11SEE 2019:

UNIT- I

1. a **What are the phases under analysis part of the compiler? Show their operations with an example.** CO1 (08)0

Ans :

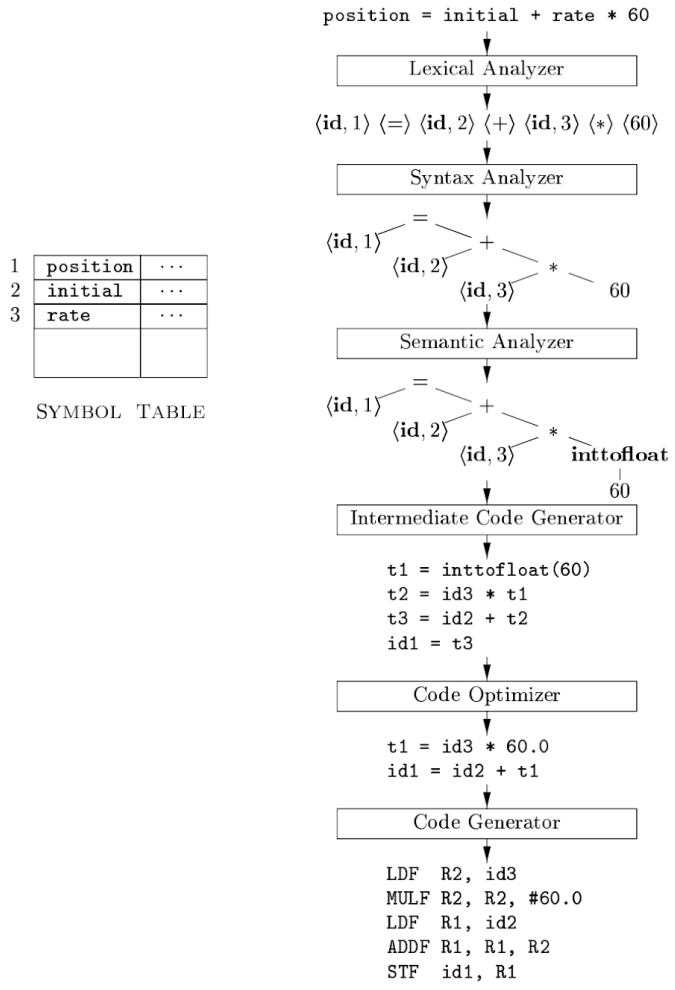


Figure 1.7: Translation of an assignment statement

b) **Justify the purposes of input buffers in compiler operations.** CO1 (08)

c) **How are panic mode and phrase level syntax error recovery schemes work?** CO1 (04)

. Hm mp

M..
M

..m.mmmm.

...MM

m
.

.pm.
..,
..
.mm

.
.pmp..
.
.
M.
M

..
.m...mP.
Mp.

M.
.
.pmp
.
M

M.

M.

.mmMmmM
M.mmp

.

.pmmmmm.
Pmm
P
Mp.m.

P.

P..

P

Pmm.p…

Pm.p..m
.
M

.
P.
.
Pm.p.
Pmm
.
.

P.
P.

P
P..
.
M

P

M.

Mm

..P.

Mmm
Me
P.mmp

MmmmM
mmp…
Mp m.

Mm.mp
.mp

M m m m m k … mm.m

2.

a) CO1 (08)

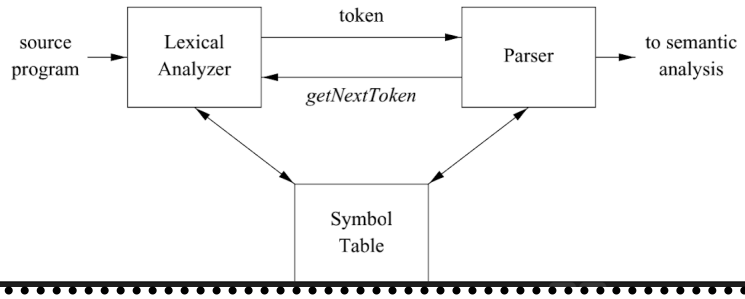
How the parser interacts with the Lexer? Explain.

3.1 The Role of the Lexical Analyzer

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the

kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

These interactions are suggested in Fig. 3.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the *getNextToken* command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.



Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and *whitespace* (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Sometimes, lexical analyzers are divided into a cascade of two processes:

a) *Scanning* consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

b) *Lexical analysis* proper is the more complex portion, which produces tokens from the output of the scanner.

b) **Define the following with proper examples:** CO1 (08)
i) Regular expressions ii) Languages.

	c)	**Show ambiguity in grammar with an example grammar and a proper input string.**	CO1	(04)

Sup2021:

SEE 2019:

UNIT- II

3.	a)	**CoL** **Is this Grammar suitable for Predictive Parsing? If not, make necessary changes and construct a LL (1) parsing table.**	CO2	(12)
	b)	**Construct a SLR parsing table for the grammar given below. Check whether the grammar is SLR or not.** **S→ AaAb | BbBa** **A→ ε** **B→ ε**	CO2	(08)
4.	a)	**Construct LALR(1) set of items for the grammar G:** **G: S→ Aa | bAc | d | bda** **A→d**	CO2	(10)
	b)	**Show the moves made by the stack of a shift reduce parser for accepting the input " id+id+id"** **G: E→E+T | T** **T→id**	CO2	(06)
	c)	**Define handle. Identify handles for the given input string : aabb** **G: S→ aSb | ab**	CO2	(04)

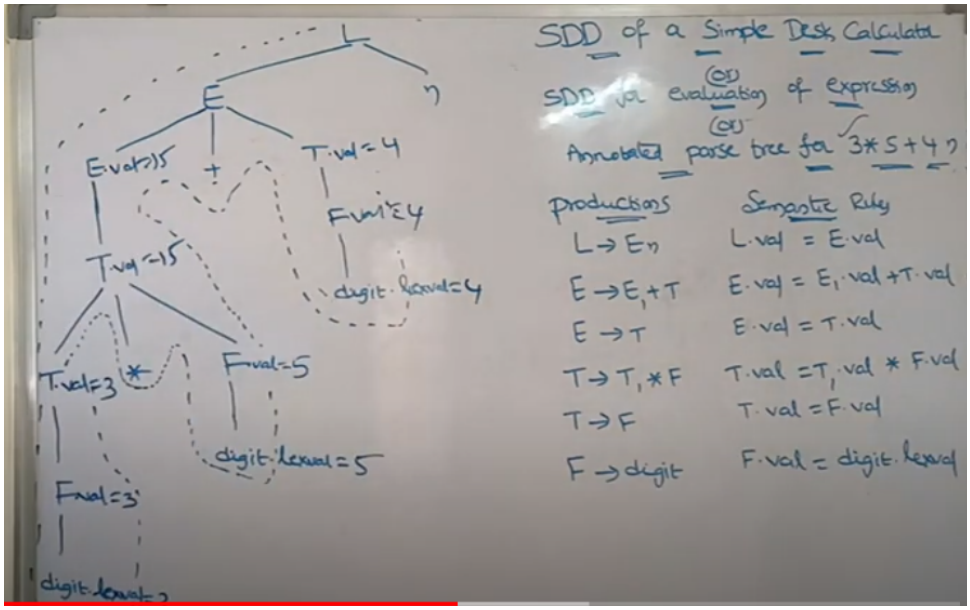
SEE 2019:

UNIT- III

5.	a)	**Generate the SDT for type setting boxes for the following grammar:** **S→B** **B→BB | B sub B | text**	CO3	(08)
	b)	**Explain how the side effects are controlled in simple type declarations by defining the SDD.** **D→T L** **T→int** **L→ L, id | id**	CO3	(08)
	c)	**Write the rules for turning L-Attributed Definitions to SDT.**	CO3	(04)
6.	a)	**Distinguish between the terms:** **i) Inherited attribute and Synthesized attribute** **ii) Annotated Parse Tree and attribute grammar**	CO3	(06)

iii) S-Attributed Definition and L- Attributed Definition.

b) **Illustrate how the desk calculator is implemented on a bottom-up parsing stack with the help of semantic actions.** CO3 (08)



Nt sure if its correct

c) **Draw the dependency graph for the following expressions:** CO3 (06)

i) int a, b, c **ii) char a, b, c, d**

SEE 2019:

UNIT- IV

7. a) **Construct DAG and VNM array for the following expressions.** CO4 (08)

i) (a+a)*(a+(a+a)*a+a)

ii) (d+f+(g+h)*(d+f))

b) **When are two type expressions equivalent? Explain.** CO4 (04)

c) **Show the SDD for flow of control statements like if...else, while.** CO4 (08)

8. a) **Write a C program segment to add two arrays using a loop. Write three address code for your program segment. Represent the same in quadruples, triples and indirect triples.** CO4 (08)

b) **Write SDD to generate three address code incrementally for arithmetic expressions.** CO4 (08)

	c)	**Write the Syntax Directed Translation for switch statement.**	CO4	(04)

SEE 2019:

UNIT- V

9.	a)	**List and Explain the issues in the development of a code generator.**	CO5	(10)
	b)	**Describe several code improving transformations on the code represented by the basic block.**	CO5	(06)
	c)	**Construct the DAG for the following basic block:** **x=a[i]** **a[j]=y** **z=a[i]**	CO5	(04)
10.	a)	**Define Basic blocks. Give an algorithm to partition three address instructions into basic block.**	CO5	(08)
	b)	**Convert the following program fragment into three address code and obtain** **i) Basic Blocks** **ii) Flow graph** **Code: int a[10][10];** **i=0; j=0;** **while(i<10){** **a[i][j]=1;** **i=i+1;** **j=j+1;** **}**	CO5	(12)

****

SUP 2021:

1.	a)	**Explain how the front end of a compiler works by translating the given statement Area=length*breadth;**	CO1	(08)
	b)	**Illustrate the processing of input buffering improvement technique "sentinels".**	CO1	(07)
	c)	**Explain how the tokens will be specified in the lexical analysis phase of the compiler.**	CO1	(05)

2.	a)	Explain about the role of lexical analyzer. Why is the analysis phase of compiler separated into lexical analysis and parsing?	CO1	(08)
	b)	Eliminate Left Recursion from the following grammar: i. S→IeTS|IeTSeS|a T→ Tb|b ii. A→Sa S→Ab|Sc|b	CO1	(06)
	c)	Construct a transition diagram to recognize the following patterns. Bat, Cat, Mat, Hat, Have, His	CO1	(06)
3.	a)	Construct the predictive parsing table for the given grammar and show the parsing steps for the string "uvuvxz" S → uBz B → Bv |vuE|vxuE|ByE E → v|vx	CO2	(14)
	b)	What is meant by handle pruning? Show the working of a shift reduce parser for accepting $id_1 * id_2$, considering the grammar: E → E + T | T T → T * F | F F → id	CO2	(06)
4.	a)	Construct LR (1) parsing table for the grammar. S → L = R | R L → *R | id R → L	CO2	(12)
	b)	Explain the rules for computing FIRST(X) and FOLLOW(X). Illustrate using the grammar: E→TE′ E′→+TE′|∈ T→FT′ T′→*FT′|∈ F→(E) |id	CO2	(08)
5.	a)	Construct Activation Tree and Activation record for the recursive code fragment for finding factorial of a number.	CO3	(07)
	b)	Design an L-attributed SDD to compute D.val G: D→BX X→BX | ε Give an Annotated Parse Tree and Dependency Graph for the input 1101$.	CO3	(07)
	c)	Define postfix SDT's. Write the rules for turning an L-attributed SDD into an SDT.	CO3	(06)
6.	a)	Generate the SDT for the following grammar S→while (C) S;	CO3	(08)
	b)	Illustrate how infix to prefix translation is implemented with the help of semantic actions.	CO3	(06)
	c)	Explain inherited attributes and synthesized attributes with examples.	CO3	(06)
7.	a)	Obtain the DAG for the expression a+b+(a+b)+(a+b)*a. Also give the sequence of steps for constructing the same.	CO4	(08)
	b)	Describe the process of generating three address code for flow control statements with help of Annotated Parse tree for if(x<0 && x>4 || !x) x=1;	CO4	(07)

c) Write and explain the SDD for switch statements. CO4 (05)

8. a) Translate the following three address codes into quadruple and triple. CO4 (06)
   i. t1=a+b; t2=c/t1; t3=i*4; t4=a[t3]; s=t4
   ii. t1=n*2; param t1; param 1; t2=call fun,2; t3=t2+2; f=t3

b) Write the algorithm for Unification of a pair of nodes in a type graph. CO4 (06

```
boolean unify(Node m, Node n) {
        s = find(m); t = find(n);
        if ( s = t ) return true;
        else if ( nodes s and t represent the same basic type ) return true;
        else if (s is an op-node with children s1 and s2 and
                        t is an op-node with children t1 and t2) {
                union(s, t);
                return unify(s1, t1) and unify(s2, t2);
        }
        else if (s or t represents a variable) {
                union(s, t);
                return true;
        }
        else return false;
}
```

c) Design an SDD for computing the type and width of basic types and array types. Also Construct Annotated Parse tree for int [2][2]. CO4 (08)

$T \rightarrow B$ C		$\{ t = B.type; w = B.width; \}$
$B \rightarrow$ **int**		$\{ B.type = integer; B.width = 4; \}$
$B \rightarrow$ **float**		$\{ B.type = float; B.width = 8; \}$
$C \rightarrow \epsilon$		$\{ C.type = t; C.width = w; \}$
$C \rightarrow$ [**num**] C_1		$\{ array(\mathbf{num}.value, C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width; \}$

Figure 6.15: Computing types and their widths

9. a) What do mean by basic block write three address code , basic block and flow graph for the following c segment CO5 (10)

```
prod=0;
i=1;
do
{
  prod=prod + a[i]* b[i];
        i=i+1;
}
while(i <=20);
```

	b)	Describe the following terms with an example i) Common sub expression elimination ii) Dead code elimination	CO5	(06)
	c)	How can algebraic identities be useful in code optimization?	CO5	(04)
10.	a)	Briefly explain the main issues in code generation.	CO5	(10)
	b)	Explain the code generation algorithm and generate code for the following expression. X=(a - b) + (a + c)	CO5	(10)

****