SIDDARTHA INSTITUTE OF SCIENCE AND TECHNOLOGY:: PUTTUR
Siddharth Nagar, Narayanavanam Road – 517583

QUESTION BANK (DESCRIPTIVE)

Subject with Code : Compiler Design (20CS0516) **Course & Branch :** B. Tech - CSE
Year & Sem : III B.Tech & I-Sem **Regulation :** R20

UNIT –I
INTRODUCTION AND LEXICAL ANALYSIS

1 a. What do you understand by language processor? [L2][CO1] **[2M]**

The computer is an intelligent combination of software and hardware. Hardware is simply a piece of mechanical equipment and its functions are being compiled by the relevant software. The hardware considers instructions as electronic charge, which is equivalent to the binary language in software programming. The binary language has only 0s and 1s.

To enlighten, the hardware code has to be written in binary format, which is just a series of 0s and 1s. Writing such code would be an inconvenient and complicated task for computer programmers, so we write programs in a high-level language, which is Convenient for us to comprehend and memorize. These programs are then fed into a series of devices and operating system (OS) components to obtain the desired code that can be used by the machine. This is known as a **language processing system**.

A language processor is a special type of computer software that has the capability to translate the source code or program codes into machine codes.

b. Describe about different language processors used in compiler design. [L2][CO1] **[4M]**

The following are the different types of language processors:-.

1. Assembler
2. Interpreter
3. Compiler

Compiler:

A compiler is a language processor which translate source code from high level language into binary machine code .

Assembler:

An assembler translates from a low level language to binary machine code.

Interpreter:

An interpreter is different from compiler and assembler . It does not convert program into machine code format instead it directs the CPU to obey each program statement.

c. Give the differences between compiler and interpreter [L4][CO1] **[6M]**

Parameter	Compiler	Interpreter	Assembler
Conversion	It converts the high-defined programming language into Machine language or binary code.	It also converts the program-developed code into machine language or binary code.	It converts programs written in the assembly language to the machine language or binary code.
Scanning	It scans the entire program before converting it into binary code.	It translates the program line by line to the equivalent machine code.	It converts the source code into the object code then converts it into the machine code.
Error Detection	Gives the full error report after the whole scan.	Detects error line by line. And stops scanning until the error in the previous line is solved.	It detects errors in the first phase, after fixation the second phase starts.
Code generation	Intermediate code generation is done in the case of Compiler.	There is no intermediate code generation.	There is an intermediate object code generation.
Execution time	It takes less execution time comparing to an interpreter.	An interpreter takes more execution time than the compiler.	It takes more time than the compiler.
Examples	C, C#, Java, C++	Python, Perl, VB, PostScript, LISP, etc…	GAS, GNU

2.a Define compiler. [L1][CO1] **[2M]**

Compiler, computer software that translates (compiles) source code written in a high-level language (e.g., C++) into a set of machine-language instructions that can be understood by a digital computer's CPU. Compilers are very large programs, with error-checking and other abilities.

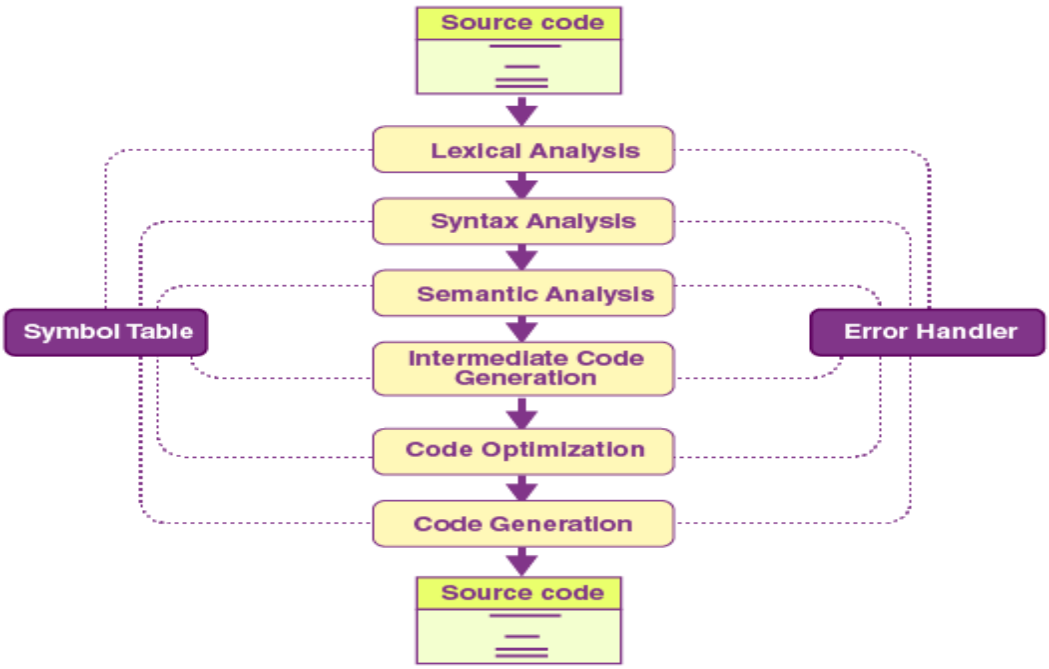
Some compilers translate high-level language into an intermediate assembly language, which is then translated (assembled) into machine code by an assembly program or assembler. Other compilers generate machine language directly. The term compiler was coined by American computer scientist Grace Hopper, who designed one of the first compilers in the early 1950s.

2.b Analyse the process of compilation while designing a compiler [L4][CO2] **[10M]**

The compilation procedure is nothing but a series of different phases. Each stage acquires input from its previous phase. In this article, we will learn more about the phases of the compiler, but before that, we need to understand the functioning of a compiler.

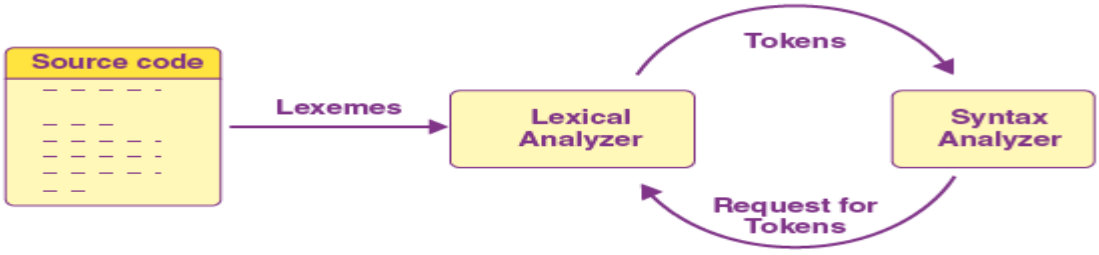
The 6 phases of a compiler are:

1. Lexical Analysis
2. Syntactic Analysis or Parsing
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation



1. **Lexical Analysis:** Lexical analysis or Lexical analyzer is the initial stage or phase of the compiler. This phase scans the source code and transforms the input program into a series of a token. A token is basically the arrangement of characters that defines a unit of information in the source code.

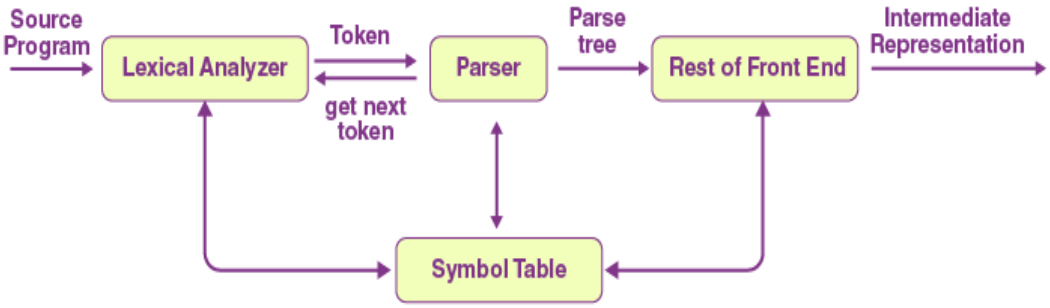
   NOTE: In computer science, a program that executes the process of lexical analysis is called a scanner, tokenizer, or lexer.



Roles and Responsibilities of Lexical Analyzer

- It is accountable for terminating the comments and white spaces from the source program.
- It helps in identifying the tokens.
- Categorization of lexical units.

2. **Syntax Analysis**: In the compilation procedure, the Syntax analysis is the second stage. Here the provided input string is scanned for the validation of the structure of the standard grammar. Basically, in the second phase, it analyses the syntactical structure and inspects if the given input is correct or not in terms of programming syntax.



It accepts tokens as input and provides a parse tree as output. It is also known as parsing in a compiler.

Roles and Responsibilities of Syntax Analyzer

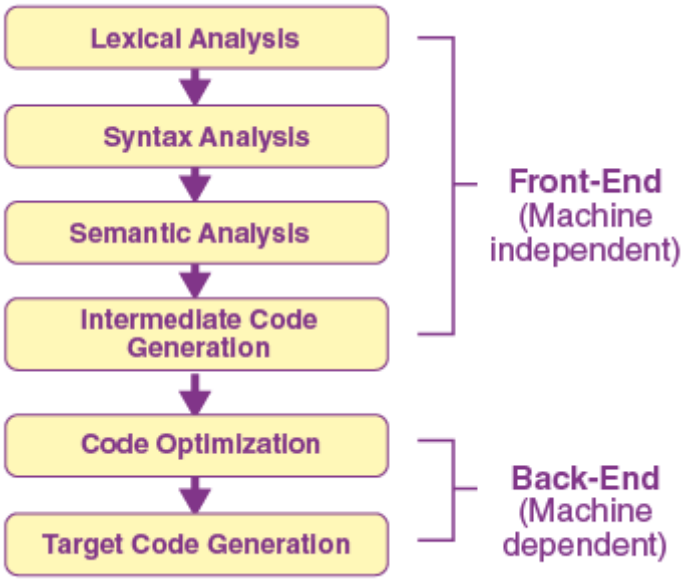
- Note syntax errors.
- Helps in building a parse tree.
- Acquire tokens from the lexical analyzer.
- Scan the syntax errors, if any.

3. **Semantic Analysis**: In the process of compilation, semantic analysis is the third phase. It scans whether the parse tree follows the guidelines of language. It also helps in keeping track of identifiers and expressions. In simple words, we can say that a semantic analyzer defines the validity of the parse tree, and the annotated syntax tree comes as an output.

Roles and Responsibilities of Semantic Analyzer:

- Saving collected data to symbol tables or syntax trees.
- It notifies semantic errors.
- Scanning for semantic errors.

4. **Intermediate Code Generation**: The parse tree is semantically confirmed; now, an intermediate code generator develops three address codes. A middle-level language code generated by a compiler at the time of the translation of a source program into the object code is known as intermediate code or text.



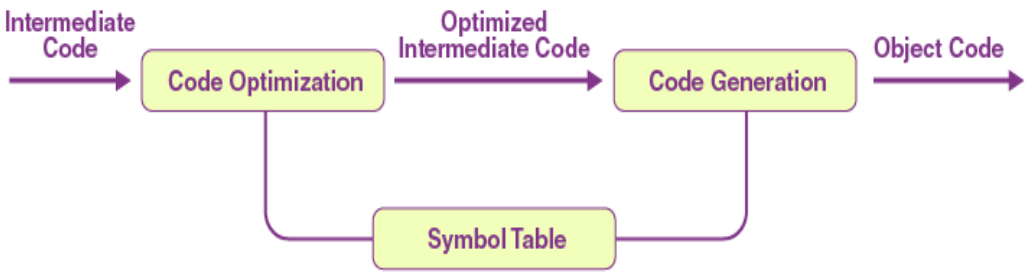
Few Important Pointers:

- A code that is neither high-level nor machine code, but a middle-level code is an intermediate code.
- We can translate this code to machine code later.
- This stage serves as a bridge or way from analysis to synthesis.

Roles and Responsibilities:

- Helps in maintaining the priority ordering of the source language.
- Translate the intermediate code into the machine code.
- Having operands of instructions.

5. **Code optimizer**: Now coming to a phase that is totally optional and it is code optimization. It is used to enhance the intermediate code. This way, the output of the program is able to run fast and consume less space. To improve the speed of the program, it eliminates the unnecessary strings of the code and organises the sequence of statements.



Roles and Responsibilities:

- Remove the unused variables and unreachable code.
- Enhance runtime and execution of the program.
- Produce streamlined code from the intermediate expression.

6. **Code Generator**: The final stage of the compilation process is the code generation process. In this final phase, it tries to acquire the intermediate code as input which is fully optimised and map it to the machine code or language. Later, the code generator helps in translating the intermediate code into the machine code.

   Roles and Responsibilities:

   - Translate the intermediate code to target machine code.
   - Select and allocate memory spots and registers.

Symbol Table: The symbol table is mainly known as the data structure of the compiler. It helps in storing the identifiers with their name and types. It makes it very easy to operate the searching and fetching process. The symbol table connects or interacts with all phases of the compiler and error handler for updates. It is also accountable for scope management.

It stores:

- It stores the literal constants and strings.
- It helps in storing the function names.
- It also prefers to store variable names and constants.
- It stores labels in source languages.

Error Handling:

In the compiler design process error may occur in all the below-given phases:

- Lexical analyzer: Wrongly spelled tokens
- Syntax analyzer: Missing parenthesis
- Intermediate code generator: Mismatched operands for an operator
- Code Optimizer: When the statement is not reachable
- Code Generator: When the memory is full or proper registers are not allocated
- Symbol tables: Error of multiple declared identifiers

Most common errors are invalid character sequence in scanning, invalid token sequences in type, scope error, and parsing in semantic analysis. The error may be encountered in any of the above phases. After finding errors, the phase needs to deal with the errors to continue with the compilation process.

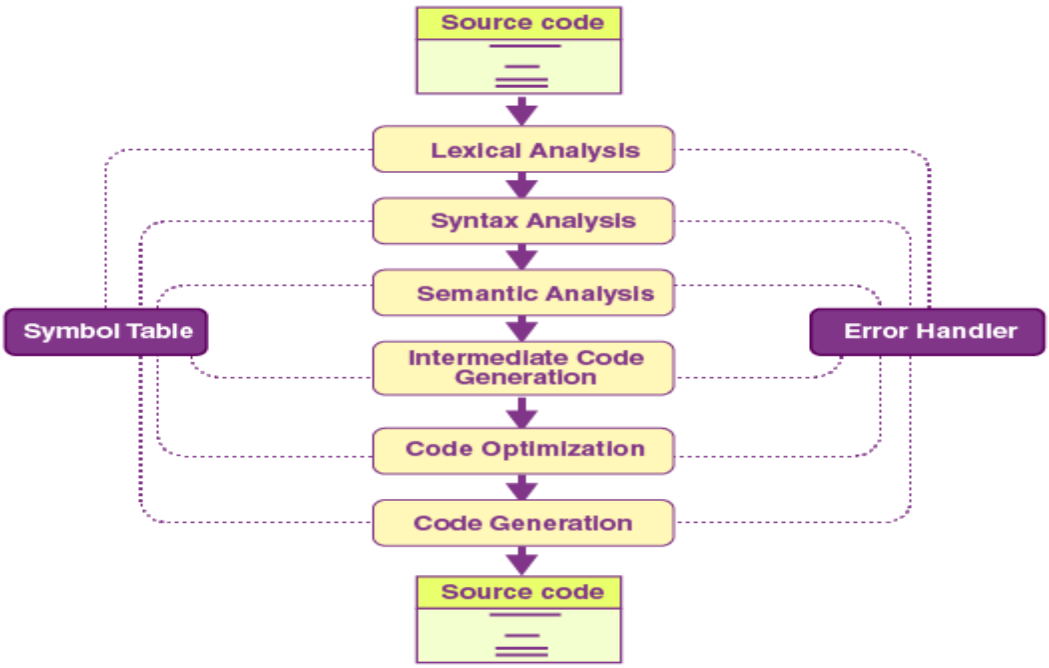
These errors need to be reported to the error handler which handles the error to perform the compilation process. Generally, the errors are reported in the form of message.

3.a List all the phases of compiler. [L1][CO2] **[2M]**

The 6 phases of a compiler are:

1. Lexical Analysis
2. Syntactic Analysis or Parsing
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation

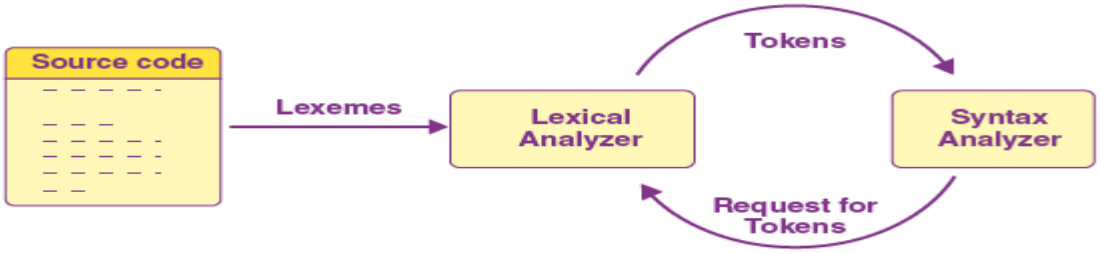
3.b. Give the neat diagram of phase of a compiler. [L2][CO2] **[4M]**



3.c Explain each phase of a compiler [L2][CO2] **[6M]**

Lexical Analysis: Lexical analysis or Lexical analyzer is the initial stage or phase of the compiler. This phase scans the source code and transforms the input program into a series of a token. A token is basically the arrangement of characters that defines a unit of information in the source code.

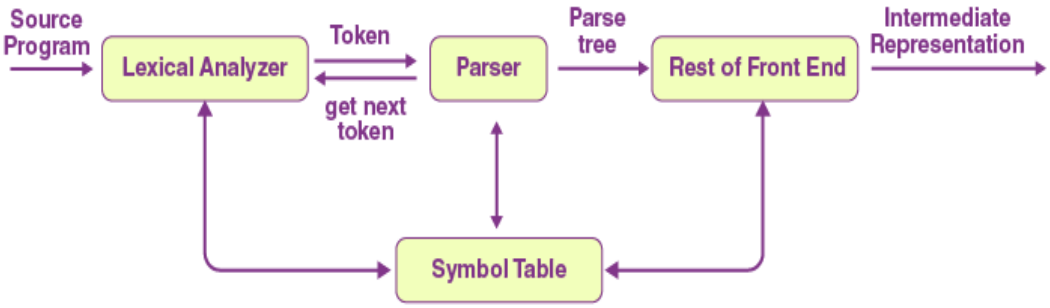
NOTE: In computer science, a program that executes the process of lexical analysis is called a scanner, tokenizer, or lexer.



Roles and Responsibilities of Lexical Analyzer

- It is accountable for terminating the comments and white spaces from the source program.
- It helps in identifying the tokens.
- Categorization of lexical units.

Syntax Analysis: In the compilation procedure, the Syntax analysis is the second stage. Here the provided input string is scanned for the validation of the structure of the standard grammar. Basically, in the second phase, it analyses the syntactical structure and inspects if the given input is correct or not in terms of programming syntax.



It accepts tokens as input and provides a parse tree as output. It is also known as parsing in a compiler.

Roles and Responsibilities of Syntax Analyzer

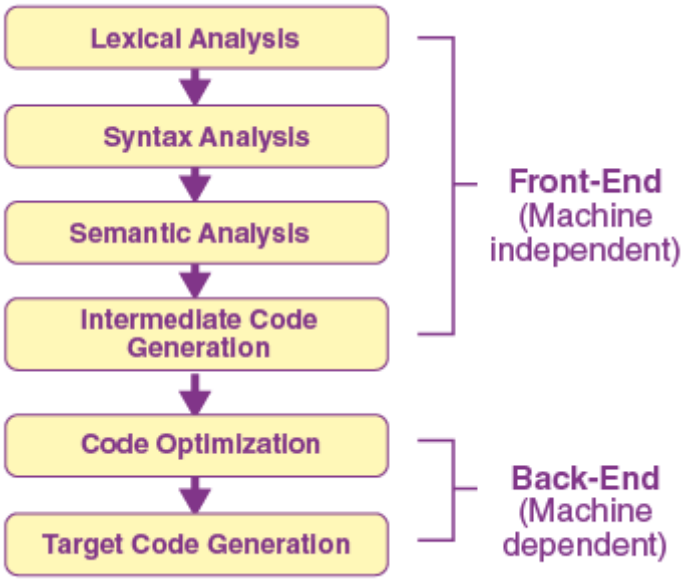
- Note syntax errors.
- Helps in building a parse tree.
- Acquire tokens from the lexical analyzer.
- Scan the syntax errors, if any.

Semantic Analysis: In the process of compilation, semantic analysis is the third phase. It scans whether the parse tree follows the guidelines of language. It also helps in keeping track of identifiers and expressions. In simple words, we can say that a semantic analyzer defines the validity of the parse tree, and the annotated syntax tree comes as an output.

Roles and Responsibilities of Semantic Analyzer:

- Saving collected data to symbol tables or syntax trees.
- It notifies semantic errors.
- Scanning for semantic errors.

Intermediate Code Generation: The parse tree is semantically confirmed; now, an intermediate code generator develops three address codes. A middle-level language code generated by a compiler at the time of the translation of a source program into the object code is known as intermediate code or text.



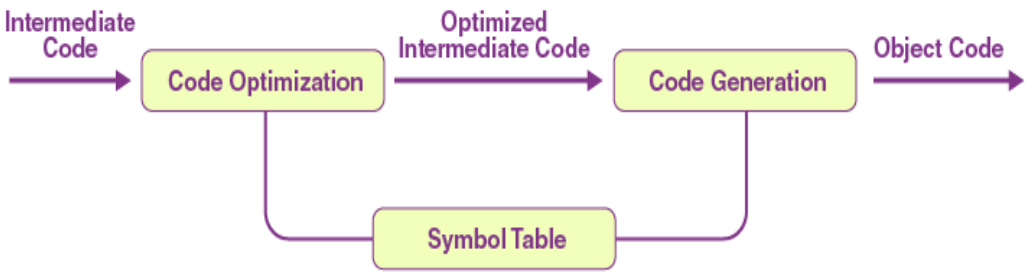
Few Important Pointers:

- A code that is neither high-level nor machine code, but a middle-level code is an intermediate code.
- We can translate this code to machine code later.
- This stage serves as a bridge or way from analysis to synthesis.

Roles and Responsibilities:

- Helps in maintaining the priority ordering of the source language.
- Translate the intermediate code into the machine code.
- Having operands of instructions.

Code optimizer: Now coming to a phase that is totally optional and it is code optimization. It is used to enhance the intermediate code. This way, the output of the program is able to run fast and consume less space. To improve the speed of the program, it eliminates the unnecessary strings of the code and organises the sequence of statements.



Roles and Responsibilities:

- Remove the unused variables and unreachable code.
- Enhance runtime and execution of the program.
- Produce streamlined code from the intermediate expression.

Code Generator: The final stage of the compilation process is the code generation process. In this final phase, it tries to acquire the intermediate code as input which is fully optimised and map it to the machine code or language. Later, the code generator helps in translating the intermediate code into the machine code.

Roles and Responsibilities:

- Translate the intermediate code to target machine code.
- Select and allocate memory spots and registers.

Symbol Table: The symbol table is mainly known as the data structure of the compiler. It helps in storing the identifiers with their name and types. It makes it very easy to operate the searching and fetching process. The symbol table connects or interacts with all phases of the compiler and error handler for updates. It is also accountable for scope management.

It stores:

- It stores the literal constants and strings.
- It helps in storing the function names.
- It also prefers to store variable names and constants.
- It stores labels in source languages.

Error Handling:

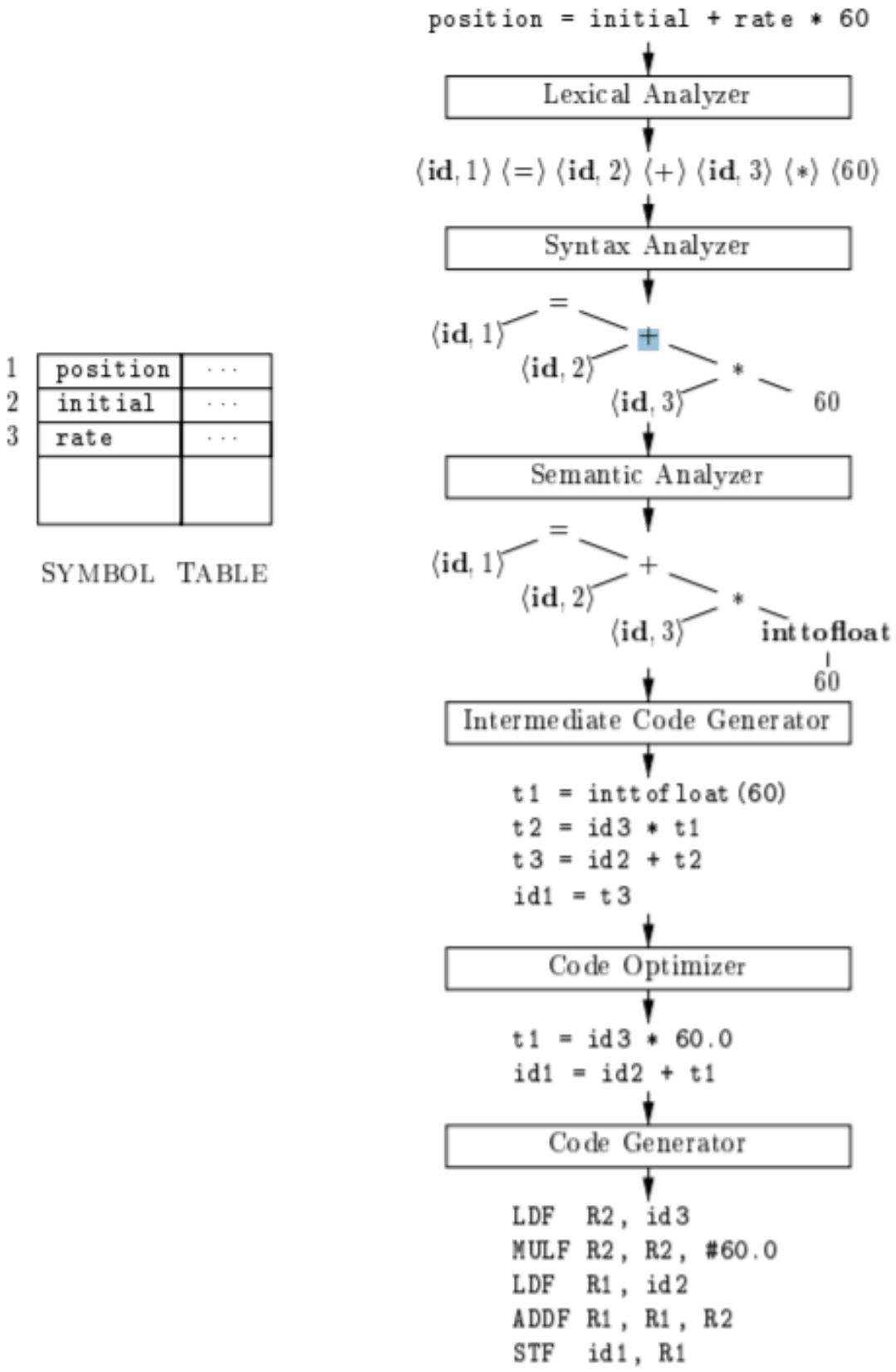
In the compiler design process error may occur in all the below-given phases:

- Lexical analyzer: Wrongly spelled tokens
- Syntax analyzer: Missing parenthesis
- Intermediate code generator: Mismatched operands for an operator
- Code Optimizer: When the statement is not reachable
- Code Generator: When the memory is full or proper registers are not allocated
- Symbol tables: Error of multiple declared identifiers

Most common errors are invalid character sequence in scanning, invalid token sequences in type, scope error, and parsing in semantic analysis. The error may be encountered in any of the above phases. After finding errors, the phase needs to deal with the errors to continue with the compilation process.

These errors need to be reported to the error handler which handles the error to perform the compilation process. Generally, the errors are reported in the form of message.

4. Design the compiler by using the source program position=intial+rate*60. [L6][CO3] **[12M]**



5 a. Analyze the reasons for separating the lexical analysis and syntax analysis [L4][CO2]**[4M]**

Reasons for separating both analyses:

Simpler design

- ✓ Separation allows the simplification of one or the other
- ✓ Example: A parser with comments or white spaces is more complex

Compiler efficiency is improved.

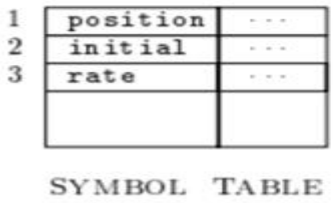
- ✓ Optimization of lexical analysis because a large amount of time is spent reading the source program and partitioning it into tokens.

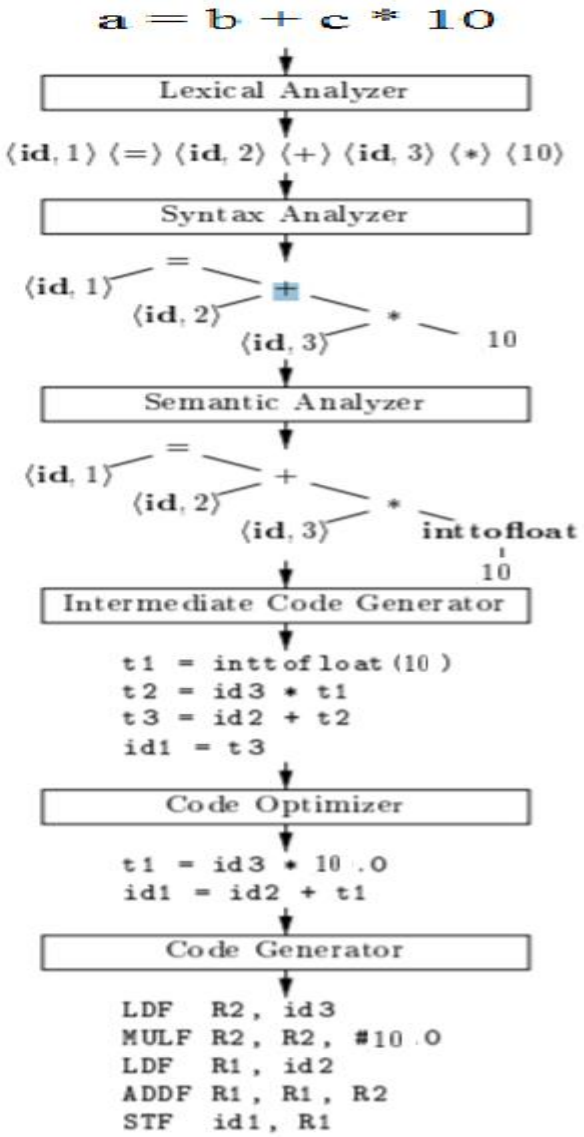
Compiler portability is enhanced.

- ✓ Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.

5 b. Illustrate the steps involved in designing the compiler by using the source program a=b+c*10

[L3][CO3] **[8M]**





6 a. Describe Bootstrapping. [L2][CO1] **[8M]**

Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on.

Writing a compiler for any high level language is a complicated process. It takes lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages. to clearly understand the **Bootstrapping** technique consider a following scenario.

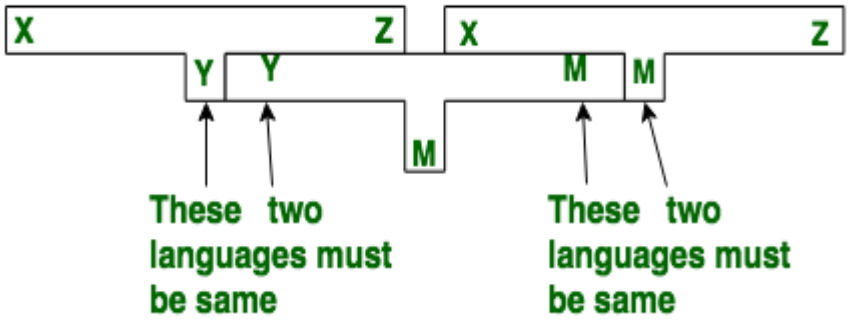
Suppose we want to write a cross compiler for new language X. The implementation language of this compiler is say Y and the target code being generated is in language Z. That is, we create XYZ. Now if existing compiler Y runs on machine M and generates code for M then it is denoted as YMM. Now if we run XYZ using YMM then we get a compiler XMZ.

That means a compiler for source language X that generates a target code in language Z and which runs on machine M.

Following diagram illustrates the above scenario.

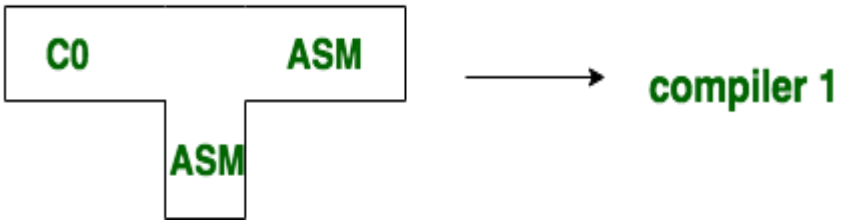
Example:

We can create compiler of many different forms. Now we will generate.



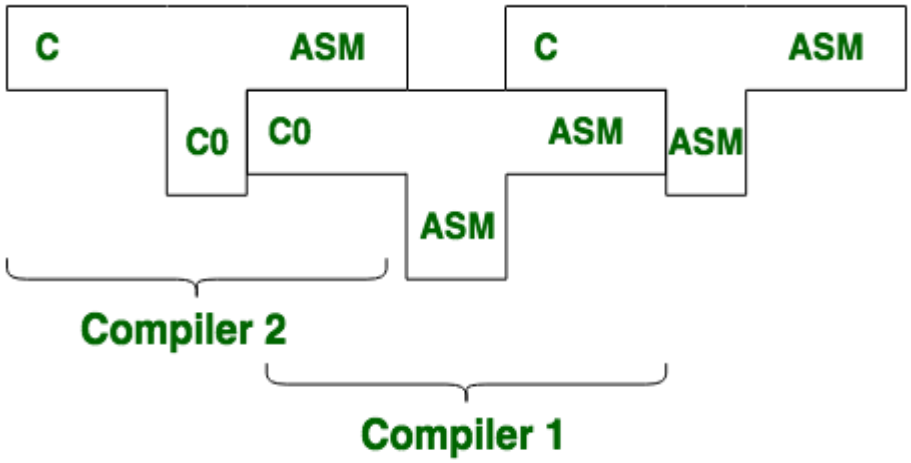
Compiler which takes C language and generates an assembly language as an output with the availability of a machine of assembly language.

- **Step-1:** First we write a compiler for a small of C in assembly language.



- **Step-2:** Then using with small subset of C i.e. C0, for the source language c the compiler is written.

- **Step-3:** Finally we compile the second compiler. using compiler 1 the compiler 2 is compiled.



- **Step-4:** Thus we get a compiler written in ASM which compiles C and generates code in ASM.

6 b. Explain the different applications of compiler technology. [L2][CO1] **[4M]**

Applications of Compiler Technology

1. Implementation of High-Level Programming Languages.
2. Optimizations for Computer Architectures
3. Design of New Computer Architectures
4. Program Translations
5. Software Productivity Tools

Implementation of High-Level Programming Languages.

A high-level programming language defines a programming abstraction: the programmer specifies an algorithm in the language, and the compiler must translate it to the target language. Higher-level programming languages are sometimes easier to develop in, but they are inefficient, therefore the target applications run slower.

Low-level language programmers have more control over their computations and, in principle, can design more efficient code. Lower-level programs, on the other hand, are more difficult to build and much more difficult to maintain. They are less portable, more prone to errors, and more complex to manage. Optimized compilers employ ways to improve the performance of generated code, compensating for the inefficiency of high-level abstractions.

In actuality, programs that utilize the register keyword may lose efficiency since programmers aren't always the best judges of extremely low-level matters like register allocation. The ideal register allocation approach is very reliant on the design of the machine.

Hardwiring low-level resource management decisions like register allocation may actually harm performance, especially if the application is executed on machines that aren't meant for it

Optimization of computer architectures

Aside from the rapid evolution of computer architectures, there is a never-ending demand for new compiler technology. Almost all high-performance computers leverage parallelism and memory hierarchies as essential methods.

Parallelism may be found at two levels: at the instruction level, where many operations are performed at the same time, and at the processor level, where distinct threads of the same program are executed on different processors. Memory hierarchies address the fundamental problem of being able to produce either extremely fast storage or extremely huge storage, but not both.

Design of new computer architectures

In the early days of computer architecture design, compilers were created after the machines were built. That isn't the case now. Because high-level programming is the norm, the performance of a computer system is determined not just by its sheer speed, but also by how well compilers can use its capabilities.

Compilers are created at the processor-design stage of contemporary computer architecture development, and the resultant code is used to evaluate the proposed architectural features using simulators.

Program Translations:

The compilation is typically thought of as a translation from a high-level language to the machine level, but the same approach may be used to translate across several languages. The following are some of the most common applications of software translation technologies.

- ✓ Compiled Simulation
- ✓ Binary translation
- ✓ Hardware Syntheses
- ✓ Database Query Interpreters

Software productivity tools:

Programs are possibly the most complex technical objects ever created; they are made up of a plethora of little elements, each of which must be accurate before the program can function properly. As a result, software mistakes are common; errors can cause a system to crash, generate incorrect results, expose a system to security threats, or even cause catastrophic failures in key systems. Testing is the most common method for discovering program flaws.

A fascinating and interesting complementary option is the use of data-flow analysis to statically discover problems (that is before the program is run). Unlike program testing, the data-flow analysis may uncover vulnerabilities along any possible execution path, not only those used by the input data sets. Many data-flow-analysis techniques, originally developed for compiler optimizations, may be used to build tools that assist programmers with their software engineering responsibilities.

7 a. Discuss the Compiler construction Tools [L2][CO3] **[6M]**

Some commonly used compiler-construction tools include

1**. Parser generators** that automatically produce syntax analyzers from a grammatical description of a programming language.

2. **Scanner generators** that produce lexical analyzers from a regular-expression description of the tokens of a language.

3. **Syntax-directed translation engines** that produce collections of routines for walking a parse tree and generating intermediate code.

4. **Code-generation** generators that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

5. **Data-flow analysis** engines that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

6. **Compiler-construction toolkits** that provide an integrated set of routines for constructing various phases of a compiler.

7 b. Differentiate tokens, patterns, and lexeme. [L4][CO1] **[6M]**

Token	Lexeme	Pattern
Keywords	specification, endspecification, type, constant, input, endinput, vop, oop, output, endoutput, variable, axioms, axiom, endaxioms, rules, endrules, rule	All reserved word or terminals of language.
Digits	0 – 9	Any numeric constants
Identifiers	A - Z , a - z	([a-z]|[A-Z]) ([a-z]|[A-Z])*. Any English latter (capital or small letters) repeted one or more times.
Operators	= , => , + , - , > , < , >= , <=	equalsign, impliessign, plussign, minussign, greatersign, lesssign, greaterequalsign, lessequalsign
Special characters	{ , } , (,) , ; , : , ^ , & , ,	lbraces, rbraces, lparen, rparen, semicolon, colon, unionsign, Ampersand, comma.
Whitespace	‘ ‘	Any white space
Newline	\n	Any new line

8 a. Explain in detail about the role of lexical analyzer in Compiler Design. [L2][CO1] **[6M]**

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. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

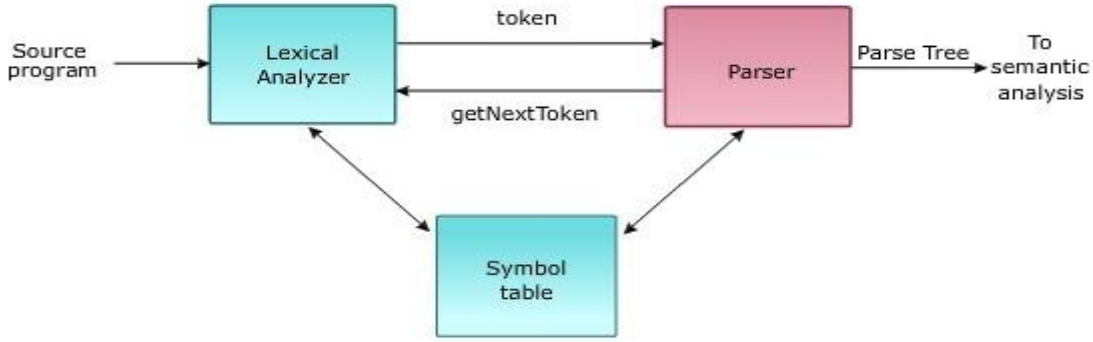


Fig: Interactions between the lexical analyzer and the parser

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.

The lexical analyzers are divided into 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, where the scanner produces the sequence of tokens as output.

8 b. Write about input buffering? [L3][CO1] **[6M]**

This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id. In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=.

There are two methods

1. Two-buffer scheme- that handles large lookaheads safely.
2. "Sentinels" that saves time checking for the ends of buffers.

Two Buffer Scheme:

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded, as suggested in Fig.

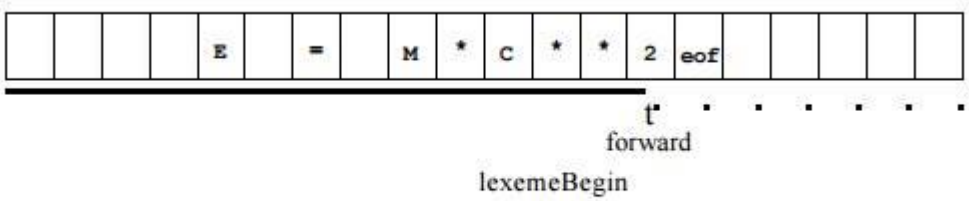


Figure 3.3: Using a pair of input buffers

Each buffer is of the same size *N*, and *N* is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read *N* characters into a buffer, rather than using one system call per character. If fewer than *N* characters remain in the input file, then a special character, represented by **eof,** marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained:

1. Pointer **lexemeBegin,** marks the beginning of the current lexeme, whose extent we are attempting to determine.

2. Pointer **forward** scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, **forward** is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, **lexemeBegin** is set to the character immediately after the lexeme just found. In above Fig**,** we see **forward** has passed the end of the next lexeme, ** (the Fortran exponentiation operator), and must be retracted one position to its left.

Algorithm :

if ***(forward*** is at end of **first buffer**)
{
reload **second buffer**;
forward = beginning of **second buffer**;
}
else if ***(forward*** is at end of **second buffer**)
{
reload **first buffer**;
forward = beginning of **first buffer**;
}
else **/* eof** within a buffer marks the end of input */
terminate lexical analysis;
break;

Sentinels:

If we use the scheme of previous, we must check, each time we advance **forward,** that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a *sentinel* character at the end.

The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof.**

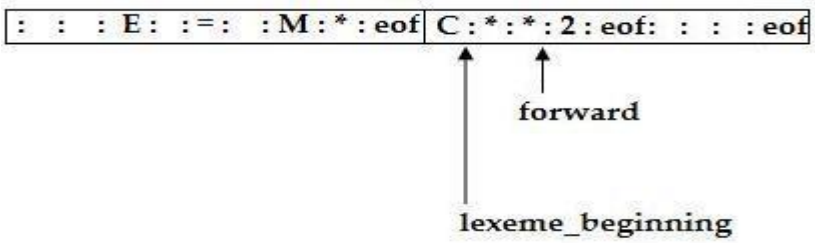


Fig 1.9 Sentinels at the end of each buffer half

Algorithm:
case eof:
if (***forward*** is at end of **first buffer**) {
reload **second buffer**;
forward = beginning of **second buffer**;
}
else if (***forward*** is at end of **second buffer**) {
reload **first buffer**;
forward = beginning of **first buffer**;
}
else /* eof within a buffer marks the end of input */
terminate lexical analysis;
break;

9. Discriminate the following terms [L5][CO1] **[12M]**

a) Specification of Tokens

Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens. In this section we shall study the formal notation for regular expressions,

Strings and Languages

An *alphabet* is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set {0,1} is the *binary alphabet.* ASCII is an important example of an alphabet; it is used in many software systems.

Terms for Parts of Strings

The following string-related terms are commonly used:

1. A ***prefix*** of string *s* is any string obtained by removing zero or more symbols from the end of *s*. For example, ban, banana, and e are prefixes of banana.
2. A ***suffix*** of string *s* is any string obtained by removing zero or more symbols from the beginning of *s.* For example, nana, banana, and e are suffixes of banana.
3. A ***substring*** of *s* is obtained by deleting any prefix and any suffix from *s.* For instance, banana, nan, and e are substrings of banana.
4. The ***proper*** prefixes, suffixes, and substrings of a string *s* are those, prefixes, suffixes, and substrings, respectively, of *s* that are not e or not equal to *s* itself.
5. A ***subsequence*** of *s* is any string formed by deleting zero or more not necessarily consecutive positions of *s.* For example, baan is a subsequence of banana.

Operations on Languages:

Operation on Languages

- The language exponentiation operation is defined as $L^0 = \{\varepsilon\}$ and $L^i = L^{i-1}L$

OPERATION	DEFINITION
union of L and M written $L \cup M$	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of L and M written LM	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes "zero or more concatenations of" L.
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of" L.

Fig. 3.8. Definitions of operations on languages.

Using the operators of Fig:
1. *L* U *D* is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. *LD* is the set c-f 520 strings of length two, each consisting of one letter followed by one digit.
3. L4 is the set of all 4-letter strings.
4. *L** is the set of all strings of letters, including e, the empty string.
5. *L(L* U *D)** is the set of all strings of letters and digits beginning with a letter.
6. *D+* is the set of all strings of one or more digits.

b) Recognition of Tokens

It means how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns. Our discussion will make use of the following running example.

stmt —>• **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
I *e*
expr —>• *term* **relop** *term*
| *term*
term -> **id**
| **number**

Figure: A grammar for branching statements

For **relop,** we use the comparison operators of languages like Pascal or SQL, where = is "equals" and <> is "not equals," because it presents an interesting structure of lexemes.
The terminals of the grammar, which are **if, then, else, relop, id,** and **number,** are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using

regular definitions, as shown in below.

digit --------> [0-9]
digits--------> *digit+*
number-----> *digits (. digits)?* **(E [+-]?** *digits)?*
letter--------> [A-Za-z]
id ----------> *letter (letter \ digit)**
if----------> if
then---------> **then**
else---------> **e l se**
relop-------> < 1 > 1 <= 1 >= 1 = 1 < >

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	-	-
if	if	-
then	then	-
else	else	-
Any *id*	id	Pointer to table entry
Any *number*	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	

Figure: Tokens, their patterns, and attribute values

Transition Diagrams:

Transition diagrams have a collection of nodes or circles, called *states.* Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of a state as summarizing all we need to know about what characters we have seen between the *lexemeBegin* pointer and the *forward* pointer.

Edges are directed from one state of the transition diagram to another. Each edge is *labeled* by a symbol or set of symbols. If we are in some state **5,** and the next input symbol is *a,* we look for an edge out of state *s* labeled by *a* (and perhaps by other symbols, as well). If we find such an edge, we advance the *forward* pointer arid enter the state of the transition diagram to which that edge leads. We shall assume that all our transition diagrams are *deterministic,* meaning that there is never more than one edge out of a given state with a given symbol among its labels.

Example: Below Figure is a transition diagram that recognizes the lexemes matching the token **relop.** We begin in state 0, the start state. If we see < as the first input symbol, then among the lexemes that match the pattern for **relop** we can only be looking at <, <>, or <=. We therefore go to state 1, and look at the next character. If it is =, then we recognize lexeme <=, enter state 2, and return the token **relop** with attribute LE, the symbolic constant representing this particular comparison operator. If in state 1 the next character is >, then instead we have lexeme <>, and enter state 3 to return an indication that the not-equals operator has been found. On any other character, the lexeme is <, and we enter state 4 to return that information. Note, however, that state 4 has a * to indicate that we must retract the input one position.

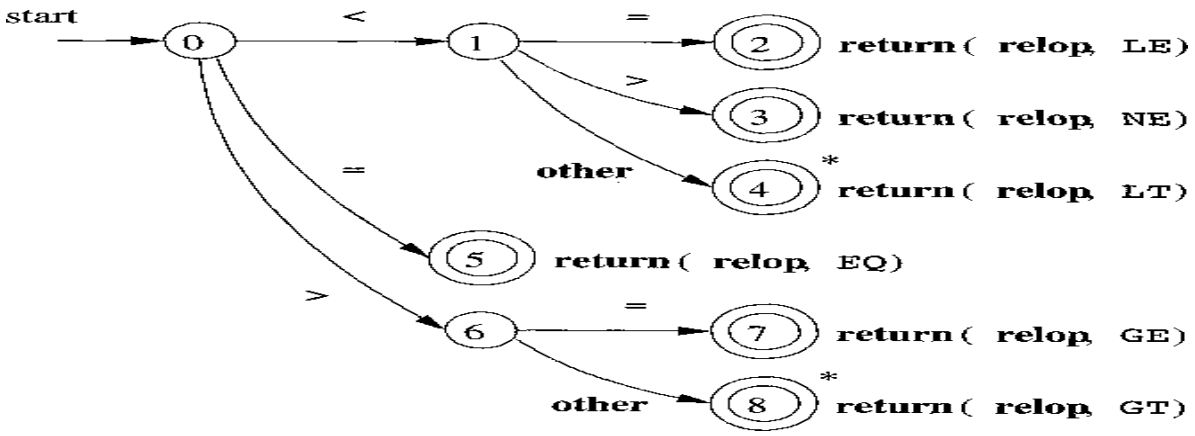


Figure: Transition diagram for **relop**

Recognition of Reserved Words and Identifiers

Recognizing keywords and identifiers presents a problem. Usually, keywords like if or **then** are reserved (as they are in our running example), so they are not identifiers even though they *look* like identifiers. Thus, although we typically use a transition diagram like that of Fig. 3.14 to search for identifier lexemes, this diagram will also recognize the keywords if, **then,** and **e l s e** of our running example.

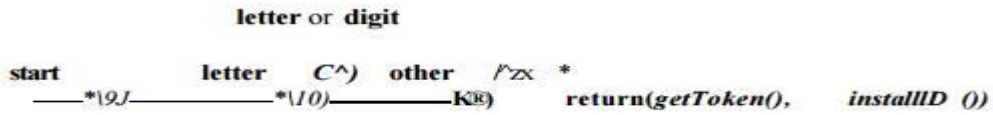
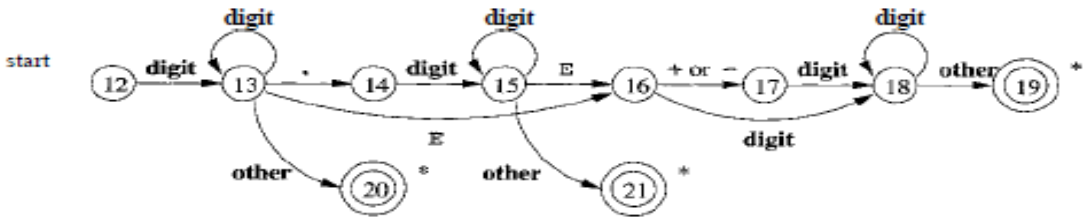
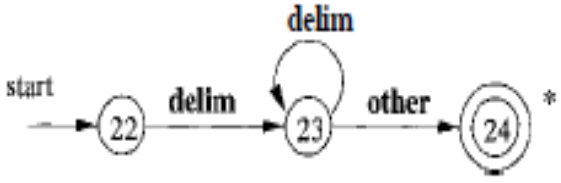


Figure 3.14: A transition diagram for **id**'s and keywords

The transition diagram for token n u m b e r is shown in Fig. 3.16, and is so far the most complex diagram we have seen. Beginning in state 12, if we see adigit, we go to state 13. In that state, we can read any number of additional digits. However, if we see anything but a digit or a dot, we have seen a number in the form of an integer; 123 is an example. That case is handled by entering state 20, where we return token n u m b e r and a pointer to a table of constants where the found lexeme is entered. These mechanics are not shown on the diagram but are analogous to the way we handled identifiers.



The final transition diagram, shown in Fig. 3.17, is for whitespace. In that diagram, we look for one or more "whitespace" characters, represented by **d e l I m** in that diagram — typically these characters would be blank, tab, newline, and perhaps other characters that are not considered by the language design to be part of any token.



10 a. What is LEX? [L2][CO3] **[2M]**

It is a tool or software which automatically generates a lexical analyzer (finite Automata). It takes as its input a LEX source program and produces lexical Analyzer as its output. Lexical Analyzer will convert the input string entered by the user into tokens as its output.

It is a tool or software which automatically generates a lexical analyzer (finite Automata). It takes as its input a LEX source program and produces lexical Analyzer as its output. Lexical Analyzer will convert the input string entered by the user into tokens as its output.

b. Explain the working of a LEX Tool [L2][CO3] **[6M]**

An input file, which we call **lex.l** , is written in the Lex language and describes the lexical analyzer to be generated. The Lex compiler transforms **lex.1** to a C program, in a file that is always named **lex.yy.c**. The latter file is compiled by the C compiler into a file called a.out , as always. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

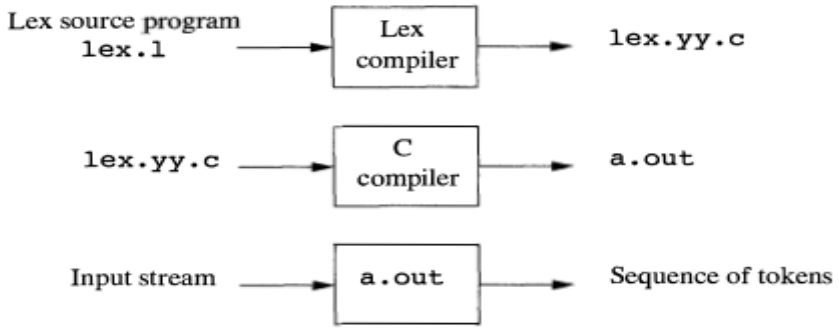


Fig: Creating a lexical analyzer with Lex

c. Give the structure of LEX program. [L2][CO3] **[4M]**

Structure of Lex Programs:

A **Lex** program has the following form:

declarations

%%

translation rules

%%

auxiliary functions

The **declarations section** includes declarations of variables, *manifest constants* (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions.

The **translation rules** each have the form

```
Pattern1 { Action1 }
Pattern2 { Action2 }
…….
…….
Pattern-n { Action-n }
```

Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C, although many variants of Lex using other languages have been created. The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

In the declarations section we see a pair of special brackets, °/.{ and %}. Anything within these brackets is copied directly to the file l e x . y y . c , and is not treated as a regular definition. It is common to place there the definitions of the manifest constants, using C #def i n e statements to associate unique integer codes with each of the manifest constants.

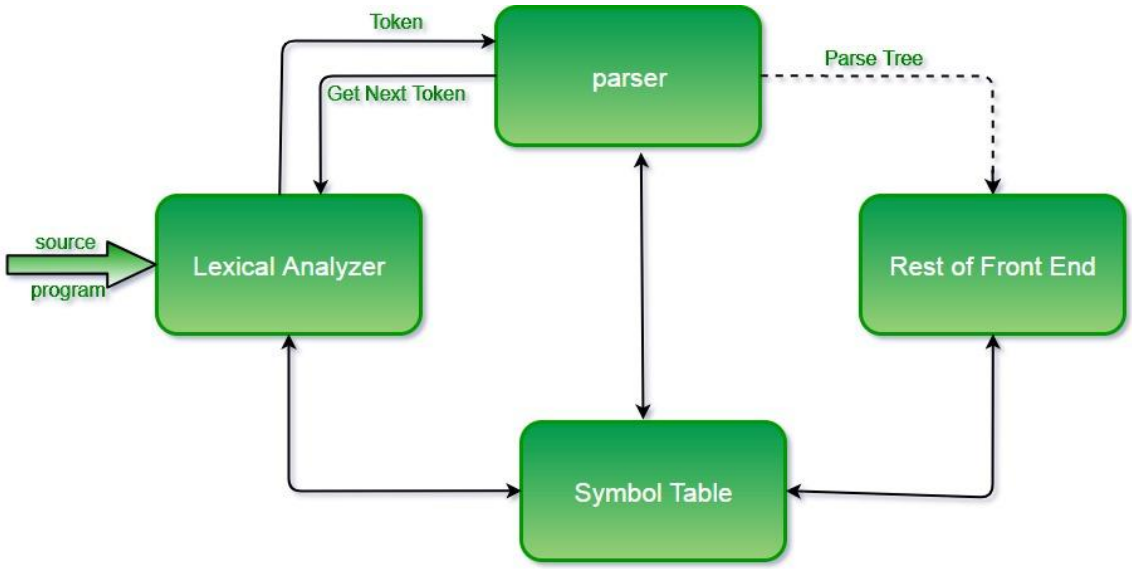
The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

UNIT –II
SYNTAX ANALYSIS AND TOP DOWN PARSING

1a) Explain the role of parser. 4M

Role of the parser :

In the syntax analysis phase, a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. This is done by a parser. The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.



1b) Define Context Free Grammar with example. 4M

Context-Free Grammar

A context-free grammar has four components :G=(V,T,P,S)

- A set of non-terminals (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as terminal symbols (Σ). Terminals are the basic symbols from which strings are formed.

- A set of productions (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings.
- One of the non-terminals is designated as the start symbol (S); from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Example

Definition (Context-Free Grammar) : A 4-tuple G = < V , Σ , S , P > is a context-free grammar (CFG) if V and Σ are finite sets sharing no elements between them, S $\in$ V is the start symbol, and P is a finite set of productions of the form X -> α , where X $\in$ V , and α $\in$ (V $\cup$ Σ)* .
A language is a context-free language (CFL) if all of its strings are generated by a context-free grammar.

Example 1: L_1 = { $a^n b^n$ | n is a positive integer } is a context-free language. For the following context-free grammar G_1 = < V_1 , Σ, S , P_1 > generates L_1 :
V_1 = { S } , Σ = { a , b } and P_1 = { S -> aSb , S -> ab }.

1c) Compare left most and right most derivations with examples ? 6M

Yield Of Parse Tree-

Concatenating the leaves of a parse tree from the left produces a string of terminals.

This string of terminals is called as yield of a parse tree.

Leftmost Derivation-

The process of deriving a string by expanding the leftmost non-terminal at each step is called as leftmost derivation.

The geometrical representation of leftmost derivation is called as a leftmost derivation tree.

Example-

Consider the following grammar-

$$S \rightarrow aB / bA$$

$$S \rightarrow aS / bAA / a$$

B → bS / aBB / b

(Unambiguous Grammar)

Let us consider a string w = aaabbabbba

Now, let us derive the string w using leftmost derivation.

Leftmost Derivation-

S → aB

→ aaBB (Using B → aBB)

→ aaaBBB (Using B → aBB)

→ aaabBB (Using B → b)

→ aaabbB (Using B → b)

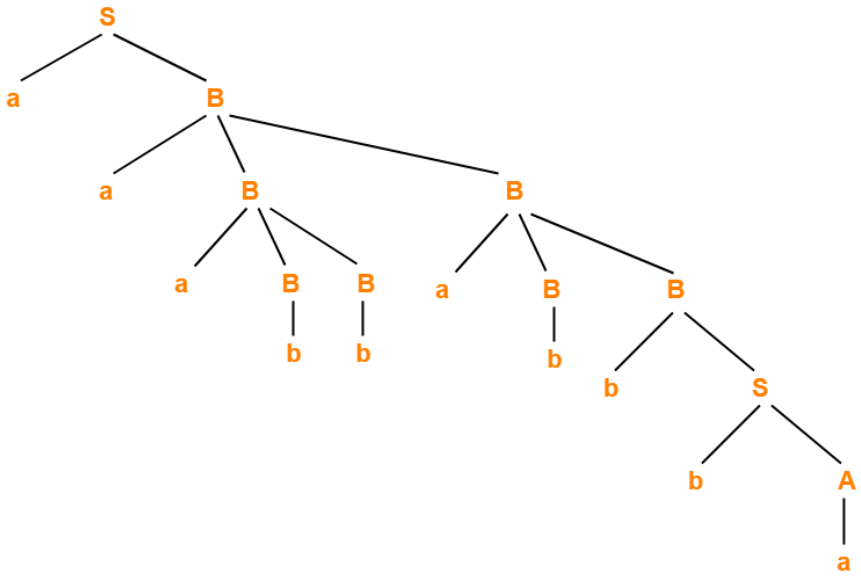
→ aaabbaBB (Using B → aBB)

→ aaabbabB (Using B → b)

→ aaabbabbS (Using B → bS)

→ aaabbabbbA (Using S → bA)

→ aaabbabbba (Using A → a)



Leftmost Derivation Tree

Rightmost Derivation-

The process of deriving a string by expanding the rightmost non-terminal at each step is called as rightmost derivation.

The geometrical representation of rightmost derivation is called as a rightmost derivation tree.

Example-

Consider the following grammar-

$$S \rightarrow aB / bA$$

$$S \rightarrow aS / bAA / a$$

$$B \rightarrow bS / aBB / b$$

(Unambiguous Grammar)

Let us consider a string w = aaabbabbba

Now, let us derive the string w using rightmost derivation.

Rightmost Derivation-

S → aB

→ aaBB (Using B → aBB)

→ aaBaBB (Using B → aBB)

→ aaBaBbS (Using B → bS)

→ aaBaBbbA (Using S → bA)

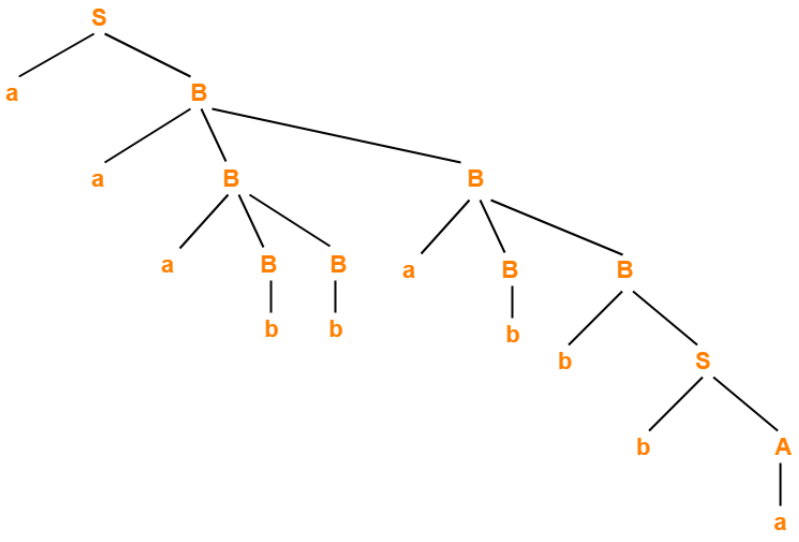
→ aaBaBbba (Using A → a)

→ aaBabbba (Using B → b)

→ aaaBBabbba (Using B → aBB)

→ aaaBbabbba (Using B → b)

→ aaabbabbba (Using B → b)



Rightmost Derivation Tree

Here,

The given grammar was unambiguous.

That is why, leftmost derivation and rightmost derivation represents the same parse tree.

2a) Define parse tree. 2M

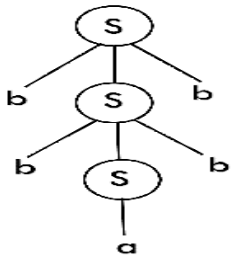
PARSE TREE :

Derivation tree is a graphical representation for the derivation of the given production rules of the context free grammar (CFG).

It is a way to show how the derivation can be done to obtain some string from a given set of production rules. It is also called as the Parse tree.

Draw a derivation tree for the string "bab" from the CFG given by

S → bSb | a | b



input string: bbabb

2b) Construct Leftmost and Rightmost derivation and parse tree for the string 3*2+5 from the given grammar. Also check it's ambiguity. Set of alphabets ∑ = {0,…,9, +, *, (,)}

E → I 8M

E →E + E

E →E * E

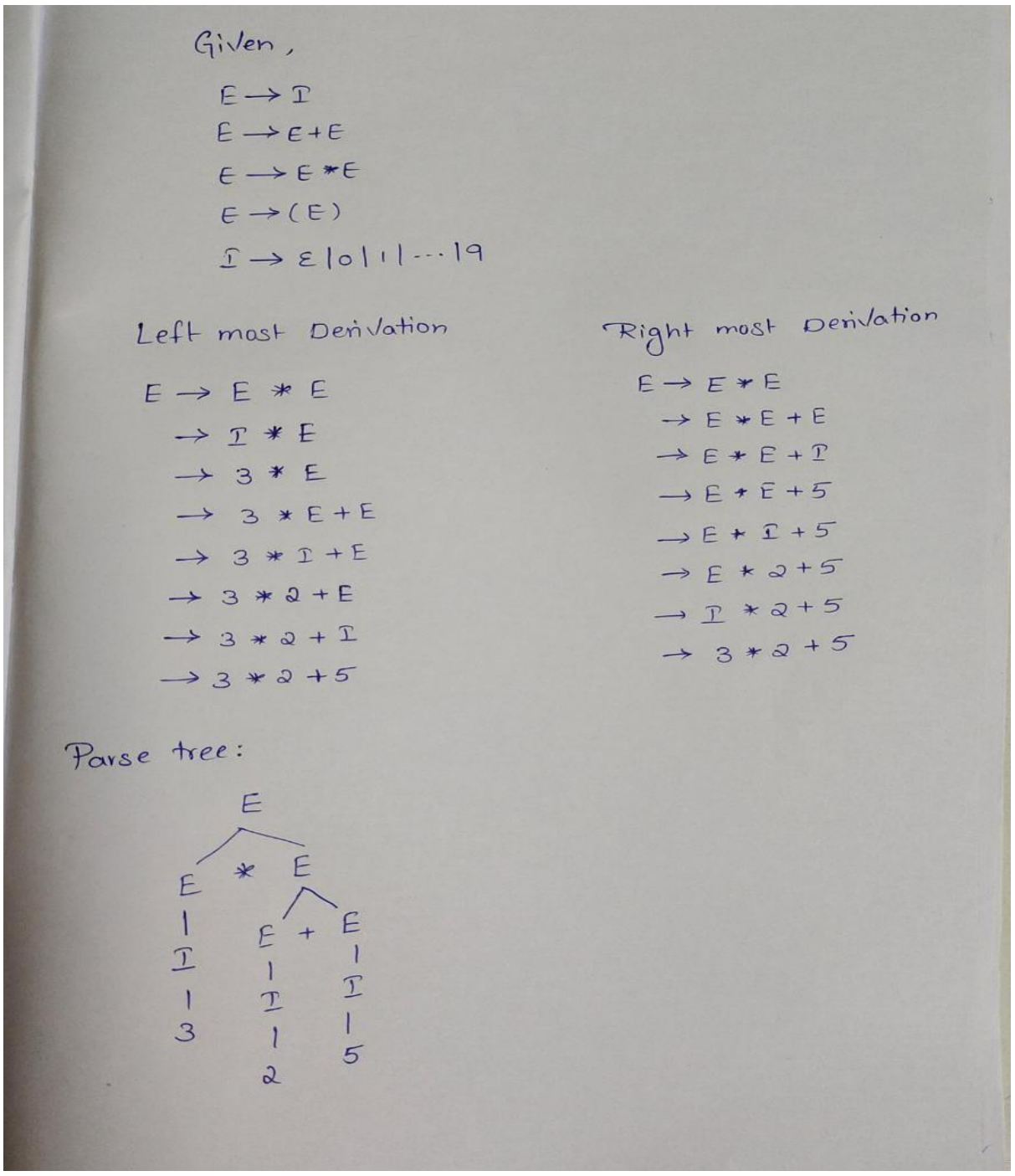
E →(E)

I →ε | 0 | 1 | … | 9

Leftmost Derivation-

- The process of deriving a string by expanding the leftmost non-terminal at each step is called as leftmost derivation.

The geometrical representation of leftmost derivation is called as a leftmost derivation tree.



Rightmost Derivation-

- The process of deriving a string by expanding the rightmost non-terminal at each step is called as rightmost derivation.
- The geometrical representation of rightmost derivation is called as a rightmost derivation tree.

3a) Define Ambiguity. 2M

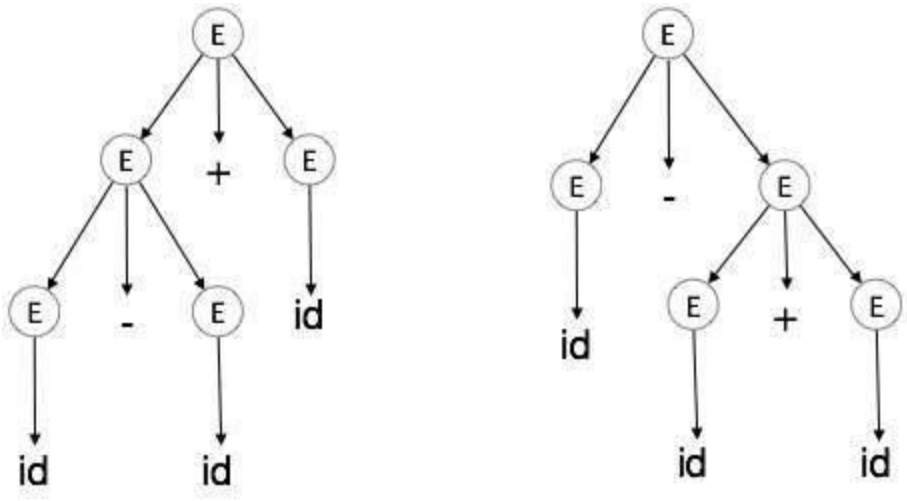
Ambiguity

A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

Example

E → E + E E → E – E E → id

For the string id + id – id, the above grammar generates two parse trees:



3b) Interpret how to eliminate ambiguity for the given Ambiguous Grammar. 10M

Converting Ambiguous Grammar Into Unambiguous Grammar-

- Causes such as left recursion, common prefixes etc makes the grammar ambiguous.
- The removal of these causes may convert the grammar into unambiguous grammar.
- However, it is not always compulsory.

Removing Ambiguity By Precedence & Associativity Rules-

An ambiguous grammar may be converted into an unambiguous grammar by implementing-

Precedence Constraints
Associativity Constraints

These constraints are implemented using the following rules-

Rule-01:

The precedence constraint is implemented using the following rules-

- ✓ The level at which the production is present defines the priority of the operator contained in it.
- ✓ The higher the level of the production, the lower the priority of operator.
- ✓ The lower the level of the production, the higher the priority of operator.

Rule-02:

The associativity constraint is implemented using the following rules-

- ✓ If the operator is left associative, induce left recursion in its production.
- ✓ If the operator is right associative, induce right recursion in its production.

PROBLEMS BASED ON CONVERSION INTO UNAMBIGUOUS GRAMMAR-

Problem-01:

Convert the following ambiguous grammar into unambiguous grammar-

$$R \rightarrow R + R / R \,.\, R / R^* / a / b$$

where * is kleen closure and . is concatenation.

Solution-

To convert the given grammar into its corresponding unambiguous grammar, we implement the precedence and associativity constraints.

We have-

- Given grammar consists of the following operators-

  $$+ \,,\, . \,,\, *$$

- Given grammar consists of the following operands-

a , b

The priority order is-

(a , b) > * > . > +

where-

- . operator is left associative
- + operator is left associative

Using the precedence and associativity rules, we write the corresponding unambiguous grammar as-

$E \rightarrow E + T / T$

$T \rightarrow T . F / F$

$F \rightarrow F^* / G$

$G \rightarrow a / b$

Unambiguous Grammar

OR

$E \rightarrow E + T / T$

$T \rightarrow T . F / F$

$F \rightarrow F^* / a / b$

Unambiguous Grammar

Note: you **can refer the handwritten pdf unit -2 for this question also .**

4a) What is left recursion? Describe the procedure of eliminating Left recursion. 4M

Left Recursion

A grammar becomes left-recursive if it has any non-terminal ‘A’ whose derivation contains ‘A’ itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

Removal of Left Recursion

One way to remove left recursion is to use the following technique:

The production

A => Aα | β

is converted into following productions

A => βA'
A'=> αA' | ε

This does not impact the strings derived from the grammar, but it removes immediate left recursion.

Example

The production set

S => Aα | β
A => Sd

after applying the above algorithm, should become

S => Aα | β
A => Aαd | βd

and then, remove immediate left recursion using the first technique.

A => βdA'
A' => αdA' | ε

Now none of the production has either direct or indirect left recursion.

4b) Eliminate left recursion for the following grammar 4M

E→E+T/T
T→T*F/F
F→(E)/id

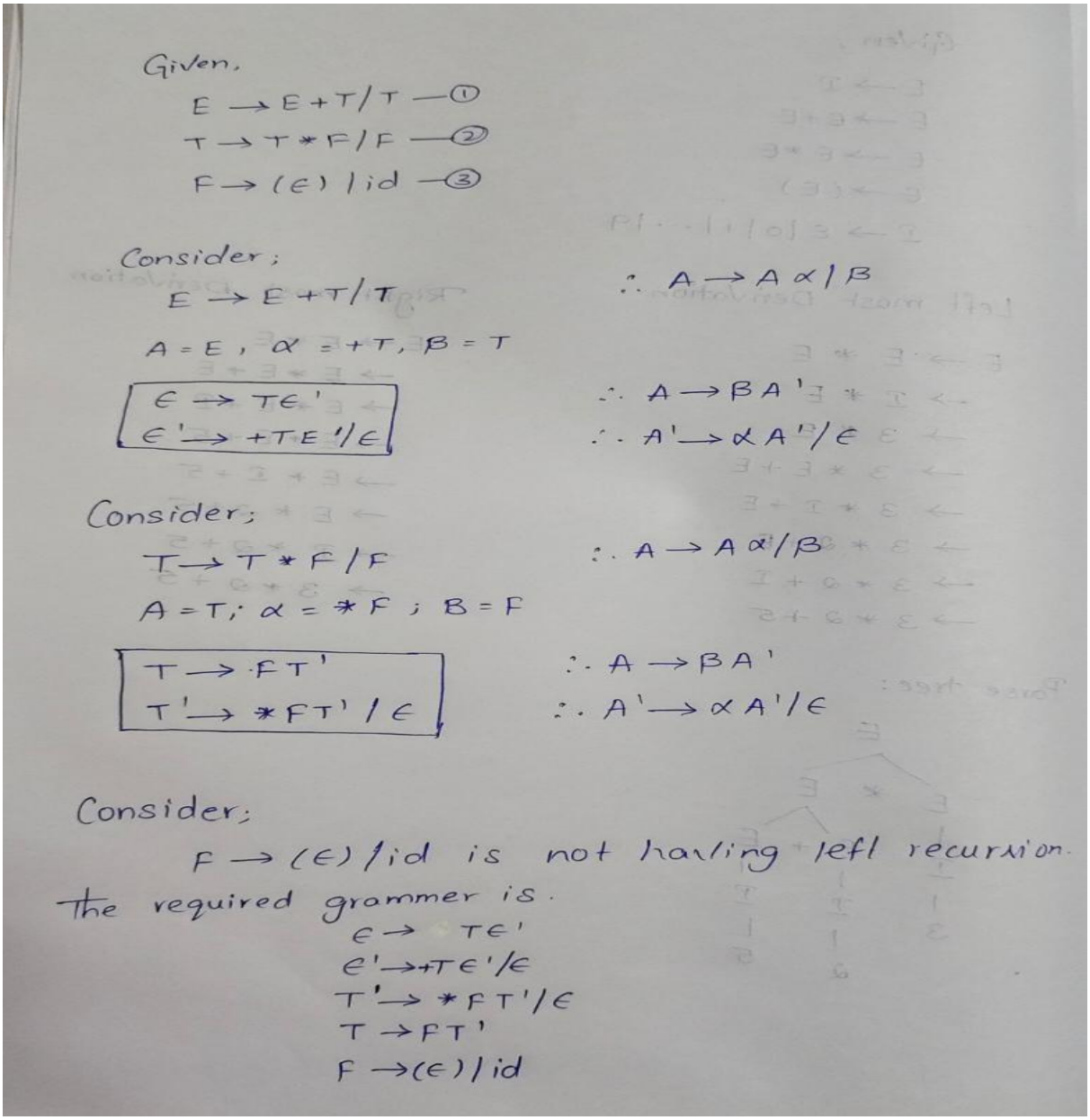
Removal of Left Recursion

One way to remove left recursion is to use the following technique:

The production

A => Aα | β is converted into following productions
A => βA'

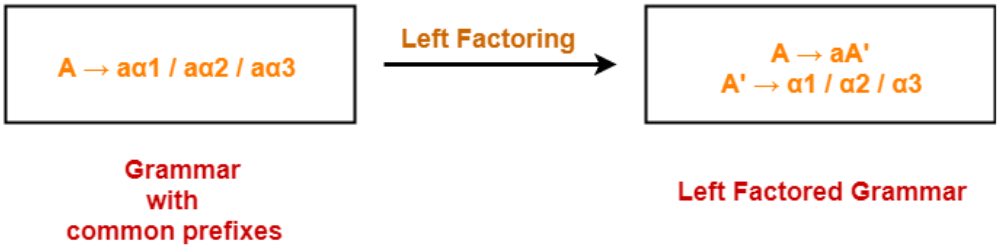
A'=> αA' | ε This does not impact the strings derived from the grammar, but it removes immediate left recursion.



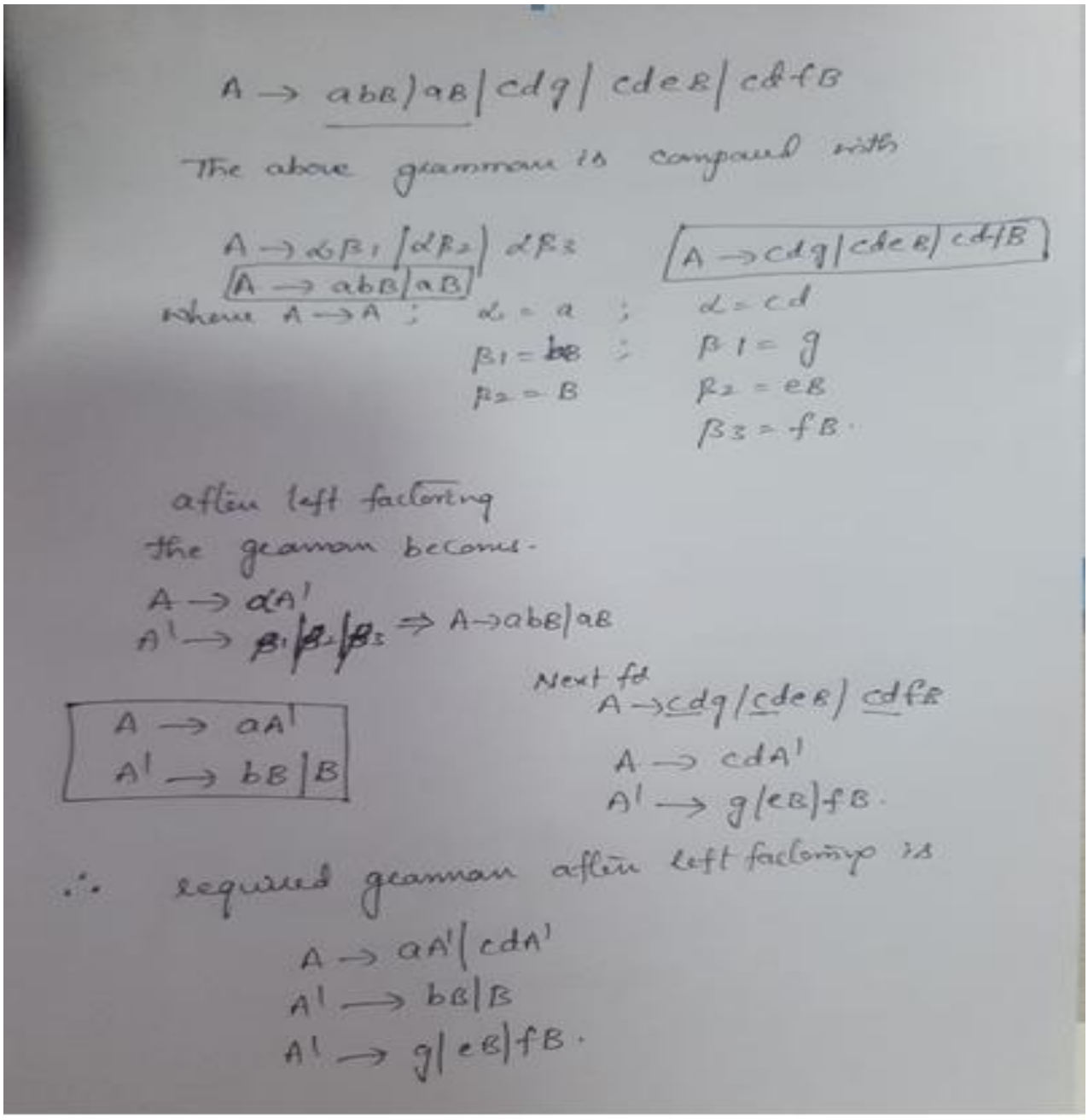
4c) Show what do you understand by Left factoring. Perform left factor for the grammar A→ abB/aB/cdg/cdeB/cdfB 4M

Left Factoring:

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.



Perform left factor for the grammar A→abB/aB/cdg/cdeB/cdfB



5a) List the types of Parsers available 4M

Parser

Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.

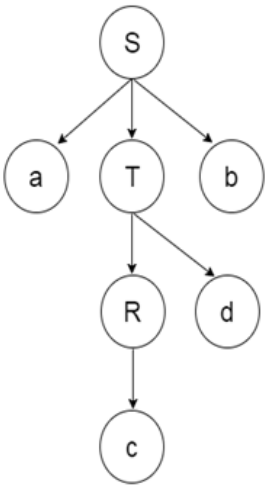
A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.

Parsing is of two types: top down parsing and bottom up parsing.

Top down paring

- The top down parsing is known as recursive parsing or predictive parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the top down parsing, the parsing starts from the start symbol and transform it into the input symbol.

Parse Tree representation of input string "acdb" is as follows:



Bottom up parsing

- Bottom up parsing is also known as shift-reduce parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse.

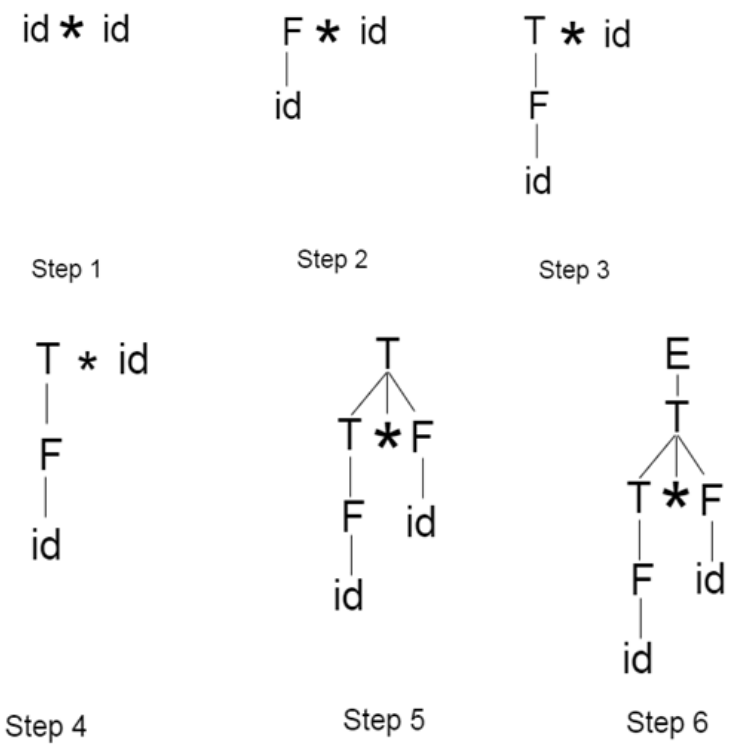
Example

Production

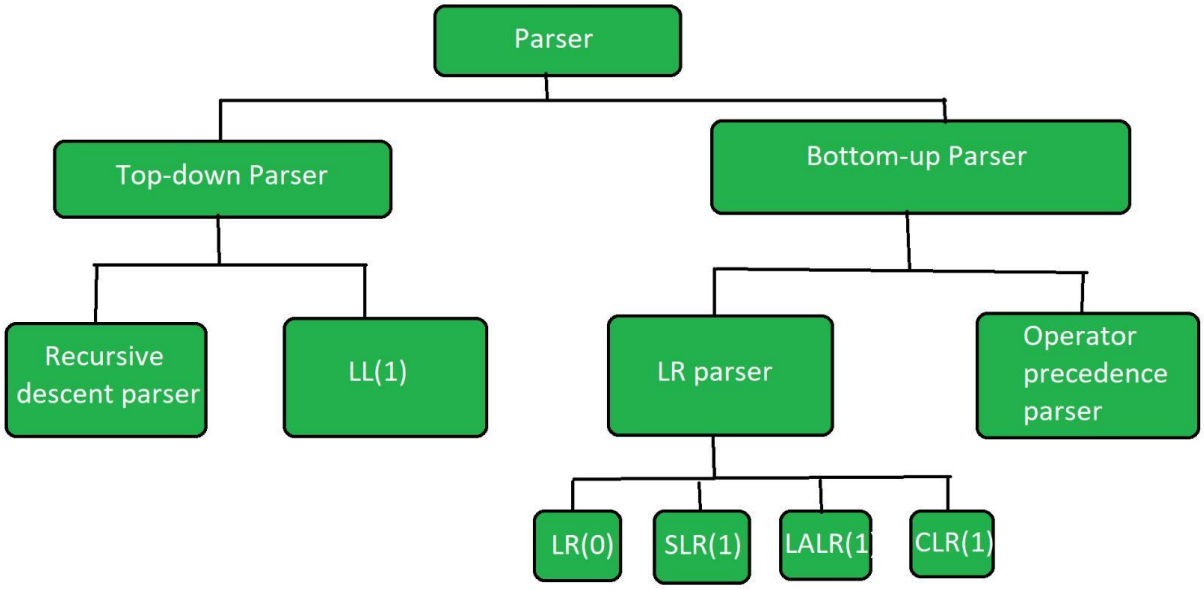
1. E → T
2. T → T * F
3. T → id
4. F → T
5. F → id

Parse Tree representation of input string "id * id" is as follows:

Parse Tree representation of input string "id * id" is as follows:



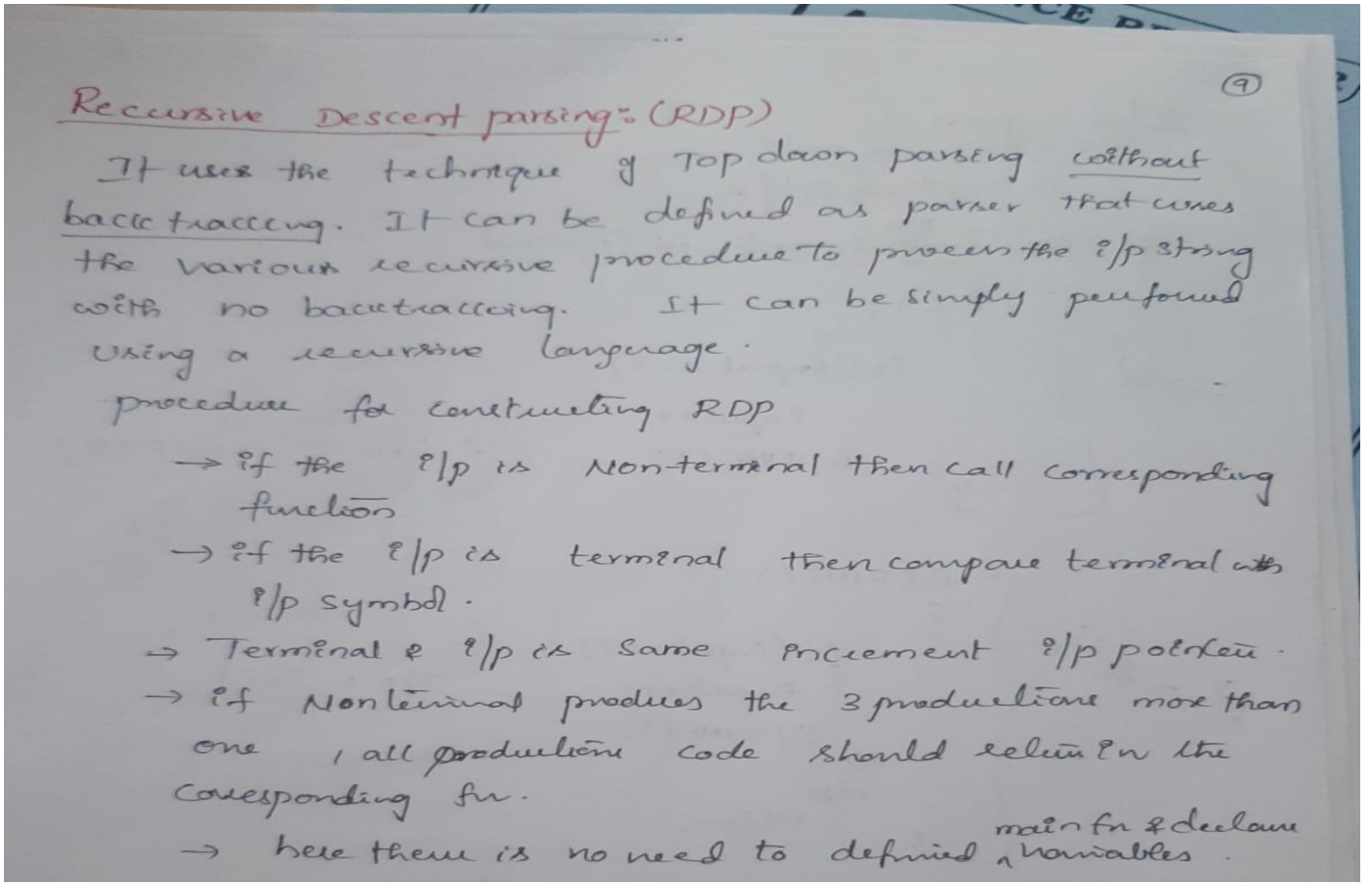
.



Classification of Parser

5b) Design the recursive decent parser for the following grammar 8M

E→ E+T/T T→T*F/F F→(E)/id



⑨

Recursive Descent parsing: (RDP)

It uses the technique of Top down parsing without backtracking. It can be defined as parser that uses the various recursive procedure to process the i/p string with no backtracking. It can be simply performed using a recursive language.

procedure for constructing RDP

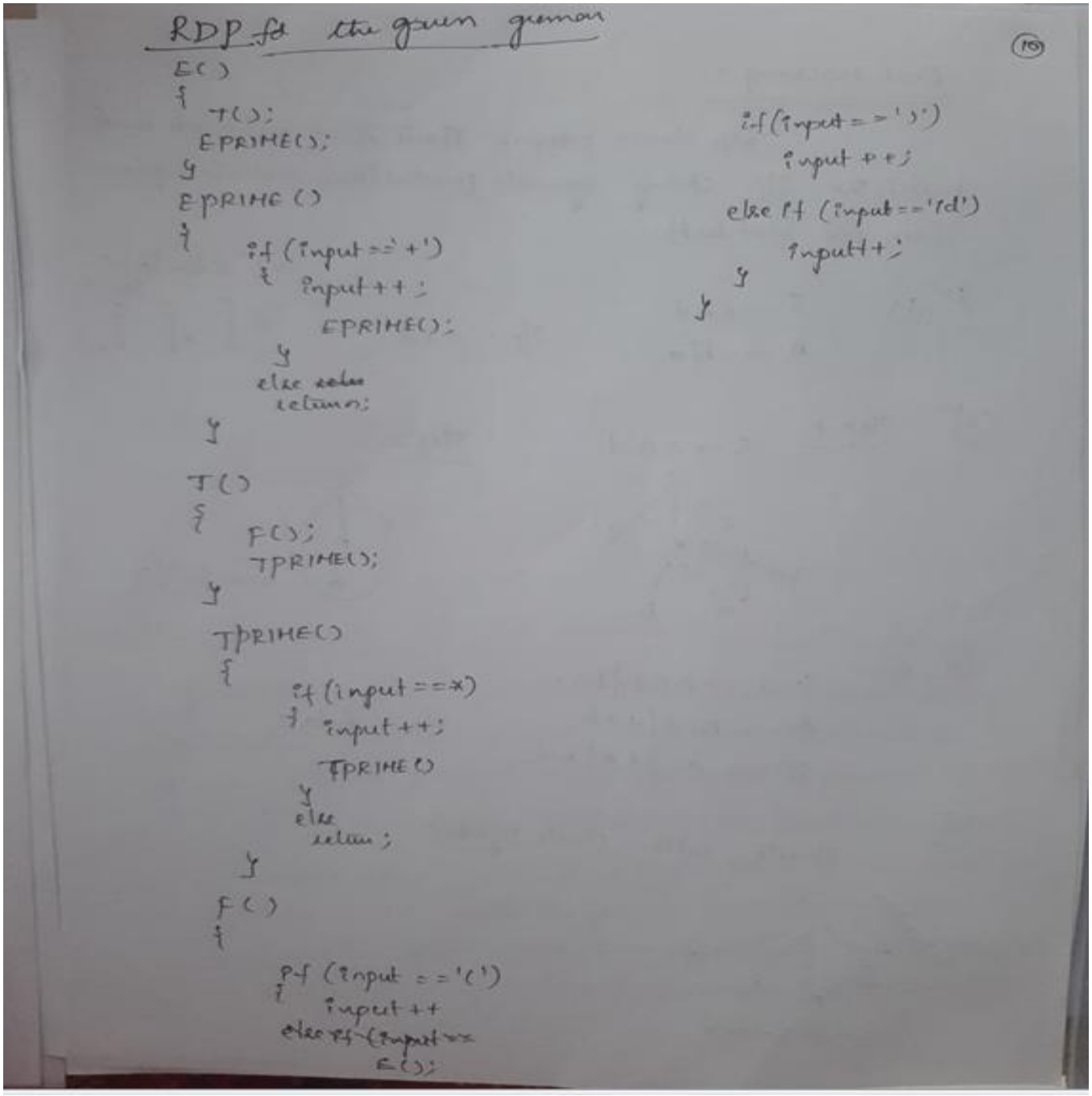
→ if the i/p is Non-terminal then call corresponding function

→ if the i/p is terminal then compare terminal with i/p symbol.

→ Terminal & i/p is same increment i/p pointer.

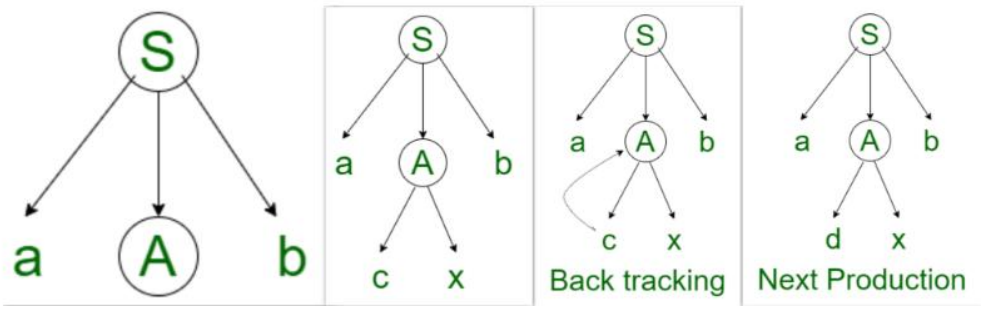
→ if Non terminal produces the 3 productions more than one, all productions code should return in the corresponding fn.

→ here there is no need to defined main fn & declare variables.



6a) What is meant by Non-recursive predictive parsing 2M

Non-Recursive Predictive Descent Parser : A form of recursive-descent parsing that does not require any back-tracking **is known as predictive parsing. It is also called as LL(1) parsing table technique since we would be building a table for string to be parsed.**



6b) Illustrate the rules to be followed in finding the FIRST and FOLLOW.

6M

First Set

This set is created to know what terminal symbol is derived in the first position by a non-terminal.

Rules For Calculating First Function-

Rule-01:

For a production rule $X \rightarrow \in$,

$$First(X) = \{ \in \}$$

Rule-02:

For any terminal symbol 'a',

$$First(a) = \{ a \}$$

Rule-03:

For a production rule $X \rightarrow Y_1Y_2Y_3$,

Calculating First(X)

If $\in \notin First(Y_1)$, then $First(X) = First(Y_1)$

If $\in \in First(Y_1)$, then $First(X) = \{ First(Y_1) - \in \} \cup First(Y_2Y_3)$

Calculating $First(Y_2Y_3)$

If $\in \notin First(Y_2)$, then $First(Y_2Y_3) = First(Y_2)$

If $\in \in First(Y_2)$, then $First(Y_2Y_3) = \{ First(Y_2) - \in \} \cup First(Y_3)$

Similarly, we can make expansion for any production rule $X \rightarrow Y_1Y_2Y_3.....Y_n$.

Follow Function-

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Rules For Calculating Follow Function-

Rule-01:

For the start symbol S, place $ in Follow(S).

Rule-02:

For any production rule A → αB,

Follow(B) = Follow(A)

Rule-03:

For any production rule A → αBβ,

If ∈ ∉ First(β), then Follow(B) = First(β)
If ∈ ∈ First(β), then Follow(B) = { First(β) – ∈ } ∪ Follow(A)

6c) Find FIRST and FOLLOW for the following grammar? 4M

E→ E+T/T
T→T*F/F
F→(E)/id

We have-

The given grammar is left recursive.
So, we first remove left recursion from the given grammar.

After eliminating left recursion, we get the following grammar-

E → TE'

E' → + TE' / ∈

T → FT'

T' →* FT' / ∈

F → (E) / id

Now, the first and follow functions are as follows-

First Functions-

First(E) = First(T) = First(F) = { (, id }
First(E') = { + , ∈ }
First(T) = First(F) = { (, id }
First(T') = { * , ∈ }
First(F) = { (, id }

Follow Functions-

Follow(E) = { $,) }

Follow(E') = Follow(E) = { $,) }

Follow(T) = { First(E') – ∈ } ∪ Follow(E) ∪ Follow(E') = { + , $,) }

Follow(T') = Follow(T) = { + , $,) }

Follow(F) = { First(T') – ∈ } ∪ Follow(T) ∪ Follow(T') = { * , + , $,) }

7) Consider the grammar **12M**

S→AB|ABad

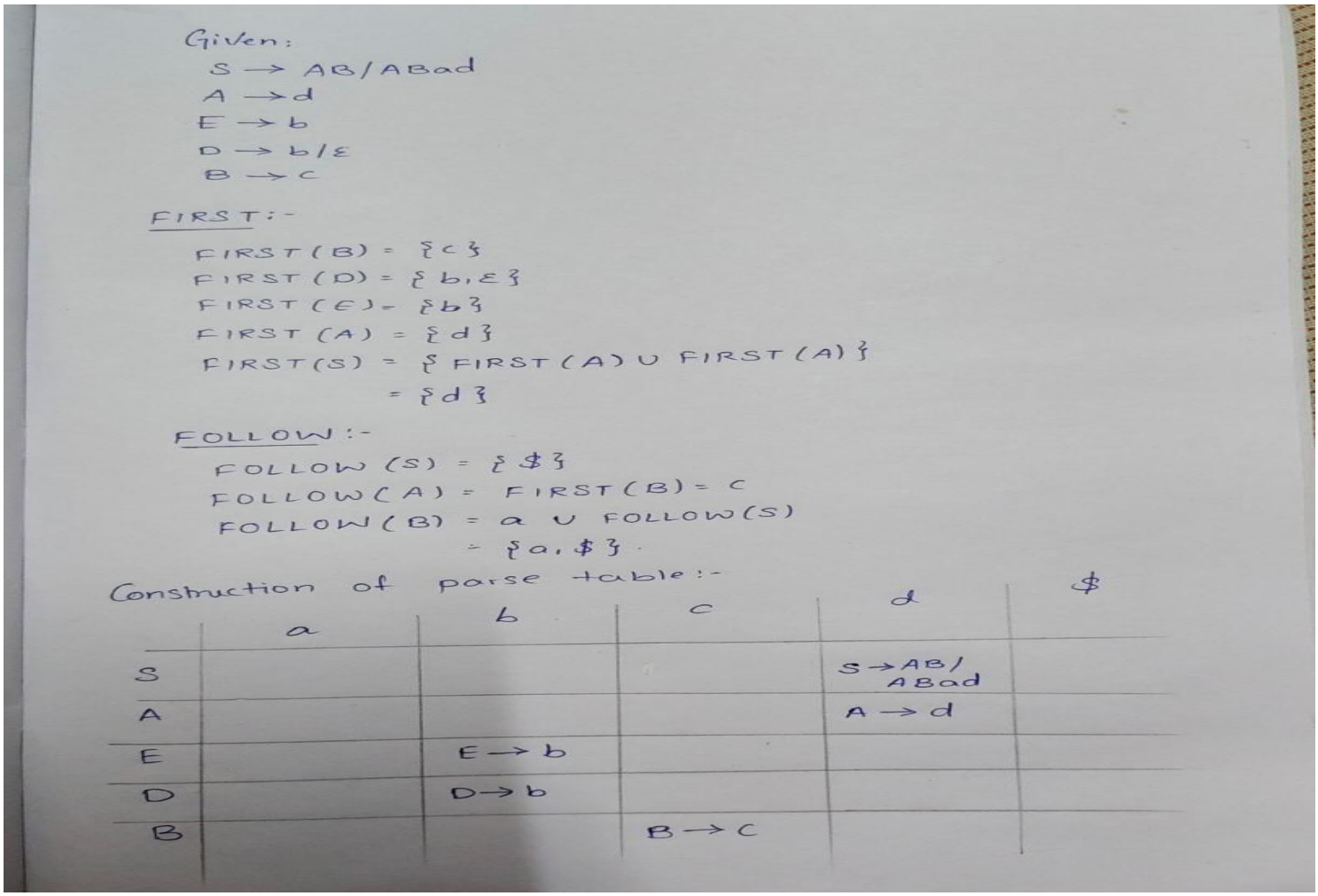
A→d

E →b

D→b| ε

B→c

Construct the predictive parse table and check whether the given grammar is LL(1) or not.



The grammar is not LL(1) .

8) Consider the grammar

$E \rightarrow TE^1$

$E^1 \rightarrow +TE^1 \mid -TE^1 \mid \varepsilon$ **12M**

$T \rightarrow FT^1$

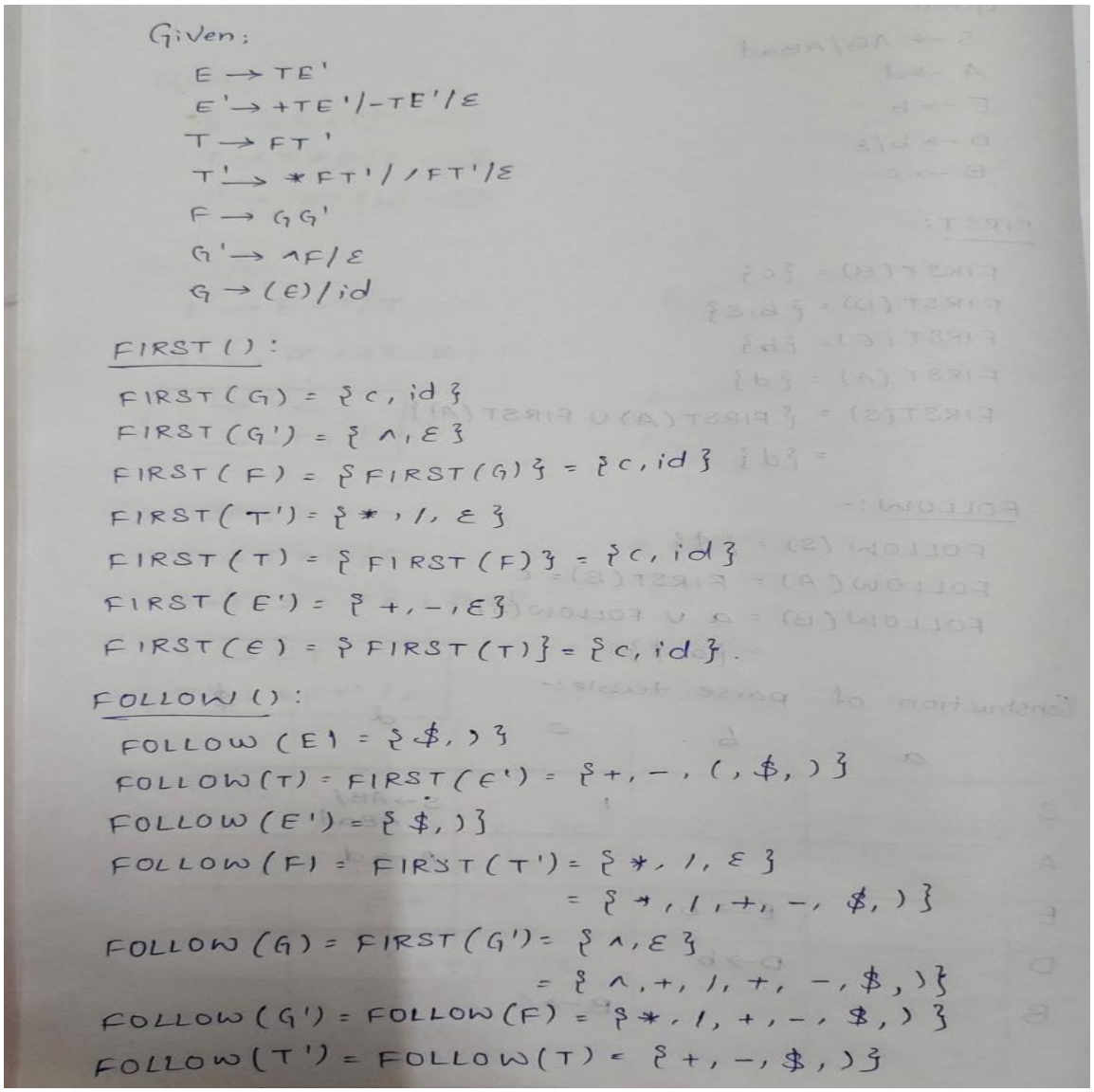
$T^1 \rightarrow *FT^1 \mid / FT^1 \mid \varepsilon$

$F \rightarrow GG^1$

$G^1 \rightarrow \wedge F / \varepsilon$

$G \rightarrow (E) / id$

Calculate FIRST and FOLLOW for the above grammar



Given:

$E \rightarrow TE'$

$E' \rightarrow +TE' / -TE' / \varepsilon$

$T \rightarrow FT'$

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

$F \rightarrow GG'$

$G' \rightarrow \wedge F / \varepsilon$

$G \rightarrow (E) / id$

FIRST ( ):

FIRST (G) = { c, id }

FIRST (G') = { ∧, ε }

FIRST (F) = { FIRST (G) } = { c, id }

FIRST (T') = { *, /, ε }

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

FIRST (E') = { +, −, ε }

FIRST (E) = { FIRST (T) } = { c, id }.

FOLLOW ( ):

FOLLOW (E) = { $,) }

FOLLOW (T) = FIRST (E') = { +, −, (, $,) }

FOLLOW (E') = { $,) }

FOLLOW (F) = FIRST (T') = { *, /, ε }

= { *, /, +, −, $,) }

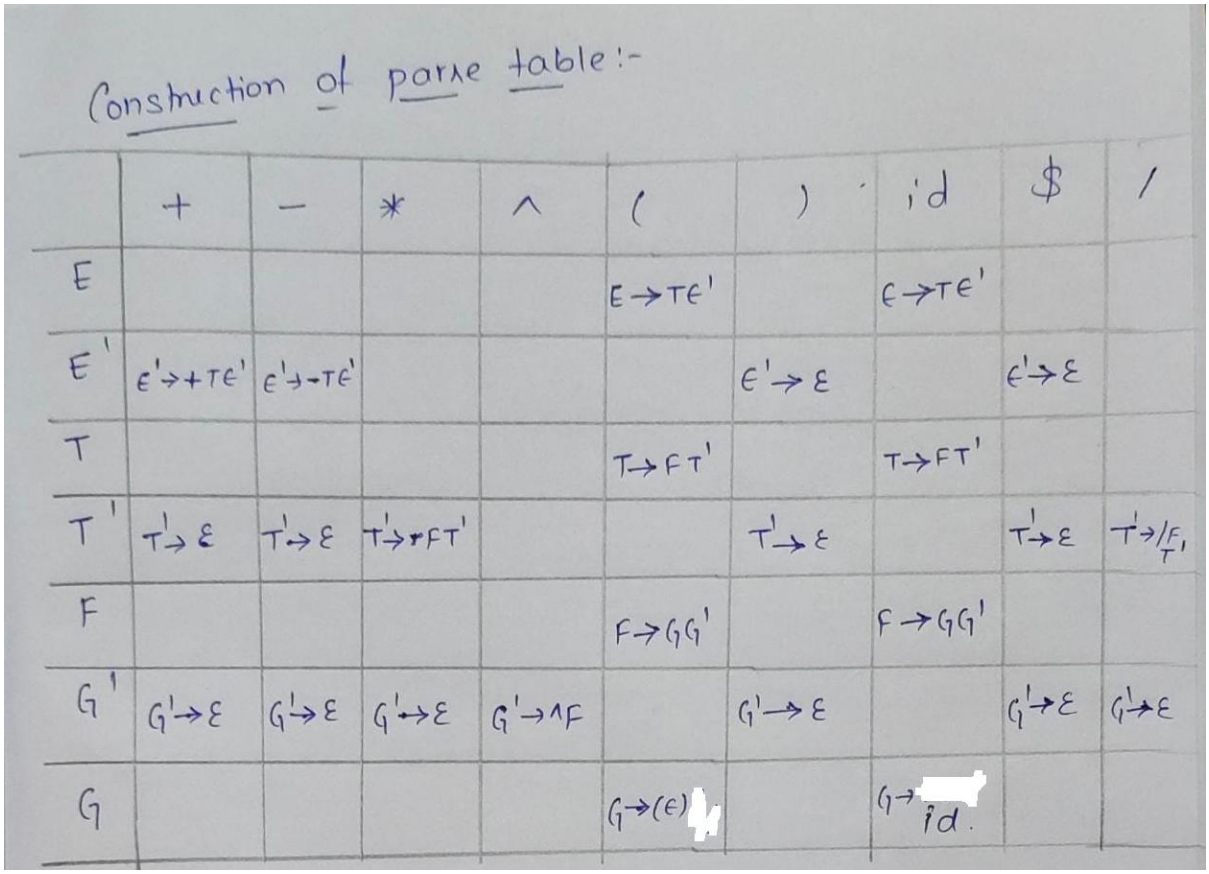
FOLLOW (G) = FIRST (G') = { ∧, ε }

= { ∧, +, /, +, −, $,) }

FOLLOW (G') = FOLLOW (F) = { *, /, +, −, $,) }

FOLLOW (T') = FOLLOW (T) = { +, −, $,) }

Construction LL(1) for the given grammar



The given grammar is LL(1).

9) Consider the grammar **12M**

E→E+T/T

T→T*F/F

F→(E)|id

Design predictive parsing table and check given grammar is LL(1) or not?

The given grammar is left recursive.

So, we first remove left recursion from the given grammar.

After eliminating left recursion, we get the following grammar-

$E \rightarrow TE'$

$E' \rightarrow + TE' / \in$

$T \rightarrow FT'$

$T' \rightarrow x FT' / \in$

$F \rightarrow (E) / id$

Now, the first and follow functions are as follows-

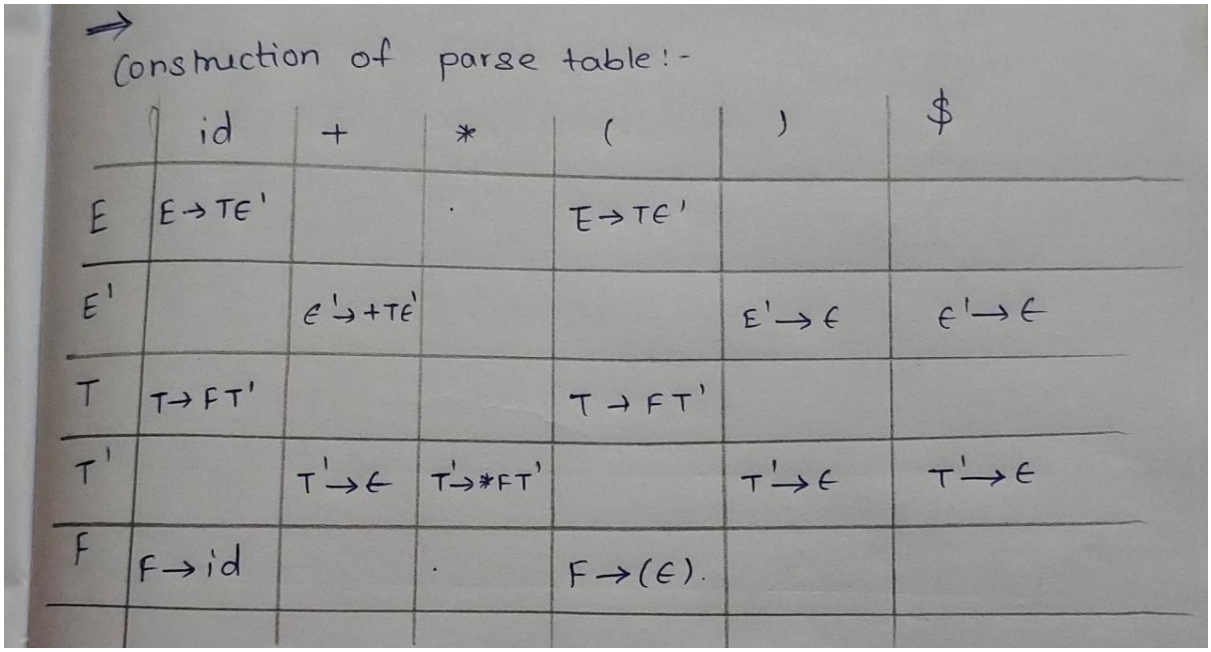
First Functions-

First(E) = First(T) = First(F) = { (, id }
First(E') = { + , ∈ }
First(T) = First(F) = { (, id }
First(T') = { x , ∈ }
First(F) = { (, id }

Follow Functions-

Follow(E) = { $,) }

Follow(E') = Follow(E) = { $,) }
Follow(T) = { First(E') – ∈ } ∪ Follow(E) ∪ Follow(E') = { + , $,) }
Follow(T') = Follow(T) = { + , $,) }
Follow(F) = { First(T') – ∈ } ∪ Follow(T) ∪ Follow(T') = { x , + , $,) }



The above grammar is LL(1) .

10a) Discuss the types of errors. **6M**

Types or **Sources of Error –** There are three types of error: logic, run-time and compile-time error:

1. **Logic errors** occur when programs operate incorrectly but do not terminate abnormally (or crash). Unexpected or undesired outputs or other behaviour may result from a logic error, even if it is not immediately recognized as such.

2. A run-time error is an error that takes place during the execution of a program and usually happens because of adverse system parameters or invalid input data. The lack of sufficient memory to run an application or a memory conflict with another program and logical error is an example of this. Logic errors occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.

3. Compile-time errors rise at compile-time, before the execution of the program. Syntax error or missing file reference that prevents the program from successfully compiling is an example of this.

Note: write examples for each type of errors.

Classification of Compile-time error –

Lexical : This includes misspellings of identifiers, keywords or operators
Syntactical : a missing semicolon or unbalanced parenthesis
Semantical : incompatible value assignment or type mismatches between operator and operand
Logical : code not reachable, infinite loop.

Finding error or reporting an error – Viable-prefix is the property of a parser that allows early detection of syntax errors.

Goal detection of an error as soon as possible without further consuming unnecessary input

How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language.

Example: for(;), this will report an error as for having two semicolons inside braces.

10b) Explain Error recovery in predictive parsing with an Example. 6M

Error Recovery Strategies In Parsing

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- **Lexical** : name of some identifier typed incorrectly
- **Syntactical** : missing semicolon or unbalanced parenthesis

- **Semantical** : incompatible value assignment
- **Logical** : code not reachable, infinite loop

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

SIDDARTHA INSTITUTE OF SCIENCE AND TECHNOLOGY:: PUTTUR
Siddharth Nagar, Narayanavanam Road – 517583

QUESTION BANK (DESCRIPTIVE)

Subject with Code : Compiler Design (20CS0516) **Course & Branch :** B. Tech - CSE

Year & Sem : III B.Tech & I-Sem **Regulation :** R20

UNIT –III
BOTTOM UP PARSING AND SEMANTIC ANALYSIS

1a) Explain about handle pruning [L2][CO1][6M]

HANDLE PRUNING:

Removing the children of the left-hand side non-terminal from the parse tree is called Handle Pruning.
A rightmost derivation in reverse can be obtained by handle pruning.

Sentential form: S => a here, 'a' is called sentential form, 'a' can be a mix of terminals and non terminals.
Consider Grammar : S -> aSa | bSb | ε

Derivation: S => aSa => abSba => abbSbba => abbbba

Left Sentential and Right Sentential Form:

- A left-sentential form is a sentential form that occurs in the leftmost derivation of some sentence.
- A right-sentential form is a sentential form that occurs in the rightmost derivation of some sentence.

Example 1:

Right Sequential Form	Handle	Reducing Production
id + id * id	id	E ⇒ id
E + id * id	id	E ⇒ id
E + E * id	id	E ⇒ id
E + E * E	E + E	E ⇒ E + E
E * E	E * E	E ⇒ E * E
E (Root)		

Example 2:

Right Sequential Form	Handle	Production
id + id + id	id	E ⇒ id
E + id + id	id	E ⇒ id
E + **E** + id	id	E ⇒ id
E + **E** + **E**	**E** + **E**	E ⇒ E + E
E + **E**	**E** + **E**	E ⇒ E + E
E (Root)		

1b) Summarize about LR parsing [L2][CO1][6M]

LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

SLR(1) – Simple LR Parser:

- ✓ Works on smallest class of grammar
- ✓ Few number of states, hence very small table
- ✓ Simple and fast construction

LR(1) – LR Parser:

- ✓ Works on complete set of LR(1) Grammar
- ✓ Generates large table and large number of states
- ✓ Slow construction

LALR(1) – Look-Ahead LR Parser:

- ✓ Works on intermediate size of grammar
- ✓ Number of states are same as in SLR(1)

For each type of LR parsing the following steps have to be followed:

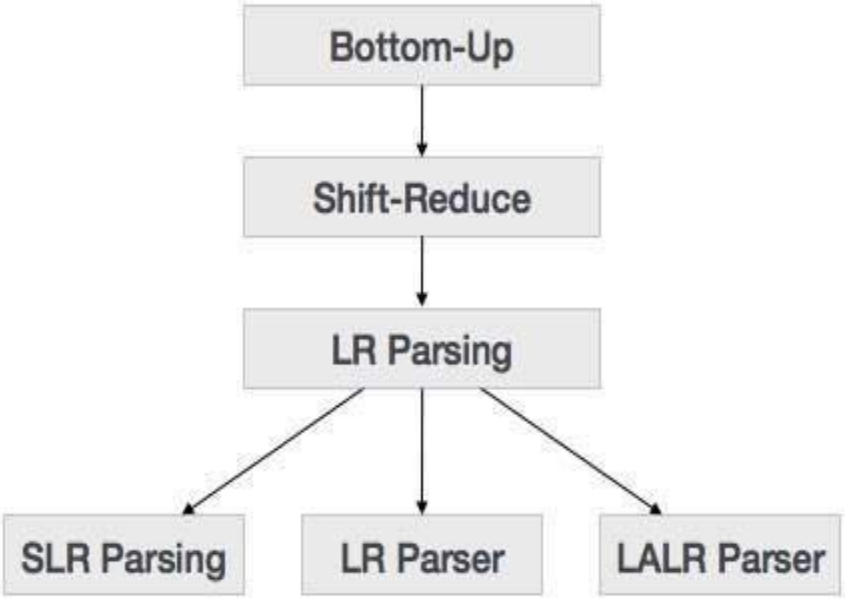
- ✓ Introduce the augmented grammar
- ✓ Construction of LR(0) items for SLR and LR(1) Items for CLR AND LALR parser
- ✓ Construction of DFA for the given parser
- ✓ Construction of Parsing Table for the given parser
- ✓ Parse the input string for the given parser

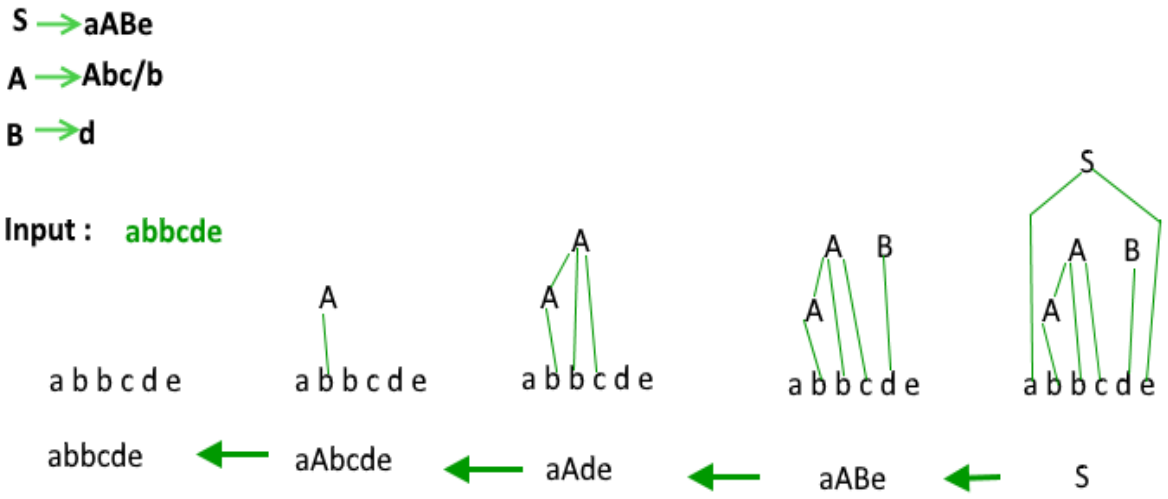
2a) Describe bottom up parsing **[L1][CO2][4M]**

BOTTOM UP PARSING

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol.

The image given below depicts the bottom-up parsers available.





2b) Differences between SLR, CLR, LALR parsers [L4][CO3][8M]

Comparison between SLR, CLR, and LALR Parser:

S. no.	SLR Parsers	Canonical LR Parsers	LALR Parsers
1	SLR parser are easiest to implement.	CLR parsers are difficult to implement.	LALR parsers are difficult to implement than SLR parser but less than CLR parsers.
2.	SLR parsers make use of canonical collection of LR(0) items for constructing the parsing tables.	CLR parsers are uses LR(1) collection of items for constructing the parsing tables part.	LALR parsers LR(1) collection , items with items having same core merged into a single itemset.
3	SLR parsers don't do any lookahead i.e., they lookahead zero	CLR parsers lookahead one symbol.	LALR parsers lookahead one symbol.
4.	SLR parsers are cost effective to construct in terms of time and space.	CLR parsers are expensive to construct in terms – of time and space.	The cost of constructing LALR parsers is i intermediate between SLR and CLR parser.
5.	SLR parsers have hundreds of states.	CLR parses have thousands of states.	LALR parsers have hundreds of state and is same as number states in SLR parsers.
6.	SLR parsers uses FOLLOW information to guide reductions.	CLR parsers uses lookahead symbol to guide reductions.	LALR parsers uses lookahead symbol to guide reductions.
7.	SLR parsers may fail to I produce a table for certain class of grammars on which ether succeed.	CLR parser works on very large class of grammar.	LALR parser works on very large class grammars.
8.	Every SLR(1) grammar is LR(1) grammar and LALR(1).	Every LR(1) grammar may not be SLR(1) grammar.	Every LALR(1) grammar may not be SLR(1) but every LALR(1) grammar is LR(1) grammar.
9.	A shift-reduce or reduce-reduce conflict may arise in SLR parsing table.	A shift-reduce or reduce-reduce conflicted may arise but chances are less than that in SLR parsing tables.	A shift-reduce conflict can not arise but a reduce-reduce conflict may arise.
10.	SLR parser is least powerful.	A CLR parsers is most powerful among the family canonical of bottom-up parsers.	A LALR parser is intermediate in power between SLR and LR parser.

3) Prepare Shift Reduce Parsing for the input string using the grammar **[L6][CO3][12M]**

S→(L)|a
L→L,S|S

a) (a,(a,a))

b) (a,a)

Consider the following grammar-

$$S \rightarrow (L) \mid a$$

$$L \rightarrow L , S \mid S$$

Parse the input string (a , (a , a)) using a shift-reduce parser.

Stack	Input Buffer	Parsing Action
$	(a , (a , a)) $	Shift
$ (	a , (a , a)) $	Shift
$ (a	, (a , a)) $	Reduce S → a
$ (S	, (a , a)) $	Reduce L → S
$ (L	, (a , a)) $	Shift
$ (L ,	(a , a)) $	Shift
$ (L , (	a , a)) $	Shift
$ (L , (a	, a)) $	Reduce S → a
$ (L , (S	, a)) $	Reduce L → S
$ (L , (L	, a)) $	Shift
$ (L , (L ,	a)) $	Shift
$ (L , (L , a	)) $	Reduce S → a
$ (L , (L , S)	)) $	Reduce L → L , S
$ (L , (L	)) $	Shift
$ (L , (L)	) $	Reduce S → (L)
$ (L , S	) $	Reduce L → L , S
$ (L	) $	Shift

$ (L)	$	Reduce S → (L)
$ S	$	Accept

Parse the input string (a,a) using Shirt Reduce parsing

For answer refer handwritten pdf unit-3

Unit-3

3(b)

parse the i/p string (a,a) using SRP

Stack	input buffer	parsing action
$	(a,a) $	shift
$(	a,a) $	shift
$(a	,a) $	reduce S→a
$(S	,a) $	reduce L→S
$(L	,a) $	shift
$(L,	a) $	shift
$(L,a	) $	reduce S→a
$(L,S	) $	reduce L→L,S
$(L	) $	shift
$(L)	$	reduce S→(L)
$S	$	accept

↓ start symbol.

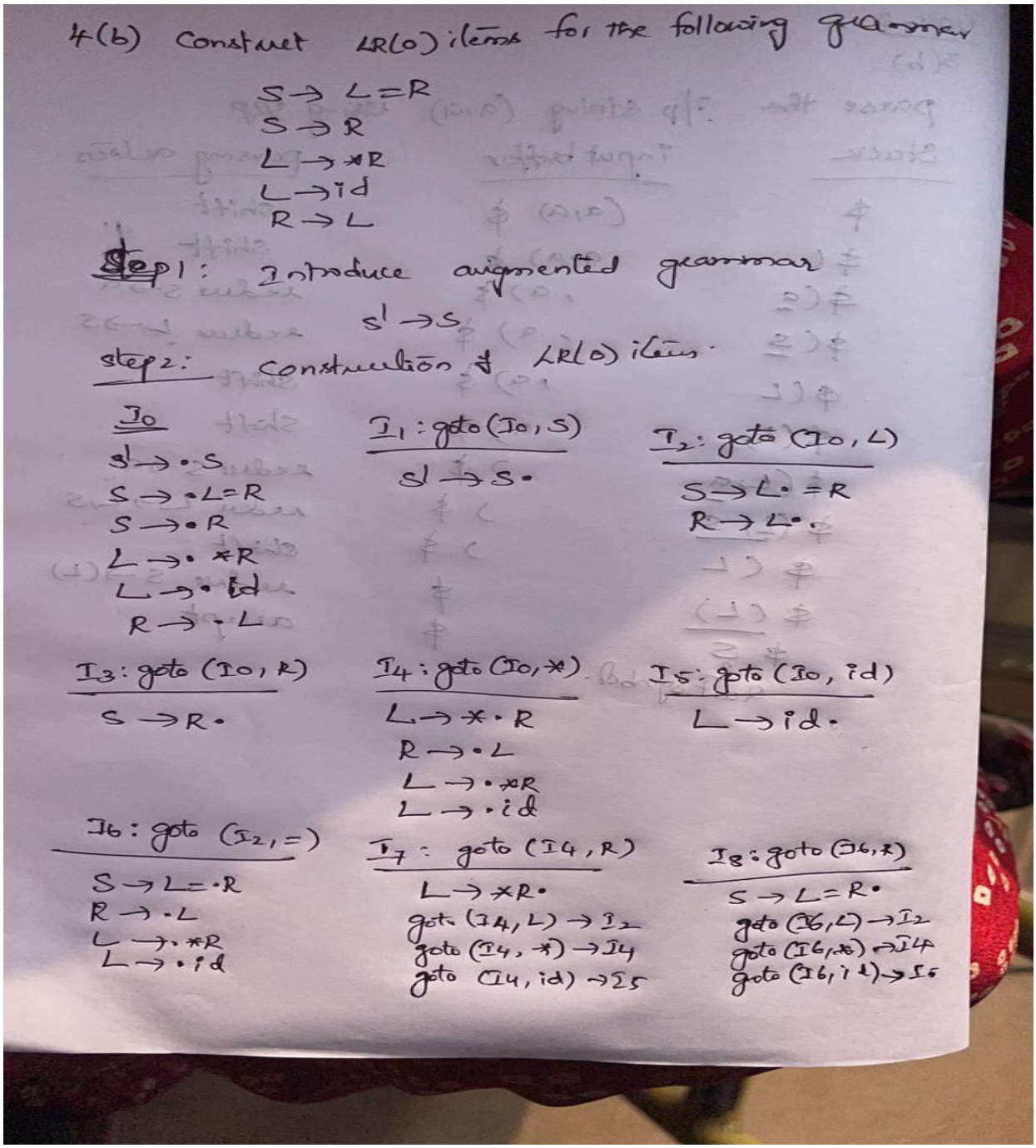
4a) Define augmented grammar. **[L1][CO2][2M]**

Augmented Grammar –

If grammar G has start symbol S, then augmented Grammar is new Grammar G′ with new start symbol S′. Also, it will contain the production S′ → S.

4b) Construct the LR(0) items for the Grammar. **[L6][CO3][10M]**

S→L=R
S→R
L→*R
L→id
R→L



5) Construct SLR Parser for the following grammar. [L6][CO3][12M]

E→E+T / T

T→TF / F

F→F* / a / b

Construct SLR parser for the following grammar

E → E+T / T

T → TF / F

F → F* / a / b

Sol:

(i) Introduce the augmented grammar

E' → E

(ii) construction of LR(0) items.

I_0

E' → .E
E → .E+T
E → .T
T → .TF
T → .F
F → .F*
F → .a
F → .b

I_1 : goto (I_0, E)

E' → E.
E → E.+T

I_2 : goto (I_0, T)

E → T.
T → T.F
F → .a
F → .b

I_3 : goto (I_0, F)

F → F.*
T → F.

I_4 : goto (I_0, a)

F → a.

I_5 : goto (I_0, b)

F → b.

I_6 : goto (I_1, +)

E → E+.T
T → .TF
T → .F
F → .F*
F → .a
F → .b

I_7 : goto (I_2, F)

T → TF.

goto (I_2, a) → I_4
goto (I_2, b) → I_5

I_8 : goto (I_3, *)

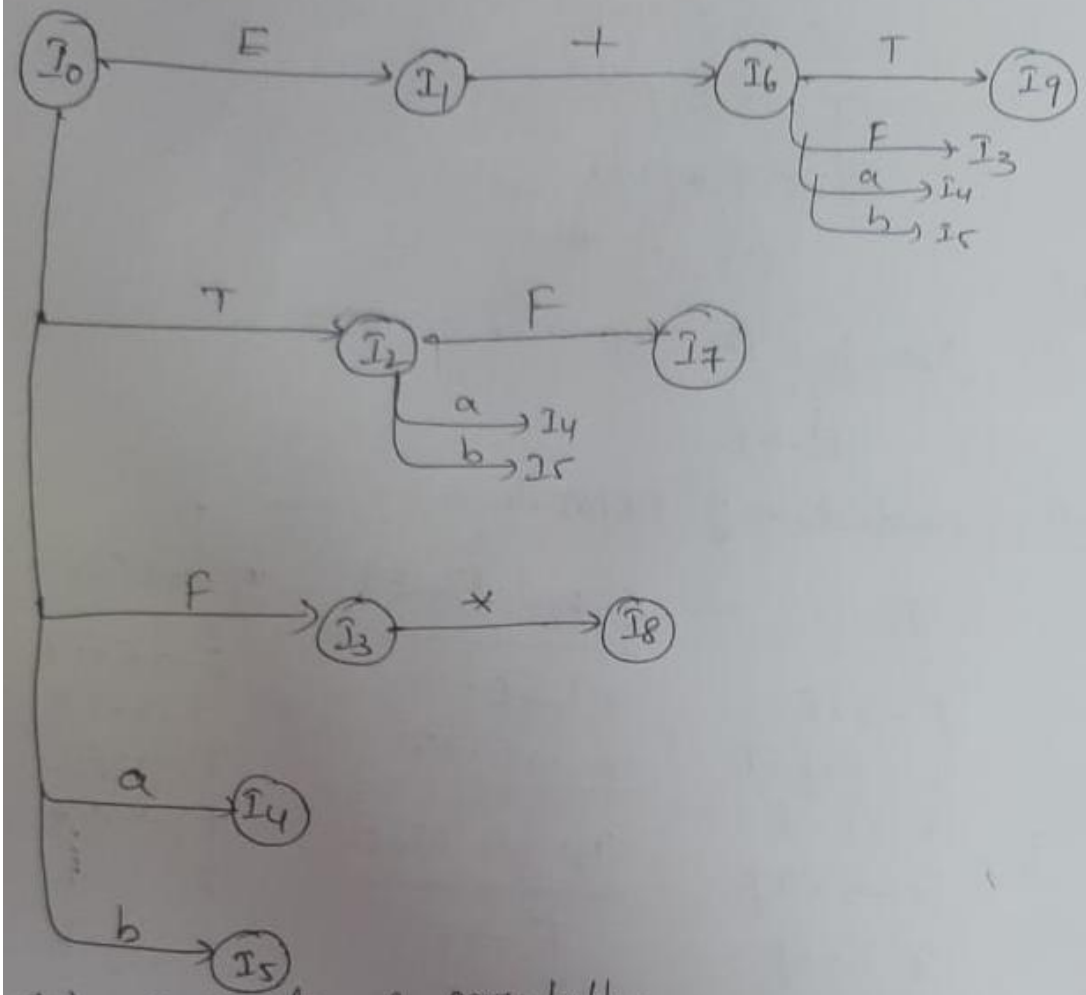
F → F*.

I_9 : goto (I_6, T)

E → E+T.
T → T.F
F → .F*
F → .a
F → .b

goto (I_6, F) → I_3
goto (I_6, a) → I_4
goto (I_6, b) → I_5

(iii) Construct DFA for the given grammar.



(iv) Construction of parse table.

States	Action					GOTO		
	+	*	a	b	$	E	T	F
0			S4	S5		1	2	3
1					accept			
2			S4	S5				7
3		S8						
4	r6	r6			r6			
5	r4	r4			r4			
6			S4	S5			9	3
7		r3	r3	r3				
8	r5	r5			r5			
9	r1				r1			

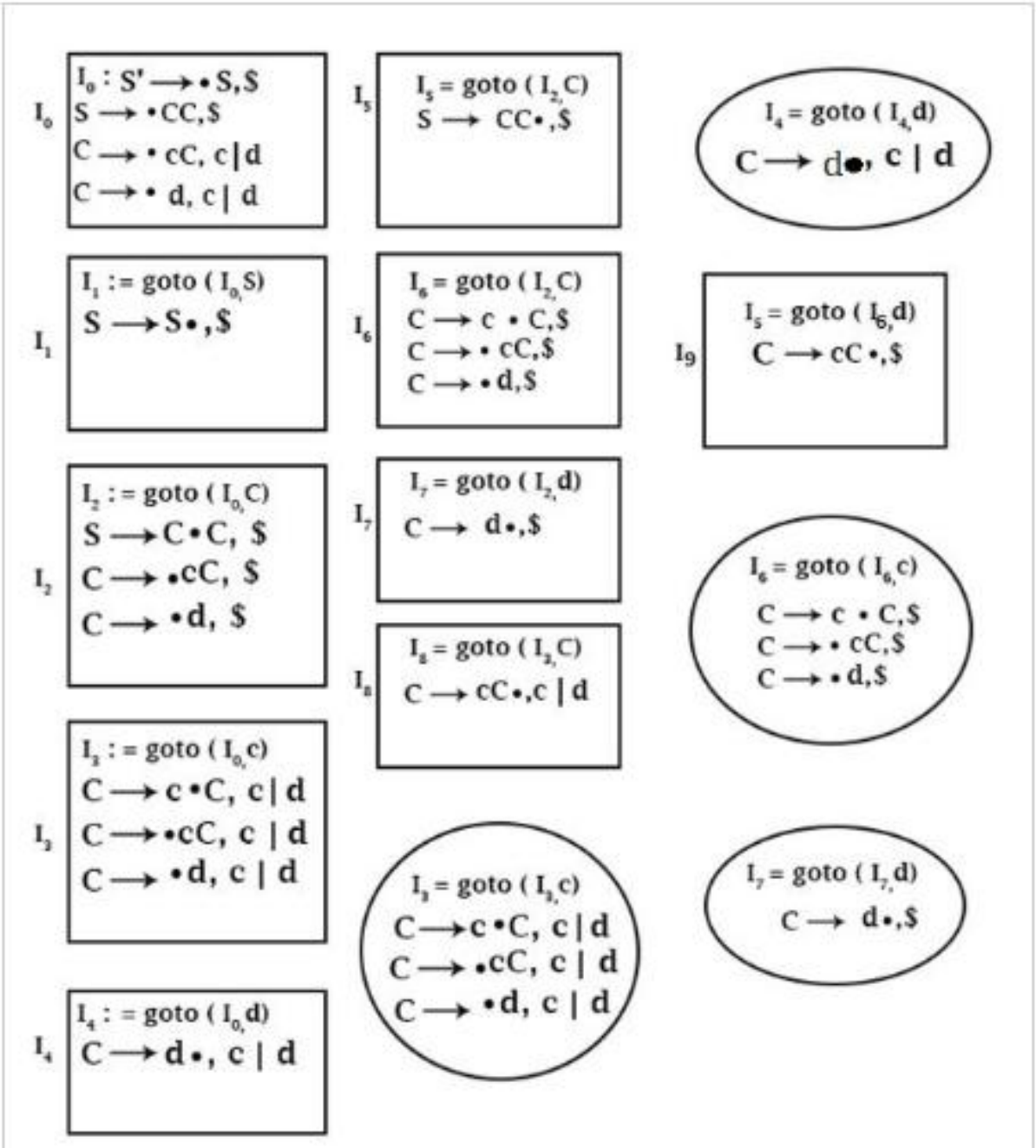
6) Construct CLR Parsing table for the given grammar. **[L6][CO3][12M]**

S→CC

C→aC/d

Solution

Step1 − Construct LR (1) Set of items. First of all, all the LR (1) set of items should be generated.



7) Design the LALR parser for the following Grammar. [L6][CO3][12M]

S → AA
A → aA
A → b

LALR (1) Grammar

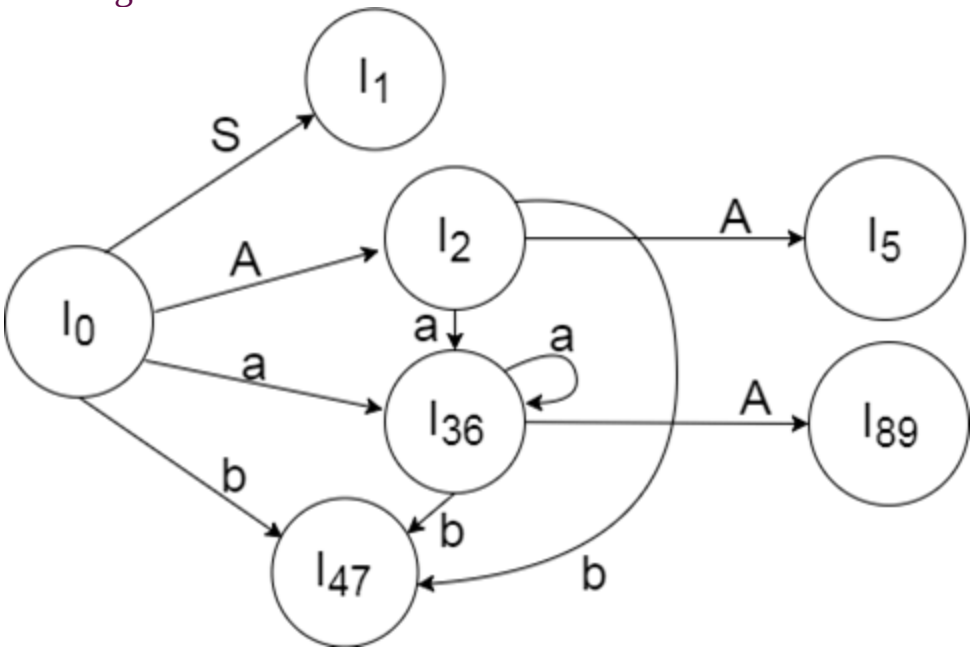
S → AA
A → aA
A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.

S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b
I0 State:
Add Augment production to the I0 State and Compute the ClosureL
I0 = Closure (S` → •S)
Add all productions starting with S in to I0 State because "•" is followed by the non-terminal. So, the I0 State becomes
I0 = S` → •S, $
S → •AA, $
Add all productions starting with A in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.
I0= S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b
I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $
I2= Go to (I0, A) = closure (S → A•A, $)
Add all productions starting with A in I2 State because "•" is followed by the non-terminal. So, the I2 State becomes
I2= S → A•A, $
A → •aA, $
A → •b, $
I3= Go to (I0, a) = Closure (A → a•A, a/b)
Add all productions starting with A in I3 State because "•" is followed by the non-terminal. So, the I3 State becomes
I3= A → a•A, a/b
A → •aA, a/b
A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)
Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)
I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b
I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $
I6= Go to (I2, a) = Closure (A → a•A, $)
Add all productions starting with A in I6 State because "•" is followed by the non-terminal. So, the I6 State becomes
I6 = A → a•A, $
A → •aA, $
A → •b, $
Go to (I6, a) = Closure (A → a•A, $) = (same as I6)
Go to (I6, b) = Closure (A → b•, $) = (same as I7)
I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $
I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
I9= Go to (I6, A) = Closure (A → aA•, $) A → aA•, $
If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.
I3 = { A → a•A, a/b
A → •aA, a/b
A → •b, a/b
}
I6= { A → a•A, $
A → •aA, $
A → •b, $
}
Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.
I36 = { A → a•A, a/b/$
A → •aA, a/b/$
A → •b, a/b/$
}
The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.
I47 = {A → b•, a/b/$}
The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.
I89 = {A → aA•, a/b/$}

Drawing DFA:



LALR (1) Parsing table:

States	a	b	S	S	A
I_0	S_{36}	S_{47}		12	
I_1		accept			
I_2	S_{36}	S_{47}			5
I_{36}	$S_{36}S_{47}$				89
I_{47}	R_3R_3	R_3			
I_5			R_1		
I_{89}	R_2	R_2	R_2		

8a) Define YACC parser? [L1][CO3][2M]

- YACC stands for Yet Another Compiler Compiler.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

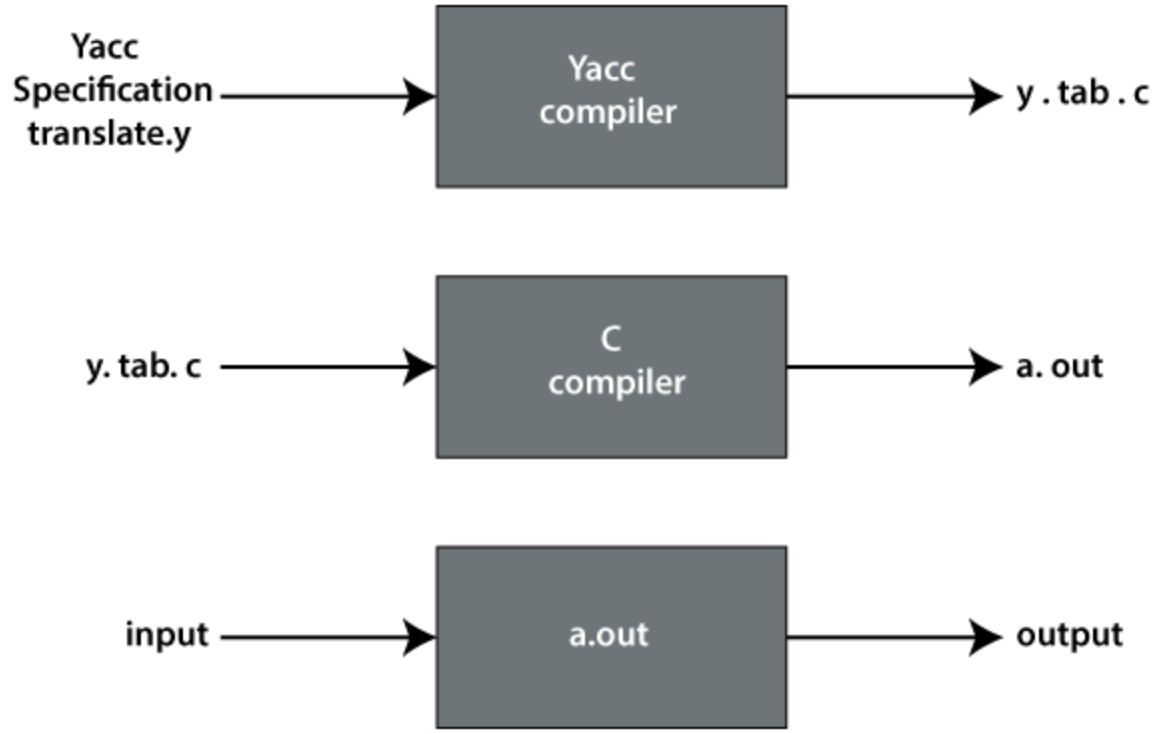
These are some points about YACC:

Input: A CFG- file.y

Output: A parser y.tab.c (yacc)

8b)Explain in detail the processing procedure of YACC Parser generator tool. [L2][CO3][6M]

YACC Tool



A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

YACC (yet another compiler-compiler) is an LALR (1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File:
YACC input file is divided into three parts.

/* definitions */

....

%%

/* rules */

....

%%

/* auxiliary routines */

....

Input File: Definition Part:

The definition part includes information about the tokens used in the syntax definition:

%token NUMBER

%token ID

Yacc automatically assigns numbers for tokens, but it can be overridden by %token NUMBER 621

Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should not overlap ASCII codes.

The definition part can include C code external to the definition of the parser and variable declarations, within %{ and %} in the first column.

It can also include the specification of the starting symbol in the grammar:

%start nonterminal

Input File: Rule Part:

The rules part contains grammar definition in a modified BNF form.

Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

The auxiliary routines part is only C code.

It includes function definitions for every function needed in rules part.

It can also contain the main() function definition if the parser is going to be run as a program.

The main() function must call the function yyparse().

Input File:

If yylex() is not defined in the auxiliary routines sections, then it should be included:

#include "lex.yy.c"

YACC input file generally finishes with:

.y

Output Files:

The output of YACC is a file named y.tab.c

If it contains the main() definition, it must be compiled to be executable.

Otherwise, the code can be an external function definition for the function int yyparse()

If called with the –d option in the command line, Yacc produces as output a header file y.tab.h with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).

If called with the –v option, Yacc produces as output a file y.output containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

Example:
Yacc File (.y)

C

```
%{
  #include <ctype.h>
  #include <stdio.h>
```

```
  #define YYSTYPE double /* double type for yacc stack */
%}
%%
 Lines :  Lines S '\n' { printf("OK \n"); }
       | S '\n'
       | error '\n' {yyerror("Error: reenter last line:");
                     yyerrok; };
 S     :  '(' S ')'
       | '[' S ']'
       |  /* empty */    ;
%%
#include "lex.yy.c"
  void yyerror(char * s)
/* yacc error handler */
{
 fprintf (stderr, "%s\n", s);
}
int main(void)
 {
 return yyparse();
 }
```

Lex File (.l)

```
C


%{

%}

%%

[ \t]     { /* skip blanks and tabs */ }

\n|.      { return yytext[0]; }

%%
```

For Compiling YACC Program:

1. Write lex program in a file file.l and yacc in a file file.y
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. type lex file.l
4. type yacc file.y
5. type cc lex.yy.c y.tab.h -ll
6. type ./a.out

NOTE: write any example program of yacc

8c)How YACC will resolve the parsing action conflicts and the error recovery. [L2][CO3][4M]

Resolve the Parsing Action Conflicts:

yacc invokes two default disambiguating rules:

In a shift-reduce conflict, the default is to shift.

In a reduce-reduce conflict, the default is to reduce by the earlier grammar rule (in the yacc specification). Rule 1 implies that reductions are deferred in favor of shifts when there is a choice.

Error Recovery:

YACC (Yet Another Compiler Compiler) is a tool for generating a parser for a specified grammar. It was developed by Stephen C. Johnson at AT&T Bell Laboratories in the 1970s. A parser is a program that takes input in the form of a sequence of tokens and checks if it conforms to a specified set of rules, called grammar. If the input is valid, the parser generates a parse tree, which represents the structure of the input according to the grammar.

YACC works by taking a file containing grammar in a specified format and generating C or C++ code for a parser that implements the grammar. The user also provides code for actions to be taken when certain grammar rules are recognized, such as creating parse tree nodes or generating code.

The file is typically named with the ".y" file extension. YACC uses LALR (Look-Ahead Left-to-Right) parsing, which is a type of bottom-up parsing that uses a stack to keep track of the input and a set of rules to determine what to do next.

YACC grammars consist of a set of rules, each of which has a left-hand side (LHS) and a right-hand side (RHS). The LHS is a nonterminal symbol, which represents a category of input, and the RHS is a sequence of the terminal and nonterminal symbols, which represents a possible sequence of input.

For example, a grammar for simple arithmetic expressions might have a rule for an expression, with the LHS being "expression" and the RHS being "term + expression" or "term – the expression" or "term", where "term" is another nonterminal symbol.

YACC also provides a number of built-in features, such as error recovery and conflict resolution, that allow for more robust and flexible parsing.

The YACC input file, also known as the YACC specification file, contains the following main components:

1. **Declarations**: This section includes any global variables or constants that are used in the program.
2. **Terminal symbols**: This section defines the terminal symbols, or the tokens, that the parser will recognize. These symbols are typically defined using the %token directive.

3. **Non-terminal symbols**: This section defines the non-terminal symbols, or the grammar rules, that the parser will use to parse the input. These symbols are typically defined using the %nonterm directive.
4. **Grammar rules:** This section defines the grammar rules for the parser. Each rule starts with a non-terminal symbol, followed by a colon and a list of terminal and non-terminal symbols that make up the rule. The grammar rules are typically separated by a vertical bar (|).
5. **Start symbol:** This section defines the start symbol for the parser. The start symbol is the non-terminal symbol that the parser will begin parsing with.
6. **Code section:** This section includes any C code that is used in the parser. This code is typically used to implement actions that are taken when a specific grammar rule is matched.
7. The YACC input file also includes additional directives and options that can be used to customize the behavior of the parser. These include %start, %left, %right, %prec, and %type, among others.

Here is an example of a simple calculator program written in C using the YACC (Yet Another Compiler Compiler) tool:

This program defines a simple calculator that can evaluate expressions containing numbers and the operators +, -, *, and /. The YACC code defines the grammar for the calculator, and the code in the curly braces specifies the actions to be taken when a particular grammar rule is matched. The yylex() function is used to read in and return the next token, and the yyerror() function is called when an error is encountered in the input.

YACC provides two types of error recovery: Panic-mode error recovery and Phrase-level error recovery. Panic-mode error recovery is used when the parser encounters a token that is not part of the grammar, and it skips input until it finds a token that can be used to resume parsing. Phrase-level error recovery is used when the parser encounters a token that is not part of the current rule, and it tries to find a way to continue parsing by matching the input to a different rule.

Additionally, YACC also provides a mechanism for resolving conflicts, such as shift-reduce and reduce-reduce conflicts, that can occur during parsing. This allows the parser to choose the correct action to take when multiple rules are applicable to the input.

Error handling in a YACC program can be done using the following techniques:

1. Error tokens: YACC allows you to define error tokens that can be used to handle errors in grammar. When an error is encountered, the error token is used to skip over the input and continue parsing.

2. Error rules: You can also define error rules in your YACC program. These rules are used to handle specific errors in grammar. For example, if a specific input is expected but not found, the error rule can be used to handle that situation.

3. Error recovery: YACC also provides an error recovery mechanism. This mechanism allows the parser to recover from errors by skipping over input and continuing parsing. This can be useful for handling unexpected input.

4. Error messages: YACC allows you to specify error messages that will be displayed when an error is encountered. This can help the user understand what went wrong and how to fix the problem.

5. yyerror() function: You can also use the yyerror() function to handle errors in your YACC program. This function is called when an error is encountered, and it can be used to display an error message or perform other error-handling tasks.

Overall, error handling in YACC requires a combination of all these methods to be successful.

Terminologies

1. **yylex():** yylex is a lexical analyzer generator, used to generate lexical analyzers (or scanners) for programming languages and other formal languages. It reads a specification file, which defines the patterns and rules for recognizing the tokens of a language, and generates a C or C++ program that can be used to scan and tokenize input text according to those rules. This generated program is typically used as a component of a larger compiler or interpreter for the language.

2. **yyerorr():** yyerror is a function in the YACC (Yet Another Compiler Compiler) library that is called when a parsing error is encountered. It is typically used to print an error message and/or take other appropriate action. The function is defined by the user and can be customized to suit their needs.

3. **Parser:** A parser is a software program or algorithm that processes and analyzes text or data, breaking it down into smaller components and identifying patterns and structures. It is often used in natural language

processing, computer programming, and data analysis to extract meaning and information from large sets of text or data. Parsers can be used to identify syntax, grammar, and semantic elements in a text or data, and can also be used to generate code or other output based on the information they extract.

4. **yyparse():** yyparse is a function used in the YACC (Yet Another Compiler-Compiler) tool to parse a given input according to the grammar defined in the YACC file. It reads the input and generates a parse tree, which is used to generate the target code or perform other actions as defined in the YACC file.

9a) Explain Syntax Directed Definition with simple examples. [L2][CO2][6M]

Syntax Directed Definition

Syntax Directed Definition (SDD) is a kind of abstract specification. It is generalization of context free grammar in which each grammar production X –> a is associated with it a set of production rules of the form $s = f(b_1, b_2, \ldots\ldots b_k)$ where s is the attribute obtained from function f.

The attribute can be a string, number, type or a memory location. Semantic rules are fragments of code which are embedded usually at the end of production and enclosed in curly braces ({ }).

Example 1:

E --> E_1 + T { E.val = E_1.val + T.val}
Example 2: Consider the following grammar
S --> E
E --> E_1 + T
E --> T
T --> T_1 * F
T --> F
F --> digit
The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> E	Print(E.val)
E --> E_1 + T	E.val = E_1.val + T.val
E --> T	E.val = T.val
T --> T_1 * F	T.val = T_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval

9b) Define a syntax-directed translation and explain with example. [L2][CO2][6M]

SYNTAX DIRECTED TRANSLATION

In syntax directed translation, along with the grammar we associate some informal notations and these notations are called as semantic rules.

So we can say that

Grammar + semantic rule = SDT (syntax directed translation)

- ✓ In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.
- ✓ In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record
- ✓ In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

Example

Production	Semantic Rules

E → E + T	E.val := E.val + T.val
E → T	E.val := T.val
T → T * F	T.val := T.val + F.val
T → F	T.val := F.val
F → (F)	F.val := F.val
F → num	F.val := num.lexval

E.val is one of the attributes of E.

num.lexval is the attribute returned by the lexical analyzer.

10a) Give the evaluation order of SDD with an example. [L5][CO2][6M]

Evaluation order of SDD

Evaluation order for SDD includes how the SDD(Syntax Directed Definition) is evaluated with the help of attributes, dependency graphs, semantic rules, and S and L attributed definitions. SDD helps in the semantic analysis in the compiler so it's important to know about how SDDs are evaluated and their evaluation order. This article provides detailed information about the SDD evaluation. It requires some basic knowledge of grammar, production, parses tree, annotated parse tree, synthesized and inherited attributes.

Terminologies:

- ✓ Parse Tree: A parse tree is a tree that represents the syntax of the production hierarchically.
- ✓ Annotated Parse Tree: Annotated Parse tree contains the values and attributes at each node.
- ✓ Synthesized Attributes: When the evaluation of any node's attribute is based on children.
- ✓ Inherited Attributes: When the evaluation of any node's attribute is based on children or parents.

Ordering the Evaluation of Attributes:

The dependency graph provides the evaluation order of attributes of the nodes of the parse tree. An edge(i.e. first node to the second node) in the dependency graph represents that the attribute of the second node is dependent on the attribute of the first node for further evaluation. This order of evaluation gives a linear order called topological order.

There is no way to evaluate SDD on a parse tree when there is a cycle present in the graph and due to the cycle, no topological order exists.

Production Table		
S.No.	Productions	Semantic Rules
1.	S ⇢ A & B	S.val = A.syn + B.syn
2.	A ⇢ A_1 # B	A.syn = A1.syn * B.syn A1.inh = A.syn
3.	A1 ⇢ B	A1.syn = B.syn
4.	B ⇢ digit	B.syn = digit.lexval

10b) Discuss Type Checking with suitable examples. [L2][CO4][12M]

TYPE CHECKING

A compiler must check that the source program follows both syntactic and semantic conventions of the source language. This checking, called static checking, detects and reports programming errors.

Some examples of static checks:

1. Type checks- A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.

2. Flow-of-control checks- Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An enclosing statement, such as break, does not exist in switch statement.

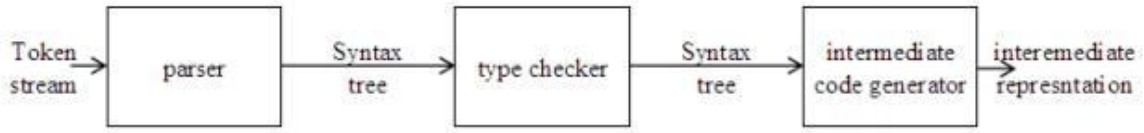


Fig. 2.6 Position of type checker

Fig. 2.6 Position of type checker

A typechecker verifies that the type of a construct matches that expected by its context. For example : arithmetic operator mod in Pascal requires integer operands, so a type checker verifies that the operands of mod have type integer. Type information gathered by a type checker may be needed when code is generated.

Type Systems

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language
constructs.
For example : " if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer "

Type Expressions

The type of a language construct will be denoted by a "type expression." A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions. The sets of basic types and constructors depend on the language to be checked. The following are the definitions of type expressions:

1. Basic types such as boolean, char, integer, real are type expressions.
   A special basic type, type_error , will signal an error during type checking; void denoting "the absence of a value" allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.
3. A type constructor applied to type expressions is a type expression.

Constructors include:

Arrays : If T is a type expression then array (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

Products : If T1 and T2 are type expressions, then their Cartesian product T1 X T2 is a type expression.

Records : The difference between a record and a product is that the names. The record type constructor will be applied to a tuple formed from field names and field types.

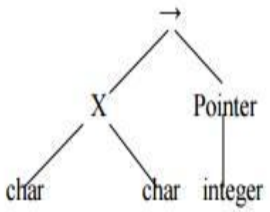
For example:

```
type row = record
address: integer;
        lexeme: array[1..15] of char
        end;
var table: array[1...101] of row;
```

declares the type name row representing the type expression record((address X integer) X (lexeme X array(1..15,char))) and the variable table to be an array of records of this type.

Pointers : If T is a type expression, then pointer(T) is a type expression denoting the type "pointer to an object of type T".
For example, var p: ↑ row declares variable p to have type pointer(row).
Functions : A function in programming languages maps a domain type D to a range type R. The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.



Fg. 5.7 Tree representation for char x char → pointer (integer)

Fg. 5.7 Tree representation for char x char → pointer (integer)

Type systems

A type system is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type system. It is specified in a syntax-directed manner. Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic. Any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

Sound type system

A sound type system eliminates the need for dynamic checking fo allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than type_error to a program part, then type errors cannot occur when the target code for the program part is run.

Strongly typed language

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

SIDDHARTH INSTITUTE OF ENGINEERING&TECHNOLOGY:: PUTTUR
(AUTONOMOUS)
Siddharth Nagar, Narayanavanam Road – 517583

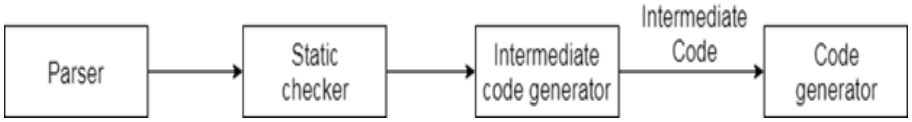
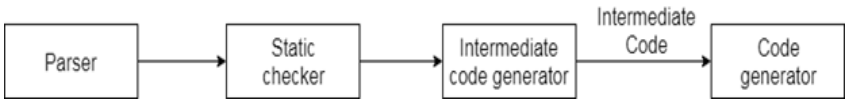
QUESTION BANK ANSWER (DESCRIPTIVE)

Subject with Code: **Compiler Design(20CS0516)** Course & Branch: **B.Tech - CSE**

Regulation: **R20** Year &Sem: **III-B.Tech & I – Sem**

UNIT –IV

Intermediate Code Generation and Run Time Environment

1.	**a**	**What do you understand by Intermediate Code**	**[L2][CO5]**	**[2M]**
		Intermediate code Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language.  Fig: Position of intermediate code generator		
	b	**Analyse different types of Intermediate Code with an example**	**[L4][CO5]**	**[10M]**
		Intermediate code Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language.  Fig: Position of intermediate code generator ✓ If the compiler directly translates source code into the machine code without generating intermediate code then a full native compiler is required for each new machine. ✓ The intermediate code keeps the analysis portion same for all the compilers that's why it doesn't need a full compiler for every unique machine. ✓ Intermediate code generator receives input from its predecessor phase and semantic analyzer phase. It takes input in the form of an annotated syntax tree. ✓ Using the intermediate code, the second phase of the compiler synthesis phase is changed according to the target machine.		

The following are commonly used intermediate code representations:

Postfix Notation: Also known as reverse Polish notation or suffix notation. The ordinary (infix) way of writing the sum of a and b is with an operator in the middle: a + b The postfix notation for the same expression places the operator at the right end as ab +.

In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +.

No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression.

In postfix notation, the operator follows the operand.

Example 1:
The postfix representation of the expression
(a + b) * c is :
ab + c *

Example 2:
The postfix representation of the expression
(a – b) * (c + d) + (a – b) is :
ab – cd + *ab -+

Three-Address Code: A statement involving no more than three references(two for operands and one for result) is known as a three address statement.

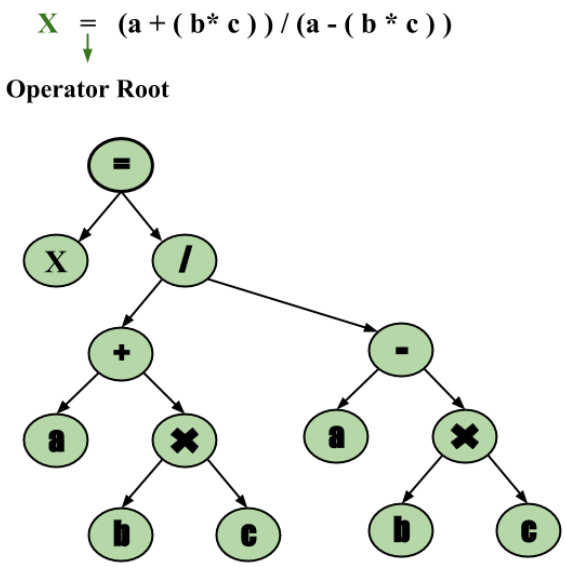
A sequence of three address statements is known as a three address code. Three address statement is of form x = y op z, where x, y, and z will have address (memory location).

Sometimes a statement might contain less than three references but it is still called a three address statement. Example:

Syntax Tree: A syntax tree is nothing more than a condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by the single link in the syntax tree the internal nodes are operators and child nodes are operands.

To form a syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Example: x = (a + b * c) / (a – b * c)



Three Address Code:

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code.

It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler.

The compiler decides the order of operation given by three address code.

General representation –

a = b op c

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

The three address code for the expression a + b * c + d :

T 1 = b * c

T 2 = a + T 1

T 3 = T 2 + d where T 1 , T 2 , T 3 are temporary variables.

Example-1: Convert the expression a * – (b + c) into three address code.

		$t_1 = b + c$
		$t_2 = uminus\ t_1$
		$t_3 = a * t_2$
		
		Example-2: Write three address code for following code
		for(i = 1; i<=10; i++)
		{
		a[i] = x * 5;
		}
		$i = 1$
		$L : t_1 = x * 5$
		$t_2 = \&a$
		$t_3 = sizeof(int)$
		$t_4 = t_3 * i$
		$t_5 = t_2 + t_4$
		$*t_5 = t_1$
		$i = i + 1$
		$if\ i<=10\ goto\ L$
		There are 3 ways to implement a Three-Address Code in compiler design are:
		i) Quadruples
		ii) Triples
		iii) Indirect Triple		

2.	**a**	**List and define various representations of Three Address Codes.**	**[L1][CO5]**	**[4M]**
		There are 3 representations of three address code namely 1. Quadruple 2. Triples 3. Indirect Triples **1.Quadruple** – It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression. **2.Triples** – This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.		

		3.Indirect Triples – This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.		
	b	**Explain representation of Three Address Codes with suitable Examples**	**[L2][CO 5]**	**[8M]**
		Implementation of Three Address Code – There are 3 representations of three address code namely 1. Quadruple 2. Triples 3. Indirect Triples **1. Quadruple –** It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression. **Advantage –** ✓ Easy to rearrange code for global optimization. ✓ One can quickly access value of temporary variables using symbol table. **Disadvantage –** ✓ Contain lot of temporaries. ✓ Temporary variable creation increases time and space complexity. **Example** – Consider expression **a = b * – c + b * – c.** The three address code is: t1 = uminus c t2 = b * t1 t3 = uminus c t4 = b * t3 t5 = t2 + t4 a = t5		

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

2. Triples –

This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage –

- ✓ Temporaries are implicit and difficult to rearrange code.
- ✓ It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example – Consider expression **a = b * – c + b * – c**

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. Indirect Triples –

This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example – Consider expression **a = b * – c + b * – c**

List of pointers to table

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

Question – Write quadruple, triples and indirect triples for following expression : **(x + y) * (y + z) + (x + y + z)**

Explanation – The three address code is:

t1 = x + y

t2 = y + z

t3 = t1 * t2

t4 = t1 + z

t5 = t3 + t4

#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation

List of pointers to table

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

3.		**Produce quadruple, triples and indirect triples for following expression: (x + y) * (y + z) + (x + y + z)**		**[L6][CO5]**	**[12M]**

The three address code is:

t1 = x + y

t2 = y + z

t3 = t1 * t2

t4 = t1 + z

t5 = t3 + t4

#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation

List of pointers to table

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

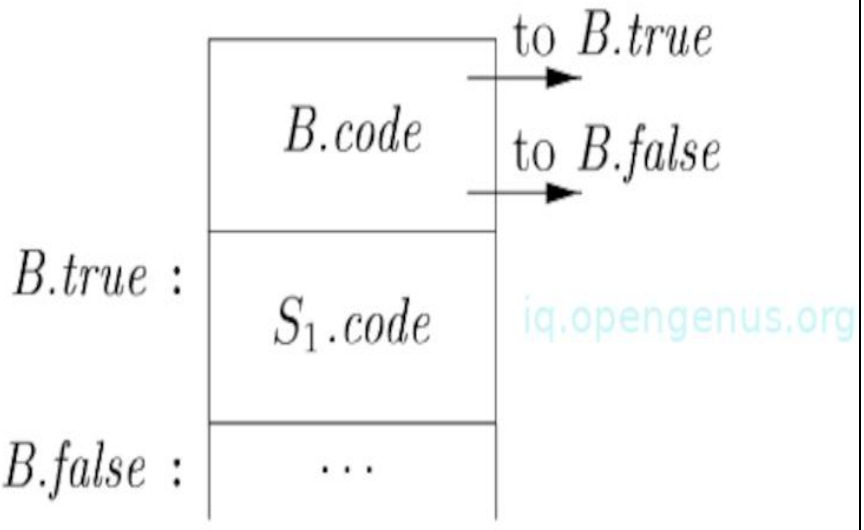
4.	**a**	**Describe scope and life time of variable.**	**[L2][CO4]**	**[2M]**
		• Scope of a variable determines the area or a region of code where a variable is available to use. • Lifetime of a variable is defined by the time for which a variable occupies some valid space in the system's memory. • Scope determines the life of a variable.		
	b	**Illustrate Control Flow Statements.**	**[L3][CO4]**	**[10M]**
		Control Flow Statements In this section, we consider the translation of control flow statements into three-address code in the context of statements generated by the following grammar;		

S → if(B) S1
S → if(B) S1 else S2
S → while(B) S1

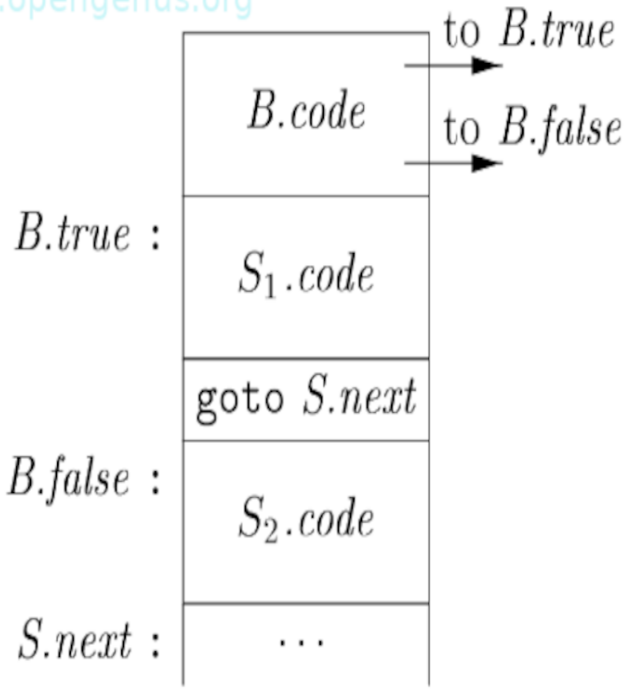
In the productions, non-terminal *B* represents a boolean expression while non-terminal *S* represents a statement.

The translation of if (B) S1 consists of *B.code* followed by S1.code as can be seen in the image below;

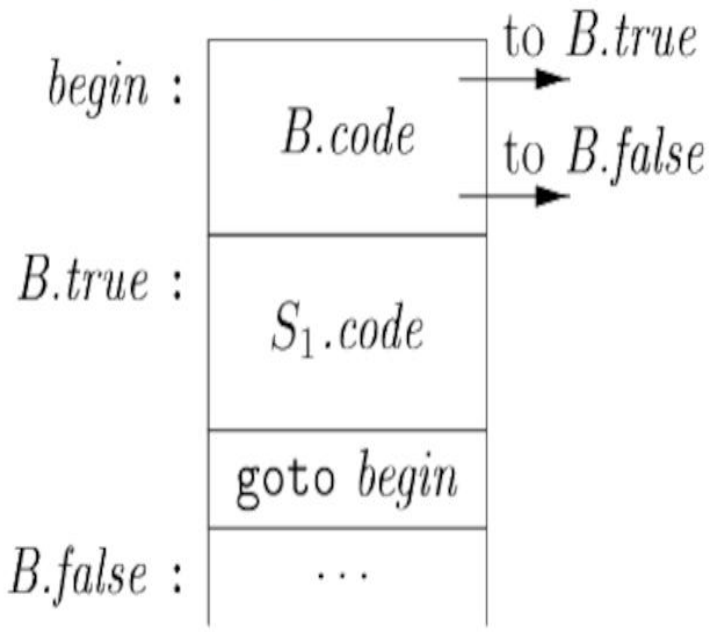
(1). **The if translation**



(2). The if-else translation



***(3).* The while translation**



Inside B.code we have jumps based on the value of *B*. If the value is true, control flows to the first instruction of S1.code otherwise *B* is false, therefore control flows to the instruction that follows *S1.code*.

Inherited attributes are used to manage labels for *B.code* and *S.code*. With a boolean expression *B*, two labels *B.true* and *B.false* are associated with each other if *B* is false.

We use statement *S* to associate the inherited attribute *S.next* that denotes a label for the instruction immediately following the code for *S*.
In other cases, the instruction that follows *S.code* is a jump to another label *L*.

A jump to *L* from *S.code* is avoided using *S.next*.

We have the Syntax-directed definition that produces a three-address code for boolean expressions in the context of *if*, *if-else* and *while* statements.

Generating three-address code for booleans;

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \textbf{ rel } E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\texttt{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \texttt{'goto'}\ B.true)$ $\mid\mid gen(\texttt{'goto'}\ B.false)$
$B \rightarrow \textbf{true}$	$B.code = gen(\texttt{'goto'}\ B.true)$
$B \rightarrow \textbf{false}$	$B.code = gen(\texttt{'goto'}\ B.false)$

We have a program that consists of a statement that is generated by P → S. Semantic rules associated with this production initialize *S.next* to a new label. P.code will consist of *S.code* which is followed by a new label *S.next*. *assign* in the production *S → assign*. This acts as a placeholder for the assignment statement.

In translating *S → if(B) S1*, semantic rules as shown in image (4) create a new label *B.true* then attach it to the first three-address instruction that is generated for *S1*.

By setting *B.false* to *S.next*, we make sure that control skips code for S1 if B results in a false.

Translating *if-else* statement that is *S → if(B) S1 else S2*, the code for the boolean expression *B* has jumps out of it to the first instruction of the code for S1 if *B* is true and to the first instruction of the code for S2, if *B* is false as can be seen from image (2).

Furthermore, control flows from S1 and S2 to the three-address instruction that immediately follows code for *S*. *S.next* is its label which is also an inherited attribute.

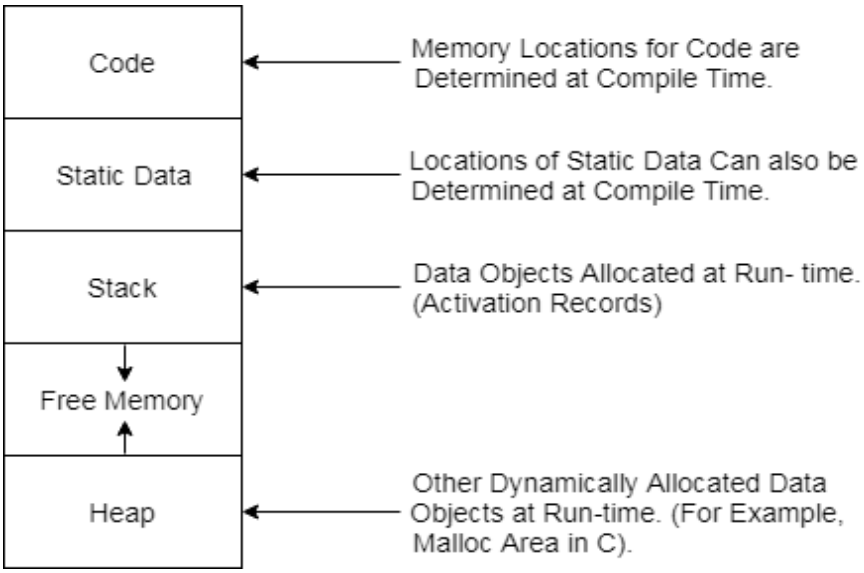
S.next is an explicit goto that appears after code for S1 to skip codefor S2. After S2, we don't need any goto since S2.next will be similar to *S.next*.

For translating *S → while(B) S1*, we form its code from *B.code* and *S1.code*. This is seen from image (3). The local variable *begin* holds a new label that is attached to

		the first instruction for this while statement, which is also the first instruction for *B*. A *variable* is preferred to an *attribute* since *begin* is local to the semantic rules for this production. Inherited label *S.next* marks the instruction where control must flow to, that is, when *B* is false, therefore, *B.false* is set to *S.next*. *B.true* which is a new label is attached to the first instruction for S1. Code for *B* generates a jump to this label, that is, if *B* is true. Following the code for S1, we place the instruction *goto begin*. This causes a jump back to the start of the code for boolean expressions. Keep in mind that S1.next is currently set to the label *begin*, therefore, jumps within S1.code can go directly to *begin*. Code for $S \rightarrow S1\ S2$ consists of code for *S1* that is followed by code for *S2*. Semantic rules manage labels, the first instruction after S1 is the beginning of the code for S2, and the instruction after code S2 code, is the instruction after code for *S*.		
5.	**a**	**Justify the need for Storage Organization.**	**[L6][CO4]**	**[4M]**
		The compiler utilizes this block of memory executing the compiled program. This block of memory is called storage management. One of the important tasks that a compiler must perform is to allocate the resources of the target machine to represent the data objects that are being manipulated by the source program**.** • The compiler utilizes this block of memory executing the compiled program. • This block of memory is called storage management. • One of the important tasks that a compiler must perform is to allocate the resources of the target machine to represent the data objects that are being manipulated by the source program		
	b	**Describe the Storage Organization with simple examples.**	**[L2][CO4]**	**[8M]**
		STORAGE ORGANIZATION ✓ When the target program executes then it runs in its own logical address space in which the value of each program has a location. ✓ The logical address space is shared among the compiler,		

operating system and target machine for management and organization. The operating system is used to map the logical address into physical address which is usually spread throughout the memory.

Subdivision of Run-time Memory:



- ✓ Runtime storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. Using the four bytes a machine word can form. Object of multibyte is stored in consecutive bytes and gives the first byte address.
- ✓ Run-time storage can be subdivide to hold the different components of an executing program:

1. Generated executable code
2. Static data objects
3. Dynamic data-object- heap
4. Automatic data objects- stack

STORAGE ALLOCATION TECHNIQUES

I. Static Storage Allocation

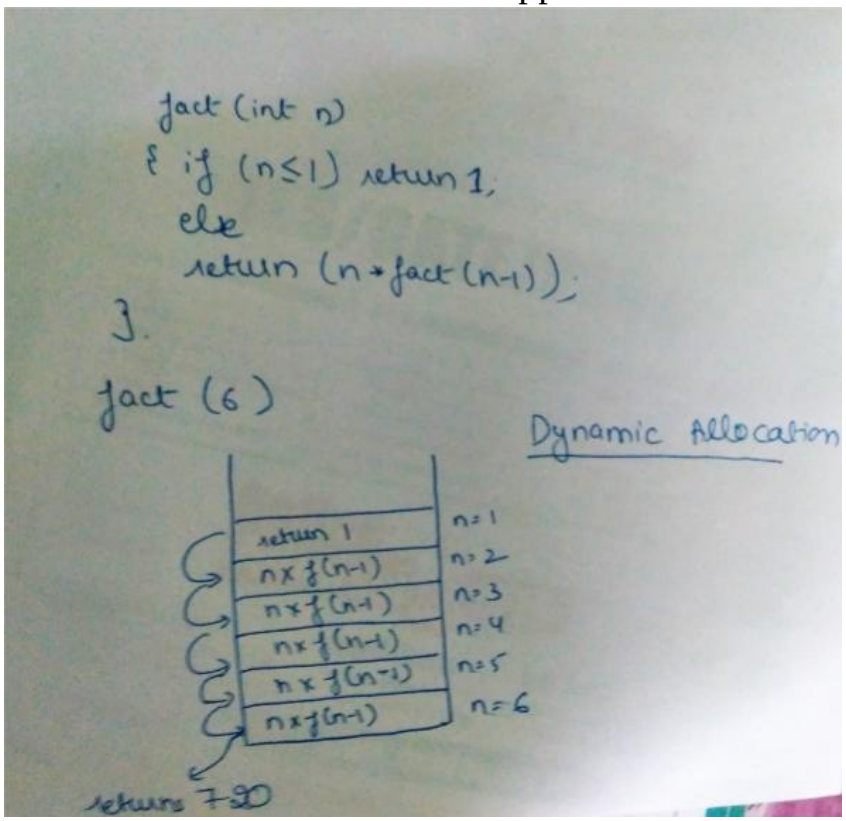
- For any program, if we create a memory at compile time, memory will be created in the static area.
- For any program, if we create a memory at compile-time only, memory is created only once.
- It doesn't support dynamic data structure i.e memory is created at compile-time and deallocated after program completion.
- The drawback with static storage allocation is recursion is not supported.
- Another drawback is the size of data should be known at compile time
- Eg- FORTRAN was designed to permit static storage allocation.

II. **Stack Storage Allocation**

- Storage is organized as a stack and activation records are pushed and popped as activation begins and end respectively.
- Locals are contained in activation records so they are bound to fresh storage in each activation.
- Recursion is supported in stack allocation

III. **Heap Storage Allocation**

- Memory allocation and deallocation can be done at any time and at any place depending on the requirement of the user.
- Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
- Recursion is supported.

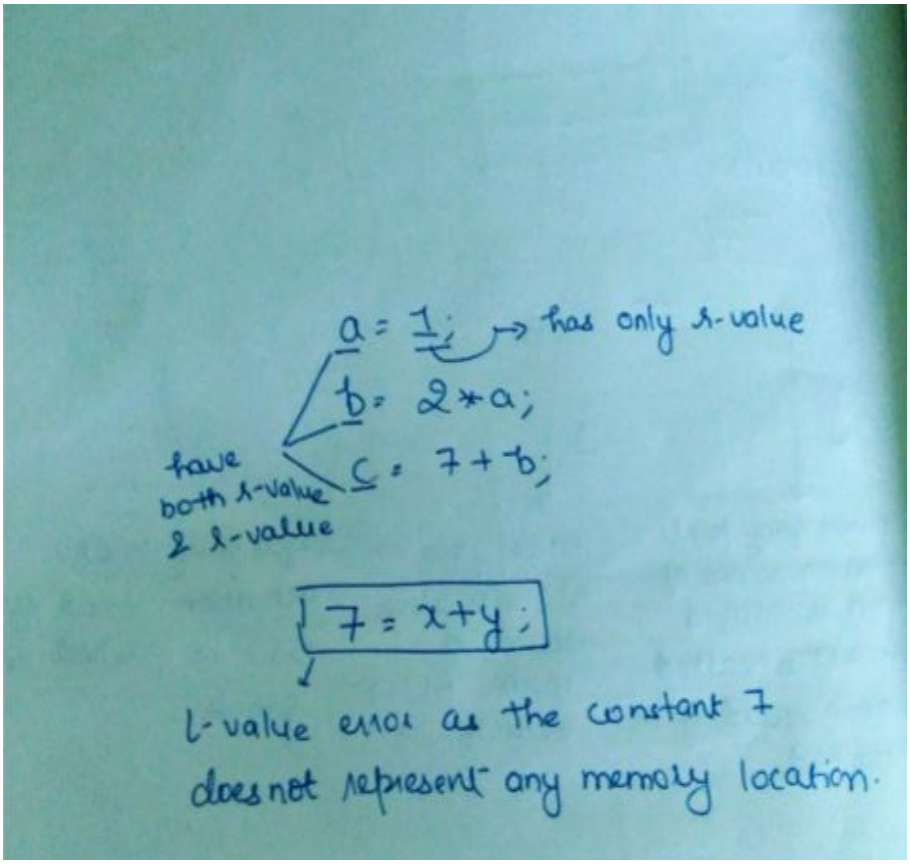


PARAMETER PASSING: The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism.

Basic terminology :

R- value: The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if its appear on the right side of the assignment operator. R-value can always be assigned to some other variable.

L-value: The location of the memory(address) where the expression is stored is known as the l-value of that expression. It always appears on the left side if the assignment operator.



i.Formal Parameter:

Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

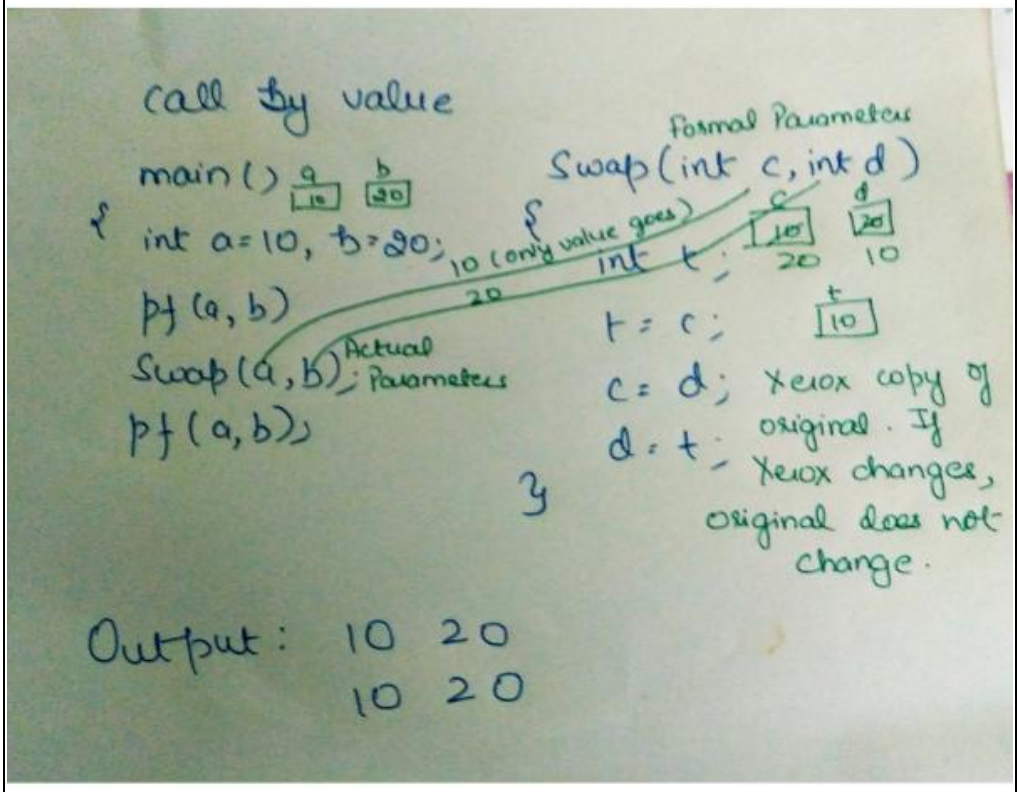
ii. Actual Parameter:

Variables whose values and functions are passed to the called function are called actual parameters. These variables are specified in the function call as arguments.

Different ways of passing the parameters to the procedure:

Call by Value: In call by value the calling procedure passes the r-value of the actual parameters and the compiler puts that into called procedure's activation record.

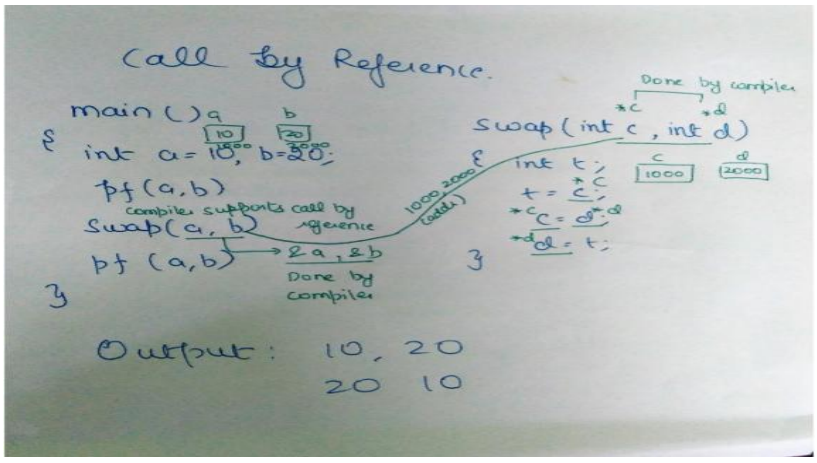
Formal parameters hold the values passed by the calling procedure, thus any changes made in the formal parameters do not affect the actual parameters.



Call by ReferenceIn call by reference the formal and actual parameters refers to same memory location. The l-value of actual parameters is copied to the activation record of the called function.

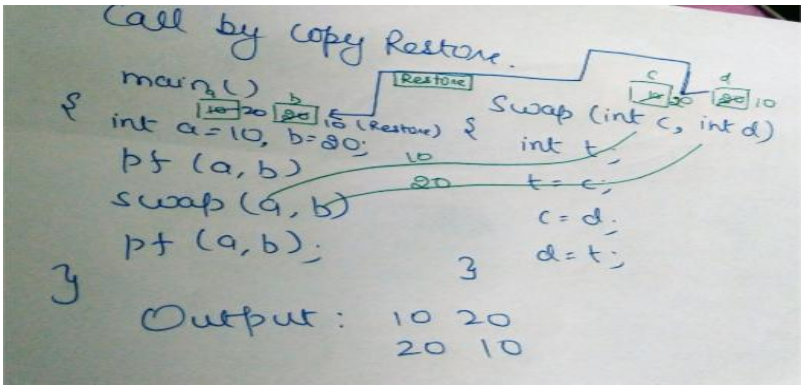
Thus the called function has the address of the actual parameters. If the actual parameters does not have a l-value (eg- i+3) then it is evaluated in a new temporary location and the address of the location is passed.

Any changes made in the formal parameter is reflected in the actual parameters (because changes are made at the address).

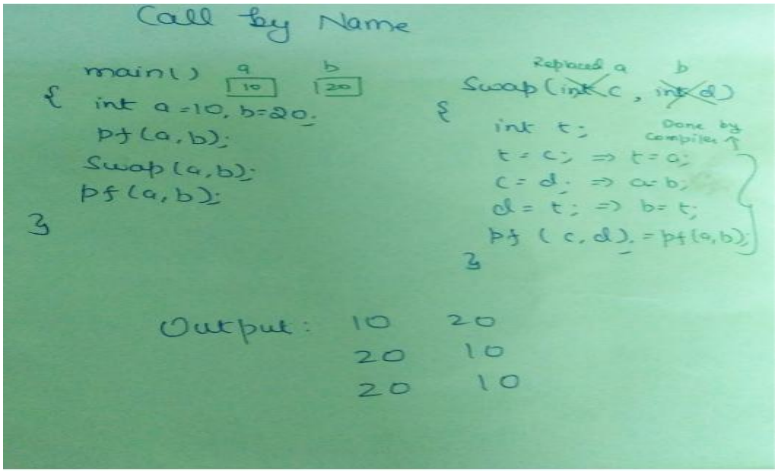


Call by Copy Restore In call by copy restore compiler copies the value in formal parameters when the procedure is called and copy them back in actual parameters when control returns to the called function.

The r-values are passed and on return r-value of formals are copied into l-value of actuals.



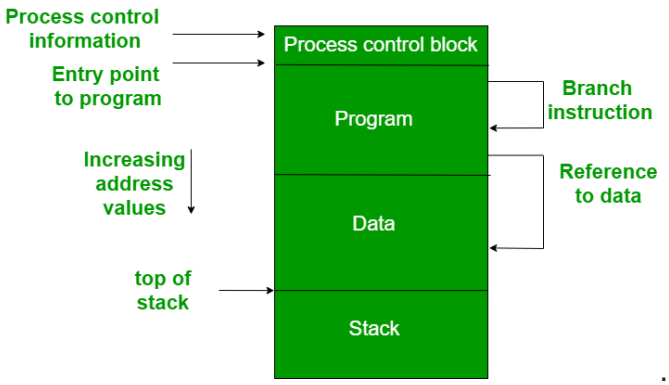
Call by Name In call by name the actual parameters are substituted for formals in all the places formals occur in the procedure. It is also referred as lazy evaluation because evaluation is done on parameters only when needed.



6.	**a**	**List out the properties of memory management**	**[L1][CO4]**	**[4M]**

Here are the requirements of memory management:

Relocation. The memory available to us gets shared among various processes present in a multiprogramming system



Protection.: There is always a danger when we have multiple programs at the same time as one program may write to the address space of another program. So every process must be protected against unwanted interference when other process tries to write in a process whether accidental or incidental.

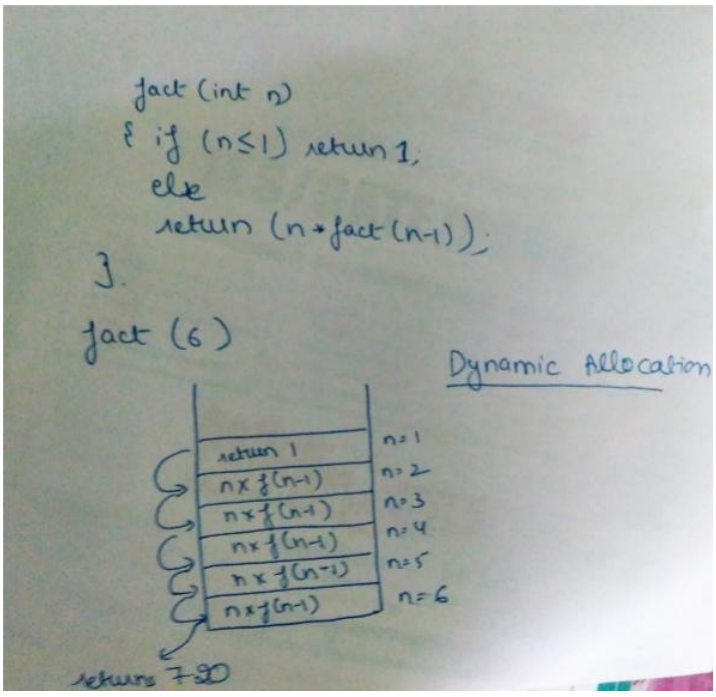
		Between relocation and protection requirement a trade-off occurs as the satisfaction of relocation requirement increases the difficulty of satisfying the protection requirement. Sharing: A protection mechanism must have to allow several processes to access the same portion of main memory. Allowing each processes access to the same copy of the program rather than have their own separate copy has an advantage. Logical Organization.: Main memory is organized as linear or it can be a one-dimensional address space which consists of a sequence of bytes or words. Most of the programs can be organized into modules, some of those are un modifiable (read-only, execute only) and some of those contain data that can be modified. Physical Organization: The structure of computer memory has two levels referred to as main memory and secondary memory. Main memory is relatively very fast and costly as compared to the secondary memory. Main memory is volatile. Thus secondary memory is provided for storage of data on a long-term basis while the main memory holds currently used programs.		
	b	**Discuss Storage allocation strategies with suitable example**	**[L2][CO4]**	**[8M]**
		Same as 5.(b) answer		
7.		**Evaluate the following terms** **i. Stack allocation** **ii. Static allocation** **iii. heap allocation**	**[L5][CO4]**	**[12M]**
		STORAGE ALLOCATION TECHNIQUES **I. Static Storage Allocation** • For any program, if we create a memory at compile time, memory will be created in the static area. • For any program, if we create a memory at compile-time only, memory is created only once. • It doesn't support dynamic data structure i.e memory is created at compile-time and deallocated after program completion. • The drawback with static storage allocation is recursion is not supported. • Another drawback is the size of data should be known at compile time • Eg- FORTRAN was designed to permit static storage allocation.		

II. **Stack Storage Allocation**

- Storage is organized as a stack and activation records are pushed and popped as activation begins and end respectively. Locals are contained in activation records so they are bound to fresh storage in each activation.
- Recursion is supported in stack allocation

III. Heap Storage Allocation

- Memory allocation and deallocation can be done at any time and at any place depending on the requirement of the user.
- Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
- Recursion is supported.

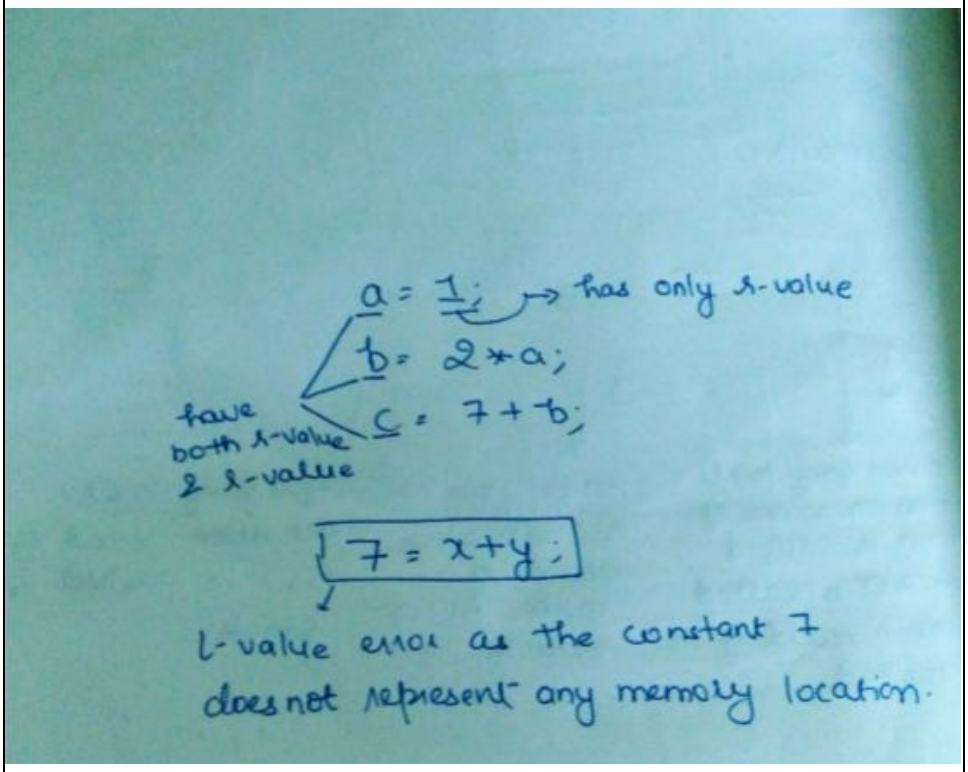


PARAMETER PASSING: The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism.

Basic terminology :

- R- value: The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if its appear on the right side of the assignment operator. R-value can always be assigned to some other variable.

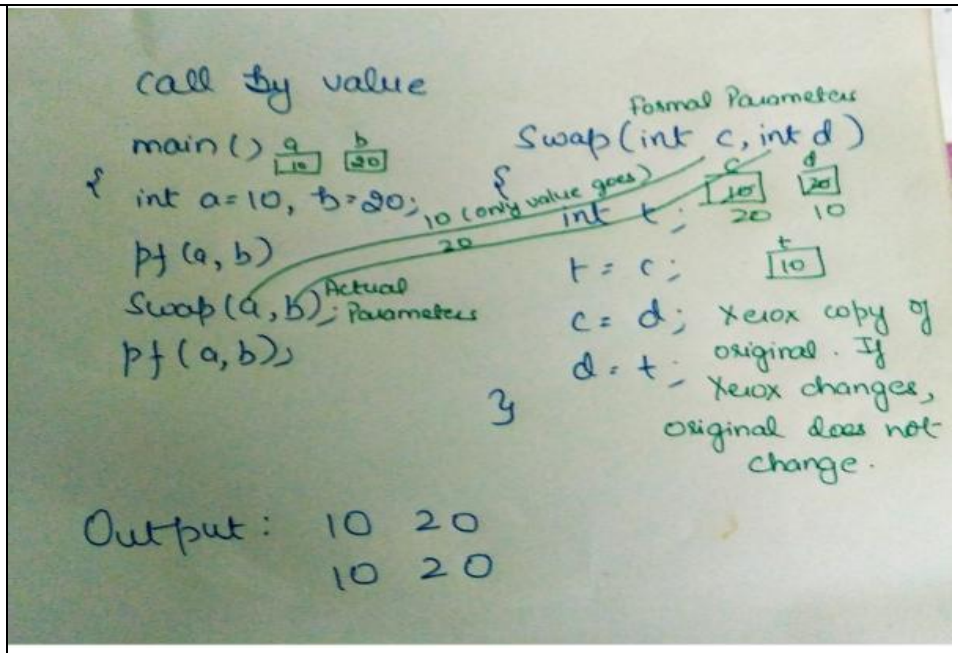
- L-value: The location of the memory(address) where the expression is stored is known as the l-value of that expression. It always appears on the left side if the assignment operator.



i.Formal Parameter: Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function. ii.Actual Parameter: Variables whose values and functions are passed to the called function are called actual parameters. These variables are specified in the function call as arguments.

Different ways of passing the parameters to the procedure:

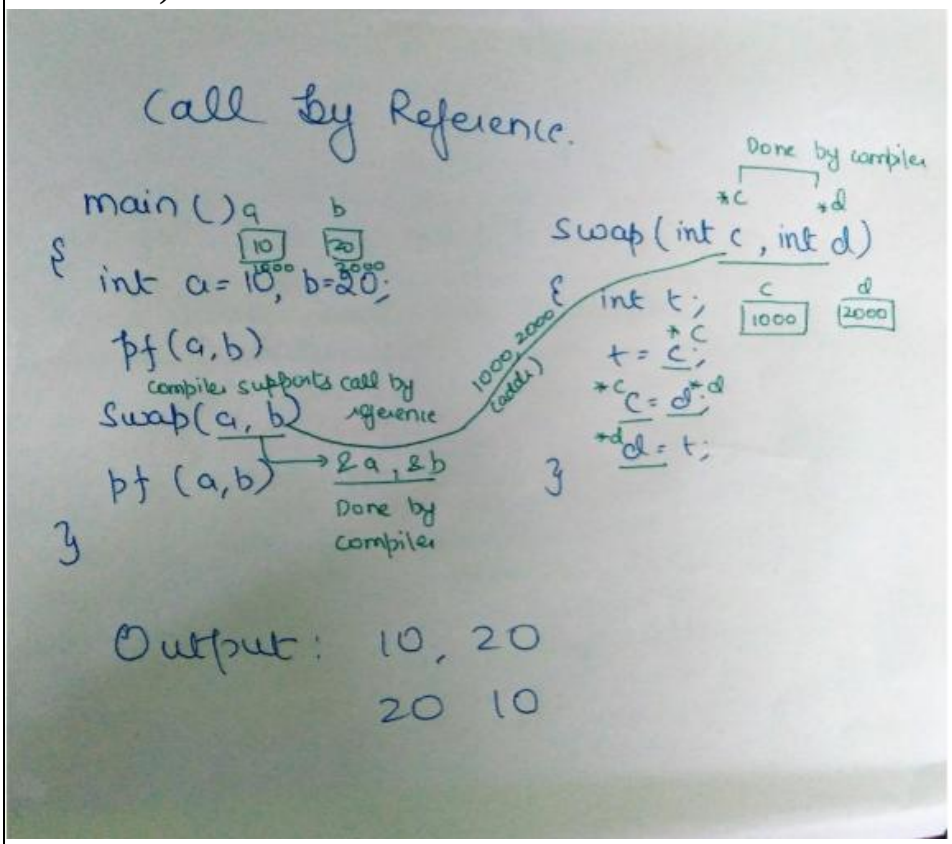
Call by Value: In call by value the calling procedure passes the r-value of the actual parameters and the compiler puts that into called procedure's activation record. Formal parameters hold the values passed by the calling procedure, thus any changes made in the formal parameters do not affect the actual parameters.



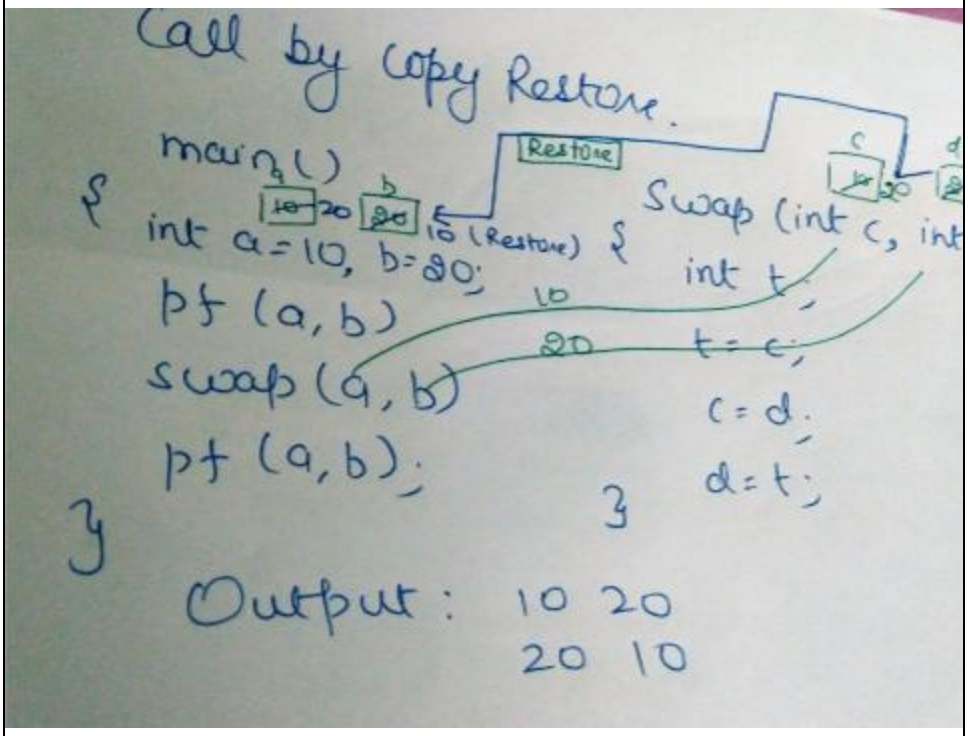
Call by Reference: In call by reference the formal and actual parameters refers to same memory location. The l-value of actual parameters is copied to the activation record of the called function.

Thus the called function has the address of the actual parameters. If the actual parameters does not have a l-value (eg- i+3) then it is evaluated in a new temporary location and the address of the location is passed.

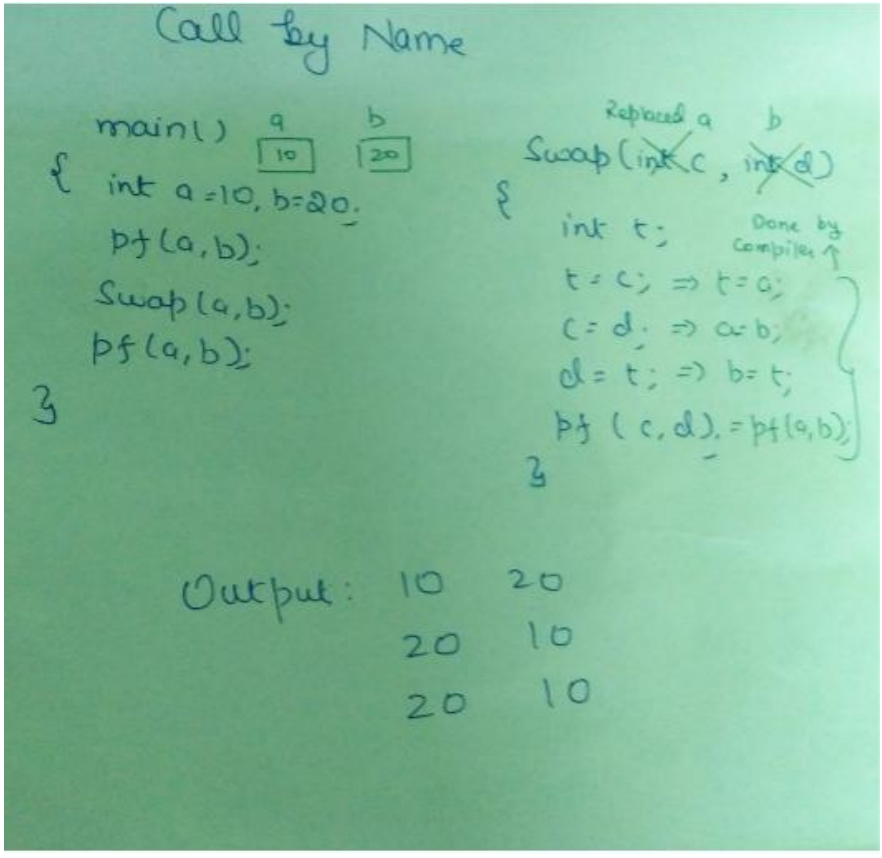
Any changes made in the formal parameter is reflected in the actual parameters (because changes are made at the address).



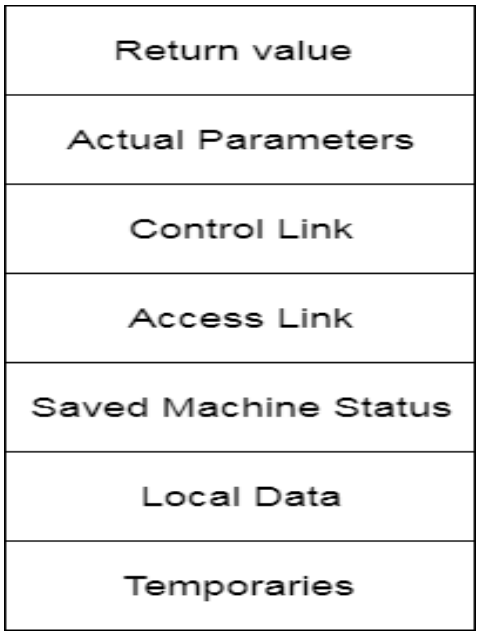
Call by Copy Restore In call by copy restore compiler copies the value in formal parameters when the procedure is called and copy them back in actual parameters when control returns to the called function. The r-values are passed and on return r-value of formals are copied into l-value of actuals.



Call by Name In call by name the actual parameters are substituted for formals in all the places formals occur in the procedure. It is also referred as lazy evaluation because evaluation is done on parameters only when needed.



8.	**a**	**Define Activation Record.**	**[L1][CO5]**	**[2M]**
		Activation Record An Activation Record is a data structure that is activated/ created when a procedure/function is invoked, and it includes the following data about the function. For example Activation Record in 'C' language consist of • Actual Parameters • Number of Arguments • Return Address • Return Value • Old Stack Pointer (SP) • Local Data in a function or procedure		
	b	**Sketch the format of Activation Record in stack allocation and explain each field in it.**	**[L3][CO5]**	**[10M]**
		➢ Control stack is a run time stack which is used to keep track of the live procedure activations i.e. it is used to find out the procedures whose execution have not been completed. ➢ When it is called (activation begins) then the procedure name will push on to the stack and when it returns (activation ends) then it will popped. ➢ Activation record is used to manage the information needed by a single execution of a procedure. ➢ An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function. The diagram below shows the contents of activation records:		



Return Value: It is used by calling procedure to return a value to calling procedure.

Actual Parameter: It is used by calling procedures to supply parameters to the called procedures.

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

Saved Machine Status: It holds the information about status of machine before the procedure is called.

Local Data: It holds the data that is local to the execution of the procedure.

Temporaries: It stores the value that arises in the evaluation of an expression.

9.	a	**Discuss about symbol table entries**	**[L2][CO4]**	**[6M]**

Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variables i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

Symbol Table entries – Each entry in the symbol table is associated with attributes that support the compiler in different phases.

Items stored in Symbol table:

- ✓ Variable names and constants
- ✓ Procedure and function names
- ✓ Literal constants and strings
- ✓ Compiler generated temporaries
- ✓ Labels in source languages

Information used by the compiler from Symbol table:

- ✓ Data type and name
- ✓ Declaring procedures
- ✓ Offset in storage
- ✓ If structure or record then, a pointer to structure table.
- ✓ For parameters, whether parameter passing by value or by reference
- ✓ Number and type of arguments passed to function
- ✓ Base Address

	b	**Describe the various operations on symbol table.**	**[L2][CO4]**	**[6M]**

Operations

A symbol table, either linear or hash, should provide the following operations.

insert()

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:

```
int a;
```

should be processed by the compiler as:

```
insert(a, int);
```

lookup()

lookup() operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

The format of lookup() function varies according to the programming language. The basic format should match the following:

lookup(symbol)

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

10.	**a**	**Define Symbol table.**	**[L1][CO4]**	**[2M]**

SYMBOL TABLE

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

➢ To store the names of all entities in a structured form at one place.
➢ To verify if a variable has been declared.
➢ To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
➢ To determine the scope of a name (scope resolution).

	b	**Explain different types of Data structure used for symbol table.**	**[L2][CO4]**	**[10M]**
		SYMBOL TABLE Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler. A symbol table may serve the following purposes depending upon the language in hand: ✓ To store the names of all entities in a structured form at one place. ✓ To verify if a variable has been declared. ✓ To implement type checking, by verifying assignments and expressions in the source code are semantically correct. ✓ To determine the scope of a name (scope resolution). Implementation of Symbol table – Following are commonly used data structures for implementing symbol table:- **List** – In this method, an array is used to store names and associated information. ✓ A pointer "available" is maintained at end of all stored records and new names are added in the order as they arrive ✓ To search for a name we start from the beginning of the list till available pointer and if not found we get an error "use of the undeclared name" ✓ While inserting a new name we must ensure that it is not already present otherwise an error occurs i.e. "Multiple defined names" ✓ Insertion is fast O(1), but lookup is slow for large tables – O(n) on average ✓ The advantage is that it takes a minimum amount of space. **Linked List –**		

- ✓ This implementation is using a linked list. A link field is added to each record.
- ✓ Searching of names is done in order pointed by the link of the link field.
- ✓ A pointer "First" is maintained to point to the first record of the symbol table.
- ✓ Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

Hash Table –

- ✓ In hashing scheme, two tables are maintained – a hash table and symbol table and are the most commonly used method to implement symbol tables.
- ✓ A hash table is an array with an index range: 0 to table size – 1. These entries are pointers pointing to the names of the symbol table.
- ✓ To search for a name we use a hash function that will result in an integer between 0 to table size – 1.
- ✓ Insertion and lookup can be made very fast – O(1).
- ✓ The advantage is quick to search is possible and the disadvantage is that hashing is complicated to implement.

Binary Search Tree –

- ✓ Another approach to implementing a symbol table is to use a binary search tree i.e. we add two link fields i.e. left and right child.
- ✓ All names are created as child of the root node that always follows the property of the binary search tree.
- ✓ Insertion and lookup are $O(\log_2 n)$ on average.

SIDDARTHA INSTITUTE OF SCIENCE AND TECHNOLOGY:: PUTTUR
Siddharth Nagar, Narayanavanam Road – 517583

QUESTION BANK ANSWER(DESCRIPTIVE)

Subject with Code : Compiler Design (20CS0516) **Course & Branch :** B. Tech - CSE

Year & Sem : III B.Tech & I-Sem **Regulation :** R20

UNIT –V

CODE OPTIMIZATION AND CODE GENERATION

1		**Interpret the principles of optimization techniques to be considered during code generation.** 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. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first. **Function-Preserving Transformations** There are a number of ways in which a compiler can improve a program without changing the function it computes. Function preserving transformations examples: Common sub expression elimination Copy propagation, Dead-code elimination Constant folding **Common Sub expressions elimination:** An occurrence of an expression E is called a common sub-expression 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. **For example** **t1: = 4*i** **t2: = a [t1]** **t3: = 4*j** **t4: = 4*i** **t5: = n** **t6: = b [t4] +t5** The above code can be optimized using the common sub-expression elimination as	**[L3][CO5]**	**[12M]**

t1: = 4*i
t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5

The common sub expression t4: =4*i is eliminated as its computation is already in t1 and the value of i is not been changed from definition to use.

Copy Propagation:

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example:

x=Pi;
A=x*r*r;

The optimization using copy propagation can be done as follows: A=Pi*r*r;

Here the variable x is eliminated.

Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

Example:

```
i=0;
if(i==1)
{
a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

		Constant folding: Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code. For example, a=3.14157/2 can be replaced by a=1.570 there by eliminating a division operation. **Loop Optimizations:** In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop. Three techniques are important for loop optimization: ✓ Code motion, which moves code outside a loop; ✓ Induction-variable elimination, which we apply to replace variables from inner loop. ✓ Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.		
2	**a**	**Discuss about function preserving transformations.** **Function Preserving transformations** • Function Preserving transformations are those transformations that are performed without changing the function it computes. • These are primarily used when global optimizations are performed. • There are four techniques as follows ❖ Common sub expression elimination ❖ Copy Propagation ❖ Dead Code Elimination ❖ Constant propagation. **Common Sub Expression Elimination:** • An occurrence of an expression E is called a common sub expression if E was previously computed and the values of the variables in E have not changed since the previous computation. • We avoid re-computing E if we can use its previously computed value;	**[L2][CO6]**	**[6M]**

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto 5
```

(a) Before.

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B1
```

(b) After.

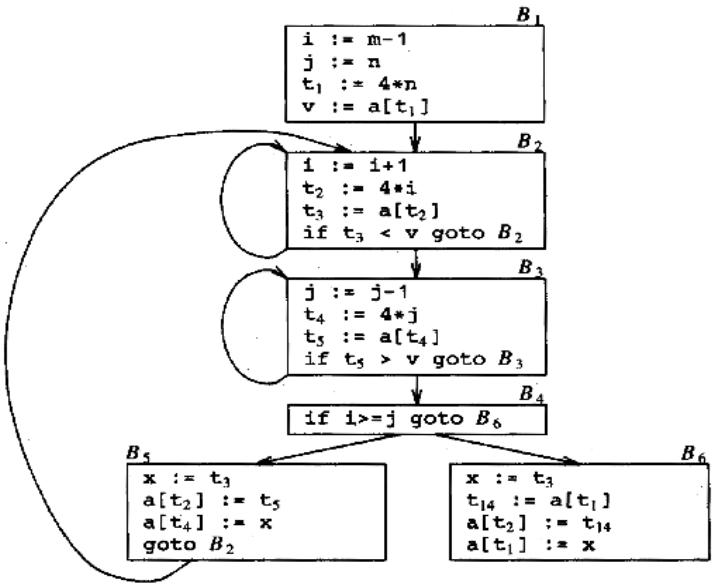


Fig. 10.7. B_5 and B_6 after common subexpression elimination.

Copy Propagation:

- Extension of constant propagation
- After y is assigned to x, use y to replace x till x is assigned again
- **Example**

  x := y; ⇒ s := y * f(y) --------> s := x * f(x)
- Reduce the copying

before	After
x := t3	x := t3
a[t7] := t5	a[t7] := t5
a[t10] := x	a[t10] := t3
Goto b2	Goto b2

Dead Code Elimination:

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
- A dead (or useless) code — statements that compute values that never get used.

		– Ineffective statements • x := y + 1 (immediately redefined, eliminate!) • y := 5 ⇒ y := 5 • x := 2 * z x := 2 * z – A variable is dead if it is never used after last definition • Eliminate assignments to dead variables **Constant propagation:** – If the value of a variable is a constant, then replace the variable by the constant • It is not the constant definition, but a variable is assigned to a constant • The variable may not always be a constant – E.g. N := 10; C := 2; for (i:=0; i<N; i++) { s := s + i*C; } ⇒ for (i:=0; i<10; i++) { s := s + i*2; }		

	b	**Describe about loop optimization technique.**
		
		Loop optimization
		• Loop optimization
		– Consumes 90% of the execution time
		⇒ a larger payoff to optimize the code within a loop
		
		• Techniques
		– Loop invariant detection and code motion
		– Induction variable elimination
		– Strength reduction in loops
		– Loop unrolling
		– Loop peeling
		
		• Loop invariant detection and code motion
		– If the result of a statement or expression does not change within a loop, and it has no external side-effect
		– Computation can be moved to outside of the loop
		– Example
		for (i=0; i<n; i++) a[i] := a[i] + x/y;
		• Three address code
		for (i=0; i<n; i++)
		{
		c := x/y;
		a[i] := a[i] + c;
		}
		⇒ c := x/y;
		for (i=0; i<n; i++) a[i] := a[i] + c;
			[L2][CO5]	**[6M]**

Code Motion:

- Move the code from inner side to outer side

Example:

```
                                    temp = x * 2
if (a<b) then                       if (a<b) then
z = x * 2          ⇒                z = temp
else                                else
 y = 10                             y = 10
```

Induction variable Elimination:

```
        t := b * c
        FOR i := 1 TO10000 DO
                BEGIN
                a := t
        d := i * 3...
                END
...
```

- *Where d* is an induction variable.

Strength Reduction:

- Replacement of an operator with a less costly one.

Example:

```
                                        temp = 5;
 for i=1 to 10 do                       for i=1 to 10 do
        …                               …
        x = i * 5          ⇒            x = temp
        …                               …
        end                             temp = temp + 5
                                        end
```

Typical cases of strength reduction occurs in address calculation of array references.

Loop unrolling:

- Execute loop body multiple times at each iteration
- Get free of the conditional branches, if possible
- Allow optimization to cross multiple iterations of the loop
- Especially for parallel instruction execution
- Space time tradeoff
- Increase in code size, reduce some instructions

Loop peeling:

- Similar to unrolling
- But unroll the first and/or last few iterations

Algebraic identities:

- Worth recognizing single instructions with a constant operand:

```
A * 1 = A
A * 0 = 0
A / 1 = A
A * 2 = A + A
        More delicate with floating-point
```

- Strength reduction:

A ^ 2 = A * A

3		**Explain the following** **i) Basic blocks ii) Flow Graphs** **Basic Blocks and Flow Graphs** Introduces a graph representation of intermediate code that is helpful for discussing code generation even if the graph is not constructed explicitly by a code-generation algorithm. **The representation is constructed as follows:** 1. Partition the intermediate code into basic blocks, which are maximal sequences of consecutive three-address instructions with the properties that a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block. b) Control will leave the block without halting or branching, except possibly at the last instruction in the block. 2. The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks. **1. Basic Blocks:** First job is to partition a sequence of three-address instructions into basic blocks. We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction. **Algorithm** 8.5 : Partitioning three-address instructions into basic blocks. **INPUT**: Asequence of three-address instructions. **OUTPUT**: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block. **METHOD**: First, we determine those instructions in the intermediate code that are leaders, that is, the first instructions in some basic block.The rules for finding leaders are: 1. The first three-address instruction in the intermediate code is a leader. 2. Any instruction that is the target of a conditional or unconditional jump is a leader. 3. Any instruction that immediately follows a conditional or unconditional jump is a leader	**[L3][CO6]**	**[12M]**

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

gure 8.7: Intermediate code to set a 10×10 matrix to an identity matrix

2. Next-Use Information:

- Knowing when the value of a variable will be used next is essential for generating good code.
- We wish to determine for each three-address statement $x = y + z$ what the next uses of x, y, and z are. For the present, we do not concern ourselves with uses outside the basic block containing this three-address statement.

3. Flow Graphs:

- Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks.
- There is an edge from block B to block *C* if and only if it is possible for the first instruction in block *C* to immediately follow the last instruction in block B.
- There are two ways that such an edge could be justified:
  - There is a conditional or unconditional jump from the end of B to the beginning of C.
  - C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

• We say that B is a predecessor of C, and *C* is a successor of B.

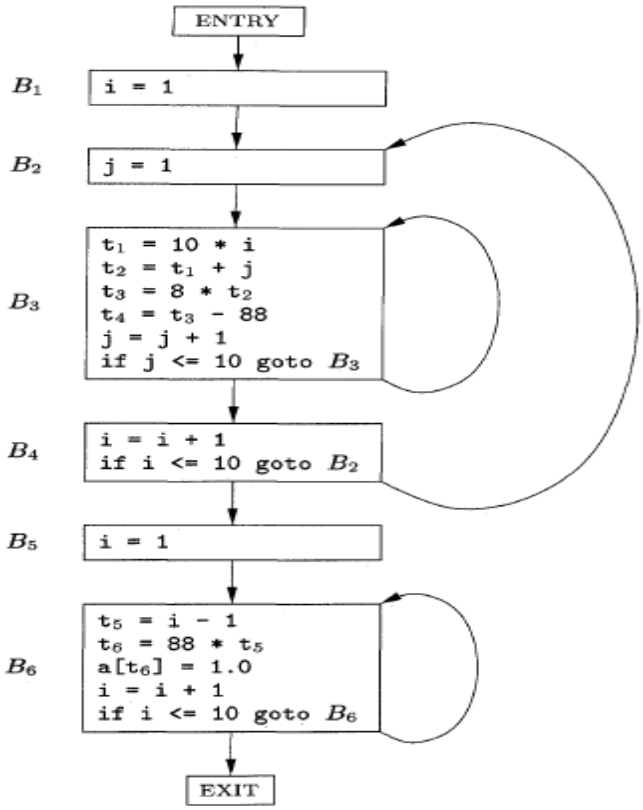


Figure 8.9: Flow graph from Fig. 8.7

4. Representation of Flow Graphs:

- In the flow graph, it is normal to replace the jumps to instruction numbers or labels by jumps to basic blocks.
- Recall that every conditional or unconditional jump is to the leader of some basic block, and it is to this block that the jump will now refer.

5. Loops:

- Programming-language constructs like while-statements, do-while-statements, and for-statements naturally give rise to loops in programs.
- Since virtually every program spends most of its time in executing its loops, it is especially important for a compiler to generate good code for loops.
- Many code transformations depend upon the identification of "loops" in a flow graph

4	**a**	**List the optimization techniques of basic blocks** **Optimization of Basic Blocks** Obtain a substantial improvement in the running time of code merely by performing *local* optimization within each basic block by itself. **1. The DAG Representation of Basic Blocks:** • Many important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph).	**[L1][CO6]**	**[4M]**

- We construct a DAG for a basic block as follows:
    1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
    2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
    3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
    4. Certain nodes are designated ***output nodes***. These are the nodes whose variables are***live on exit*** from the block; that is, their values may be used later, in another block of the flow graph.
- The DAG representation of a basic block lets us perform several code improving transformations on the code represented by the block.
    1. We can **eliminate*local common subexpressions***, that is, instructions that compute a value that has already been computed.
    2. We can **eliminate *dead code***, that is, instructions that compute a value that is never used.
    3. We can **reorder statements** that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
    4. We can **apply algebraic laws to reorder operands** of three-address instructions, and sometimes t hereby simplify the computation.

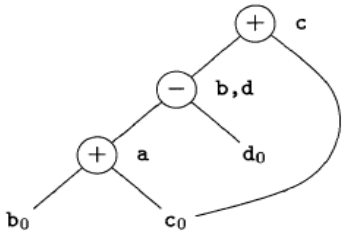
2. Finding Local Common Subexpressions:

- 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.

Example 8.10: A DAG for the block

```
a = b + c
b = a - d
c = b + c
d = a - d
```

is shown in Fig. 8.12. When we construct the node for the third statement c = b + c, we know that the use of b in b + c refers to the node of Fig. 8.12 labeled -, because that is the most recent definition of b. Thus, we do not confuse the values computed at statements one and three.



ıre 8.12: DAG for basic block in Example 8.10

However, the node corresponding to the fourth statement d = a - d has the operator - and the nodes with attached variables a and do as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add d to the list of definitions for the node labeled -.

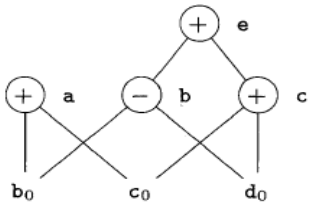
3. Dead Code Elimination:

- The operation on DAG's that corresponds to dead-code elimination can be implemented as follows.

We delete from a DAG any root (**node with no ancestors**) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

- **Example 8.12:** If, in Fig. 8.13, a and b are live but c and e are not, we can immediately remove the root labeled ***e.*** Then, the node labeled c becomes a root and can be removed. The roots labeled a and b remain, since they each have live variables attached.

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



ure 8.13: DAG for basic block in Example 8.11

4. The Use of Algebraic Identities:

- Algebraic identities represent another important class of optimizations on basic blocks.
- For example, we may apply arithmetic identities, such as to eliminate computations from a basic block.

$$x + 0 = 0 + x = x \qquad x - 0 = x$$
$$x \times 1 = 1 \times x = x \qquad x/1 = x$$

- Another class of algebraic optimizations includes local reduction in strength, that is, replacing a more expensive operator by a cheaper one as in:

		$$\begin{array}{lcl} \text{EXPENSIVE} & & \text{CHEAPER} \\ x^2 & = & x \times x \\ 2 \times x & = & x + x \\ x/2 & = & x \times 0.5 \end{array}$$ • A third class of related optimizations is **constant folding**. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression *2* * 3.14 would be replaced by 6.28.		
	b	**Analyse different types of optimization techniques of basic blocks** Optimization process can be applied on a basic block. While optimization, we don't need to change the set of expressions computed by the block. There are two type of basic block optimization. These are as follows: 1. Structure-Preserving Transformations 2. Algebraic Transformations **1. Structure preserving transformations:** The primary Structure-Preserving Transformation on basic blocks is as follows: ✓ Common sub-expression elimination ✓ Dead code elimination ✓ Renaming of temporary variables ✓ Interchange of two independent adjacent statements **(a) Common sub-expression elimination:** In the common sub-expression, you don't need to be computed it over and over again. Instead of this you can compute it once and kept in store from where it's referenced when encountered again. a : = b + c b : = a - d c : = b + c d : = a - d In the above expression, the second and forth expression computed the same expression. So the block can be transformed as follows:	**[L4][CO6]**	**[8M]**

a : = b + c

b : = a - d

c : = b + c

d : = b

(b) Dead-code elimination

- It is possible that a program contains a large amount of dead code.
- This can be caused when once declared and defined once and forget to remove them in this case they serve no purpose.
- Suppose the statement x:= y + z appears in a block and x is dead symbol that means it will never subsequently used. Then without changing the value of the basic block you can safely remove this statement.

(c) Renaming temporary variables

A statement t:= b + c can be changed to u:= b + c where t is a temporary variable and u is a new temporary variable. All the instance of t can be replaced with the u without changing the basic block value.

(d) Interchange of statement

Suppose a block has the following two adjacent statements:

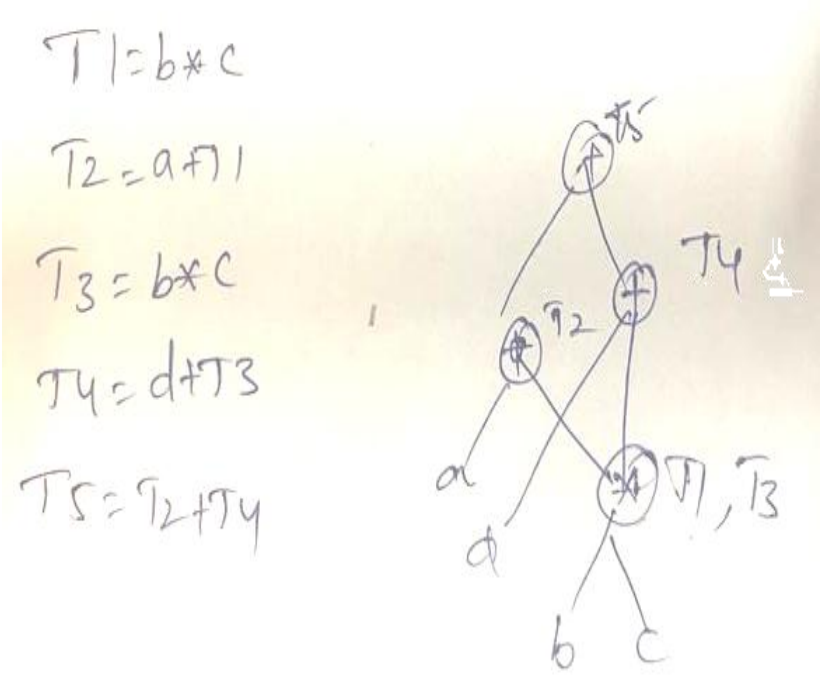
t1 : = b + c

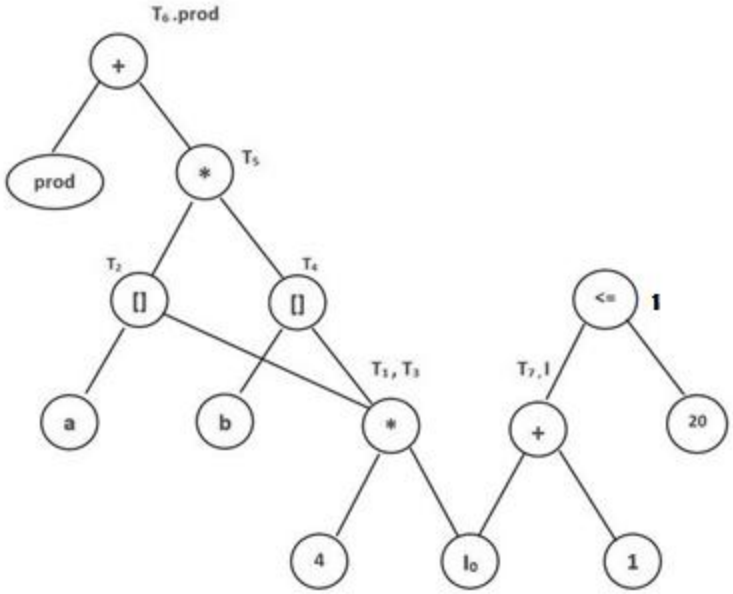
t2 : = x + y

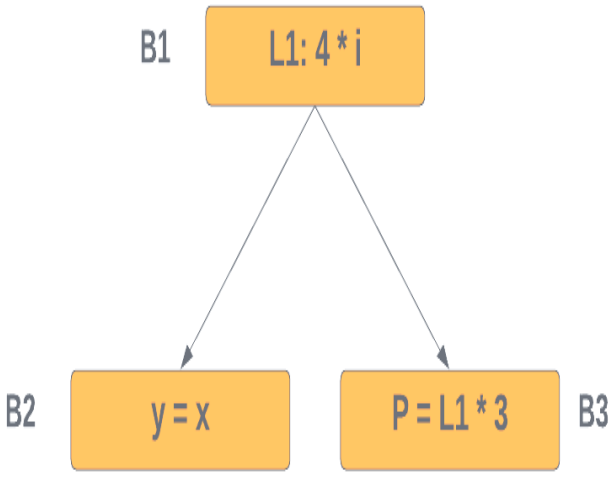
These two statements can be interchanged without affecting the value of block when value of t1 does not affect the value of t2.

2. Algebraic transformations:

- In the algebraic transformation, we can change the set of expression into an algebraically equivalent set. Thus the expression x:= x + 0 or x:= x *1 can be eliminated from a basic block without changing the set of expression.
- Constant folding is a class of related optimization. Here at compile time, we evaluate constant expressions and replace the constant expression by their values. Thus the expression

		5*2.7 would be replaced by13.5. ➢ Sometimes the unexpected common sub expression is generated by the relational operators like <=, >=, <, >, +, = etc. ➢ Sometimes associative expression is applied to expose common sub expression without changing the basic block value. if the source code has the assignments a:= b + c e:= c +d +b The following intermediate code may be generated: a:= b + c t:= c +d e:= t + b		
5	**a**	**Create the DAG for following statement. a+b*c+d+b*c** First convert given expression into 3-address code then construct the DAG for the converted 3-address code  	**[L6][CO6]**	**[6M]**

	b	**Construct the DAG for the following basic blocks** 1. t1:=4*i 2. t2:=a[t1] 3. t3:=4*i 4. t4:=b[t3] 5. t5:=t2*t4 6. t6:=prod+t5 7. prod:=t6 8. t7:=i+1 9. i:=t7 **if i<=20 goto 1** First convert given expression into 3-address code then construct the DAG for the converted 3-address code 	**[L6][CO6]**	**[6M]**
6	**a**	**List out the properties of global data flow analysis and explain it.** properties of the data flow analysis are- ✓ Available expression ✓ Reaching definition ✓ Live variable ✓ Busy expression We will discuss these properties one by one. **Available Expression** An expression **a + b** is said to be available at a program point **x** if none of its operands gets modified before their use. It is used to eliminate common sub expressions. An expression is available at its evaluation point. Example:	**[L2][CO6]**	**[6M]**

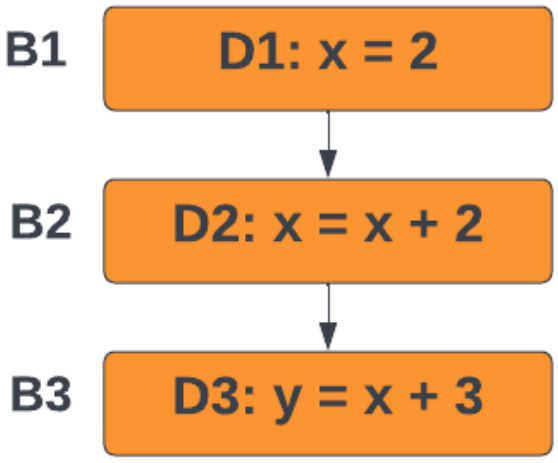


In the above example, the expression L1: 4 * i is an available expression since this expression is available for blocks B2 and B3, and no operand is getting modified.

Reaching Definition

A definition D is reaching a point x if D is not killed or redefined before that point. It is generally used in variable/constant propagation.

Example:



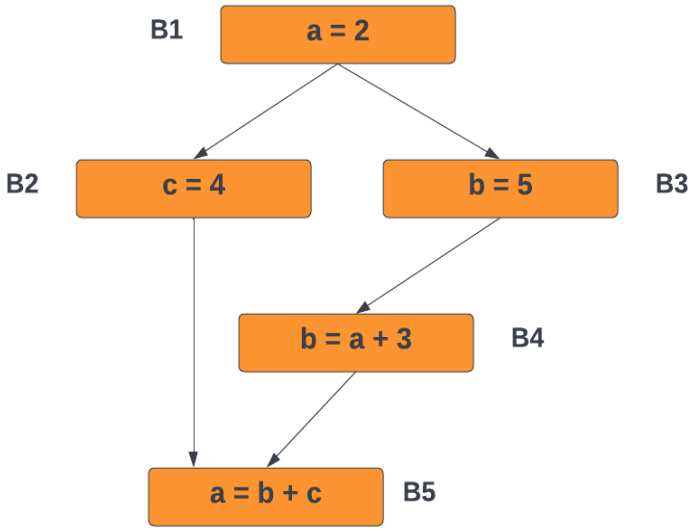
In the above example, D1 is a reaching definition for block B2 since the value of x is not changed (it is two only) but D1 is not a reaching definition for block B3 because the value of x is changed to x + 2. This means D1 is killed or redefined by D2.

Live Variable

A variable **x** is said to be live at a point **p** if the variable's value is not killed or redefined by some block. If the variable is killed or redefined, it is said to be dead.

It is generally used in register allocation and dead code elimination.

Example:

		 In the above example, the variable a is live at blocks B1,B2, B3 and B4 but is killed at block B5 since its value is changed from 2 to b + c. Similarly, variable b is live at block B3 but is killed at block B4. **Busy Expression** An expression is said to be busy along a path if its evaluation occurs along that path, but none of its operand definitions appears before it. It is used for performing code movement optimization		
	b	**Discuss about machine dependent optimization.** Machine dependent code optimization is a type of code optimization that focuses on optimizing code for a specific type of hardware. This type of optimization is usually done with assembly language, which is a low-level programming language that is designed to be used with a specific type of hardware. By taking advantage of the hardware's specific features, the code can be optimized to run faster and more efficiently. The advantage of machine dependent code optimization is that it can provide significant performance gains. This is because the code is specifically designed to take advantage of the specific features of the hardware. The downside is that the code can only run on that specific hardware, meaning that it cannot be transferred to a different type of hardware without significant reworking. **Continue Q.No 7: Peephole Optimization**	**[L2][CO5]**	**[6M]**
7		**Explain the peephole optimization Technique with examples.** **Peephole Optimization** ➢ Generally code generation algorithms produce code statement by statement. ➢ This may contain redundant instructions. ➢ This efficiency of such code can be improved by applying peephole optimization.	**[L2][CO5]**	**[12M]**

- It is a simple but effective technique for locally improved target code.
- It is a small window moving on target code and transformations can be made.

Characteristics of Peephole Optimization

- The peephole optimization can be applied on the target code using following characteristics.
  - Redundant Instruction elimination.
  - Flow of control optimization.
  - Algebraic Simplifications.
  - Reduction in strength.

Redundant Instruction elimination

- Especially the redundant loads and stores can be eliminated in this type of transformations.

Example:

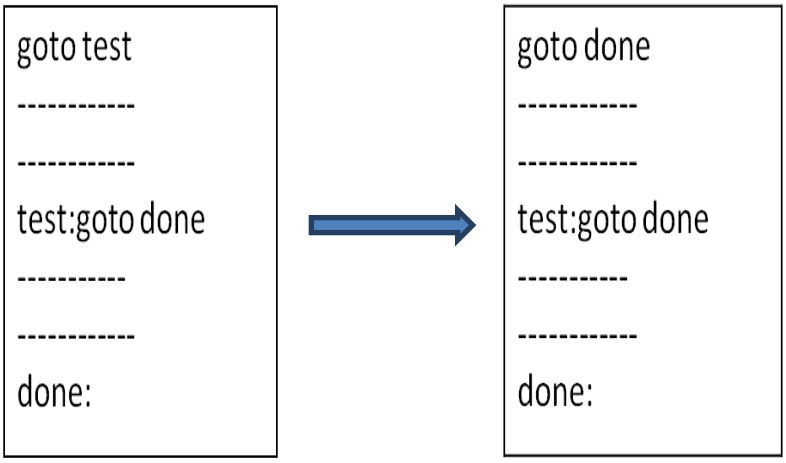
```
MOV R0,X
MOV X,R0
```

In the above code second instruction can be eliminated because x is already in R0**.**

Flow of control optimization.

- By using Peephole optimization , unnecessary jumps on jumps can be eliminated.

Example:



Algebraic Simplifications

- Using peephole optimization we simplyfy algebraic expressions.
- Example :

  X=x+0 or x=x*1

can be eliminated by using peephole optimization.

Reduction in strength

- Reduction in strength means substitute lowest operators in place of highest operators.

  X=2*2 → X=2+2

  X*X ---> X^2

8	**a**	**List all the issues in the design of a code generator** **Issues in the Design of a Code Generator** • The most important criterion for a code generator is that it should produce correct code. • Correctness takes on special significance because of the number of special cases that a code generator might face. • Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal. There are 6 issues in the generation code as follows: • Input to the Code Generator • Memory manager • The Target Program • Instruction Selection • Register Allocation • Evaluation Order **1. Input to the Code Generator:** • The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table. • The many choices for the IR include – **Three-address** representations such as quadruples, triples, indirect triples; – **Linear** representations such as postfix notation; and – **Graphical** representations such as syntax trees and DAG's. **2. The Target Program:** • The most common target-machine architectures are **RISC** (reduced instruction set computer), **CISC** (complex instruction set computer), and stack based. – A **RISC machine** typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture. – A **CISC machine** typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects. • In a **stack-based machine**, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. • **Stack-based architectures** were re-energized with the introduction of the **Java Virtual Machine** (JVM). The **JVM** is a software interpreter for Java bytecodes, an intermediate	**[L2][CO6]**	**[4M]**

language produced by Java compilers.

3. Instruction Selection:

- The code generator must map the IR program into a code sequence that can be executed by the target machine.
- The complexity of performing this mapping is determined by a factors such as
  - the level of the IR
  - the nature of the instruction-set architecture
  - the desired quality of the generated code.
- For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct.
  - For example, every three-address statement of the form **x = y + z**, where x, y, and z are statically allocated, can be translated into the code sequence

```
LD  R0, y       // R0 = y        (load y into register R0)
ADD R0, R0, z   // R0 = R0 + z   (add z to R0)
ST  x, R0       // x = R0        (store R0 into x)
```

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

a = b + c
d = a + e

would be translated into

```
LD  R0, b       // R0 = b
ADD R0, R0, c   // R0 = R0 + c
ST  a, R0       // a = R0
LD  R0, a       // R0 = a
ADD R0, R0, e   // R0 = R0 + e
ST  d, R0       // d = R0
```

Instead of writing the above statements we can write the assembly language as the following

```
LD    R0 , b        //R0=b
ADD   R0 , R0, c    //R0=R0+c
ADD   R0,R0,e       //R0=R0+e
ST    d,R0          //d=R0
```

For example if the target machine has an increment instruction (INC), then the three address statement a=a+1 may be implemented more efficiently by the single instruction INC a, rather than the following code

```
LD    R0,a          // R0=a
ADD   R0,R0,#1      //R0=R0+1
ST    a,R0          //a=R0
```

4. Register Allocation:

- A key problem in code generation is deciding what values to hold in what registers.
- Registers are the fastest computational unit on the target

		machine,
		• The use of registers is often subdivided into two subproblems:
		1. **Register allocation**, during which we select the set of variables that will reside in registers at each point in the program.
		2. **Register assignment**, during which we pick the specific register that a variable will reside in.
		
		t = a + b
		t = t * c
		t = t / d
		Optimal Three address code for the above code is
		
		MOV R0,R0,a
		ADD R0,R0,b
		MUL R0,R0,c
		DIV R0, R0,d
		ST t,R0
		
		5. Evaluation Order:
		• The order in which computations are performed can affect the efficiency of the target code. As we shall see, some computation orders require fewer registers to hold intermediate results than others.
		• Initially, we shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.		

	b	**Explain the issues to be handled when code generator is designed.** The following issues arise during the code generation phase: 1. Input to code generator 2. Target program 3. Memory management 4. Instruction selection 5. Register allocation 6. Evaluation order **1. Input to the code generator** ✓ The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front end. ✓ Intermediate representation has the several choices: a) Postfix notation	**[L2][CO6]**	**[8M]**

b) Syntax tree

c) Three address code

- ✓ We assume front end produces low-level intermediate representation i.e. values of names in it can directly manipulated by the machine instructions.
- ✓ The code generation phase needs complete error-free intermediate code as an input requires.

2. Target program:

The target program is the output of the code generator. The output can be:

a) **Assembly language:** It allows subprogram to be separately compiled.

b) **Relocatable machine language:** It makes the process of code generation easier.

c) **Absolute machine language:** It can be placed in a fixed location in memory and can be executed immediately.

3. Memory management

- ✓ During code generation process the symbol table entries have to be mapped to actual p addresses and levels have to be mapped to instruction address.
- ✓ Mapping name in the source program to address of data is co-operating done by the front end and code generator.
- ✓ Local variables are stack allocation in the activation record while global variables are in static area.

4. Instruction selection:

- ✓ Nature of instruction set of the target machine should be complete and uniform.
- ✓ When you consider the efficiency of target machine then the instruction speed and machine idioms are important factors.
- ✓ The quality of the generated code can be determined by its speed and size.

Example:

The Three address code is:

a:= b + c
d:= a + e

Inefficient assembly code is:

```
MOV b, R0        R0→b
ADD c, R0        R0→ c + R0
MOV R0, a        a → R0
MOV a, R0        R0 →  a
ADD e, R0        R0 →  e + R0
MOV R0, d        d  →  R0
```

5. Register allocation

Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.

The following sub problems arise when we use registers:

Register allocation: In register allocation, we select the set of variables that will reside in register.

Register assignment: In Register assignment, we pick the register that contains variable.

Certain machine requires even-odd pairs of registers for some operands and result.

For example:

Consider the following division instruction of the form:

D x, y

Where,

x is the dividend even register in even/odd register pair

y is the divisor

Even register is used to hold the reminder.
Old register is used to hold the quotient.

6. Evaluation order

The efficiency of the target code can be affected by the order in which the computations are performed. Some computation orders need fewer registers to hold results of intermediate than others.

9	**a**	**Analyse the different forms in target program.** a) **Assembly language:** It allows subprogram to be separately compiled. b) **Relocatable machine language:** It makes the process of code generation easier. c) **Absolute machine language:** It can be placed in a fixed location in memory and can be executed immediately.	**[L4][CO6]**	**[6M]**
	b	**Explain the target machine in code generator.** **The Target Language** we shall use as a target language assembly code for a simple computer that is representative of many register machines. There are Two Methods: ➢ A Simple Target Machine Model ➢ Program and Instruction Costs **1. A Simple Target Machine Model:** Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps. The underlying computer is a byte addressable machine with n general-purpose registers, RO, R1, . . . ,Rn - 1. • Our target computer models a three-address machine done following Operations ❖ Load operation. ❖ Store operation. ❖ Computation operations. ❖ Unconditional jumps ❖ Conditional jumps **Load operation:** ➢ The instruction **LD dst, addr** loads the value in location addr into location dst This instruction denotes the assignment dst = addr. ➢ The most common form of this instruction is **LD r, x** which loads the value in location x into register r. ➢ An instruction of the form **LD r, r2** is a register-to-register copy in which the contents of register r2 are copied into register r. **Store operation:** ➢ The instruction ST x, r Which Stores the value in register r into the location x. ➢ This instruction denotes the assignment x = r.	**[L2][CO6]**	**[6M]**

Computation operations:

- Computation operations of the form OP dst, src1,src2,
  Where OP is a operator like ADD or SUB, and dst, src1, and src2 are locations, not necessarily distinct.
- The effect of this machine instruction is to apply the operation represented by OP to the values in locations src1 and src2, and place the result of this operation in location dst.
- For example, SUB n,r2,r3 computes
  n = r2 - r3. Any value formerly stored in n is lost, but if r1 is r2 or r 3 , the old value is read first. Unary operators that take only one operand do not have a src2.

Unconditional jumps:

Unconditional jumps: The instruction BR L causes control to branch to the machine instruction with label L. (BR stands for branch.)

Conditional jumps

Conditional jumps of the form Bcond r, L,

where r is a register, L is a label, and cond stands for any of the common tests on values in the register r.

For example, BLTZ r, L causes a jump to label L if the value in register r is less than zero, and allows control to pass to the next machine instruction if not.

- We may execute the three-address instruction b = **a [i]** by the machine instructions:

```
LD  R1, i          // R1 = i
MUL R1, R1, 8      // R1 = R1 * 8
LD  R2, a(R1)      // R2 = contents(a + contents(R1))
ST  b, R2          // b = R2
```

The machine-code equivalent would be something like:

```
LD   R1, x         // R1 = x
LD   R2, y         // R2 = y
SUB  R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M         // if R1 < 0 jump to M
```

2. Program and Instruction Costs

- Some common cost measures are the length of compilation time and the size, running time and power consumption of the target program.

- Determining the actual cost of compiling and running a program is a complex problem. Finding an optimal target program for a given source program is an undecidable problem in general.

The Addressing Modes are used as follows

Addressing Mode	Form	Address	Cost
Absolute	M	M	1
Register	R	R	0
Indexed	C(R)	C+Contents(R)	1
Indirect Register	*R	Contents(R)	0
Indirect Indexed	*c(R)	Contents(c+contents(R))	1
literal	#c	c	1

10	**a**	**Analyze Simple code generator.** **A Simple Code Generator** • Consider an algorithm that generates code for a single basic block. • It considers each three-address instruction in turn, and keeps track of what values are in what registers so it can **avoid generating unnecessary loads** and **stores**. • One of the primary **issues** during code generation is deciding **how to use registers** to best advantage. • There are four principal uses of registers: 1. In most machine architectures, some or all of the operands of an operation must be in registers in order to perform the operation. 2. Registers make good temporaries - places to hold the result of a subexpression while a larger expression is being evaluated, 3. Registers are used to hold (global) values that are computed in one basic block and used in other blocks. 4. Registers are often used to help with run-time storage management. **1. Register and Address Descriptors:** • Our code-generation algorithm considers each three-address instruction in turn and decides what loads are necessary to get the **needed operands into registers.** • In order to make the needed decisions, we require a data structure that tells us what program variables currently have their value in a register, and which register or registers. • The desired data structure has the following descriptors: 1. a**register descriptor** keeps track of the variable names whose current value is in that register. 2. an**address descriptor** keeps track of the location or locations where the current value of that variable can be found. **2. The Code-Generation Algorithm:** • An essential part of the algorithm is a function **getReg(I)**, which selects registers for each memory location associated	**[L4][CO6]**	**[6M]**

with the three-address instruction I.

- A possible improvement to the algorithm is to generate code for both x = y + z and x = z + y whenever + is a commutative operator, and pick the better code sequence.

Machine Instructions for Operations:

- For a three-address instruction such as x = y + z, do the following:
    1. Use getReg (x = y + z) to select registers for x, y, and z. Call these R_x, $R_y$and R_z.
    2. If y is not in R_y(according to the register descriptor for R_y), then issue an instruction LD R_y, y', where y' is one of the memory locations for y (according to the address descriptor for y).
    3. Similarly, if z is not in R_z, issue and instruction LD R_z, z', where z' is a location for z .
    4. Issue the instruction ADD R_x, R_y, R_z.

Machine Instructions for Copy Statements:

- There is an important special case: a three-address copy statement of the form x = y.
- We assume that getReg will always choose the same register for both x and y.

Ending the Basic Block:

As we have described the algorithm, variables used by the block may wind up with their only location being a register. If the variable is a temporary used only within the block, that is fine; when the block ends, we can forget about the value of the temporary and assume its register is empty.

Managing Register and Address Descriptors:

As the code-generation algorithm issues load, store, and other machine instructions, it needs to update the register and address descriptors. The rules are as follows:

1. For the instruction LD R, x.

   (a) Change the register descriptor for register R so it holds only x.

   (b) Change the address descriptor for x by adding register R as an additional location.

2. For the instruction ST x, R, change the address descriptor for x to include its own memory location.

3. For an operation such as ADD R_x, R_y ,R_z, implementing a three-address instruction x = y + z.

   (a) Change the register descriptor for R_x so that it holds only x.

   (b). Change the address descriptor for x so that its only location is R_x.

(c). Remove R_x from the address descriptor of any variable other than x.

4. When we process a copy statement x = y, after generating the load for y into register R_y, if needed, and after managing descriptors as for all load statements (per rule 1):

   (a) Add x to the register descriptor for R_y.

   (b) Change the address descriptor for x so that its only location is R_y.

Example : Let us translate the basic block consisting of the three-address statements

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

A summary of all the machine-code instructions generated is in following Fig. The figure also shows the register and address descriptors before and after the translation of each three-address instruction.

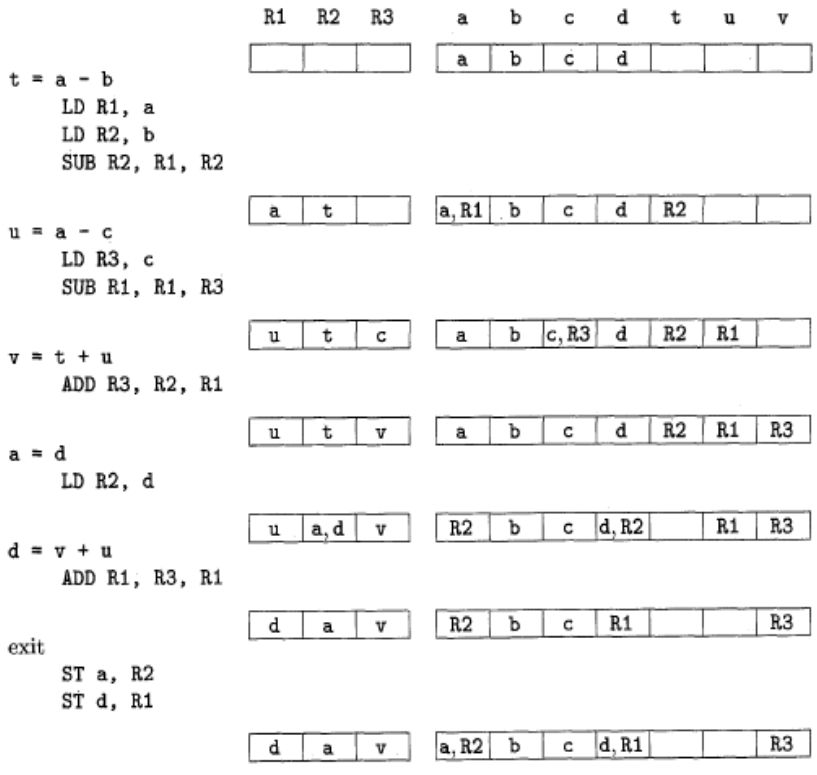


Figure 8.16: Instructions generated and the changes in the register and address descriptors

	b	**Evaluate Register allocation and register assignment techniques.**	**[L5][CO6]**	**[6M]**

Register Allocation and Assignment

- Instructions involving only register operands are faster than those involving memory operands.
- One approach to register allocation and assignment is to assign specific values in the target program to certain registers.

1. Global Register Allocation:

- Since programs spend most of their time in inner loops, a natural approach to global register assignment is to try to keep a frequently used value in a fixed register throughout a loop.
- One strategy for global register allocation is to assign some fixed number of registers to hold the most active values in each inner loop.

2. Usage Counts:

- Assume that the savings to be realized by keeping a variable x in a register for the duration of a loop L is one unit of cost for each reference to x if x is already in a register.
- On the debit side, if x is live on entry to the loop header, we must load x into its register just before entering loop L. This load costs two units.
- Thus, an approximate formula for the benefit to be realized from allocating a register x within loop L is

$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B) \qquad (8.1)$$

Example 8.17 : Consider the the basic blocks in the inner loop depicted in Fig. 8.17, where jump and conditional jump statements have been omitted. Assume registers RO, R1, and R2 are allocated to hold values throughout the loop.

For example, notice that both e and f are live at the end of B1, but of these, only e is live on entry to B2 and only f on entry to B3. In general, the variables live at the end of a block are the union of those live at the beginning of each of its successor blocks.

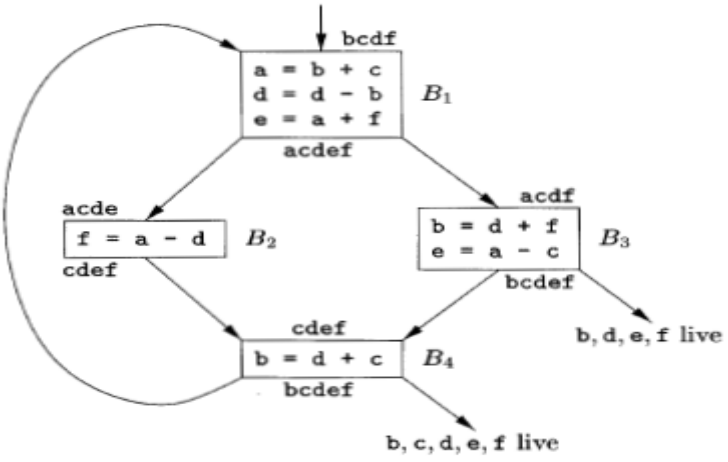


Figure 8.17: Flow graph of an inner loop

		3. Register Assignment for Outer Loops: • Having assigned registers and generated code for inner loops, we may apply the same idea to progressively larger enclosing loops. • If an outer loop L1 contains an inner loop L2, the names allocated registers in L2 need not be allocated registers in L1 - L2. **4. Register Allocation by Graph Coloring:** • When a register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored (spilled) into a memory location in order to free up a register. • Graph coloring is a simple, systematic technique for **allocating** registers and **managing** register spills.		