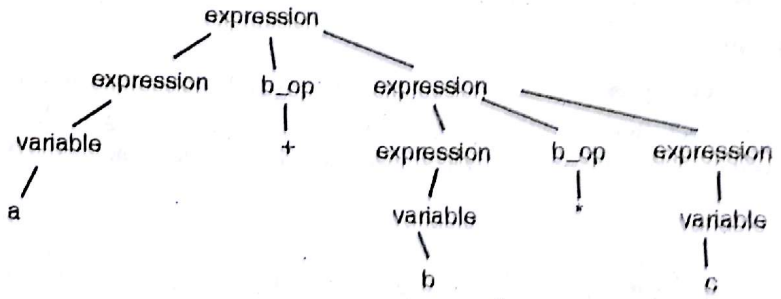


Fig. 2-Q.4(a) : Syntax tree for Arithmetic Expression

Directed Acyclic Graph

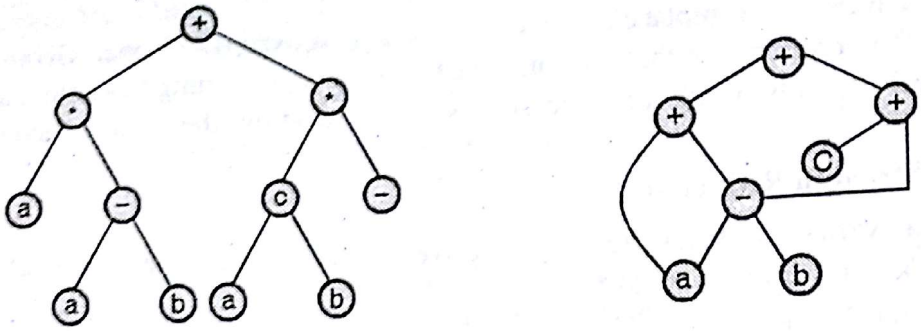
In compilers a tree has a one route from the parent (root) to child (leaf) node. *Directed acyclic graph* (DAG) is used to share the common subtree of a given expression. DAG is used represent more than one path from start symbol to terminals. DAG's are difficult to construct, but provide an obvious savings in space. They also highlight equivalent sections of code and that will be useful later when we use different optimization techniques.

A **directed acyclic graph** is a collection of vertices and directed edges. Each edge connecting one vertex to another, such that there is no way to start at some vertex *v* and follow a sequence of edges that eventually loops back to *v* again. **DAG has** no directed cycles.

Example 1

A syntax tree and DAG representation for the string a * (a-b) + c * (a-b) is given in Fig. 3-Q. 4(a).

Scanned by CamScanner



(a) Syntax Tree Representation (b) DAG Representation

Fig. 3-Q.4(a) : Graphical Representation of a:=b* -c+b* -c

Chapter 12 : Code Optimization [Total Marks : 10]

Q. 2(b) Explain various loop optimization techniques with example. **(10 Marks)**

Ans. :

Loops plays a very important role in code optimizations. Since the body of the loop is executed again and again, and moreover the loops are nested so small changes in the execution of the loop body can results in a remarkable improvement in the performance of the program. Loop optimization can be viewed as the application of a sequence of specific loop transformations. (listed below) to the source code or intermediate representation, with each transformation having an associated test for legality. A transformation (or sequence of transformations) generally must preserve the temporal sequence of all dependencies if it is to preserve the result of the program (i.e., be a legal transformation). It is most important set of optimizations in this programs are likely to spend more time in loops. The commonly used loop optimization techniques are :

1. Code Motion
2. Strength Reduction
3. Induction Variables and Strength Reduction
4. Loop Induction Variable
5. Loop Fission
6. Loop Fusion or Loop Combining
7. Loop Interchange
8. Loop Unrolling
9. Loop Splitting

Code Motion

Code motion (also called code hoisting) unifies sequences of code common to one or more basic blocks to reduce the code size and potentially avoid expensive re-evaluation. The most common form of code motion is loop-invariant code motion that moves statements that evaluate to the same value every iteration of the loop to somewhere outside the loop. In code motion invariant expressions should be executed only once.

Code Motion is of two types :

Code Space Reduction : Similar to common sub-expression elimination but with the objective to reduce code size. Execution Frequency Reduction : It reduces execution frequency of partially available expressions (expressions available at least in one path).

Scanned by CamScanner

Strength Reduction

Operator strength reduction replaces an operator by a "less expensive" one. Given each group of identities below, which operations are the most and least expensive, assuming **f** is a float and **i** is an int ? (Trick question : it all depends on the architectures we need to know the target machine to optimize well!)

Induction Variable Strength Reduction

Basic induction variables are created from derived induction variables. The rules required for creating the basic induction variables are as :

x := y op z; It states that

op can be any *, <<, +, or –

y is an induction variable and z is an invariant

x is a register and no other statement modifies x

x is not y or z

Loop Header

It is also known as the entry point of the loop. It dominates all blocks in the loop body.

Loop Pre-Header

For example block B is a dominator with several incoming edges. , some of them being back edges (so M is a loop header). Create a loop preheader - a basic block before the loop header block. When a basic block is divided into two parts B_{pre} and B_{loop} the contents of B and back edges are moved to B_{loop}, the rest of the edges are moved to point into B_{pre}, and a new edge from B_{pre} to B_{loop} is inserted. B_{pre} is called the loop pre-header, and B_{loop} would be the loop header.

Loop Terminology

Loop	:	This is strongly connected component of a control flow graph
Loop entry edge	:	Source entry not in loop but target in loop
Loop exit edge	:	Source entry in loop but target not in loop
Loop header node	:	Target of loop entry edge
Natural loop	:	Loop made up of single loop header
Back edge	:	Target is loop header & source is in the loop
Loop tail node	:	Source of back edge
Loop preheader node	:	Single node that's source of the loop entry edge
Nested loop	:	Loop whose header is inside another loop
Reducible flow graph	:	CFG whose loops are all natural loops

Loop Induction Variable

Induction variables values are either incremented or decremented whenever the value of the variable is changed. There are three types of induction variable :

Basic Induction Variable :

These type of variables are assigns within a loop. There are two types of induction variable. For example

K = K +/- D where D is a constant.

Primary Induction Variable : The loop execution is controlled in it.

(for j=0; j<100; j++)

i is the primary induction variable.

Scanned by CamScanner

Derived Induction Variable :

Derived variable is a linear function of a basic induction variable.

Loop Fission

Loop Fission tries to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. It improves the locality of reference.

Loop Fusion or Loop Combining

The transformation opposite to loop fission is loop fusion. When two adjacent loops would iterate the same number of times, their bodies can be combined as long as they make no reference to each other's data.

Loop Interchange

These optimizations exchange inner loops with outer loops.

Loop Unrolling

It duplicates the body of the loop multiple times, in order to decrease the number of times the loop conditions needs to be tested. It also reduces the number of jumps from the end of the loop to the beginning.

Loop Splitting

Loop Splitting simplifies a loop or eliminate dependencies by breaking the loop into multiple loops which have the same bodies but iterate over different portions.

Chapter 13 : Run Time Storage [Total Marks : 10]

Q. 5(b) Explain Run time organization in detail. **(10 Marks)**

Ans. : **Please refer Q. 5(b) of May 2015.**

Chapter 14 : Compiler-Compilers [Total Marks : 05]

Q. 6(a) Write short note on LEX and Y ACC **(5 Marks)**

Ans. :

YACC : Please refer Q.6(a) of May 2015.

LEX

LEX and YACC is a matched pair of tools. A lexer or scanner is used to perform lexical analysis or the breaking up of an input stream into meaningful units, or tokens. LEX is a tool for automatically generating a lexer or scanner given a lex specification (.l file). For example, consider breaking a text file up into individual words. LEX's output is mostly designed to be fed into some kind of parser. LEX and YACC are tools for building programs.

□□□

Scanned by CamScanner

May 2015

Q. 1 (a) Differentiate between application program and system program. **(5 Marks)**

(b) State the reason for assembler to be multi pass program. **(5 Marks)**

(c) Explain Functions of loader. **(5 Marks)**

(d) What is flow graph ? State its signification in code generation. **(5 Marks)**

Q. 2 (a) For following code what will be output generated by Pass-I and Pass-II for two pass assembler. Explain with database. **(10 Marks)**

```
ABC     Start   0
        USING   *,15
        L       1, FIVE
        A       1, FOUR
        ST      1, TEMP
FOUR    DC      F'4'
FIVE    DC      F'5'
TEMP    DS      1F
        END
```

(b) Explain operator precedence parser along with example. **(10 Marks)**

Q. 3 (a) Generate three address code for following code.

```
While (a < b) do
if (c < d) then
x = y + 2
else
x = y – 2
```

(10 Marks)

(b) Discuss with example quadruple, triple and indirect triple. **(10 Marks)**

Q. 4 (a) Construct predictive parsing table for following grammar. **(10 Marks)**

$S \rightarrow A$

$A \rightarrow aB \mid Ad$

$B \rightarrow bBC \mid f$

$C \rightarrow g$

(b) Explain loop optimization with example. **(10 Marks)**

Q. 5 (a) What are different issues in code generation, explain in detail. **(10 Marks)**

(b) Explain run time storage organization in details. **(10 Marks)**

Q. 6 Write short note on : **(20 Marks)**

(a) Code motion. (b) LEX and YACC

(c) Software tools. (d) Left recursion and left factoring removal technique.

□□□

Scanned by CamScanner

Dec. 2015

Q. 1 (a) Differentiate between Application program and system program. Indicate the order in which following system programs are used, from developing program upto its execution. **(5 Marks)**
Assemblers, Loaders, Linker, Macro processor, compiler, Editor.

(b) Eliminate Left recursion in the following grammar (Remove Direct and Indirect recursion)
$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \varepsilon$ **(5 Marks)**

(c) What is an activation record? Draw diagram of General Activation record and explain the purpose of different fields of an activation record. **(5 Marks)**

(d) What are the different functions of loader ? **(5 Marks)**

Q. 2 (a) For a given grammar below, construct an operator precedence relation matrix, assuming *, + are binary operators and id as terminal Symbol and E as non-terminal. **(10 Marks)**
$E \rightarrow E + E$ $E \rightarrow E * E$ $E \rightarrow id$
Apply operator precedence parsing algorithm for the statement
id + id * id

(b) Explain the role of code optimization in compiler designing? Explain Peephole optimization along with an example. **(10 Marks)**

Q. 3 (a) Write a note on JAVA compiler environment. **(5 Marks)**

(b) Write a brief note on design of an Editor. **(5 Marks)**

(c) Explain synthesized and Inherited attributes used in Syntax Directed Definition. **(5 Marks)**

(d) Find FIRST and FOLLOW Set for given grammar below **(5 Marks)**
$E \rightarrow T E'$ $E' \rightarrow + T E' \mid \varepsilon$
$T \rightarrow F T'$ $T' \rightarrow * F T' \mid \varepsilon$
$F \rightarrow (E)$ $F \rightarrow id$

Q. 4 (a) Explain Design of Dynamic Linking Loader along with example. **(10 Marks)**

(b) For the following grammar construct LL(1) parser table. **(10 Marks)**
$S \rightarrow F$ $S \rightarrow (S - F)$ $F \rightarrow a$
And Parse the string (a – a). Show contents of stack and i/p buffer and action taken after each step.

Q. 5 (a) Explain different pseudo-ops used for conditional macro expansion along with an example. **(10 Marks)**

(b) What are the different phases of Compiler? Illustrate compilers internal representation of source program for following statement after each phase. **(10 Marks)**
Position = initial + rate * 60

Q. 6 (a) With reference to Assembler explain following tables with suitable example. **(10 Marks)**
(i) POT (ii) MOT (iii) ST (iv) LT

(b) Explain Backpatching with an example. **(10 Marks)**

❑❑❑

Scanned by CamScanner

May 2016

Q. 1 (a) What is the role of an automata in compiler design? **(5 Marks)**

(b) Eliminate Left recursion in the following grammar (Remove Direct and Indirect recursion) **(5 Marks)**

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

(c) What is an activation record ? Draw diagram of General Activation record and explain the purpose of different fields of an activation record. **(5 Marks)**

(d) What is the difference between Compiler and Interpreter? **(5 Marks)**

Q. 2 (a) Explain with an example Quadruples, Triples, Indirect triples. **(10 Marks)**

(b) What is the difference between Dynamic Loading and Dynamic Linking explain with an example? **(5 Marks)**

Q. 3 (a) Write a note on JAVA compiler environment. **(5 Marks)**

(b) Write a brief note on Design of an Editor. **(5 Marks)**

(c) Explain synthesized and Inherited attributed used in Syntax Directed Definition. **(5 Marks)**

(d) Find FIRST and FOLLOW Set for given grammar below **(5 Marks)**

$E \rightarrow T E'$ $\quad$ $E' \rightarrow + TE' \mid \varepsilon$

$T \rightarrow F T'$ $\quad$ $T' \rightarrow * FT' \mid \varepsilon$

$F \rightarrow (E)$ $\quad$ $F \rightarrow id$

Q. 4 (a) Explain different Code Optimization techniques along with an example. **(10 Marks)**

(b) For the following grammar construct LR(0) parser table **(10 Marks)**

$S \rightarrow aCDe$

$C \rightarrow Cbc$

$C \rightarrow b$

$D \rightarrow d$

And Parse the string abbcbcde. Show contents of stack and input buffer and action taken after each step.

Q. 5 (a) Draw and explain DAG and represent the following example with it. **(10 Marks)**

(a / b) + (a / b) * (c * d)

(b) What are the different phases of Compiler? Illustrate compilers internal representation of source program or following statement after each phase. **(10 Marks)**

Amount = P + P * N * R / 100

Q. 6 (a) With reference to Assembler explain following tables with suitable examples. **(10 Marks)**

(i) POT (ii) MOT (iii) ST (iv) LT

(b) What are the different issues in design of Code Generator ? Explain with an example. **(10 Marks)**

□□□

Scanned by CamScanner

Dec. 2016

Q. 1 (a) What is Handle pruning ? **(5 Marks)**

(b) What is role of finite automata in compiler theory ? **(5 Marks)**

(c) What are different type of attributes in SDD? Explain with examples. **(5 Marks)**

(d) Backpatching with example. **(5 Marks)**

Q. 2 (a) Explain two pass macro processor with flowchart and databases. **(10 Marks)**

(b) Explain various loop optimization techniques with example. **(10 Marks)**

Q. 3 (a) Construct SLR parsing table for following grammar. Show how parsing actions are done for the input string ( ) () $. Show stacks content, action. buffer,

S -> (S)S

S -> E **(12 Marks)**

(b) What are various databases used in two pass assembler design: Explain with example. **(8 Marks)**

Q. 4 (a) Discuss various intermediate code forms in detail. **(10 Marks)**

(b) What is Loader ? Explain functions of loader with examples. **(10 Marks)**

Q. 5 (a) For the given grammar below, constmct operator precedence relations matrix, assuming *, + are binary operators and id as terminal symbol and E as non terminal symbol.

E +E

E E

E

Apply operator precedence parsing algorithm to obtain skeletal syntax tree

for the statement

id. + id * id **(10 Marks)**

(b) Explain Run time organization in detail. **(10 Marks)**

Q. 6 Write short notes. **(20 Marks)**

(a) LEX andY ACC (b) Design of an Editor

(c) Syntax Directed Translation (d) Recursive Descent parsing

❑❑❑

Scanned by CamScanner