INDEX

SR. NO.	TITLE	DATE	TEACHER'S SIGN / REMARK
	Dhanal mam (C.C)		
	T.Y. CSE.		

Unit I

Compilers – is a program that reads a program written in one language – the source language – and translates it into an equivalent program in another language – the target language.

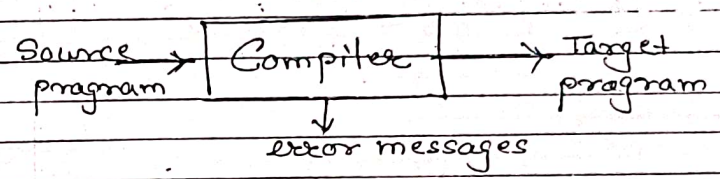


fig: 1 A Compiler

Compilers are sometimes classified as single-pass, multi-pass, load-and-go, debugging, or optimizing depending on how they have been constructed or on what function they are supposed to perform.

There are two parts to Compilation: Analysis and Synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation.

The process by which a software accepts a text in a certain language as input and produces a text in another language, while preserving the meaning of that text, is called translation. A translator is a generalised form of Compiler.

The language in which the translator is written is called the implementation language.
When the object language is a low level lang; such a translator is called a Compiler.
This conversion process is essential for the hardware to interpret and perform the semantics of the input program.

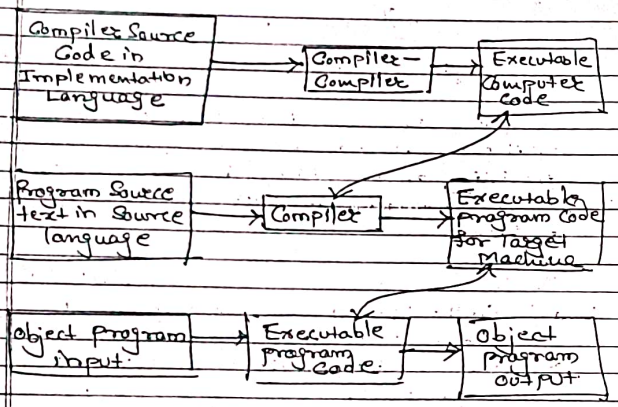


Fig: Compiling and Running a Compiler.

Thus Compiler is just a giant file conversion program. To carry out its translation functionality, the semantics and the grammar of the source and target languages should be specified.

When the source language is also the implemen language and the source text to be compiled actually a new version of the Compiler itself the process is called bootstrapping.

Issues in Compilation
Following issues have to be considered in th design of Compilers due to the nature of H.L

1. Compiler must handle simple expressions an maintain hierarchy of operations.
2. Compiler should be capable of maintaining data integrity
3. Handle various user defined data types.
4. Must develop appropriate storage mappings fo data structures.
5. Must resolve occurance of each variable nam in a program to determine the namespace t which a referenced variable belongs to.
6. Handle different control structures.
7. Should perform Optimization.

Analysis-Synthesis Model of Compilation—

Analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.

Synthesis part constructs the desired target program from the IR.

Many s/w tools that manipulate source programs first perform some kind of analysis—
For e.g.

1. Structure editors — A structure editor takes as input a sequence of commands to build a source program. It not only performs the text-creation and modification functions of an ordinary text editor but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
2. Pretty Printers — It analyzes a program and prints it in such a way that the structure of the program becomes clearly visible.
3. Static Checkers — It reads a program, analyzes it and attempts to discover potential bugs without running the program.
4. Interpreters — Instead of producing a target program as a translation, an interpreter performs the operations implied by the source program.

Unit II

Role of Lexical Analyzer

The Lexical Analyzer is the first phase of compiler. Its main task is to read the in that the parser uses for syntax analysis.

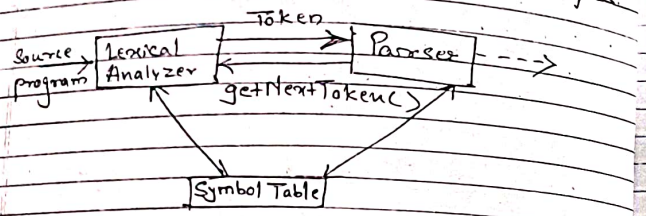


fig. Interaction of Lexical Analyzer with parser also called Block diagram of Lexical Analyzer.

Lexical analyzer acts as subroutine or comm of the parser on receiving "getNextToken" comm for the parser. The lexical analyzer reads input characters until it can identify the ne token.

Lexical analyzer not only reads the sour text but also perform certain secondary task At user interface one such task is stripping out from the source program comments & white space in the form of blank, tab & newline characters. Another is correcting error messages from the compiler with the source program.

Lexical analyzers can be divided into a cascade of two phases, the first called "Scanni and the second "Lexical Analysis".
The scanner is responsible for doing simple tasks, whi Lexical analyzer does the complex operations.

Issues in Lexical Analysis—

There are several reasons for separating analysis phase of compiling into lexical analysis and parsing

1.) The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases. A parser which needs to perform secondary functions like removing comments and white space will be more complex than one that can assume comments and white space have already been removed by a lexical analyzer. Hence, If we are designing a new language, separating the lexical and syntactic conventions can lead to a cleaner overall language design.

2.) Compiler efficiency is improved. A separate lexical analyzer allows us to construct a specialized and potentially more efficient processor for the task. A large amount of time is spent reading the source program and partitioning it into tokens.
Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler.

3.) Compiler portability is enhanced.

Tokens, Patterns, Lexemes

In general, there is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token. The pattern is said to match each string in the set. A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
for. e.g. Const pi = 3.1416
Here the substring pi is a lexeme for the token "identifier".

e.g.

Token	Sample lexeme
relation	<, <=, =, <>, >, >=
id	Pi, count, a, y
num	3.1416, 25, 6.02E23

Tokens are treated as Terminal symbols in the grammar for the source language. The lexemes matched by the pattern for the token represent strings of characters in the source program that can be treated together as a lexical unit.
In most programming languages the following constructs are treated as tokens: keywords, operators, identifiers, constants, literal strings and punctuation symbols such as parantheses, commas, and semicolons.
A pattern is a rule describing the set of lexemes that can represent a particular token in source programs.

Attributes For Tokens:

When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler.

The lexical analyzer collects information about tokens into their associated attributes.

Example: The tokens and associated attributes values for the fortran statement

E = M * C ** 2

<id, pointer to symbol-table entry for E>
<assign-op, >
<id, pointer to symbol-table entry for M>
<Mult-op, >
<id, pointer to symbol-table entry for C>
<exp-op, >
<num, integer value 2>

LEXICAL ERRORS

A situation in which the lexical analyzer is unable to proceed because none of the pattern for tokens matches a prefix of the remaining input then the token cannot be produced for the lexical unit and there exists a lexical error.

e.g. 5x is a lexical error as it does not match with any pattern.
Whereas if (a == f(x)) ... in this statement if is not a lexical error. Rather its token is produced as it matches with the pattern "identifier". But it certainly a syntatic error which is handled by some other subsequent phase.

Possible error Recovery strategies for lexical errors

1. Panic mode
2. deleting an extraneous character
3. Replacing an incorrect character by a correct character.
4. Transposing two adjacent characters.
5. Inserting a missing character.

In panic mode recovery we delete successive characters from the remaining input until the lexical analyzer can find a well-formed token.

Error transformations like inserting deleting replacing or transposing may be tried in an attempt to repair the input.

The strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by just a single error transformation.

This strategy assumes most lexical errors are the result of a single error transformation, but this may not be always.

TE CSE Part VI Sem-VI April 2017
Compiler Construction (New)

Sub. Code: 86858 SL-1017

Total Marks: 50 26-04-2017

SECTION-I

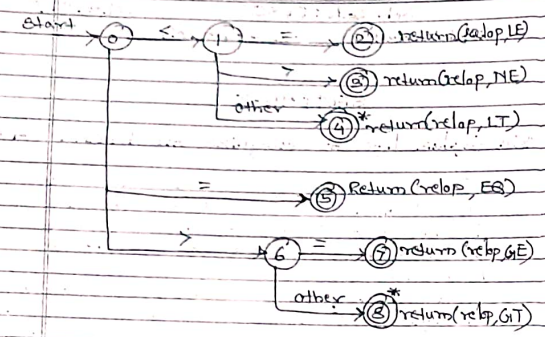
Q1(a) Explain how to recognize the tokens. 4 mks

Tokens are recognized using Finite Automata. Lexical analyzer should isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute-value.

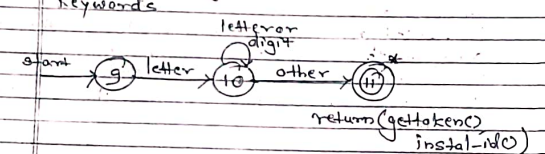
Finite Automata can be constructed using a stylized flowchart called a Transition diagram. Positions in a transition diagram are drawn as circles and are called states. The states are connected by arrows, called edges. These edges are called transitions. One state is labelled as start state. It is the initial state of the transition diagram where control resides when we begin to recognize a token. On entering a state we read the next input character.

Some sample FAs are as follows:-

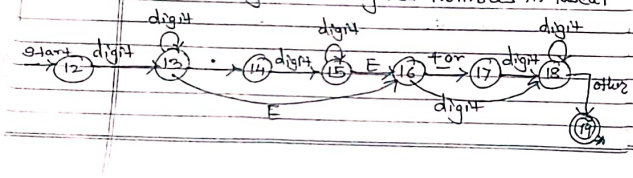
1. Transition diagram for relational operators.



2.) Transition diagram for identifiers and keywords



3.) Transition dig. for unsigned numbers in Pascal



In general, there may be several transition diagrams, each specifying a group of tokens. If failure occurs while we are following a transition diagram, then we retract the forward pointer to where it was in the start state of diagram and activate the next transition diagram. Since the lexeme-beginning and forward pointers marked the same position, the start state of the diagram, the forward pointer is retracted to the position in the start state of the diagram marked by the lexeme-beginning pointer. If failure occurs in all transition diagrams, then a lexical error has been detected and we invoke an error-recovery routine.

Implementing a Transition Diagram

A sequence of transition diagrams can be converted into a program to look for the tokens specified by the diagrams.

```
e.g. int state=0, start=0
     int lexical-value;
     int fail()
     {
       forward = token-beginning;
       switch(start) {
         case 0:  start = 9; break;
         case 9:  start = 12; break;
         case 12: start = 20; break;
         case 20: recover(); break;
         default: /* Compiler error */
       }
       return start;
     }
```

Note: Ans. for 1a can be cut short for 4 mks, write only 2 examples and reduce a little bit of theory. This ans as it is can be used for 6mks and 8mks.

1 b.) Consider the following grammar

$S \to CC$
$C \to cC$
$C \to d$

Find the canonical collection of sets of LR(1) items and construct the action goto table. 6 mks.

$$\left.\begin{array}{l} S' \to S \\ S \to CC \\ C \to cC \\ C \to d \end{array}\right\} \text{Augmented Grammar}$$

$S' \to \cdot S, \$$ // default
$A \to \alpha \cdot B\beta, a$

$A = S \quad \alpha = \wedge \quad B = S \quad \beta = \$ \wedge \quad a = \$$
$FIRST(\beta a) = FIRST(\$\$) = \{\$\}$

add $B \to \cdot\gamma, b$

$S \to \cdot CC, \$$
$A \to \alpha \cdot B\beta, a$

$A = S \quad \alpha = \wedge \quad B = C \quad \beta = C \quad a = \$$
$First(\beta a) = First(C\$) = First(C) = \{c, d\}$

$C \to \cdot cC, c|d$
$C \to \cdot d, c|d$

I_0: $S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot cC, c|d$
$C \to \cdot d, c|d$

$I_1 : (I_0, S)$ $S' \to S \cdot, \$$

$I_2 : (I_0, C)$ $S \to C \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

$I_3 : (I_0, c)$
$C \to c \cdot C, c|d$
$C \to \cdot cC, c|d$
$C \to \cdot d, c|d$

$I_4 : (I_0, d)$ $C \to d \cdot, c|d$

$I_5 : (I_2, C)$ $S \to CC \cdot, \$$

$I_6 : (I_2, c)$ $C \to c \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

$I_7 : (I_2, d)$ $C \to d \cdot, \$$

$I_8 : (I_3, C)$ $C \to cC \cdot, c|d$

$I_3 : (I_3, c)$ $C \to c \cdot C, c|d$
$C \to \cdot d, c|d$
$C \to \cdot cC, c|d$

I_4 : (I_3, d) : $C \to d \cdot, cd$

I_5 : (I_3, C) $C \to cC \cdot, \$$

I_6 : (I_3, c) $C \to c \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

I_7 : (I_6, d) : $C \to d \cdot, \$$

Canonical Parsing table for grammar LR(1)

State	c	d	$	S	C
0	S_3	S_4		1	2
1			acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

Q.2 a) List and explain various Compiler Cons Tools — 4 mks.

Shortly after the first compilers were writ Systems to help with the Compiler writing process appeared. These systems have been referred to as Compiler-Compilers, Compiler-generators or translator-writing Systems, Compiler writers.

Following is the list of some useful Comp Construction tools:-

1. Parser Generators
2. Scanner Generators
3. Syntax-directed translation engines
4. Automatic Code Generators.
5. Data flow Engines
6. Compiler-Construction Tool kit.

1. Parser Generators :- These produce Syntax an normally from i/p that is based on a CF Many parser generators utilize powerful pars algorithms that are too complex to be c out by hand. Hence, this phase has now bec easy to implement.

2. Scanner Generator :- These automatically ge lexical analyzers, normally from a specification based on regular expressions. The basic organization of the resulting lexical analyzer effect a Finite Automata. That means the o/p is in the form of a F.A.

3. Syntax Directed translation engines :- These produce collections of routines that walk the parse tree, generating I.C. The basic idea is that one or more "translations" are associated with each node of the parse tree and each translation is defined in terms of translations at its neighbor nodes in the tree.

4. Automatic Code Generators :- Such tools take a collection of rules that define the translation of each operation of the intermediate lang. into the m/c lang. for the target m/c. The I.C. stmts are replaced by "templates" that represent sequences of m/c instructions.

5. Data Flow Engines :- These tools collect the information needed to perform good code optimization. It involved data flow analysis and control flow analysis.

Q.2 b) Write the code for recursive descent parser for the following grammar — 6 mks

$S \to AbS$
$A \to a \mid da$

```
void main()
{
  // Enter the string
  S()
  //String is Syntatically Correct
}

S()
{
  A()
  match(b);
  S()
}

A()
{
  if (LS == 'a')
    match('a');
  else if (LS == 'd')
  {
    match('d');
    match('a');
  }
  else
    error();
}

match(token tmp)
{ scanner
  (if (LS == tmp) = c
  then LS++
  else error())
}

error()
{
  //String Syntatically incorr
  exit(0);
}
```

Q.2 b.) What are conflicts in Shift-Reduce Parser? Explain with the help of an example. — 6 mks

There are CFG for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next i/p symbol, cannot decide ~~which of the several reds which~~ whether to shift or to reduce (a shift/reduce conflict) or cannot decide which of several reductions to make (a reduce/reduce conflict).

These grammars are not in the LR(k) class of grammars. (non-LR grammar)

e.g.

Consider the grammar

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

Consider the string id+id*id$

Step	Stack	input	Action
1	$ —	id+id*id$	Shift
2	id	+id*id$	Reduce F→id
3	F	+id*id$	Reduce T→F
4	T	+id*id$	Reduce E→T
5	E	+id*id$	Shift
6	E+	id*id$	Shift
7	E+id	*id	Reduce F→id
8	E+F	*id	Reduce T→F
9	E+T	*id	Reduce E→E+T

Reduce-Reduce Conflict

E	*id	Shift	E+T	*id	Redu
E*	id	Shift	E+E	*id	Shi
E*id	—	Reduce	E+E*	id	Shi
E*F	—	Reduce	E+E*id	—	Red
E*T	—	Reduce	E+E*F	—	Red
E*E	—	?	E+E*T	—	Red
		Error			Error
					Cannot p

Instead of Reducing a step 9 if we shift following occurs—

	Stack	i/p	Action
9.	E+T	*id	Shift
10.	E+T*	id	Shift
11	E+T*id	—	Reduce F→id
12	E+T*F	—	Reduce T→T*F
13.	E+T	—	Reduce E→E+T
14	E	—	String Accepted

Hence, at step 9 if we choose to red the string is not accepted whereas if shift string is accepted. This is shift reduce conflict.

Q.3 What is Left recursion? Explain elimination of left recursion with example. — 5 mks

A grammar is left recursive if it has a nonterminal A' such that there is a derivation $A \xrightarrow{+} A\alpha$ for some string α. Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left-recursion is needed.

Left recursive rule can be of the form

$A \to A\alpha \mid \beta$

Replacement Algo.

$A \to \beta A'$

$A' \to \alpha A' \mid \wedge$

e.g. $E \to E+T \mid T$, $T \to T*F \mid F$, $F \to (E) \mid id$ } Left recursion

After applying replacement Alg.

$E \to TE'$

$E' \to +TE' \mid \wedge$

$T \to FT'$

$T' \to *FT' \mid \wedge$

$F \to (E) \mid id$

It can be generalized as follows:-

$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

Replacement Algo:-

$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \wedge$

Consider another example where we have indirect left recursion

$S \to Aa \mid b$

$A \to Ac \mid Sd \mid \wedge$

Here $S \Rightarrow Aa \Rightarrow Sda$ i.e. $S \xrightarrow{+} Sda$ is not immediate but indirect recursion

Replacement Algo. applied on direct / immediate recursion but for indirect we must replace/substitute $A \to Sd$ to obtain following prod^n

$A \to Ac \mid Aad \mid bd \mid \wedge$

Now we can eliminate direct left-recursion

o/p grammar

$S \to Aa \mid b$

$A \to bdA' \mid A'$

$A' \to cA' \mid adA' \mid \wedge$

optional

Algorithm :- Eliminating left recursion Having indirect left-recursion.

input: G with no cycles or $\wedge$-productions

o/p: Equivalent G with no left recursion

Method:

1. Arrange the non-terminals in some order $A_1, A_2, \ldots, A_n$

2. for $i = 1$ to n do begin
   for $j = 1$ to $i-1$ do begin
   replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$.
   Where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j-productions.
   end
   eliminate the immediate left-recursion among the A_i-productions.
   end.

Note: Algo. for eliminating left-recursion can be skipped for 5 mks but can be added if required or asked in the question.

OR

Q3. Explain i/p buffering in lexical analysis. Give an example lexeme involving exactly two 1's for the given pattern:

$(01+21+02)^*(1+\wedge)(1+22)$

There are three general approaches to the implementation of a lexical analyzer

1. Use a lexical-analyzer generator, such as LEX Compiler
2. Write the lexical analyzer in a conventional system programming lang.
3. Write the lexical analyzer in assembly lang and explicitly manage the reading of input

Three choices are listed in order of increasing difficulty for the implementor.

Since the lexical analyzer (LA) is the only phase of the compiler that reads the source program (SP) character-by-character, it is to spend a considerable amount of time in the LA phase, even though the later phases are conceptually more complex.

Thus, the speed of L.Analysis is a Co in Compiler design. Because a large amount of time can be consumed moving characters, spe buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Buffer Pair :-

A buffer is divided into two N-character Typically N is the no. of characters on one d block

E	=	M	*	C	*	*	2	eof	

forward

lexeme-beginning

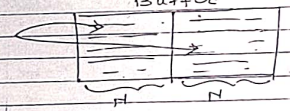
An input buffer in two halves.

We read N input characters into each half of buffer with one system read command, rather invoking a read command for each input char

eof marks the end of the source file. Two pointers are maintained namely lexeme-beginning and forward. Initially, both pointers point to the first character of the next lexeme to be found. The string of characters betw the two pointers is the current lexeme. Forward pointer scans ahead until a match for a pattern is found. Once, the next lexeme is determined, the forward ptr. is set to the character at its right end. After the lexeme is processed, both pointers are set to the character immediatly past the lexeme. With this scheme, comments and white space are treated as patterns that yield no token.

If the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters. If the forward is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.

Buffer



N N

Sentinels —

One must check each time we move the forward pointer that we have not moved off one half of the buffer, if we do then we must reload the other half. Hence, we need to perform two tests 1. Whether forward ptr points to eof 2. Whether forward ptr has reached end of buffer half

We can reduce two tests to one if we extend each buffer half to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program. A natural choice is eof.

E	=	M	*	eof	C	*	*	2	eof

Fig: Sentinels at the end of each buffer

Lexeme involving exactly two 1's for the given pattern

$(01+21+02)^*(1+\wedge)(1+22)$

1) 01122
2) 0211
3) 21122
4) 11
5) 011

Section-II

SL-1017

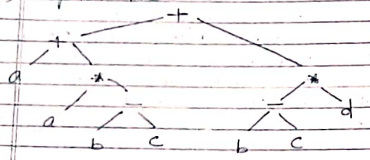
Q4 a) Give — 4 mks

i) Syntax Tree

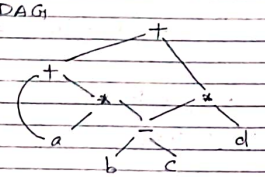
ii) DAG for the expression

$a + a * (b-c) + (b-c) * d$

i) Syntax Tree



ii) DAG



Q4 b) What is SDD? Explain the construction of Syntax tree for Expressions using SDD with the help of an example. — 6 mks

SDD = Grammar + Semantic Actions

A Syntax directed definition is a generalization of CFG in which each grammar symbol has an associated set of attributes partitioned into two subsets called the Synthesized and Inherited attributes of the grammar symbol.

The value of an attribute at a parse tree node is defined by a Semantic Rule associated with the production used at that node.

The value of Synthesized attribute at a node is computed from the values of attributes at children of that node in the parse tree.

The value of an Inherited attribute is computed from the values of attributes at the siblings and parent of that node.

Semantic rules set up dependencies between attributes that will be represented by a graph.

From the dependency graph, we derive an evaluation order for the semantic rules. Evaluation of the semantic rules defines the values of the attributes at the nodes in the parse tree for the input string.

Definition

In SDD each grammar production $A \to \alpha$ has associated with it a set of semantic rules of the form $b = f(c_1, c_2, \ldots, c_k)$ where f is function, and either

1. b is a synthesized attribute of A and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of the production or

2. b is an inherited attribute of one of the grammar symbols on the RHS of the production, and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of the production.

SDD for constructing Syntax tree for an Expression

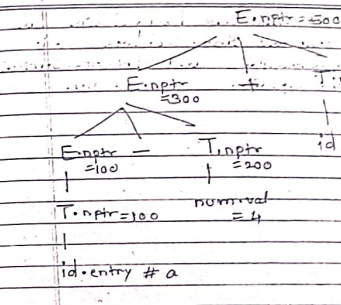
Production	Semantic Rules
$E \to E_1 + T$	{ E.nptr = mknode('+', E_1.nptr, T.nptr) }
$E \to E_1 - T$	{ E.nptr = mknode('−', E_1.nptr, T.nptr) }
$E \to T$	{ E.nptr = T.nptr }
$T \to (E)$	{ T.nptr = E.nptr }
$T \to id$	{ T.nptr = mkleaf(id, id.entry) }
$T \to num$	{ T.nptr = mkleaf(num, num.val) }

The synthesized attribute nptr for E and T keeps track of the pointers returned by the function calls mkleaf(), mknode().

Attributes id.entry and num.val are the lexical values assumed to be returned by the Lexical Analyzer with the tokens id and num.

Consider the string $a - 4 + c$

Parse tree for the string is as follows—



P_1 = mkleaf(id, #a)

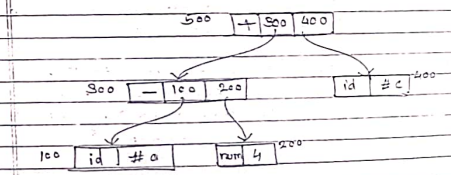
E.nptr = T.nptr

T.nptr = mkleaf(num, 4)

E.nptr = mknode(−, E.nptr, T.nptr)

T.nptr = mkleaf(id, #c)

E.nptr = mknode(+, E.nptr, T.nptr)



Q4 b) Write an SDD to show how backpatching can be used to generate three address code for Boolean expressions and flow-of-control statements and explain. — 6 mks.

Main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. This problem can be solved by generating a series of branching statements with the targets of the jumps temporarily left unspecified. Each such stmt. will be put on a list of goto stmts. whose labels will be filled in when the proper label can be determined. This subsequent filling in of labels is called backpatching.

(1) $E \to E_1$ or $M E_2$ { backpatch(E_1.falselist, M.quad);
E.truelist = merge(E_1.truelist, E_2.truelist);
E.falselist = E_2.falselist }

(2) $E \to E_1$ and $M E_2$ { backpatch(E_1.truelist, M.quad);
E.truelist = E_2.truelist;
E.falselist = merge(E_1.falselist, E_2.falselist) }

(3) $E \to$ not E_1 { E.truelist = E_1.falselist;
E.falselist = E_1.truelist }

(4) $E \to (E_1)$ { E.truelist = E_1.truelist
E.falselist = E_1.falselist }

(5) $E \to id_1$ relop id_2 { E.truelist = makelist(next
E.falselist = makelist(nextquad
emit(if id_1.place relop.op id_2.place
emit(goto —); }

(6) $E \to$ true { E.truelist = makelist(nextquad);
emit(goto —); }

(7) $E \to$ false { E.falselist = makelist(nextquad
emit(goto —); }

(8) $M \to \epsilon$ { M.quad = nextquad }

Synthesized attributes truelist and falselist of E are used to generate jumping code for expressions.
The variable nextquad holds the index of next quadruple to follow.

1. makelist(i) creates a new list containing an index into the array of quadruples, makelist returns a pointer to the list made.

2. merge(P_1, P_2) concatenates the lists pointed by P_1 and P_2 and returns a pointer to the concatenated list.

3. backpatch(p, i) inserts i as the target for each of the statements on the list pointed to by p.

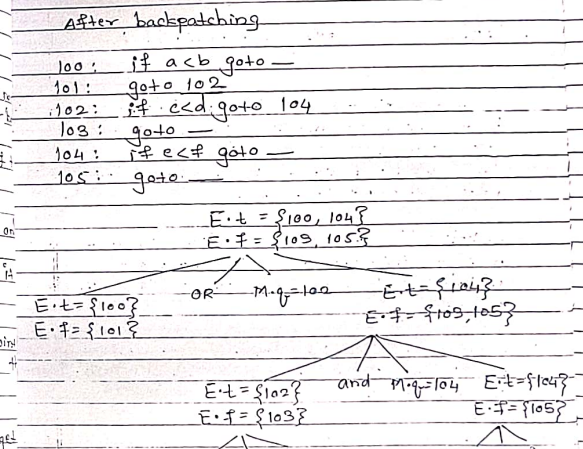
Consider the expression —
a<b or c<d and e<f

Before Backpatching

100: if a<b goto —
101: goto —
102: if c<d goto —
103: goto —
104: if e<f goto —
105: goto —

After backpatching

100: if a<b goto —
101: goto 102
102: if c<d goto 104
103: goto —
104: if e<f goto —
105: goto —



Q5 a) Using code generation algorithm, generate IC for the expression a = (a-b) + (a-c) + (a-c). Also show the values of the register descriptor and address descriptor as code generation progresses. — 4 mks.

d = (a-b) + (a-c) + (a-c) is translated into following three-address code sequence —

t = a-b
u = a-c
v = t+u
d = v+u

t, u, and v are temporaries and we assume that they are not in memory unless we explicitly store their values with a MOV instruction.

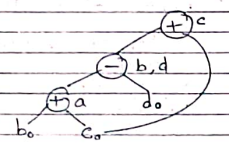
Statements	Code Generated	Register Descriptor	Address Descriptor
		registers empty	
t = a-b	Mov a, R_0 Sub b, R_0	R_0 contains t	t in R_0
u = a-c	Mov a, R_1 Sub c, R_1	R_0 contains t R_1 contains u	t in R_0 u in R_1
v = t+u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d = v+u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

5.a) Explain optimization of basic blocks. — 4 mks.

Many of the structure-preserving transformations can be implemented by constructing a dag for a basic block.

Common subexpressions can be detected by noticing, as a new node m is about to be added, whether there is an existing node n with the same children, in the same order, and with the same operator. If so, n computes the same value as m and may be used in its place.

e.g. A DAG for the block
$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$ is

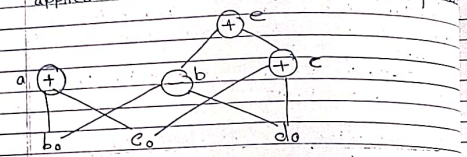


o/p optimized code is
$a = b + c$
$d = a - d$
$c = d + c$

In common subexpressions, we really are looking for expressions that are guaranteed to compute the same value, no matter how that value is computed.

$a = b + c$
$b = b - d$
$c = c + d$
$e = b + c$

DAG method will miss the fact that the expr. computed by the first and fourth statement in the sequence. However, algebraic ident. applied to the DAG can solve this prob.



The operation on DAGs that corresponds to dead code elimination is straightforward to implement. We delete from a DAG any (node with no ancestors) that has no live v. Repeated application of this transformation remove all nodes from the DAG that correspond to dead code.

Use of Algebraic Identities –
Algebraic identities represent another imp. class of optimizations on basic blocks. Following transformations can be tried during optimization —
$x + 0 = 0 + x = x$
$x - 0 = x$
$x * 1 = 1 * x = x$
$x / 1 = x$

Reduction in strength — is replacing a more expensive operator by a cheaper one as follows –
$x * x 2 = x * x$
$2.0 * x = x + x$ (Multn replaced by addition)
$x / 2 = x * 0.5$

Constant folding — Here we evaluate constant expressions at compile time and replace the constant expressions by their values.
e.g $2 * 3.14$ can be replaced by 6.28.
$22/7 = 3.142$

Commutative and associative laws
$x * y = y * x$

Code: $a = b + c$, $e = c + d + b$ → I.C: $a = b + c$, $t = c + d$, $e = t + b$

↓

$a = b + c$
$e = a + b$ optimized code.

(Q5b) ~~Using code~~ What is peephole optimization? Describe different transformations that are characteristics of Peephole Optimization. — 6 mks.

A simple but effective technique for locally improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence whenever possible.

The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous.

It is a characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements. In general, repeated passes over the target code are necessary to get the maximum benefit.

Following are the transformations that are characteristics of peephole optimization –

- Redundant – instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

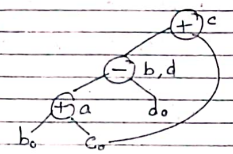
Redundant loads and stores –
Consider the following sequence of instructions
1. MOV R_0, a
2. MOV a, R_0

5.a) Explain optimization of basic blocks. —4 mks.

Many of the structure-preserving transformations can be implemented by constructing a dag for a basic block.

Common subexpressions can be detected by noticing, as a new node m is about to be added, whether there is an existing node n with the same children, in the same order, and with the same operator. If so n computes the same value as m and may be used in its place.

e.g. A DAG for the block
$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$ is

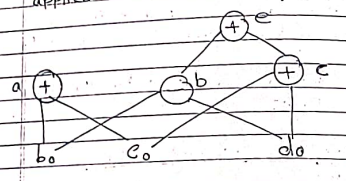


o/p optimized code is
$a = b + c$
$d = a - d$
$c = d + c$

In common subexpressions, we really are looking for expressions that are guaranteed to compute the same value, no matter how that value is computed.

$a = b + c$
$b = b - d$
$c = c + d$
$e = b + c$

DAG method will miss the fact that the expr. computed by the first and fourth statements in the sequence. However, algebraic ide... applied to the DAG can solve this prob...



The operation on DAGs that corresponds to dead code elimination is straightforward to implement. We delete from a DAG any (node with no ancestors) that has no live... Repeated application of this transformation remove all nodes from the DAG that com... to dead code.

Use of Algebraic Identities –
Algebraic identities represent another imp... class of optimizations on basic blocks. Following transformations can be tried duri... optimization —
$x + 0 = 0 + x = x$
$x - 0 = x$
$x * 1 = 1 * x = x$
$x / 1 = x$

Reduction in Strength — is replacing a more expensive operator by a cheaper one as follows –
$x * * 2 = x * x$
$2 \cdot 0 * x = x + x$ (Multn replaced by addition)
$x / 2 = x * 0.5$

Constant folding – Here we evaluate constant expressions at compile time and replace the constant expressions by their values.
e.g $2 * 3.14$ can be replaced by 6.28.
$22/7 = 3.142$

Commutative and associative Laws
$x * y = y * x$

C.E { $a = b + c$, $e = c + d + b$ } I.C { $a = b + c$, $t = c + d$, $e = t + b$ }

↓

$a = b + c$
$e = a + b$ optimized code.

(Q5b) ~~Using code~~ What is peephole optimization? Describe different transformations that are characteristics of Peephole Optimization. —6 mks.

A simple but effective technique for locally improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence whenever possible.

The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous.

It is a characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements. In general, repeated passes over the target code are necessary to get the maximum benefit.

Following are the transformations that are characteristics of peephole optimization –

- Redundant – instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Redundant Loads and Stores –
Consider the following sequence of instructions
1. MOV R_0, a
2. MOV a, R_0

We can delete instruction (2) because whenever (2) is executed, (1) will ensure that the value of a is already in register R_0.

Unreachable Code —
Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed.

```
# define debug 0
...
if( debug )
{
  print debugging information
}
```

We can also eliminate jumps over jumps.

Flow of Control Optimizations
I.C generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the I.C or T.C. by using peephole optimization.

```
goto L1
...
L1 : goto L2
```
→ can be replaced by `goto L2`

```
if a<b goto L1
...
L1 : goto L2
```
→ can be replaced by `if a<b goto L2`

Algebraic Simplification
There is no end to the amount of simplification that can be attempted through peephole optimization.
e.g $x = x + 0$ or $x = x * 1$
these can be eliminated easily through peephole optimization.

Reduction in Strength — It replaces expensive operations by equivalent cheaper ones on target machine.
Multiplication can be replaced by addition.
e.g $x^2 = x * x$

Use of Machine Idioms —
The target machine may have hardware instructions to implement certain specific operations efficiently.
e.g INCR, DCR
Push and Pop instructions.

Q6.) With reference to code generation, what is a basic block? Describe an algorithm to partition three address sequences into basic blocks. —5 mks

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

e.g. $t_1 = a * a$
$t_2 = a * b$
$t_3 = 2 * t_2$
$t_4 = t_1 + t_3$
$t_5 = b * b$
$t_6 = t_4 + t_5$

A three-address stmt $x = y + z$ is said to define x and to use (or reference) y and z.
A name in a basic block is said to be live at a given point if its value is used after that point in the program, perhaps in another basic block.

Following Algo. can be used to partition a sequence of three-address statements into basic blocks.

Input: A sequence of three-address statements
Output: A list of basic blocks with each three-address statement in exactly one block.

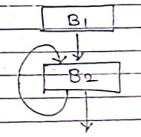
Method:-
1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are the following:-
   i) The first statement is a leader
   ii) Any statement that is the target of a conditional or unconditional goto is a leader.
   iii) Any statement that immediately follows a goto or conditional goto statement is a leader
2. For each leader, its basic block consists of the leader and all statements upto but not including the next leader or the end of the program.

Consider the following three-address code.

```
1  Prod = 0          ← Leader By Rule (i)
2  i = 1
3  t1 = 4 * i        ← Leader By Rule (ii)
4  t2 = a[t1]
5  t3 = 4 * i
6  t4 = b[t3]
7  t5 = t2 * t4
8  t6 = prod + t5
9  prod = t6
10 t7 = i + 1
11 i = t7
12 if i <= 20 goto (3)
13.                  ← Leader By rule iii
```

By rule iii statement following (12) is a leader

Basic Block 1 Has Stmts 1,2
" " 2 Has Stmts 3 to 12



Q6 Explain loops in flow graphs. — 5 mks.

A Loop is a collection of nodes in a flow graph such that

1. All nodes in the collection are strongly connected, that is, from any node in the loop to any other, there is a path of length one or more, wholly within the loop, and

2. The collection of nodes has a unique entry, that is, a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.

A loop that contains no other loops is called an inner loop.

One can use the notion of a node "dominating" another to define natural loops and the important special class of "reducible" flow graphs.

Dominators

We say node d of a flow graph dominates node n, written d dom n, if every path from initial node of the flow graph to n goes through d. Every node dominates itself and the entry of a loop dominates all nodes in the loop.

Dominator information can be presented in the form of a tree called Dominator Tree, in which initial node is the root and each node d dominates only its descendants in the t

The existence of dominator trees follows fr property of dominators, each node n has unique immediate dominator m that is last dominator of n on any path from initial node to n. In terms of the relation, the immediate dominator m has property that if $d \neq n$ and d dom n, t d dom m.

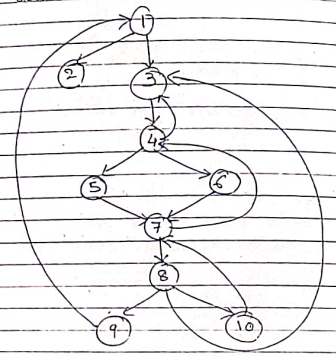
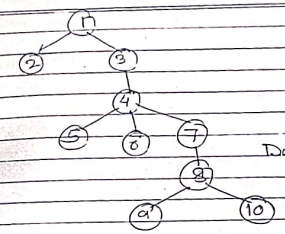


fig: flow

Dominator Tree for the above flow graph as follows —



Dominator Tree

Natural Loops

Important application of dominator inform. is in determining the loops of a flow graph suitable for improvement. There are two essential properties of such loops.

1. A loop must have a single entry point, called the "header". This entry point dominates all nodes in the loop or it would not be the sole entry to the loop.

2. There must be atleast one way to iterate the loop. i.e. atleast one path back to the header.

A good way to find all the loops in flow graph is to search for edges in the flow graph whose heads dominate their tails. If $a \to b$ is an edge, b is the head and a is the tail, we call such edges "back" edges

Given a back edge $n \to d$, we define the natural loop of the edge to be 'd' plus the set of nodes that can reach n, without going through d. Node d is the header of the loop.

eg in fig. there is a back edge from $7 \to 4$
$\therefore$ (4,5,7), (4,6,7) forms a loop and 4 is header.

Unless two loops have the same header, they are either disjoint or one is entirely contained (nested within) the other.

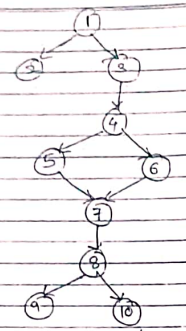
Hence inner loop, is one that contains no other loops.

A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, often called forward edges and back edges, with the following two properties

1. The forward edges form an acyclic graph in which every node can be reached from the initial node of G.

2. The back edges consists only of edges whose heads dominate their tails.

If we remove the back edges from the considered flow graph it looks like follows —



Since the above graph is acyclic after removing back edges, the graph is said to be reducible.

Whereas the following graph is not reducible

fig: Non-reducible flow graph.

Note: Shorten the above ans for 5 mks.

SF-977 Nov-2017 01-11-2017

Section I

Q1(a) How to specify and recognize tokens

Specification of Tokens —

Tokens are specified using regular expressions.

Regular expressions are written as Notations for regular languages.

Since lexical rules are simple structure, regular expressions are sufficient to specify the tokens.

R.E r denotes the language $L(r)$ and defined as follows —

1. ε is a regular expression that denotes lang. containing empty string.
2. If a is a symbol in Σ, then a is that denotes $L = \{a\}$ i.e. a set contains single string 'a'.
3. Suppose r and s are REs denoting and $L(s)$ then

(a) $(r+s)$ is R.E for $L(r) \cup L(s)$ Union

(b) $(r)(s)$ is R.E for $L(r) \cdot L(s)$ Concatenation

(c) $(r)^*$ is R.E denoting $(L(r))^*$ closure

(d) (r) is a R.E for $L(r)$

e.g. $\Sigma = \{a, b\}$ R.E $= (ab)^*a$

$(a)^* + (b)^*$

R.E. for an identifier in a P.L. can be written as letter (letter + digit)*

letter → A|B|...|Z|a|b|...|z

digit → 0|1|...|9

R.E. for unsigned numbers can be written as

digit → 0|1|...|9

digits → digit digit*

optional_fraction → . digit | ε

optional_exponent → (E (+|-|ε) digits) | ε

num → digits optional_fractions optional_exponent

Non-regular sets — Some lang. cannot be described by any R.E. They cannot count or compare. In such cases CFG can be used.

Recognition of Tokens —

Tokens can be recognized using Finite Automaton. Transition diagrams can be drawn and then converted into programs.

DFA is defined with 5 tuples = $\langle Q, \Sigma, \delta, q_0, A \rangle$

Q = set of states (finite)

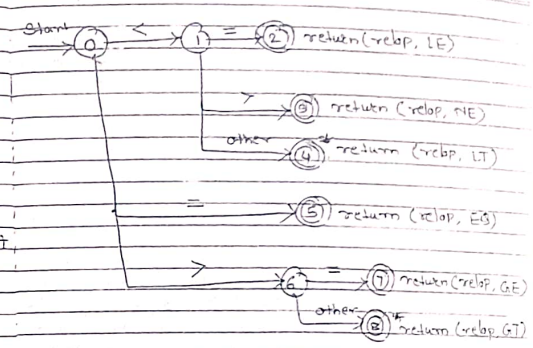
Σ = set of i/p alphabets (finite)

δ: Transition function $Q \times \Sigma \rightarrow Q$

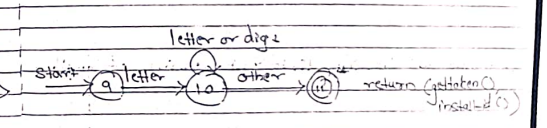
q_0 initial state (Unique state)

A = set of accepting states. $A \subseteq Q$

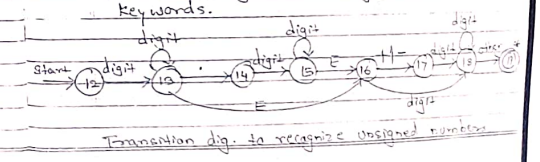
Following are some transition diagrams recognizing certain tokens —



Transition dig. to recognize relational operators



Transition diagram to recognize identifiers and keywords.



Transition dig. to recognize unsigned numbers

Such transition dig. are then converted into a program

```
int state=0, start=0
int lexical-value
int fail()
{
  forward = token_beginning;
  Switch (start)
  {
    Case 0:  start=9; break;
    Case 9:  start=12; break;
    Case 12: start=20; break;
    Case 20: recover(); break;
    default: Compiler error
  }
  return start;
}
```

Q1 a) List and describe different Cousins of Compiler — 6 mks.

The Input to a Compiler may be produced by one or more preprocessors and further processing of the Compiler's output may be needed before running machine code. The Lang processors which help to do this are called Cousins of Compiler.

Following are two of the Cousins of Compiler

1. Preprocessors
2. Assemblers
3. Loaders and Link Editors

1. Preprocessors — Preprocessors produce to Compilers. They may perform following functions —

i. Macro Preprocessing : It allows users define macros that are shorthands for constructs and also helps in its expansion.

ii. File inclusion : A preprocessor may header files into the program text. e.g. #include<stdio.h>

iii. Rational Preprocessors — These augment languages with more modern flow of and data-structuring facilities. eg. built-in Macros.

iv. Language extensions — These processors add capabilities to the lang. by what to built-in macros.
eg. Stmts beginning with ## are the preprocessor to be database stmt and are translated into procedures or routines that perform the data access.

2. Assemblers — Some compilers produce code that is passed to an assembler further processing. Other compilers perform the job of the assembler, producing relocatable m/c code that can be passed directly the loader/link editor.

Assembly code is the mnemonic version machine code.

Assemblers can be Single-pass, Two-pass Multi-pass.

3. Loaders and Link-Editors

A program called a Loader performs the two functions of loading and link-editing.

Loading consists of taking relocatable machine code, altering the relocatable addresses and placing the altered instructions and data in memory at the proper locations.

Linker allows us to make a single program from several files of relocatable m/c code. These files may have been result of several different compilations, one or more library files and the result of linking is a single Binary file where all unresolved addresses are resolved and the file is ready for execution.

Linker resolves the addresses of all External references in the obj. module. It is like finding the unknown addresses and putting them in the correct instruction where it is required.

2) Describe Tokens Patterns and Lexeme — 4 mks

The lexical analysis phase is said to produce tokens by matching the lexeme with the patterns.

There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

The pattern is said to match each string in the set.

A lexeme is a sequence of characters in the S.P that is matched by the pattern for a token.

Forex in the statement

```
const pi = 3.1416;
```

the substring pi is a lexeme for the token identifier.

Token	Sample Lexemes	Informal Description of Pattern
keywords	const	const
	if	if
operators	relation	<, <=, =, <>, >, >=	< or <= or = or <> or > or >=
identifiers	id	pi, count, D2	Letter followed by letters and digits
	num	3.1416, 0, 6, 6.02E23	any numeric constant

Tokens are the terminal symbols in the grammar for the source lang. In most PL keywords, operators, identifiers, constants, literal strings and punctuation symbols such as parentheses, commas, semicolons are treated as tokens.

A pattern is a rule describing the set of lexemes that can represent a particular token in source program.

e.g $x = y + z$; x, y, z are identifiers
(6 tokens are produced) =, + are operators
; punctuation symbol

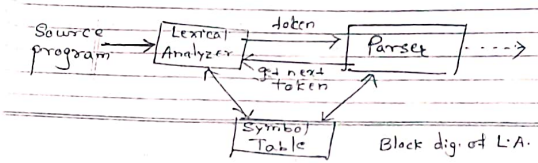
Tokens can have attributes to describe them.

e.g. E = M * C ** 2

	Token produced
E	<id, ptr to symbol-table entry for E>
=	<assign-op, >
M	<id, ptr to symbtable entry for M>
*	<Mult-op, >
C	<id, ptr to symbol table entry for C>
**	<exp-op, >
2	<Num, integer value 2>

Q2a) Role of Lexical Analyzer — 6 mks

Lexical Analyzer is the first phase of a Compiler. The main task is to read the i/p characters and produce as output a sequence of tokens that the parser uses for Syntax Analysis.



Block dig. of L.A.

LA is the part of Compiler that reads source text. Upon receiving a "get next token" command from the parser, the Lexical reads input characters until it can identify the next token. It produces tokens for lexemes. This is its primary function.

It also performs certain secondary tasks. For e.g. task is (1) Stripping out from the source comments and white space in the blank, tab, and newline characters.

(2) Correlating error messages from the compiler with the source program (SP).

(3) It may keep track of the number of newline characters seen, so that a number can be associated with an error message.

(4) May create copy of SP with error messages marked in it.

(5) Sometimes may also process the macros in SP.

Activities of Lexical Analyzer are divided into two phases — 1. Scanning (Simple)
2. Lexical Analysis. (Complex)

Issues in Lexical Analysis —

Following are the reasons for separating the analysis phase of Compiling into Lexical and Syntax analysis (Parsing)

1. Simpler Design — Separation of Lexical and from Syntax Analysis often allows us to simplify one or the other of these phases.

2. Compiler efficiency is improved — Separate lexical analyzer allows us to construct a specialized and potentially more efficient processor for the task.

Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler.

3. Compiler Portability is enhanced.

Lexical analyzer produces tokens. There is a set of strings in the i/p for which the same token is produced as output. This set of ~~rules~~ strings is described by a rule called a pattern associated with the token. A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

e.g. int x = 5;

int	keyword
x	identifier
=	operator
5	Numeric Constant
;	Punctuation Symbol.

Lexical analyzer collects information about tokens into their associated attributes.

x	<id, ptr to symbol table entry for x>
=	<assign-op, >

Q2a) What are the features of LR parser. Write the algo. for LR parsing — 6 mks

LR parsing is an efficient bottom-up parsing technique that can be used to parse a large class of Context-free grammars.
LR(k) L = stands for Left to right scanning of the input.
R = for right most derivation in reverse
k = No. of look ahead symbols.

Features of LR parser

1. It can be constructed to recognize, virtually all programming lang constructs for which CFG can be written.

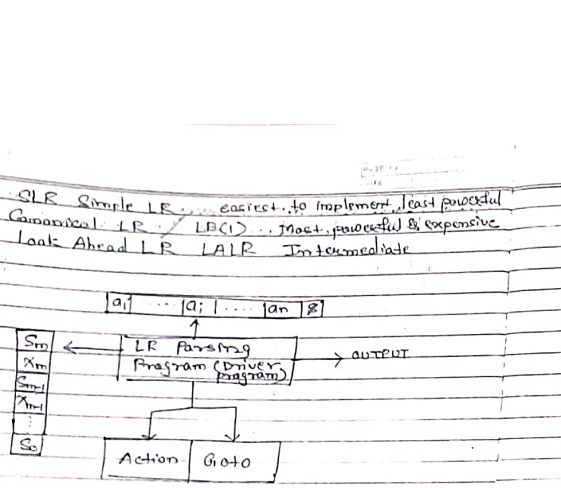
2. It is most general nonbacktracking shift-reduce parsing methods and can be implemented efficiently as other shift-reduce methods.

3. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

4. An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

We can use specialized tool — LR parser generator. There are three techniques for constructing LR parsing table.

SLR Simple LR ... easiest to implement, least powerful
Canonical LR / LR(1) .. Most powerful & expensive
Look Ahead LR LALR Intermediate



Block dig of an LR parser.

Algorithm of LR parsing

Input: String w and an LR parsing table with action goto for Grammar G.

Output: If w is in $L(G)$, a bottom-up parse for w otherwise an error indication

Method: Initially, the parser has S_0 on its stack where S_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the following program until an accept or error action is encountered.

Algorithm
set i/p to point to the first symbol o
repeat forever begin
let s be the state on top of the s
'a' the symbol pointed by i/p
if action[s, a] = shift s' then begin
push 'a' then s' on top of the stack
Advance ip to the next input symbol
end
else if action[s, a] = reduce $A \to \beta$ then be
pop $2*|\beta|$ symbols off the stack.
Let s' be the state now on top of the
push A then goto[s', A] on top of the s
output the production $A \to \beta$
end
else if action[s, a] = accept then
return
else error()
end

Q2 b.) Explain Removing Left Recursion and Left of Grammar with the help of examp — 4

Removing Left Recursion
A grammar is left recursive if it h nonterminal 'A' such that there is a der $A \overset{+}{\Rightarrow} A\alpha$ for some string α. Top-D parsing methods cannot handle left-rec grammars, so a transformation that el left recursion is needed.

Left Recursive Rule $A \to A\alpha \mid \beta$
Can be replaced by non-left recursive productions $A \to \beta A'$
$A' \to \alpha A' \mid \wedge$

e.g(1) $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

After removing left-recursion

$E \to TE'$
$E' \to +TE' \mid \wedge$
$T \to FT'$
$T' \to *FT' \mid \wedge$
$F \to (E) \mid id$

In general $A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

Replacement Algo
$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$
$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \wedge$

e.g(2.) $S \to Aa \mid b$
$A \to Ac \mid Sd \mid \wedge$

$S \to Aa \mid b$
$A \to Ac \mid Aad \mid bd \mid \wedge$
(Substitute value of S)

After removing Left recursion (indirect recursion)

$S \to Aa \mid b$
$A \to bdA' \mid A'$
$A' \to cA' \mid adA' \mid \wedge$

Left Factoring —
Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
The basic idea is that when it is not clear which of two alternative productions to use to expand a non-terminal A, we may be able to rewrite the A-productions to postpone the decision until we have seen enough of the input to make the right choice.

In general if $A \to \alpha\beta_1 \mid \alpha\beta_2$ are two productions then after left factoring the original productions become

$A \to \alpha A'$
$A' \to \beta_1 \mid \beta_2$

e.g. $S \to iEtS \mid iEtSeS \mid a$
$E \to b$

After left factoring
$S \to iEtSS' \mid a$
$S' \to eS \mid \wedge$
$E \to b$

Q3) Write an algo. for operator precedence parsing and explain. — 5 mks.

Operator Precedence parsing algorithm

Input: string w and a table of precedence

Output: If w is well formed, a skeletal parse tree otherwise an error indication.

Method: Initially, the stack contains $ and the input buffer the string w$. Following prog. is used for parsing –

```
Set ip to point to the first symbol of w$
repeat forever
    if $ is on tos and ip points to $ then
        return
    else begin
        Let a be the TOS terminal symbol on the stack
        and Let b be the symbol pointed to by ip.
    if a<·b or a≐b then begin
        push b onto the stack.
        advance ip to the next input symbol.
    end
    else if a·>b then        /* reduce */
        repeat
            pop the stack
        until the top stack terminal is related by <·
            to the terminal most recently popped.
    else error()
end
```

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

Consider the string id + id * id $

Stack	input	a(TOS)	b(IP)	Acti
$	id+id*id$	$	id	$<·
$ id	+id*id$	id	+	pop
$	+id*id$	$	+	$<·
$ +	id*id$	+	id	+<·
$ +id	*id$	id	*	pop
$ +	*id$	+	*	+<· *
$ + *	id$	*	id	*<·
$ + * id	$	id	$	Pop
$ + *	$	*	$	pop *
$ +	$	+	$	pop +
$	$	$	$	Accept

id id id * + ← postfix notation

Q.4 a) What are S attributed definitions? Explain with example. —6 mks.

A syntax-directed definition that uses synthe attributes exclusively is said to be an S-attrib definition.

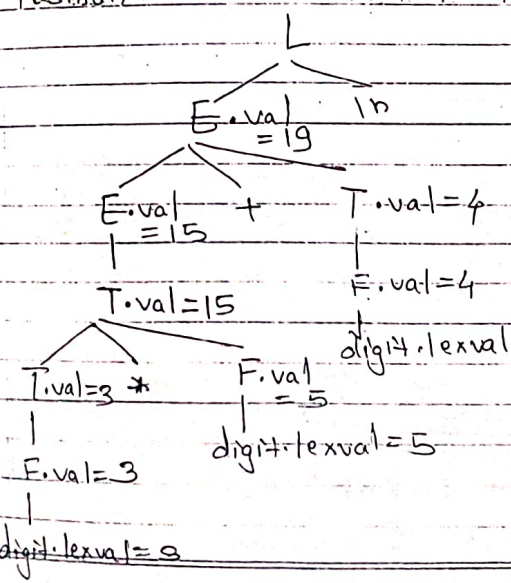
A parse tree for an S-attributed definit can always be annotated by evaluating the sem rules for the attributes at each node bottom from the leaves to the root.

Consider the following SDD which is a Simple Desk Calculator. It has two attributes val and lexval. Both are synthesized attribute Hence, it is an S-attributed definition.

$L \rightarrow E\ \backslash n$ { print(E.val) }
$E \rightarrow E_1 + T$ { E.val = E_1.val + T.val }
$E \rightarrow T$ { E.val = T.val }
$T \rightarrow T_1 * F$ { T.val = T_1.val * F.val }
$T \rightarrow F$ { T.val = F.val }
$F \rightarrow (E)$ { F.val = E.val }
$F \rightarrow$ digit { F.val = digit.lexval }

Since val is computed from the children ∴ it is a synthesized attribute.
lexval is associated with terminal symbol digit which has no child. But its value is provided by the lexical analyzer.

Consider the string 3*5+4 \n
The Annotated parse tree will be as follows —
The values are computed in Bottom-up Fashion



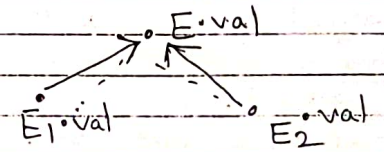
Q4 a) Define Dependency Graph. Describe methods for evaluating the semantic rules. — 6 mks.

The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph.

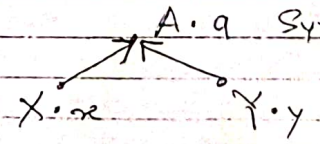
Before constructing a dependency graph for a parse tree, we put each semantic rule into the form $b = f(c_1, c_2, \cdots c_k)$ by introducing a dummy synthesized attribute b for each semantic rules that consists of a procedure call.

The graph has a node for each attribute and an edge to the node for b from the node c if attribute b depends on attribute c.

e.g. $E \rightarrow E_1 + E_2$ { E.val = E_1.val + E_2.val }



$A \rightarrow XY$
$A.a = f(X.x, Y.y)$ then dependancy graph is



Inherited attribute.

$A \rightarrow XY \quad X.i = g(A.a, Y.y)$ then the dependancy graph is

$A.a$

$X.i \leftarrow Y.y$

Algo. to construct dependency graph.

for each node n in the parse tree do
  for each attribute a of Grammar Symbol at n do
    Construct a node in dependency graph for a.

for each node n in the parse tree do
  for each semantic rule $b = f(c_1, c_2, \ldots, c_k)$ associated with the production used at n do
    for i = 1 to k do
      Construct an edge from the node for c_i to the node for b;

Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated. i.e. helps find the evaluation order of semantic actions.

Several Methods for evaluating semantic rules are —

1. Parse-tree methods — At Compile time, these methods obtain an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input. These methods will fail to find an evaluation order from a topologic sort based on dependency graph if the graph has a cycle.

2. Rule-based methods — At Compiler-Constr time, the semantic rules associated w productions are analyzed, either by hand by a specialized tool.

Inherited attribute.

$A \rightarrow XY \quad X.i = g(A.a, Y.y)$ the dependancy graph is

$A.a$

$X.i \leftarrow Y.y$

Algo. to construct

for each node
  for each a
    Constr.

for e

For each production, the order in which the attributes associated with that production are evaluated is predetermined at Compile Construction time.

3. Oblivious methods — An evaluation order is chosen without considering the semantic rules. If translation takes place during parsing then the evaluation order is forced by the parsing method, independent of the semantic rules.

Rule based and oblivious methods need not explicitly construct the dependency graph at compile time, so they can be more efficient in their use of compile time and space.

A syntax directed definition is said to be circular if the dependency graph for some parse tree generated by its grammar has a cycle.

Q4 b) What are the principle sources of code optimization? — 4 mks.

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block, otherwise it is called global.

Sources of optimization are —

1. Common Sub-expression elimination
2. Copy propagation
3. Dead-Code elimination

4. Constant folding
5. Loop optimizations
6. Code Motion
7. Induction variables and Reduction in strength

1. Common subexpressions —
An occurrence of an expression E is called a common subexpression if E was previously computed and the values of variables in E have not changed since the previous computation.
We can avoid recomputing the expression if we can use the previously computed value.

e.g

Block B5

Common subexpr	$t_6 = 4*i$
	$x = a[t_6]$
	$t_7 = 4*i$
	$t_8 = 4*j$
Common subexpr?	$t_9 = a[t_8]$
	$a[t_7] = t_9$
	$t_{10} = 4*j$
	$a[t_{10}] = x$
	goto B_2

(a) Before optimization

Block B5

$t_6 = 4*i$
$x = a[t_6]$
$t_8 = 4*j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B_2

(b) After Optimization

2. Copy Propagation —
Assignments of the form $f = g$ are called copy statements or copies for short.
The idea behind the copy-propagation transformation is to use g for f, wherever possible after the copy statement $f = g$.

e.g.

$a = d+e$	$b = d+e$	$t = d+e$; $a = t$;	$t = d$ $b =$
$c = d+e$		$c = t$	

(a) Before (b) After

3. Dead-Code Elimination —
A variable is live at a point in a p if its value can be used subsequently, otherwise, it is dead at that point.
Dead or useless code are statements that compute values that are never. While the programmer is unlikely to int any dead code intentionally, it may appe as the result of previous transformati

e.g.

if (debug) print

if we replace debug with false the pri stmt is dead because it cannot be re

4. Constant Folding
Knowing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

e.g. $x = 22/7$ can be replaced by $x = 3.142$

Loop Optimizations
Programs tend to spend the bulk of their time in loops. Hence, running time of a program may be improved if we decrease the no. of instructions in an inner loop.
Optimization techniques like code-motion (which moves code outside the loop), induction-variable elimination and reduction in strength (which replaces an expensive operation by a cheaper one e.g. multiplication by addition) are used for loop optimizations.

Q5 a) What is a basic block? Give an algo. to convert three address stmts into basic blocks with example — 6 mks.
Repeated

OR

a) What is Backpatching? Write SDD that uses backpatching for IC generation of Boolean expressions and flow of control statements. — 6 mks.
Repeated

Q5b) Explain various transformations on Basic Blocks — 4 mks

Structure Preserving Transformations
1. Common Subexpression elimination
2. Dead-Code elimination
3. Renaming of temporary variables
4. Interchange of two independent adjacent stmts

1. Common Subexpression elimination —
Consider the basic block
$a = b+c$
$b = a-d$
$c = b+c$
$d = a-d$
The second and fourth statements compute the same expression, namely $b+c-d$, and hence this block may be transformed into the equivalent block
$a = b+c$
$b = a-d$
$c = b+c$
$d = b$
Stmt no. 1 and 3 are not common subexpr as value of b gets changed in stmt 2.

2. Dead Code elimination :- Suppose x is dead, that is, never subsequently used, at the point where the stmt $x = y+z$ appears in a basic block. Then, this stmt may be safely removed without changing the value of the basic block.

3. Renaming temporary variables :—

Suppose we have a stmt $t = b + c$, where t is a temporary. If we change this stmt to $u = b + c$, where u is a new temporary variable, and change all uses of this instance of t to u, then the value of the basic block is not changed.

We can always transform a basic block into an equivalent block in which each stmt that defines a temporary defines a new temporary. We call such a basic block a normal-form block.

4. Interchange of statements —

Suppose we have a block with the two adjacent statements

$t_1 = b + c$

$t_2 = x + y$

Then we can interchange the two stmts without affecting the value of the block if and only if neither x nor y is t_1 and neither b nor c is t_2.

Algebraic Transformations —

Countless algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

e.g. $x = x + 0$ or $x = x * 1$

$x = y * * 2$ can be replaced by $x =$

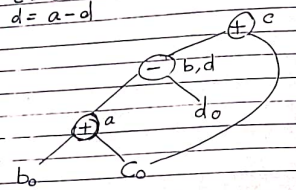
Q6 Draw DAG for basic block.

$a = b + c$

$b = a - d$

$c = b + c$

$d = a - d$



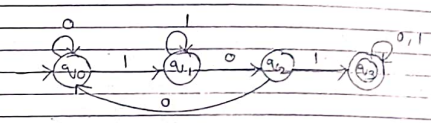
TE CSE Part-II Sem. VI Revised SV-201

May-2018 CC 3/5/2018 Marks 50

Section I

Q1a) Construct finite automata that will accept strings having the substring 101 where the lang. is defined on $\Sigma = \{0, 1\}$ — 4 mks

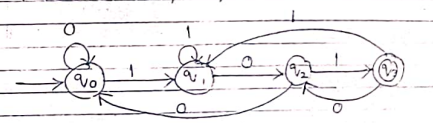
$L = \{101, 1101, 0101, 1011, 1010, 01010, 11011, 11010, 11011100, 101111, 10110, 10101, 00101, 1011000101, \ldots\}$



OR

Q1a) Construct FA accepting the set of all strings ending with 101 where $\Sigma = \{1, 0\}$ — 4 mks

$L = \{101, 1101, 0101, 10101, 01101, 000101, 1010101, 0101101, 00001111101, 1111000101, 1010010101010101, \ldots\}$



Q1 b) Enumerate and describe diff. Compiler Constr. tools — 6 mks.

Repeated.

Q2 a) Compute FIRST and FOLLOW sets for the grammar given below — 6 mks.

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

First(+) = {+} First {*} = *
First (() = {(} First ()) = {)}
First (^) = {^} First (id) = {id}

FIRST (E) = FIRST (TE') = FIRST (T) = {(, id}

FIRST (F) = FIRST ((E)) = {(}
FIRST (F) = FIRST (id) = {id}

$\boxed{\text{FIRST}(F) = \{(, id\}}$

FIRST (T') = FIRST (*FT') ∪ FIRST (^)
FIRST (*FT') = {*} FIRST (^) = {^}

$\boxed{\text{FIRST}(T') = \{*, \wedge\}}$

FIRST(T) = FIRST (F) = {(, id}

$\boxed{\text{FIRST}(T) = \{(, id\}}$

FIRST (E') = FIRST (+TE') ∪ FIRST (^)

$\boxed{\text{FIRST}(E') = \{+, \wedge\}}$

$\boxed{\text{FIRST}(E) = \{(, id\}}$

$\boxed{\text{FOLLOW}(E) = \{\$,)\}}$

$F \rightarrow (E)$
$A \rightarrow \alpha B \beta$
First (β) = First (()) = {)}

$\boxed{\text{FOLLOW}(E') = \{\$,)\}}$

$E \rightarrow TE'$ FIRST (β) = First (^) = {^}
$A \rightarrow \alpha B \beta$

Everything in FOLLOW (E) is in FOLLOW

$\boxed{\text{FOLLOW}(T) = \{+,), \$\}}$

$E \rightarrow TE'$ First (E') = {+, ^}
$A \rightarrow \alpha B \beta$

$E' \rightarrow +TE'$ First (β) = {+, ^}
$A \rightarrow \alpha B \beta$ First (E')

$\boxed{\text{FOLLOW}(T') = \{+,), \$\}}$

$T \rightarrow FT'$ First (β) = First (^) = {^}
$A \rightarrow \alpha B \beta$

FOLLOW (T) is put in FOLLOW (T')

$\boxed{\text{FOLLOW}(F) = \{*, +,), \$\}}$

$T' \rightarrow *FT'$ First (β) = First (T') = {*, ^}
$A \rightarrow \alpha B \beta$

Everything in FOLLOW (T') is in FOLLOW (F)

2 b) What is bottom-up parsing? Explain the shift-reduce bottom-up parsing algorithm. — 4 mks.

Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves (at bottom) and working up towards the root (the top). We can think of this process as one of "reducing" a string w to the start symbol of a grammar.

At each reduction step a particular substring matching the RHS of a production is replaced by the symbol on the left of that production, and if the substring is chosen correctly at each step, a rightmost derivation is traced out in reverse.

Bottom-up parser performs following primary operations —

1. Shift action — next i/p symbol is shifted onto the top of the stack.

2. reduce action — as soon as the handle that matches RHS of production appears on the TOS, it is popped out and nonterminal on LHS is pushed in.

3. accept action — parser announces successful completion of parsing.

4. error action — parser discovers that a syntax error has occurred and calls an error recovery routine.

Consider the Grammar
$E \rightarrow E+E \mid E*E \mid E-E \mid E/E \mid (E) \mid -E \mid id$

String : id + id * id $
state input Action
First we identify the handles by doing a Rightmost derivation.

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

Stack implementation is as follows —

Stack	input	Action
$	id+id*id$	shift
$ id	+id*id$	reduce E→id
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce E→id
$E+E	*id$	shift

Stack	Input	Action
$ E + E *	id $	shift
$ E + E * id	$	reduce E → id
$ E + E * E	$	reduce E → E * E
$ E + E	$	reduce E → E + E
$ E	$	Accepted.

Q2 b Explain different error recovery techniques used in Syntax Analysis. — 6 mks

There are many different strategies that a parser can employ to recover from a syntactic error. Although no one strategy has proven itself to be universally acceptable.
Following are the error recovery strategies:

1. Panic mode
2. Phrase level
3. Error productions
4. Global corrections

1. Panic mode — It is simplest method to implement and can be used by most parsing methods.
On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon, comma, space etc.
While panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has advantage of simplicity.
It is guaranteed not to go into an infinite loop. In situations where multiple errors in the same statement are rare, this method may be quite adequate.

2. Phrase level Recovery — On discovering an error, a parser may perform local correction on the remaining input that it may replace a prefix of the remaining input by some string that allows the parser to continue.
e.g. replace a comma by a semicolon,
delete an extraneous semicolon,
insert a missing semicolon.

But care must be taken that it does not enter into an infinite loops. This method was first used with Top-Down Parsing. But its major difficulty, it has in coping with situation in which the actual error has occured before the point of detection.

3. Error productions —
If we have a good idea of the common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. We then use the grammar augmented by these error productions to construct a parser. If an error production is used by the parser, we can generate appropriate error diagnostics to indicate the erroneous construct that has been recognized in the input.

4. Global Correction —
Ideally, we would like a compiler to make as few changes as possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction.
Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y, such that the number of insertions, deletions and changes of tokens required to transform x into y is as small as possible.
Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.

Q3 Explain input buffering used in lexical Analysis. — 5 mks.

Repeated.

Q4 a) With respect to Intermediate code generation explain different forms of I.C. Languages. — 6 mks.

Intermediate code languages can be one of the following forms —
1. Syntax Trees (Graphical Representation)
2. Postfix notations (Linear Representation)
3. Three-address code (Linearized Representation)

1. Graphical representations —
Syntax tree depicts the natural hierarchical structure of a source program. A DAG gives the same information but in a more compact way because common subexpressions are identified.
consider the expression
e.g. a = b * -c + b * -c

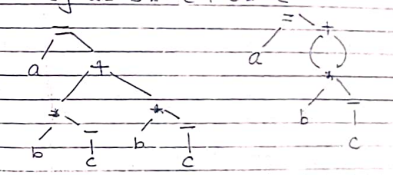


fig (a) Syntax tree (b) DAG

2. Postfix notation —
It is a linearized representation of a syntax tree. It is a list of the nodes of the tree in which a node appears immediately after its children.

e.g. for the expression $a = b * -c + b * -c$

$a\ b\ c\ - *\ b\ c\ - *\ + =$

The edges in a syntax tree do not appear explicitly in postfix notation.
Edges can be recovered using a stack

3. Three-Address Code —
It is a sequence of statements of the general form $x = y\ op\ z$
where x, y and z are names, constants, or compiler-generated temporaries. op stands for any operator. Only one operator is permitted on on the RHS.

Thus the source language expression like $x + y * z$ is translated into three address code as
$t_1 = y * z$
$t_2 = x + t_1$

where t_1 and t_2 are compiler-generated temporary names.
This simplification of complicated arithmetic expressions and of nested flow-of-control stmts makes three-address code desirable for target code generation and optimization.

Three-address code is linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.

e.g. Consider the DAG

then its corresponding three-address code is

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = t_2 + t_2$
$a = t_3$

The reason for the term "Three Address" is that each statement usually contains three addresses, two for the operands and one for the result.

Q4 a) What are L attributed definitions? Ex with the help of an example. —6m

L-Attributed Definitions —
A SDD is L-attributed if each inherited attribute of X_j, $1 \le j \le n$, on the right side of $A \to X_1 X_2 \cdots X_n$ depends only on

1. the attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$ to the left of X_j in the production and
2. the inherited attributes of A.

Note: Every S-attributed definition is L-attributed, because the restrictions (1) and (2) apply only to inherited attributes.

Following is an example of L-attributed definition.

$D \to T\ L$	$\{L.in = T.type\}$
$T \to int$	$\{T.type = int\}$
$T \to real$	$\{T.type = real\}$
$L \to L_1, id$	$\{L_1.in = L.in;\ addtype(id.entry, L.in)\}$
$L \to id$	$\{addtype(id.entry, L.in)\}$

The inherited attribute "In" is computed from the sibling on the left or parent. ∴ it is L-attributed definition.
The rest are synthesized attribute.

Consider the following SDD which is not an example of L-attributed definition.

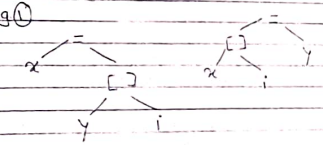
$A \to LM$	$L.i = l(A.i)$
	$M.i = m(L.s)$
	$A.s = f(M.s)$
$A \to QR$	$R.i = r(A.i)$
	$Q.i = q(R.s)$
	$A.s = f(Q.s)$

Above SDD is Non L-attributed definition because the inherited attribute $Q.i$ of the grammar symbol Q depends on the attribute $R.s$ of the grammar symbol to its Right.

Q4 b) How are array indexing instructions represented by DAGs? Explain with the help of example. —4 mks

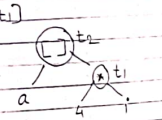
Indexed assignments are of the form $x = y[i]$ and $x[i] = y$. The first of these sets x to the value in the location i memory units beyond location y. The statement $x[i] = y$ sets the contents of the location i units beyond x to the value of y.
DAG can be used to represent it as follows—

eg ①



DAG for $x = y[i]$ DAG for $x[i] = y$

e.g. ② $t_1 = 4 * i$
$t_2 = a[t_1]$



1. Any evaluation of or assignment to an element of array 'a' must follow the previous assignment to an element of that array if there is one.

3. Any assignment to an element of array 'a' must follow any previous evaluation of 'a'.

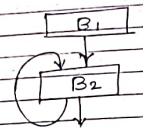
Q5 a) Write a short note on flow graph representation of basic blocks. — 6 mks.

We can add the flow-of-control information to the set of basic blocks making up a program by constructing a directed graph called a flow graph. The nodes of the flow graph are the basic blocks. One node is distinguished as initial. It is a block whose leader is the first statement. There is a directed edge from block B_1 to Block B_2 if B_2 can immediately follow B_1 in some execution sequence, that is, if

1. there is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2, or

2. B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump.

We say B_1 is a predecessor of B_2, B_2 is a successor of B_1.

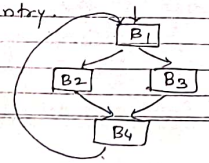


It is important to note that an edge in the flow graph from block B to block B' does not specify the conditions under which control flows from B to B'. This info can be recovered when needed from the jump statement in B.

Loops — Loop is a collection of nodes in a flow graph such that

1. All nodes in the collection are strongly connected, that is, from any node in the loop to any other, there is a path of one or more, wholly within the loop, and

2. The collection of nodes has a unique entry, that is, a node in the loop such that the only way to reach a node of the loop from a node outside the loop, is to first go through the entry.



b) What is peephole optimization? Describe diff. characteristics of peephole optimization — 6 mks

Repeated.

b) Discuss different issues in design of code generator? — 4 mks

The generic issues in the design of code generators are —

1. Input to the Code Generator
2. Target Programs
3. Memory Management
4. Instruction Selection
5. Register Allocation
6. Choice of Evaluation Order

1. Input to the Code Generator —

Input to the code generator consists of the Intermediate representation of the source program produced by Front End, together with the information in the symbol table that is used to determine the run-time addresses.

It is necessary that the front end has scanned, parsed and translated the S.P. into a detailed I.C., so the values of the names are represented by quantities that the target m/c can directly manipulate (bits, bytes, int, etc)

Type checking, type-conversions and semantic errors have to be detected before giving I.C as an i/p to T.C generator.

I.C. should be free of errors.

2. Target Programs :-

T.P can be in the form of Absolute m/c lang., relocatable m/c lang or Assembly lang.

- Absolute m/c lang program as an output has the advantage that it can be placed in a fixed location in memory and immediately executed. A small program can be compiled and executed quickly.

- Relocatable m/c Lang. program (object module) allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. Flexibility is the major advantage here.

- Assembly Lang. prog. as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help generate T.C.

3 Memory Management —

Mapping names in the source program to addresses of data objects in runtime memory is done cooperatively by the front end and

the code generator. If m/c code is being generated, labels in three-address stmts. have to be converted to addresses of instructions.

4. Instruction Selection —

The nature of the instruction set of the target machine determines the difficulty of instruction selection. If the target m/c does not support each data type in a uniform manner, then each exception to the general rule requires special handling.

Instruction speeds and machine idioms are other important factors.

Consider the three address code

a = b + c
d = a + e } the corresponding T.C. will be

1 Mov b, R0
2 ADD C, R0
3 Mov R0, a
4 Mov a, R0
5 ADD e, R0
6 Mov R0, d

Here stmt. No 4 is redundant. and so is 8.

The quality of the generated code is determined by its speed and size.

For example if three-address stmt is a = a + 1 then the corresponding T.C will be

Mov a, R0
ADD #1, R0 } But instead of using target instructions we can a single INCR instruction
Mov R0, a

INCR a

But deciding which m/c code sequence is for a given three-address code

5. Register Allocation

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore efficient utilization of registers is particularly important in generating good code. The use of registers is subdivided into two subproblems —

1. During register allocation, we select the set of variables that will reside in registers at a point in the program.

2. During a register assignment phase, we pick the specific register that a variable will reside in.

Register Allocation and assignment is an NP hard problem

6. Choice of Evaluation Order — The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. But these algorithms are NP

Q6) What is a Basic Block? Describe an algo. to partition a three address sequences into basic blocks? — 5 mks.

Repeated.

TE CSE Part-II Sem VI Exam Nov 2018
Compiler Construction (Revised)
12-11-2018 50 mks Sub Code: 66352
SE-205

Section I

Q1a) List and describe different Causes of Compiler — 4 mks. Repeated

OR

Q1a) Write a note on grouping of phases and passes. — 4 mks

In an implementation, activities from more than one phase are often grouped together.

Front and Back Ends —

Front End consists of those phases that depend on the source lang. and are largely independent of the target m/c.

↓ SP

Front End

Lexical Analysis → Error Handling
Syntax Analysis
Semantic Analysis → Symbol Table formation

↓ I.C generation

Back end includes those portions that depend on the target m/c

↓ I.C

Back End

Code optimization
Target Code generation

↓ TC

One can use the same front end and only redo its associated Back end to produce a compiler for the same source lang on a different m/c.

Sometimes we can also compile several different languages into the same Intermediate lang. (I.C) and then use a common back end for the different front ends.

Passes

Several phases of compilation are usually implemented in a single pass consisting of reading an i/p file and writing an o/p file. It is common for several phases to be grouped into one pass and for the activities of these phases to be interleaved during the pass.

e.g. Lexical, Syntax, Semantic — Analysis All in one pass phase

Two pass Translation — Optimization, Code gen? — Synthesis In one pass phase

OR

We can also have three passes by separating lexical analysis as it is slowest phase.

Lexical Analy — Pass I

Syntax + Semantic Analysis — Pass II

Back End Activities — Pass III

T.C.

Reducing the Number of Passes —

It is desirable to have relatively few passes since it takes time to read and write intermediate files.

But on the other hand, if we group several phases into one pass, we may be forced to keep the entire program in memory.

For some phases, grouping into one pass presents few problems — as follows: —

1. the interface betw^n Lexical and syntactic analysis are often limited to a single token for synchronization
2. It is often hard to perform T.C. generation until I.R. has been completely generated.
3. There could be a problem of forward reference which then needs backpatching.

Thus based on time and space complexity one must choose appropriate number of passes.

Q1 b) Explain input buffering in Lexical Analysis 6 mks. Repeated.

Q2 a) Explain Role of Lexical Analyzer — 6 mks Repeated.

OR

Q2 (b) What is operator precedence grammar?

The largest class of grammars for which shift-reduce parsers can be built successfully are LR parsers. However, for a small but important class of grammars we can easily construct efficient shift-reduce parsers by hand. These grammars have the property that no production Right side is ∧ (null)/ε or has two adjacent nonterminals. A grammar with the latter property is called as Operator Grammar.

e.g. $E \to E+E \mid E-E \mid E/E \mid E*E \mid (E) \mid -E \mid id$

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

Write algo. for operator precedence parsing Repeated.

12 b) $S \to iEtS \mid iEtSeS \mid a$
$E \to b$

Is the grammar ambiguous? If yes then remove the ambiguity — 4 mks.

Yes the grammar is ambiguous.

Unambiguous Grammar
$S \to iEtSS' \mid a$
$S' \to eS \mid \wedge$
$E \to b$

Q3 Construct LR(1) parsing table for the following grammar — 5 mks.
$S \to CC$
$C \to cC$
$C \to d$
Repeated

Section-II

Q4 (a) Explain with example:
i) inherited attributes
ii) Synthesized Attributes — 6 mks

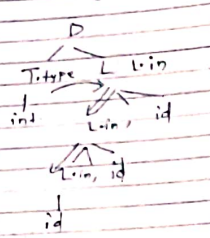
i) Inherited Attributes —
The value of an attribute at a parse-tree node is defined by a semantic rule associated with the production used at that node.
The value of an inherited attribute is computed from the values of attributes at the siblings and/or parent of that node.
Inherited attributes are convenient for expressing the dependencies. Although it is always possible to rewrite a SDD to use only synthesized attributes, it is often more natural to use SDD with inherited attributes.

Example — of Inherited attribute.

$D \to TL$	{L.in = T.type}
$T \to int$	{T.type = integer}
$T \to real$	{T.type = real}
$L \to L, id$	{L1.in = L.in, addtype(id.entry, L.in)}
$L \to id$	{addtype(id.entry, L.in)}

Fig: SDD for Declarative Stmt. showing inherited attribute in

Type and entry are synthesized attribute. L.in is an inherited attribute as its value is computed from its parent and/or sibling.



ii) Synthesized Attribute —
The value of a synthesized attribute at a node is computed from the values of the attributes at the children of that node in the parse tree.

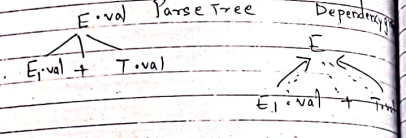
A syntax directed definition that uses synthesized attributes exclusively is said to be an S-attributed definition.

A parse tree for an S-attributed def^n can always be annotated by evaluating the semantic rules for the attributes at each node in bottom up, from the leaves to the root.

Following is an example of SDD having synthesized attributes named val and lexval. Its value is computed from attributes of children

$L \rightarrow E\ln$	{print (E·val)}
$E \rightarrow E_1 + T$	{E·val = E_1·val + T·val}
$E \rightarrow T$	{E·val = T·val}
$T \rightarrow T_1 * F$	{T·val = T_1·val * F·val}
$T \rightarrow F$	{T·val = F·val}
$F \rightarrow (E)$	{F·val = E·val}
$F \rightarrow$ digit	{F·val = digit·lexval}

fig. SDD of a Simple Desk Calculator



Q4 (a) Define Dependency Graph. Describe methods for evaluating the semantic rules. — 6 mks.
Repeated.

Q4 (b) What are the principle sources of code optimization
Repeated.

Q5 (a) Write the code generation algorithm and write of the function getreg(). — 6 mks

Code generation Algorithm —
The code generation algorithm takes as input a sequence of three-address stmts. constituting a basic block. For each three-address stmt of the form $x = y$ op z we perform the foll. actn

1. Invoke a function getreg to determine the location L where the results of the computation y op z should be stored. L will usually be a register, but it could also be a memory location.

2. Consult the address descriptor for y to determine y' (the current location of y). Prefer register for y', if the value of y is currently in both memory and register. If value of y is not already in L, generate the instruction MOV y', L to place a copy of y in L.

3. Generate the instruction OP z', L where z' is a current location of z. Again prefer a register to a memory location, if z is in both. Update the address descriptor for x to indicate that x is in location L. If L is a register, update its descriptor to indicate that it contains the value of x, and remove x from all other register descriptors.

4. If the current value of y and/or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of $x = y$ op z, those registers no longer will contain y and/or z, respectively.

If three-address stmt. has a unary operator, the steps are analagous to those above.

If the stmt is $x = y$ and if y is in reg. then simply change the register and address descriptor to record that the value of x is now found only in register holding the value of y.

The Function getreg
The function getreg returns the location L to hold the value of x for the assignment $x = y$ op z.

1. If the name y is in a register that holds the value of no other names and y is not live and has no next use after execution of $x = y$ op z, then return the register of y for L. Update the address descriptor of y to indicate that y is no longer in L.

2. Failing (1), return an empty register for L if there is one.

3. Failing (2), if x has a next use in the block or op is an operator such as indexing, that requires a register, find an occupied register R. Store the value of R into a memory location, if it is not already in the proper memory location M, update the address descriptor for M and return R.

If R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored. A suitable occupied register might be one whose datum is referenced furthest in the future.

or one whose value is also in memory.

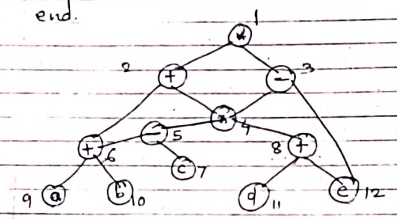
4. If x is not used in the block, or no suitable occupied register can be found, then select the memory location of x as L.

Q5 (a) With respect to generating code from DAGs explain node listing algorithm and labelling algo — 6 mks.

Ordering for DAGs :—

Node listing Algorithm — (Optimal Order of Eval^n)

(1) While unlisted nodes remain do begin
(2) Select an unlisted node 'n', all of whose parents have been listed;
(3) list 'n'
(4) While the leftmost child m of n has no unlisted parents and is not a leaf do begin
(5) list m;
(6) n = m;
end
end.



According to node listing algo. the resulting sequence of nodes will be

1, 2, 3, 4, 5, 6, 8

So the suggested order of evaluation is 8, 6, 5, 4, 3, 2, 1

Hence, the sequence of exec^n will be

$t_8 = d + e$
$t_6 = a + b$
$t_5 = t_6 - c$
$t_4 = t_5 * t_8$
$t_3 = t_4 - e$
$t_2 = t_6 + t_4$
$t_1 = t_2 * t_3$

This sequence will yield an optimal code for the DAG.

Optimal Ordering for Trees —

The Labelling Algorithm — it has two parts

1. First part labels each node of the tree in bottom-up sequence with an integer that denotes the fewest number of registers (RR) required to evaluate the tree without storing of intermediate results.

2. Second part of the algo. is a tree traversal whose order is governed by the computed node labels. The o/p code is generated during the tree traversal.

If the register requirements of both operands are the same, either operand can be evaluated first.

Algo.
(1) if n is a leaf then
(2) if n is the leftmost child of its parent then
(3) label (n) = 1
(4) else label (n) = 0
(4) else begin /* n is an interior node */
(5) let $n_1, n_2, \ldots n_k$ be the children of n ordered by label,
So $label(n_1) \geq label(n_2) \geq \ldots \geq label(n_k)$.
(6) $label(n) = \max_{1 \leq i \leq k} (label(n_i) + i - 1)$
end

If all operators are binary i.e. if tree is binary then

$$label(n) = \begin{cases} \max(l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1 + 1 & \text{if } l_1 = l_2 \end{cases}$$

Consider the Binary tree.

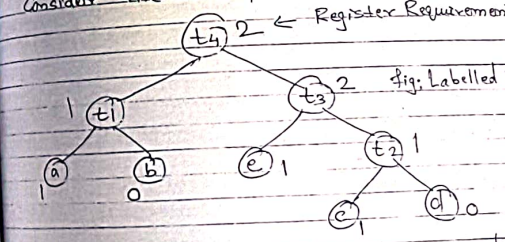


fig: Labelled Tree

Postorder is always an appropriate order in which to do the label computation

a b t1 e c d t2 t3 t4

Q5 (b) What are the criteria for code improving transformations? — 4 mks

Criteria for Code improving transformations —

The best program transformations are those that yield the most benefit for the least effort.

The transformations provided by an optimizing compiler should have several properties (criterias) —

1. First a transformation must preserve the meaning of programs. That is an optimization must not change the o/p produced by a program for a given i/p.

2. Second, a transformation must, on the average, speed up programs by a measurable amount. Occasionally an "optimization" may slow down a program slightly.

3. Third, a transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed.

1. Some transformations can only be applied after detailed, time consuming, analysis of the source program, so there is little point in applying them to programs that will be run only a few times.

e.g. A fast non-optimizing compiler is likely to be more helpful during debugging or for student jobs (programs) that will be run successfully a few times and thrown away.

Q6 What are register and address descriptors. Generate target code sequence for following three-address code along with register and address descriptor. — 5 mks.

Register and Address Descriptors —
The code generation algo uses descriptors to keep track of register contents and addresses for names —

1. A register descriptor keeps track of what is currently in each register. It is consulted whenever a new register is needed. We assume that initially the register descriptor shows that all registers are empty. As the code generation for the block progresses, each register will hold the value of zero or more names at any given time.

2. An Address descriptor keeps track of the location (or locations) where the current value of the name can be found at run time. The location might be a register, a stack location, a memory address or some set of these. This information can be stored in the symbol table and is used to determine the accessing method for a name.

Target code sequence for the given code.

t = a+b
u = a−c
v = t+u
d = v+u

t, u, v are temporaries and we assume they are not in memory unless we explicitly store their values with MOV instr.

Statement	Code Generated	Register Descriptor	Address Descriptor
		Registers Empty	
t = a+b	MOV a, R_0 ADD b, R_0	R_0 contains t	t in R_0
u = a−c	MOV a, R_1 SUB c, R_1	R_0 contains t R_1 contains u	t in R_0 u in R_1
v = t+u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d = v+u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory.

TE(CSE) Part - II Sem - VI Revised
May 2019 Compiler Construction
SV-206 Sub Code: 66858 Marks: 50

Section - I

Q1(a) Describe the use of DFA for a construction of a lexical analyzer. Illustrate with example — 4 mks.

Note:- The question is same as "Explain how to recognize the tokens?". Hence, write the same answer which is already written. The question is simply made tricky.

Q1(a) Construct DFA for all string that end with 101. $\Sigma = \{0,1\}$ — 4 mks.

Repeated.

Q1(b) Enumerate and describe diff. Compiler Construction Tools — 6 mks.

Repeated.

Q2(a) Compute FIRST and FOLLOW sets for the grammar given below — 6 mks.

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

⇒ Repeated.

Q2 (a) Explain different error-recovery techiques used in Syntax Analysis — 6 mks

⇒ Repeated

Q2 (b) Explain Specification of Tokens in lexical Analysis — 4 mks

⇒ Repeated.

Q3. What are features of LR parser? Write the LR parsing algorithm. — 5 mks.

⇒ Repeated.

Section II

Q4 (a) What are S-attributed definitions? Explain with the help of an example. — 6 mks.

⇒ Repeated

OR

Q4 (a) What is backpatching? Explain it with a suitable SDD — 6 mks.

⇒ Repeated

OR

Q4 (a) What is backpatching? Explain with the help of a suitable SDD. — 6 mks.

⇒ Repeated.

Q4 (b) Generate three-address code for the following block: — 4 mks

a = b * (c+d)
e = ((a+b) * (a+c))

(i) a = b * (c+d)

Corresponding three-address code.

$t_1 = c+d$
$t_2 = b * t_1$
$a = t_2$

(ii) e = ((a+b) * (a+c))

$t_1 = a+b$
$t_2 = a+c$
$t_3 = t_1 * t_2$
$e = t_3$

Q5 (a) What is a dominator Tree? Draw the dominator tree for the given flow graph. — 6 mks.

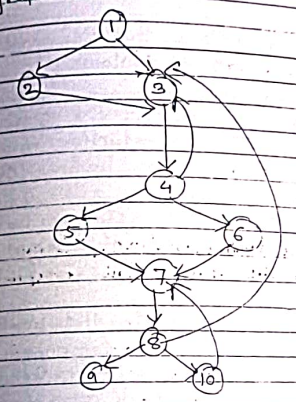
We say node d of a flow graph dominates node n, written d dom n, if every path from initial node of the flow graph to 'n' goes through d. Every node dominates itself and the entry of a loop dominates all nodes in the loop.

Dominator information can be presented in the form of a tree called Dominator Tree, in which initial node is the root and each node d dominates only its descendants in the tree.

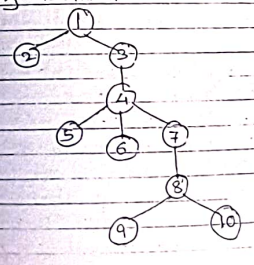
The existence of dominator trees follows from a property of dominators, each node n has a unique immediate dominator m that is the last dominator of n on any path from the initial node to n.

In terms of dom relation, the immediate dominator m has that property that if $d \neq n$ and d dom n, then d dom m.

Flow graph



Corresponding Dominator Tree



Q5 (a) What is peephole optimization? Describe different characteristics of peephole optimization — 6 mks

⇒ Repeated

Q5 (b) Discuss different issues in design of Code generator — 4 mks

⇒ Repeated

Q6 What is a basic block? Draw DAG for the given basic block — 5 mks.

a = b+c
b = b-d
c = c+d
e = b+c

⇒ Repeated

Compiler Construction

Nov. 2019 SC-163

Q5 (a) What are the rules for writing the semantic actions in an SDT when we have both inherited and synthesized attributes. Explain how to convert SDD into SDT with the help of an example. — 6 mks.

Rules for writing the semantic actions in an SDT are as follows —

1. An inherited attribute for a symbol on the right side of a production must be computed in an action before that symbol.

2. An action must not refer to a synthesized attribute of a symbol to the right of the action.

3. A synthesized attribute for the nonterminal on the left can only be computed after all attributes it references have been computed. The action computing such attributes can usually be placed at the end of the right side of the production.

$S \to B \quad \{B.ps = 10;\ S.ht = B.ht\}$

$B \to B_1 B_2 \quad \{B_1.ps = B.ps,\ B_2.ps = B.ps,\ B.ht = max(B_1.ht, B_2.ht)\}$

$B \to B_1\ sub\ B_2 \quad \{B_1.ps = B.ps,\ B_2.ps = Shrink(B.ps),\ B.ht = disp(B_1.ht, B_2.ht)\}$

$B \to text \quad \{B.ht = text.h \times B.ps\}$

fig. SDD for size and height of Boxes.

Translation scheme for the same SDD, which is converted using the rules.

$S \to \{B.ps = 10\}\ B\ \{S.ht = B.ht\}$

$B \to \{B_1.ps = B.ps\}\ B_1\ \{B_2.ps = B.ps\}\ B_2\ \{B.ht = max(B_1.ht, B_2.ht)\}$

$B \to \{B_1.ps = B.ps\}\ B_1\ sub\ \{B_2.ps = shrink(B.ps)\}\ B_2\ \{B.ht = disp(B_1.ht, B_2.ht)\}$

$B \to text\ \{B.ht = text.h \times B.ps\}$

ps (point size) is the inherited attribute

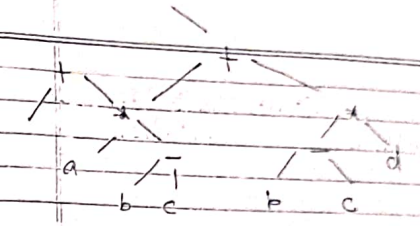
ht (height) is a synthesized attribute associated with non-terminal B.

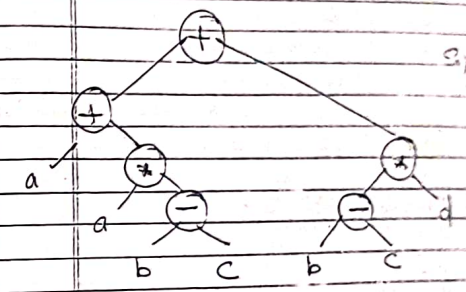
h (height) is synthesized attribute associated with terminal symbol text.

Draw syntax tree and DAG for the expression and also write the intermediate code in three address code format.

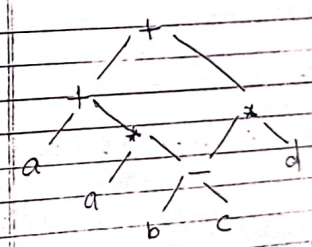
$a + a * (b - c) + (b - c) * d$

Repeated





Syntax Tree



DAG

Three address code from DAG.

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = t_1 * d$

$t_4 = a + t_2$

$t_5 = t_4 + t_3$

Few Additional Practice Questions

Q. Find First and Follow sets for the following grammar — 5 mks.

$S \to aBDh$
$B \to cC$
$C \to bC \mid \wedge$
$D \to EF$
$E \to g \mid \wedge$
$F \to f \mid \wedge$

First(S) = {a}
First(B) = {c}
First(C) = {b, ∧}
First(D) = {f, g, ∧}
First(E) = {g, ∧}
First(F) = {f, ∧}

① Follow(S) = {$}

② Follow(B) = {g, f, h}

$S \to aBDh$
$A \to \alpha B \beta$

First(β) = First(Dh) = First(D) = {f, g, ∧} ∪ First(h) = {f, g, h}

③ Follow(C) = {g, f, h}

$B \to cC$
$A \to \alpha B \beta$

First(β) = First(∧) = {∧}

∴ Follow(B) is in Follow(C)

④ Follow(D) = {h}

$S \to aBDh$
$A \quad \alpha \quad B \quad \beta$

First(β) = First(h) = {h}

⑤ Follow(E) = {f, h}

$D \to EF$
$A \to \alpha B \beta$

First(β) = First(F) = {f, ∧}

Follow(D) is in Follow(E)

⑥ Follow(F) = {h}

$D \to EF$
$A \to \alpha B \beta$

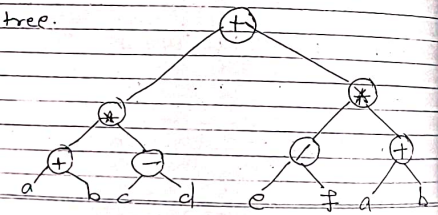
First(β) = First(∧) = {∧}

Follow(D) is in Follow(F)

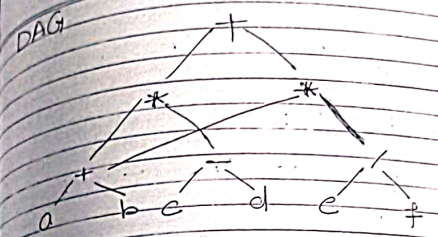
Q. Draw DAG and Syntax Tree for the given expression —

(a+b) * (c−d) + ((e/f) * (a+b))

⇒ Syntax tree.



DAG



Q. Find First and Follow sets for the following grammar

$S \to iEtSS' \mid a$
$S' \to eS \mid \wedge$
$E \to b$

First(S) = First(iEtSS') ∪ First(a)
= {i, a}

First(S') = First(eS) ∪ First(∧)
= {e, ∧}

First(E) = {b}

Follow(S) = {$, e}

$S \to iEtSS'$ First(S') = {e, ∧}
$A \quad \alpha \quad B \quad \beta$

$S' \to eS$ First(∧) = {∧}
$A \to \alpha B \beta$

Follow(S') = {$, e}

$S \to iEtSS'$ First(β) = First(∧) = {∧}
$A \quad \alpha \quad B \quad \beta$

Everything in Follow(S) is in Follow(S')

Follow(E) = {t}

$S \to iEtSS'$ First(tSS') = First(t)
$A \quad \alpha \quad B \quad \beta$

Q. Write an SDD to construct Syntax Tree for assignment stmts

$S \to id = E$	{S.nptr = mknode(assign, mkleaf(id, id.place), E.nptr)}
$E \to E_1 + E_2$	{E.nptr = mknode(+, E₁.nptr, E₂.nptr)}
$E \to E_1 * E_2$	{E.nptr = mknode(*, E₁.nptr, E₂.nptr)}
$E \to -E_1$	{E.nptr = mkunod(uminus, E₁.nptr)}
$E \to (E_1)$	{E.nptr = E₁.nptr}
$E \to id$	{E.nptr = mkleaf(id, id.place)}

Above SDD produces Syntax tree for given assignment stmts

The function mkunode (op, child) creates unary node

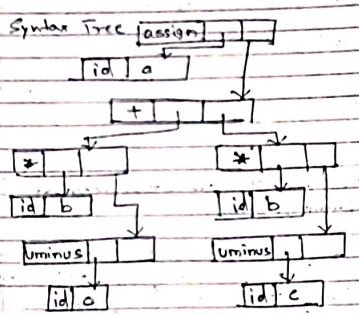
The function mknode (op, left, right) creates

internal binary node
mkleaf(id, id.place) creates leaf node.

These functions return a pointer to an existing node whenever possible, instead of creating new nodes. (DAG).

e.g. a = b * -c + b * -c

Syntax Tree



nptr, place are synthesized attributes.

Q. Define S-attributed and L-attributed definition and give an example of each.

Repeated
⇒ But the question was asked separately for S-attributed and L-attributed.
So club those two answers together and write.

Q. Explain structure preserving transformation and Algebraic transformations on Basic Block.

Repeated.
Note:
Refer to the question "Explain various transformations on Basic Blocks."

Q. Write an SDD to produce three-address code for assignment stmts. and generate three address code for the stmt.

a = b * -c + b * -c

When three-address code is generated, temporary names are made up for the interior nodes.

In the SDD to produce three-address code for assignment stmt, the non-terminal E is associated with two attributes —

E.place : the name that will hold the value of E and
E.code : the sequence of three-address stmts. evaluating E

The function newtemp returns a sequence of distinct names $t_1, t_2, \ldots$ in response to successive calls.

$S \rightarrow id = E$ { S.code = E.code || gen(id.place = E.place) }

$E \rightarrow E_1 + E_2$ { E.place = newtemp;
E.code = E_1.code || E_2.code ||
gen(E.place = E_1.place + E_2.place) }

$E \rightarrow E_1 * E_2$ { E.place = newtemp;
E.code = E_1.code || E_2.code ||
gen(E.place = E_1.place * E_2.place) }

$E \rightarrow -E_1$ { E.place = newtemp;
E.code = E_1.code ||
gen(E.place = uminus E_1.place) }

$E \rightarrow (E_1)$ { E.place = E_1.place;
E.code = E_1.code; }

$E \rightarrow id$ { E.place = id.place;
E.code = ''; }

fig: SDD for generating Three-address code for Assignment stmt.

Three - address code for the expression

$a = b * -c + b * -c$

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

} Three-address code

Note: Rest all questions in Nov. 2019 are repeated.

Unit IV SDT
MCQs

(1) In the conceptual view of Syntax Directed Definition the sequence of steps for evaluating semantic rules is

(a) Parse tree, i/p string, dependency graph, evaluation order for semantic rules.

(b) i/p string, Parse tree, dependancy graph, evaluation order for semantic rules. ✓

(c) Dependancy graph, evaluation order of semantic rules, parse tree, i/p string.

(d) Any sequence may do.

(2) The types of attributes associated with the grammar symbols are

(a) Inherited
(b) Synthesized
(c) L-attributed
(d) Both a and b ✓

(3) In L-attributed definition the value of the inherited attribute is computed from its

(a) Parent only
(b) Children
(c) Parent and/OR Right Siblings
(d) Parent and/OR Left Siblings ✓

(4) The interdependencies among the inherited and synthesized attributes can be depicted by

(a) Parse Tree
(b) Annotated Parse Tree
(c) Dependency Graph ✓
(d) None of the above

(5) The value of the synthesized attribute of a symbol is computed from its

(a) Parent
(b) Children ✓
(c) Both Parent and/or children
(d) from siblings on Left side

(6) The values of the inherited attribute of a symbol are computed from its

(a) Children
(b) Parent
(c) Parent and/or Siblings ✓
(d) Siblings on the Left side.

(7) The value of the attribute is determined by _____ associated with the production used at that node

(a) Semantic actions ✓
(b) Children
(c) CFG
(d) All of the above

(8) Parse tree showing the values of attributes at each node is called

(a) Dependancy graph
(b) Annotated Parse tree
(c) Decorated Parse tree
(d) Both b and c ✓

(9) Attributes are associated to _____ in a CFG.

(a) Non-terminals Only
(b) Terminals Only
(c) Both Terminals and non-terminals ✓
(d) Parent and Children.

(10) A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node in _____

(a) Bottom-up fashion ✓
(b) Top-Down fashion
(c) Both a and b
(d) Depth First Fashion

(11) Which of the following methods have been proposed for evaluating semantic rules?

(a) Oblivious methods
(b) Rule-based methods
(c) Parse tree methods
(d) All of the above ✓

(12) The functions mknode(), mkleaf() are used in an

(a) SDD for Construction Syntax tree for expression
(b) SDT for Constructing three address code for stmt
(c) SDD for Constructing Parse Tree
(d) All of the above

(13) A _____ for an expression identifies the common subexpressions in the expression

(a) Syntax Tree
(b) Parse Tree
(c) Directed Acyclic Graph
(d) Spanning Tree

(14) A _____ is a CFG in which attributes are associated with the grammar symbols and semantic actions are inserted within (embedded) the right sides of productions.

(a) SDD
(b) SDT
(c) S-attributed definition
(d) L-attributed definition

(15) If we have both inherited and synthesized attributes in a SDT then _____ for a symbol on the right side of a production must be computed in an action before that symbol.

(a) Synthesized attributed
(b) Inherited attribute
(c) Both Inherited and Synthesized attributes
(d) Synthesized attribute of the non-terminal on the LHS.

(16) If we have both inherited and synthesized attributes in SDT then an action must refer to a synthesized attribute of a symbol to the right of the action.

(a) True
(b) False

(17) In a SDT, a synthesized attribute for the nonterminal on the left can only be computed after all attributes it references have been computed. The actions i.e. semantic actions computing such attributes can usually be placed _____.

(a) at the end of the RHS of the production
(b) before the RHS
(c) depends on dependency graph & topological sort
(d) Can't Say

(18) Which of the following are Intermediate Code languages

(a) Syntax Trees / DAGs
(b) Postfix Notations
(c) Three-address Code
(d) All of the above

(19) Three address code is a _____ of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.

(a) Non-Linearized representation
(b) Linearized representation
(c) Graphical representation
(d) Postfix representation

(20) The reason for the term three address code is that each stmt usually contains _____.

(a) three addresses
(b) exactly three addresses
(c) More than three addresses
(d) All of the above.

(21) In an SDD to produce three address code for assignment stmt the attributes E.place and E.code are _____

(a) Both Inherited
(b) Both Synthesized
(c) E.place is inherited and E.code is synthesized
(d) E.place is synthesized and E.code is inherited.

(22) In a Compiler, three-address stmts can be implemented using

(a) Triples
(b) Indirect triples
(c) Quadruples
(d) All of the above

(23) In the SDD for backpatching while generating code for boolean expressions and flow-of-control stmts, the function backpatch(p, i)

(a) creates a new list containing only i and returns a pointer p
(b) Concatenates the lists pointed by p and returns a pointer to the concatenated list
(c) Inserts i as the target label for each of the statements on the list pointed to by p.
(d) All of the above.

Code Generation and Code Optimization

(1) Some code optimizations are carried out on the intermediate code because –

A. they enhance the portability of the compiler to other target processors
B. program analysis is more accurate on intermediate code than on m/c code
C. information from dataflow analyses cannot otherwise be used for optimization
D. the information from the front end cannot otherwise be used for optimization

(2) Which of the following is FALSE?

(A) Basic Block is a sequence of instructions where control enters the sequence at the beginning and exits at the end
(B) Available expression analysis can be used for common subexpression elimination

(c) Live variable analysis can be used for dead code elimination.

✓ (d) $x = 4*5 \Rightarrow x = 20$ is an example of common subexpression elimination

(3) Consider the following C code segment

```
for (i=0, i<n, i++)
  {
    for(j=0 ; j<n, j++)
      {
        if(i%2)
        {
          x = x+(4*j+5*i);
          y = y+(7+4*j);
        }
      }
  }
```

Which of one of the following is false?

(A) The code contains loop invariant computation
(B) There is a scope of common sub-expression elimination in this code.
(C) There is scope of strength reduction in this code
✓ (D) There is scope of dead code elimination in this code.

(4) Consider the grammar rule $E \rightarrow E_1 - E_2$ for arithmetic expressions. The code generated is targeted to a CPU having a single user register. The subtraction operation requires the first operand to be in the register. If E_1 and E_2 do not have any common sub-expression, in order to get the shortest possible code.

A. E_1 should be evaluated first.
✓ B. E_2 should be evaluated first.
C. Evaluation of E_1 and E_2 should necessarily be interleaved.
D. Order of evaluation of E_1 and E_2 is of no consequence.

(5) Consider the intermediate code given below:-

Code Generation

(1) The quality of the target code and hence, the design of code generator is dependent on —

(a) I/P to code generator
(b) Memory Management
(c) Target program
✓ (d) All of the above.

(2) During register allocation —

✓ (a) we select the set of variables that will reside in registers at a point in the program.
(b) we pick the specific register that a variable will reside in.
(c) Both (a) and (b)
(d) None of the above.

(3) During Register assignment, we pick the specific registers that a variable will reside in.

✓ (a) True
(b) False

(4) _____ and _____ are important factors in instruction selection during T.C. generation.

(a) i/p and o/p
✓ (b) Instruction speed and machine idioms
(c) Syntax tree and postfix notations.
(d) Memory and Evaluation order.

(5) The order in which computations are performed can affect the efficiency of target code and finding best evaluation order is an NP-complete problem.

✓ (a) True
(b) False

(6) The most important criteria for a code generator is that it produces _____ code

(a) Flexible
(b) meaningful
✓ (c) Correct
(d) Reusable

(7) Address modes involving registers have _____ cost, while those with memory location or literal in them have a _____ cost

(a) one, one
✓ (b) zero, one
(c) one, zero
(d) zero, zero

(8) A three-address stmt $x = y + z$ is used to _____ x and to _____ y and z

(a) reference, use
✓ (b) define, reference or use
(c) use, reference
(d) reference, define

(9) Which of the following are/is the rules to determine the set of leaders in a sequence of three-address stmts.

(a) The first stmt. is a leader.
(b) Any stmt. that is the target of a conditional or unconditional goto is a leader.
(c) Any stmt. that immediately follows a goto or conditional goto statement is a leader.
✓ (d) All of the above.

(10) 1 Common sub-expression elimination
2 Dead code elimination
3 renaming of temporary variables
4 interchange of two independent adjacent stmts.
are _____ transformations

(a) Global and Algebraic
(b) Local transformations
(c) Structure-preserving transformations
✓ (d) Both (b) and (c)

(11) $x = x + 0$, $x = x * 1$ are examples of _____ transformation
(a) Structure preserving
✓ (b) Algebraic
(c) Constant folding
(d) Renaming temporary variables.

GREAT THOUGHTS

I assure you that a learned fool is more foolish than an ignorant fool.

Moliere

For every minute you are angry, you lose sixty seconds of happiness.

Emerson

The beginnings of all things are small.

Cicero

Well begun is half done.

Horace

Four things cannot come back-the spoken word, the sped arrow, past life and neglected opportunity.

Anon

When we have not what we like, We must like what we have.

--- *Bussy-*

An invasion of armies can be resisted, but not an idea whose time has come.

--- *Victor*

Thoreau

The miracle is this --- the more we share, the more we have.

--- *Leonard*