

## INDEX

<b>Unit #</b>	<b>Ref. Book</b>	<b>Page Numbers</b>
Unit 1	Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, “Compilers – Principles, Techniques and Tools”, 2nd Edition, Pearson Education, 2007.	<b>Page 1 -25</b>
Unit 2	Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, “Compilers – Principles, Techniques and Tools”, 2nd Edition, Pearson Education, 2007.	<b>Page 109-185</b>
Unit 3	Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, “Compilers – Principles, Techniques and Tools”, 2nd Edition, Pearson Education, 2007.	<b>Page 191-287</b>
Unit 4	Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, “Compilers – Principles, Techniques and Tools”, 2nd Edition, Pearson Education, 2007.	<b>Page 303-440</b>
Unit 5	Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, “Compilers – Principles, Techniques and Tools”, 2nd Edition, Pearson Education, 2007.	<b>Page 505-553</b>

## UNIT I INTRODUCTION TO COMPILERS

Structure of a compiler – Lexical Analysis – Role of Lexical Analyzer – Input Buffering – Specification of Tokens – Recognition of Tokens – Lex – Finite Automata – Regular Expressions to Automata – Minimizing DFA.

S. No	Question
1	<p><b>Define Token.</b><u>APRIL/MAY2011,MAY/JUNE 2013</u></p> <p>The token can be defined as a meaningful group of characters over the character set of the programming language like identifiers, keywords, constants and others.</p>
2	<p><b>Define Symbol Table.</b><u>NOV/DEC 2016, MAY/JUNE 2014</u></p> <p>A Symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.</p>
3	<p><b>What is a Compiler?</b> <u>MAY/JUNE 2007</u></p> <p>A Compiler is a program that reads a program written in one language-the source language-and translates it in to an equivalent program in another language-the target language. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.</p>
4	<p><b>What is an interpreter?</b> <u>NOV/DEC 2017</u></p> <p>Interpreter is a program which converts source language to machine language line by line. No intermediate object code is nerated, hence are memory efficient. Ex: Python, COBOL.</p>
5	<p><b>What do you mean by Cross-Compiler?</b> <u>NOV/DEC 2017</u></p> <p>A <b>cross compiler</b> is a <u>compiler</u> capable of creating <u>executable</u> code for a <u>platform</u> other than the one on which the compiler is run. (ie). A compiler may run on one machine and produce target codefor another machine.</p>
6	<p><b>What are the cousins of compiler?</b>  <u>APRIL/MAY2004,APRIL/MAY2005,APRIL/MAY 2012,MAYY/JUNE 2013, MAY/JUNE 2012, APRIL/MAY</u></p>

	<p><b>2017</b> The following are the cousins of compilers</p> <ul style="list-style-type: none"> <li>i. Preprocessors</li> <li>ii. Assemblers</li> <li>iii. Loaders</li> <li>iv. Link editors.</li> </ul>
7	<p><b>What are the four obsoletes of quality What are the main two parts of compilation? What are they performing? MAY/JUNE 2016 , APRIL/MAY 2010, APRIL/MAY 2017, APRIL/MAY 2018</b></p> <p>The two main parts are</p> <p>-<b>Analysis</b> part breaks up the source program into constituent pieces and creates . An intermediate representation of the source program.</p> <p>-<b>Synthesis</b> part constructs the desired target program from the intermediate representation.</p>
8	<p><b>What are an assembler and interpreter?</b> <b>APRIL/MAY 2011</b></p> <p>Assembler is a program, which converts the assembly language in to machine language.</p> <p>Interpreter is a program which converts source language into machine language line by line.</p>
9	<p><b>State any two reasons as to why phases of compiler should be grouped.</b> <b>MAY/JUNE 2014</b></p> <p>The reasons for grouping,</p> <ul style="list-style-type: none"> <li>1. Implementation purpose</li> <li>2. Compiler work is based on two things; one is based on language other one is based on machine.</li> </ul>
10	<p><b>State some software tools that manipulate source program?</b></p> <ul style="list-style-type: none"> <li>i. Structure editors</li> <li>ii. Pretty printers</li> <li>iii. Static checkers</li> <li>iv. Interpreters.</li> </ul>

11	<p><b>What is a Structure editor?</b></p> <p>A structure editor takes as input a sequence of commands to build a source program .The structure editor not only performs the text creation and modification functions of an ordinary text editor but it also analyzes the program text putting an appropriate hierarchical structure on the source program.</p>
12	<p><b>What are a Pretty Printer and Static Checker?</b></p> <p>A Pretty printer analyses a program and prints it in such a way that the structure of the program becomes clearly visible.</p> <ul style="list-style-type: none"> <li>• A static checker reads a program, analyses it and attempts to discover potential bugs with out running the program.</li> </ul>
13	<p><b>How many phases does analysis consists?</b></p> <p>Analysis consists of three phases</p> <ol style="list-style-type: none"> <li>i .Linear analysis</li> <li>ii .Hierarchical analysis</li> <li>iii. Semantic analysis</li> </ol>
14	<p><b>What happens in Hierarchical analysis?</b></p> <p>This is the phase in which characters or tokens are grouped hierarchically in to nested collections with collective meaning.</p>
15	<p><b>What happens in Semantic analysis?</b></p> <p>This is the phase in which certain checks are performed to ensure that the components of a program fit together meaningfully</p>
16	<p><b>State some compiler construction tools?</b></p> <p><b><u>2008</u></b></p> <p style="text-align: right;"><b><u>NOV/DEC 2016, APRIL /MAY</u></b></p> <ol style="list-style-type: none"> <li>v. Parse generator</li> <li>vi. Scanner generators</li> <li>vii. Syntax-directed</li> <li>viii. translation engines</li> <li>ix. Automatic code generator</li> <li>x. . Data flow engines.</li> </ol>

17	<p><b>What is a Loader? What does the loading process do?</b></p> <p>A Loader is a program that performs the two functions i. Loading ii .Link editing The process of loading consists of taking relocatable machine code, altering the relocatable address and placing the altered instructions and data in memory at the proper locations.</p>
18	<p><b>What does the Link Editing does?</b></p> <p>Link editing: This allows us to make a single program from several files of relocatable machine code. These files may have been the result of several compilations, and one or more may be library files of routines provided by the system and available to any program that needs them</p>
19	<p><b>What is a preprocessor? Nov/Dev 2004</b></p> <p>A preprocessor is one, which produces input to compilers. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a distinct program called a preprocessor. The preprocessor may also expand macros into source language statements.</p>
20	<p><b>State some functions of Preprocessors</b></p> <ul style="list-style-type: none"> <li>i) Macro processing</li> <li>ii) File inclusion</li> <li>iii) Relational Preprocessors</li> <li>iv) Language extensions</li> </ul>
21	<p><b>State the general phases of a compiler</b></p> <ul style="list-style-type: none"> <li>i) Lexical analysis</li> <li>ii) Syntax analysis</li> <li>iii) Semantic analysis</li> <li>iv) Intermediate code generation</li> <li>v) Code optimization</li> <li>vi) Code generation</li> </ul>
22	<p><b>What is an assembler?</b></p> <p>Assembler is a program, which converts the source language in to assembly language.</p>
23	<p><b>Depict diagrammatically how a language is processed.</b></p> <p><b><u>MAY/JUNE 2016</u></b></p>

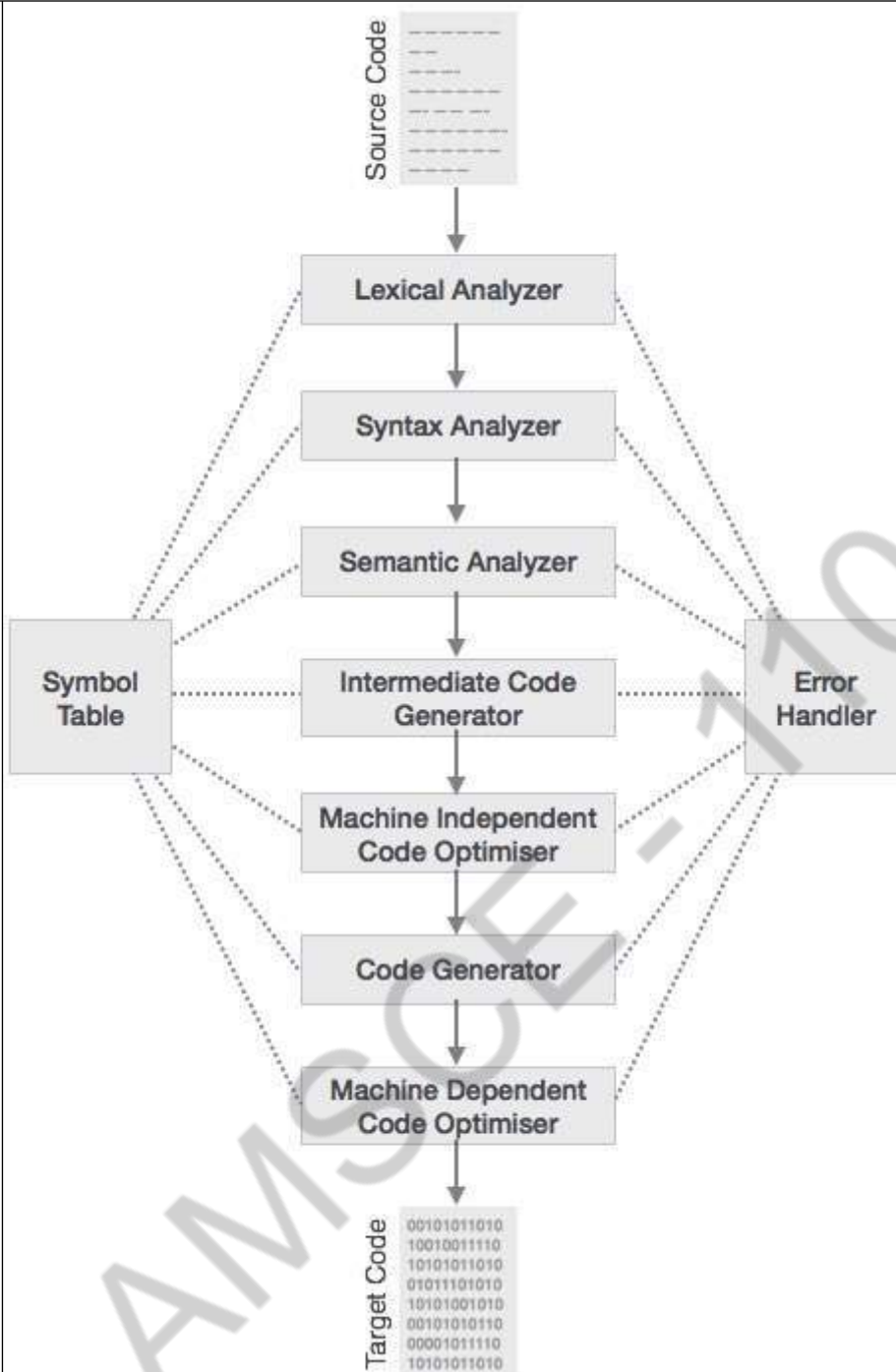
	<pre> graph TD     A[Skeletal Source Program] --&gt; B[Preprocessor]     B -- Source program --&gt; C[Compiler]     C -- Target Assembly program --&gt; D[Assembler]     D -- Relocatable Machine Code --&gt; E[Loader/Linker-editor]     F[Library, relocatable obj file] --&gt; E     E --&gt; G[Absolute Machine Code]   </pre>
24	<p><b>What is linear analysis?</b>        Linear analysis is one in which the stream of characters making up the source program is read from left to right and grouped into tokens that are sequences of characters having a collective meaning. Also called lexical analysis or scanning.</p>
25	<p><b>What are the classifications of a compiler?</b>        Compilers are classified as:</p> <ul style="list-style-type: none"> <li>· Single- pass Multi-pass</li> <li>· Load-and-go</li> <li>· Debugging or optimizing</li> </ul>
26	<p><b>List the phases that constitute the front end of a compiler.</b>        The front end consists of those phases or parts of phases that depend primarily on the source language and are largely independent of the target machine. These include</p> <ul style="list-style-type: none"> <li>• · Lexical and Syntactic analysis</li> <li>• · The creation of symbol table</li> <li>• · Semantic analysis</li> <li>• · Generation of intermediate code</li> </ul> <p>A certain amount of code optimization can be done by the front end as well. Also includes error handling that goes along with each of these phases.</p>

27	<p><b>Mention the back-end phases of a compiler.</b></p> <p>The back end of compiler includes those portions that depend on the target machine and generally those portions do not depend on the source language, just the intermediate language. These include</p> <ul style="list-style-type: none"> <li>• Code optimization</li> <li>• Code generation, along with error handling and symbol- table operations.</li> </ul>
28	<p><b>Define compiler-compiler.</b></p> <p>Systems to help with the compiler-writing process are often been referred to as compiler-compilers, compiler-generators or translator-writing systems. Largely they are oriented around a particular model of languages , and they are suitable for generating compilers of languages similar model.</p>
29	<p><b>What are the advantages of a interpreter ?</b></p> <p>Modification of user program can be easily made and implemented as execution proceeds. Type of object that denotes a various may change dynamically. Debugging a program and finding errors is simplified task for a program used for interpretation. The interpreter for the language makes it machine independent.</p>
30	<p><b>What are the disadvantages of a interpreter</b></p> <p>The execution of the program is <i>slower</i>. <i>Memory</i> consumption is more</p>
31	<p><b>What are the components of a Language processing system?</b></p> <p>Preprocessor Compiler Assembler</p> <p>Loader-Linker editor</p>
32	<p><b>Mention the list of compilers.</b></p> <p>BASIC compilers</p> <p>C# compilers</p> <p>C compilers C++ compilers</p> <p>COBOL compilers</p>
33	<p><b>What is the main difference between phase and pass of a compiler?</b></p> <p>A phase is a sub process of the compilation process whereas combination of one or more phases into a module is called pass.</p>
34	<p><b>Write short notes on error handler?</b></p> <p>The error handler is invoked when a flaw in the source program is detected. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can proceed. So that as many errors as possible can be detected in one compilation.</p>
35	<p><b>How will you group the phases of compiler?</b></p> <p><b>Front and Back Ends:</b> The phases are collected into a front end and a back end.</p> <p><b>Front End:</b> Consists of those phases or parts of phases that depend primarily on the source language and are largely independent of target machine</p> <p><b>Back End:</b> Includes those portions of the compiler that depend on the target machine and these</p>

	portions do not depend on the source language. <b>Passes:</b> It is common for several phases to be grouped into one pass, and for the activity of these phases to be interleaved during the pass.
36	<b>Why lexical and syntax analyzers are separated out?</b>  Reasons for separating the analysis phase into lexical and syntax analyzers: Simpler design. Compiler efficiency is improved. Compiler portability is enhanced.
37	<b>Mention the basic issues in parsing.</b>  There are two important issues in parsing. Specification of syntax Representation of input after parsing.
38	<b>Define parser.</b>  Hierarchical analysis is one in which the tokens are grouped hierarchically into nested collections with collective meaning. Also termed as Parsing.
39	<b>What happens in linear analysis?</b> This is the phase in which the stream of characters making up the source program is read from left to right and grouped in to tokens that are sequences of characters having collective meaning.
40	<b>Give the properties of intermediate representation?</b> a) It should be easy to produce. b) It should be easy to translate into the target program
41	<b>What are the tools available in analysis phase?</b> •Structure editors •Pretty printer •Static checkers •Interpreters.
42	<b>Define assembler and its types?</b>  It is defined by the low level language is assembly language and high level language is machine language is called assembler. •One pass assembler •Two pass assembler
43	<b>What are the functions performed in synthesis phase?</b> •Intermediate code generation •Code generation •Code optimization
45	<b>What do you meant by phases?</b> Each of which transforms the source program one representation to another. A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation.
46	<b>Write short notes on symbol table manager?</b> The table management or bookkeeping portion of the compiler keeps track of the names used by program and records essential information about each, such as its type (int, real etc.,) the



	data structure used to record this information is called a symbol table manger.
47	<p><b>What is front end and back end?</b></p> <p>The phases are collected into a front end and a back end. The front end consists of those phases or parts of phases, that depends primarily on the source language and is largely independent of the target machine. The back ends that depend on the target machine and generally these portions do not depend on the source language.</p>
48	<p><b>What do you meant by passes?</b></p> <p>A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases and writes output into an intermediate file, which may then be read by a subsequent pass. In an implementation of a compiler, portions of one or more phases are combined into a module called pass.</p>
49	<p><b>What Are The Various Types Of Intermediate Code Representation?</b></p> <p>There are mainly three types of intermediate code representations.</p> <ol style="list-style-type: none"> <li>1. Syntax tree</li> <li>2. Postfix</li> <li>3. Three address code</li> </ol>
50	<p><b>Define Token.</b></p> <p>Sequence of characters that have a collective meaning.</p>
1	<p>What are the various phases of a compiler? Explain each phase in detail</p> <p><u>APRIL/MAY 2011, APRIL/MAY 2012, MAY/JUNE 2014, MAY/JUNE 2013, NOV/DEC 2016, NOV/DEC 2017</u></p> <p>The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.</p>



### Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

### Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical

analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

### Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

### Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

### Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

### Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

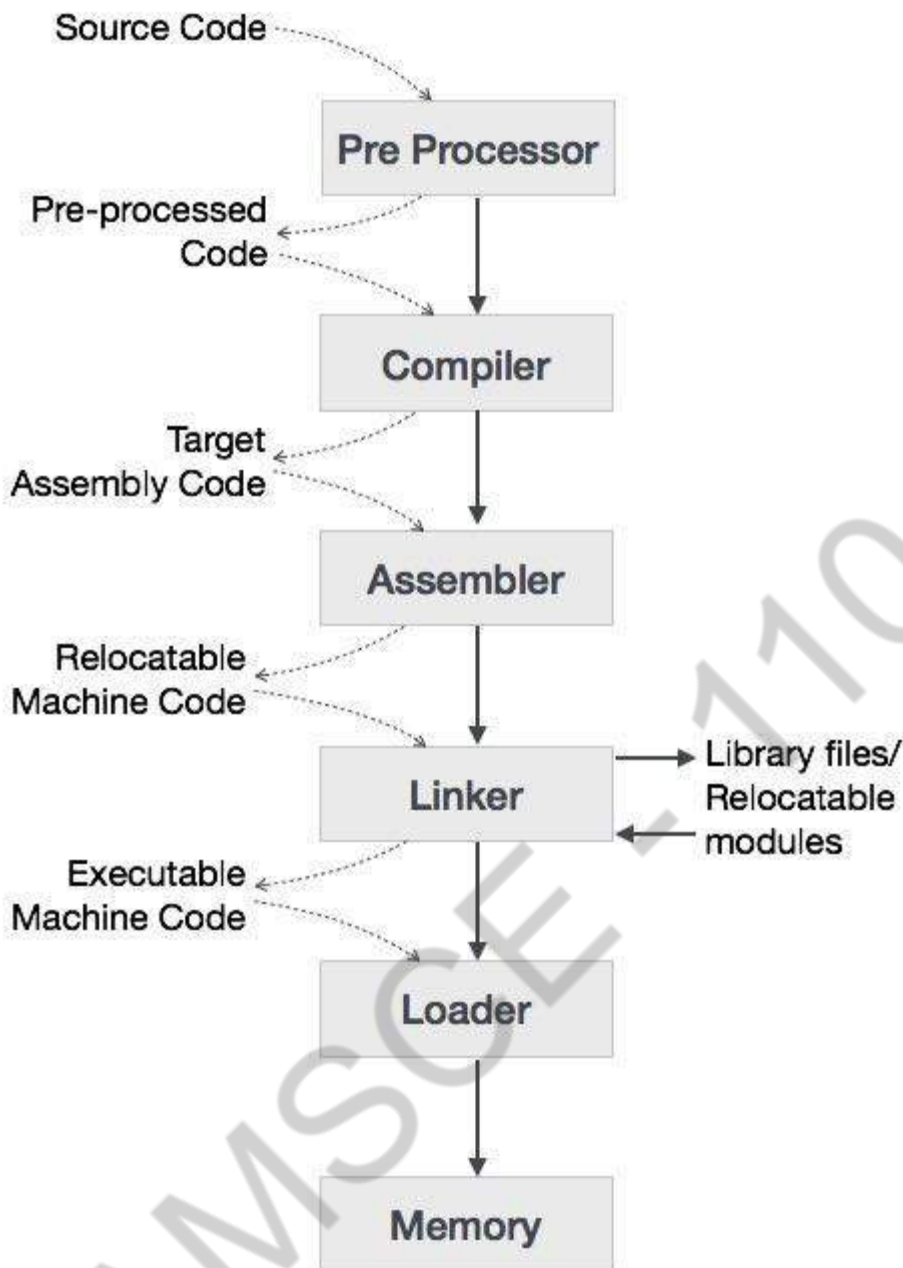
### Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

- 2 Explain the various Cousins of Compiler. (Page No.22) APRIL/MAY 2011, APRIL/MAY 2012, NOV/DEC2014,MAY/JUNE 2015, NOV/DEC 2016, APRIL/MAY 2017, NOV/DEC 2017

### *Language Processing System*

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.



The high-level language is converted into binary language in various phases. A **compiler** is a program that converts high-level language to assembly language. Similarly, an **assembler** is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using C compiler, is executed on a host machine.

- User writes a program in C language (high-level language).
- The C compiler, compiles the program and translates it to assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object).
- A linker tool is used to link all the parts of the program together for execution (executable machine code).

- A loader loads all of them into memory and then the program is executed.

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

#### Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

#### Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

#### Assembler

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

#### Linker

Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

#### Loader

Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

#### Cross-compiler

A compiler that runs on platform (A) and is capable of generating executable code for platform (B) is called a cross-compiler.

#### Source-to-source Compiler

A compiler that takes the source code of one programming language and translates it into the source code of another programming language is called a source-to-source compiler.

3 What are the cousins of a Compiler? Explain them in detail.

Explain the need for grouping of phases of compiler. . (Page No.16) NOV/DEC 2014,

APRIL/MAY 2017

4

Write about the Error handling in different phases. (OR) Explain various Error encountered in different phases of compiler. MAY/JUNE 2016, NOV/DEC 2016

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.

#### 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

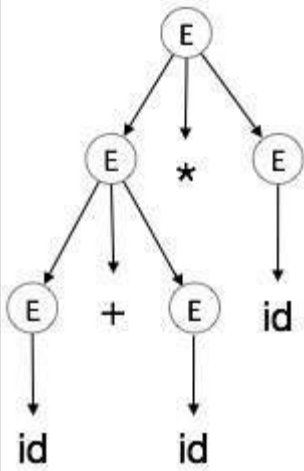
Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

#### Global correction

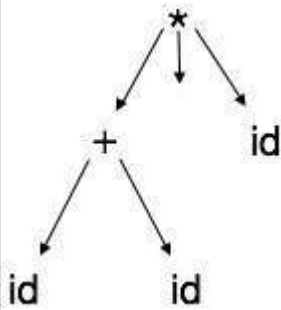
The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow 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.

#### Abstract Syntax Trees

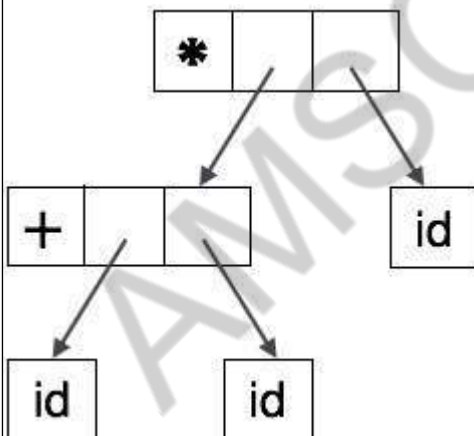
Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed. Take the following parse tree as an example:



If watched closely, we find most of the leaf nodes are single child to their parent nodes. This information can be eliminated before feeding it to the next phase. By hiding extra information, we can obtain a tree as shown below:



Abstract tree can be represented as:



ASTs are important data structures in a compiler with least unnecessary information. ASTs are more compact than a parse tree and can be easily used by a compiler.

5 Draw the transition diagram for relational operators and unsigned numbers.(Page No.131&133)

APRIL/MAY 2017

Transition Diagram has 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

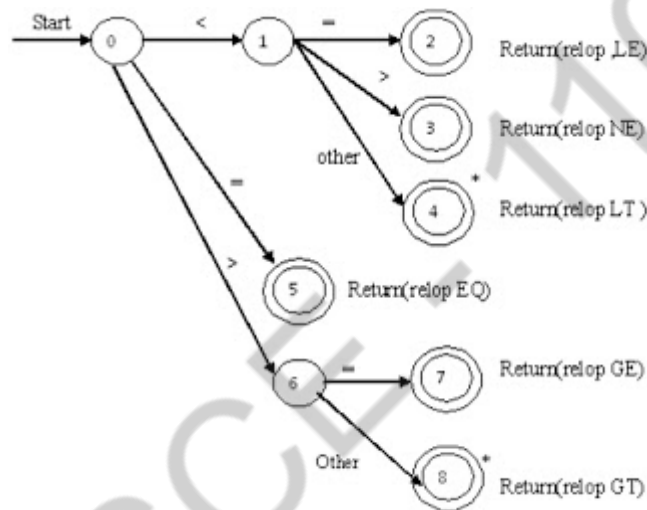
matches one of several patterns .

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 one state  $s$ , and the next input symbol is  $a$ , we look for an edge out of state  $s$  labeled by  $a$ . if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.
2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a \* near that accepting state.
3. One state is designed the state ,or initial state , it is indicated by an edge labeled “start” entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.



6 For the following expression

MAY/JUNE 2016, APRIL/MAY 2017

Position: =initial+ rate\*60. Write down the output after each phase. (Page No.13)

Position:= initial + rate \*60

∧

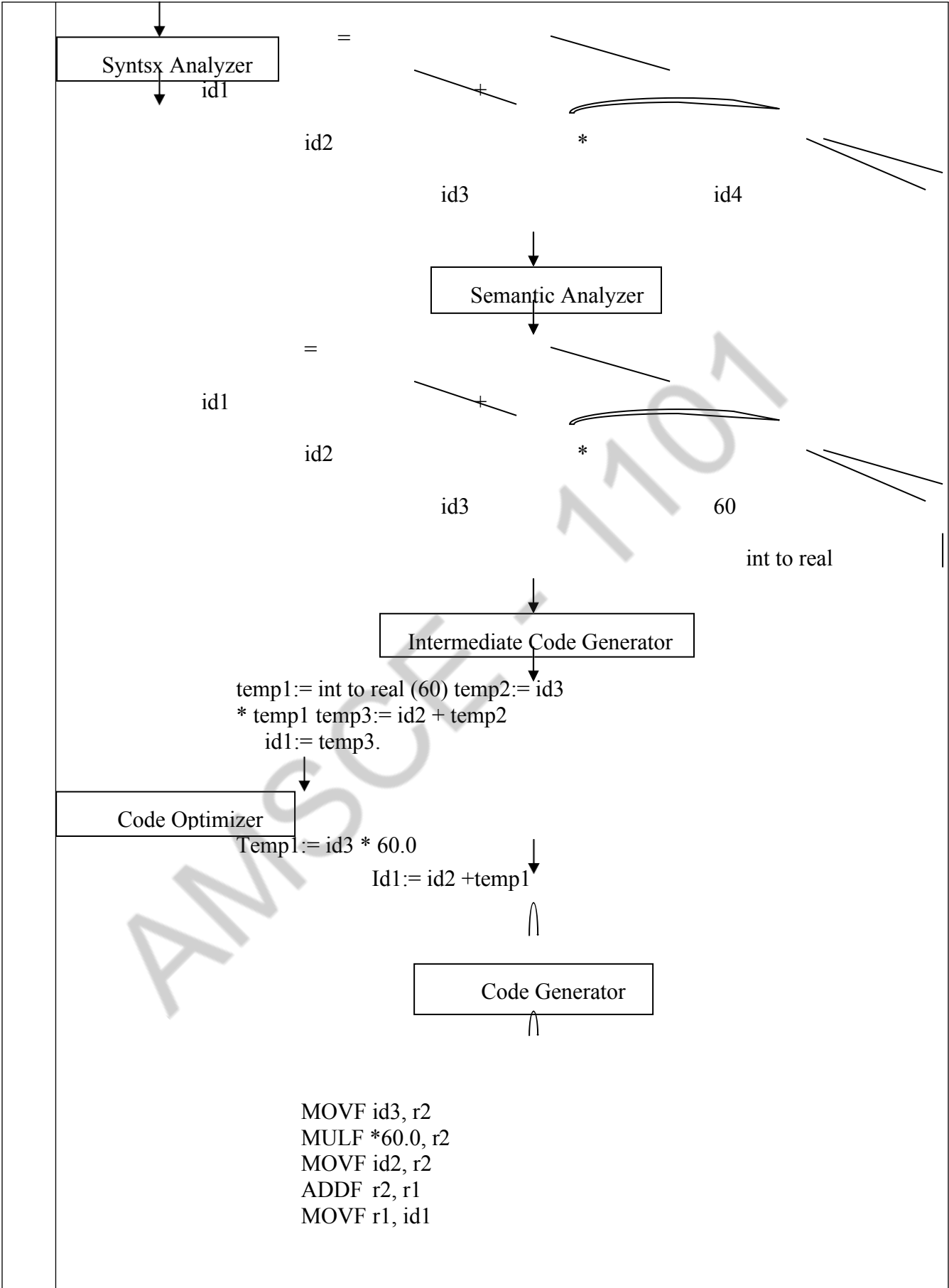
Lexical Analyzer

↓

Tokens

id1 = id2 + id3 \* id4





i) Explain language processing system with neat diagram. (Page No.5)  
MAY/JUNE 2016

7

### *Compiler*

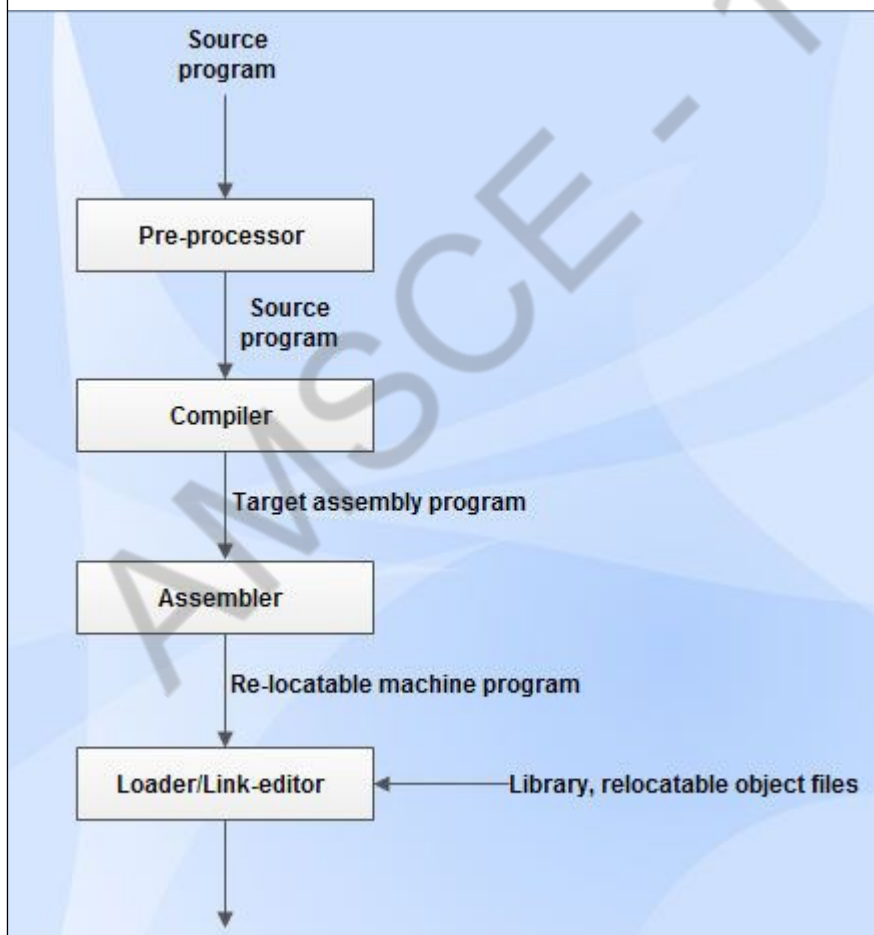
Compiler is a program that takes source program as input and produces assembly language program as output.

### *Assembler*

Assembler is a program that converts assembly language program into machine language program. It produces re-locatable machine code as its output.

### *Loader and link-editor*

- The re-locatable machine code has to be linked together with other re-locatable object files and library files into the code that actually runs on the machine.
- The linker resolves external memory addresses, where the code in one file may refer to a location in another file.
- The loader puts together the entire executable object files into memory for execution



--	--

---

---

AMSCE - 1101

8 Explain Compiler Construction Tools?

1. Parser generators.
2. Scanner generators.
3. Syntax-directed translation engines.
4. Automatic code generators.
5. Data-flow analysis engines.
6. Compiler-construction toolkits.

### ***Parser Generators***

**Input:** Grammatical description of a programming language

**Output:** Syntax analyzers.

Parser generator takes the grammatical description of a programming language and produces a syntax analyzer.

### ***Scanner Generators***

**Input:** Regular expression description of the tokens of a language

**Output:** Lexical analyzers.

Scanner generator generates lexical analyzers from a regular expression description of the tokens of a language.

### ***Syntax-directed Translation Engines***

**Input:** Parse tree.

**Output:** Intermediate code.

Syntax-directed translation engines produce collections of routines that walk a parse tree and generates intermediate code.

### ***Automatic Code Generators***

**Input:** Intermediate language.

**Output:** Machine language.

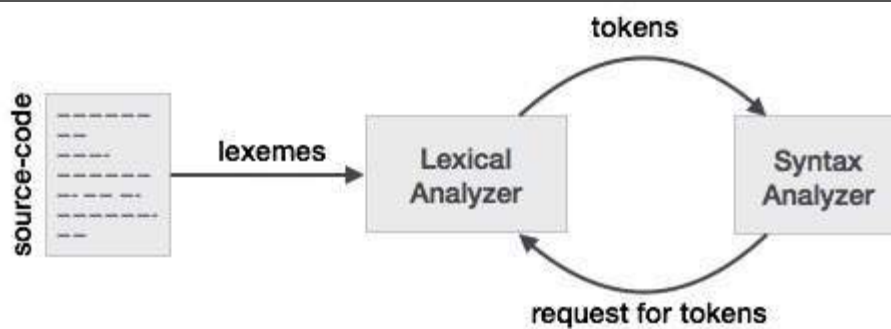
Code-generator takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for a target machine.

### ***Data-flow Analysis Engines***

Data-flow analysis engine gathers the [information](#), that is, the values transmitted from one part of a program to each of the other parts. Data-flow analysis is a key part of code optimization.

### ***Compiler Construction Toolkits***

	<p>The toolkits provide integrated set of routines for various phases of compiler. Compiler construction toolkits provide an integrated set of routines for construction of phases of compiler.</p>
9	<p>(i).Tell the various phases of the compiler and examine with programs segment (Page No.10)</p> <p>(ii).Discuss in detail about symbol table. (Page No.5)</p> <p><b>Symbol Table</b> is an important data structure created and maintained by the compiler in order to keep track of semantics of variable i.e. it stores information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.</p> <ul style="list-style-type: none"> <li>• It is built in lexical and syntax analysis phases.</li> <li>• The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code.</li> <li>• It is used by compiler to achieve compile time efficiency.</li> <li>• It is used by various phases of compiler as follows :-       <ol style="list-style-type: none"> <li>1. <b>Lexical Analysis:</b> Creates new table entries in the table, example like entries about token.</li> <li>2. <b>Syntax Analysis:</b> Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.</li> <li>3. <b>Semantic Analysis:</b> Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.</li> <li>4. <b>Intermediate Code generation:</b> Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.</li> <li>5. <b>Code Optimization:</b> Uses information present in symbol table for machine dependent optimization.</li> <li>6. <b>Target Code generation:</b> Generates code by using address information of identifier present in the table.</li> </ol> </li> </ul> <p><b>Symbol Table entries</b> – Each entry in symbol table is associated with attributes that support compiler in different phases.</p>
10	<p>What is meant by lexical analysis? Identify the lexemes that makeup the token in the following program segment.indicate the correspond token and pattern.</p> <pre>Void swap(int i, int j) {   int t; t = i ; i = j ; j = t ; }</pre> <p>Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.</p> <p>If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.</p>



### Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

```
int value = 100;
```

contains the tokens:

```
int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).
```

### Specifications of Tokens

Let us understand how the language theory undertakes the following terms:

#### Alphabets

Any finite set of symbols  $\{0,1\}$  is a set of binary alphabets,  $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$  is a set of Hexadecimal alphabets,  $\{a-z, A-Z\}$  is a set of English language alphabets.

#### Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by  $|tutorialspoint| = 14$ . A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by  $\epsilon$  (epsilon).

#### Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=

Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&,  ,   , !
Shift Operator	>>, >>>, <<, <<<

### Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

AMSCCE - 1701

## Extra Questions For Unit- I

S. No.	Question
1	<p><b>Write a grammar for branching statements. <u>MAY/JUNE 2016</u> Stmt-&gt; if expr then</b></p> <p style="padding-left: 40px;"><b>stmt</b></p> <p style="padding-left: 80px;"><b>  if expr then stmt else stmt</b></p> <p style="padding-left: 120px;"><b>  ε</b></p> <p style="padding-left: 40px;"><b>expr-&gt; term relop term</b></p> <p style="padding-left: 80px;"><b>  term term -&gt; id</b></p>
2	<p><b>What is a lexeme? Define a regular set. <u>APRIL/MAY2011,MAY/JUNE2013</u></b>  <b><u>MAY/JUNE2014, NOV/DEC 2017</u></b></p> <p>A Lexeme is a sequence of characters in the source program that is matched by the pattern for a token. A language denoted by a regular expression is said to be a regular set.</p>
3	<p><b>What is a regular expression? State the rules, which define regular expression?</b>  <b><u>MAY/JUNE 2007,APRIL/MAY 2018</u></b></p> <p>Regular expression is a method to describe regular Language <u>Rules:</u></p> <ol style="list-style-type: none"> <li>1) <math>\epsilon</math>-is a regular expression that denotes <math>\{\epsilon\}</math> that is the set containing the empty string</li> <li>2) If <math>a</math> is a symbol in <math>\Sigma</math>, then <math>a</math> is a regular expression that</li> </ol>



	<p>denotes {a}</p> <p>3) Suppose r and s are regular expressions denoting the languages L(r) and L(s) Then,</p> <p>a) (r) (s) is a regular expression denoting L(r) U L(s).</p> <p>b) (r)(s) is a regular expression denoting L(r)L(s)</p> <p>c) (r)* is a regular expression denoting L(r)*.</p> <p>d) (r) is a regular expression denoting L(r).</p>
4	<p><b>What are the Error-recovery actions in a lexical analyzer?</b> <u>APRIL/MAY 2012, MAY/JUNE 2013, APRIL/MAY 2015, APRIL/MAY 2018</u></p> <p>Deleting an extraneous character Inserting a missing character</p> <p>Replacing an incorrect character by a correct character</p> <p>Transposing two adjacent characters</p>
5	<p><b>Draw a transition diagram to represent relational operators.</b> <u>NOV/DEC 2007</u></p>
6	<p><b>What are the issues to be considered in the design of lexical analyzer?</b> <u>MAY/JUNE 2009</u></p> <p>How to Precisely Match Strings to Tokens How to Implement a Lexical Analyzer</p>

7	<p><b>Write short notes on buffer pair. <u>APRIL/MAY 2008</u></b></p> <p>Lexical analyzer will detect the tokens from the source language with the help of input buffering. The reason is, the lexical analyzer will scan the input character by character, it will increase the cost file read operation. So buffering is used. The buffer is a pair in which each half is equal to system read command.</p>
8	<p><b>How the token structure is is specified? Or Define Patterns. <u>APRIL/MAY 2010, MAY/JUNE 2013</u></b></p> <p>Token structure is specified with the help of Pattern. The pattern can be described with the help of Regular Expression</p>
9	<p><b>What is the role of lexical analyzer? <u>NOV/DEC 2011, NOV/DEC 2014, NOV/DEC 2017</u></b></p> <p>Its main task is to read input characters and produce as output a sequence of tokens that parser uses for syntax analysis. Additionally task is removing blank, new line and tab characters</p>
10	<p><b>Give the transition diagram for an identifier. <u>NOV/DEC 2011</u></b></p> <div data-bbox="332 1192 1088 1291" data-label="Diagram"> <pre> graph LR     start((start)) -- letter --&gt; 9((9))     9 -- letter --&gt; 10((10))     10 -- "letter or digit" --&gt; 10     10 -- other --&gt; 11(((11)*))     11 -- "return(gettoken(), install_id())" --&gt; 11   </pre> </div> <p><b>Fig. 3.13. Transition diagram for identifiers and keywords.</b></p>
11	<p><b>Why is buffering used in lexical analysis? What are the commonly used buffering methods?<u>MAY/JUNE 2014</u></b></p> <p>Lexical analyzer needs to get the source program statement from character by character, without buffering it is difficult to synchronize the speed between the read write hardware and the</p>

	lexical program. Methods are two way buffering and sentinels.
12	<p><b>Write regular expression to describe a languages consist of strings made of even numbers a and b.</b></p> <p style="text-align: right;"><b><u>NOV/DEC 2014</u></b></p> <p><math>((a+b)(a+b))^*</math></p>
13	<p><b>What are the various parts in LEX program? <u>APRIL/MAY 2017</u></b></p> <p>Lex specification has three parts declarations</p> <p>%%</p> <p>pattern specifications</p> <p>%%</p> <p>support routines</p>
14	<p><b>Write regular expression for identifier and number. <u>NOV/DEC 2012, APRIL/MAY 2017</u></b></p> <p>For identifier <math>(a-z)((a-z) (0-9))^*</math> other symbols For numbers <math>(0-9)(0-9)^*</math></p>
15	<p><b>What is the need for separating the analysis phase into lexical analysis and parsing? (Or) What are the issues of lexical analyzer?</b></p> <ul style="list-style-type: none"> <li>• Simpler design is perhaps the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.</li> <li>• Compiler efficiency is improved</li> <li>• Compiler portability is enhanced</li> </ul>
16	<p><b>What is Lexical Analysis?</b></p> <p>The first phase of compiler is Lexical Analysis. This is also known as linear analysis in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a</p>

	collective meaning.
17	<p><b>What is a sentinel? What is its usage?</b>  <u>April/May 2004</u></p> <p>A Sentinel is a special character that cannot be part of the source program. Normally we use 'eof' as the sentinel. This is used for speeding-up the lexical analyzer.</p>
18	<p><b>What is a regular expression? State the rules, which define regular expression?</b></p> <p>Regular expression is a method to describe regular language</p> <p><b>Rules:</b></p> <ol style="list-style-type: none"> <li>1) <math>\epsilon</math>-is a regular expression that denotes <math>\{\epsilon\}</math> that is the set containing the empty string</li> <li>2) If <math>a</math> is a symbol in <math>\Sigma</math>, then <math>a</math> is a regular expression that denotes <math>\{a\}</math></li> <li>3) Suppose <math>r</math> and <math>s</math> are regular expressions denoting the languages <math>L(r)</math> and <math>L(s)</math>  Then, <ol style="list-style-type: none"> <li>a) <math>(r) (s)</math> is a regular expression denoting <math>L(r) \cup L(s)</math>.</li> <li>b) <math>(r)(s)</math> is a regular expression denoting <math>L(r)L(s)</math></li> <li>c) <math>(r)^*</math> is a regular expression denoting <math>L(r)^*</math>.</li> <li>d) <math>(r)</math> is a regular expression denoting <math>L(r)</math>.</li> </ol> </li> </ol>
19	<p><b>Construct Regular expression for the language <math>L = \{w \in \{a,b\}^*/w \text{ ends in } abb\}</math></b></p> <p>Ans: <math>\{a/b\}^*abb</math>.</p>
20	<p><b>What is recognizer?</b></p> <p>Recognizers are machines. These are the machines which accept the strings belonging to certain language. If the valid strings of such language are accepted by the machine then it is said that the corresponding language is accepted by that machine, otherwise it is rejected.</p>
21	<b>Differentiate tokens, patterns, lexeme.</b>

	<p style="text-align: center;"><b><u>NOV/DEC 2016</u></b></p> <ul style="list-style-type: none"> <li>· Tokens- Sequence of characters that have a collective meaning.</li> <li>· Patterns- There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token</li> <li>· Lexeme- A sequence of characters in the source program that is matched by the pattern for a token.</li> </ul>
22	<p><b>List the operations on languages.</b></p> <p style="text-align: center;"><b><u>MAY/JUNE 2016</u></b></p> <ul style="list-style-type: none"> <li>· <b>Union</b> – <math>L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}</math></li> <li>· <b>Concatenation</b> – <math>LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}</math></li> <li>· <b>Kleene Closure</b> – <math>L^*</math> (zero or more concatenations of L)</li> <li>· <b>Positive Closure</b> – <math>L^+</math> ( one or more concatenations of L)</li> </ul>
23	<p><b>Write a regular expression for an identifier.</b></p> <p>An identifier is defined as a letter followed by zero or more letters or digits. The regular expression for an identifier is given as <b>letter (letter   digit)*</b></p>
24	<p><b>Mention the various notational short hands for representing regular expressions.</b></p> <ul style="list-style-type: none"> <li>· One or more instances (+)</li> <li>· Zero or one instance (?)</li> <li>· Character classes ([abc] where a,b,c are alphabet symbols denotes the regular expressions a   b   c.)</li> <li>· Non regular sets</li> </ul>
25	<p><b>What is the function of a hierarchical analysis?</b></p> <p>Hierarchical analysis is one in which the tokens are grouped hierarchically into nested collections with collective meaning. Also termed as Parsing.</p>
26	<p><b>What does a semantic analysis do?</b></p> <p>Semantic analysis is one in which certain checks are performed to ensure that components of a program fit together meaningfully. Mainly performs type checking.</p>

27	<p><b>What is a lexical error ?</b></p> <p>Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognise a <i>lexeme</i> as a valid <i>token</i> for you lexer. Syntax errors, on the other side, will be thrown by your scanner when a given set of already recognised valid tokens don't match any of the right sides of your grammar rules.</p>
28	<p><b>State the conventions of a transition diagram.</b></p> <p>Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.</p> <p>In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.</p> <p>One state is designed the state ,or initial state ., it is indicated by an edge labeled “start” entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.</p>
29	<p><b>What is DFA?</b></p> <ul style="list-style-type: none"> <li>• A Deterministic Finite Automaton (DFA) is a special form of a NFA.</li> <li>• No state has <math>\epsilon</math>- transition</li> <li>• For each symbol a and state s, there is at most one labeled edge a leaving s. i.e. transition function is from pair of state- symbol to state (not set of states).</li> </ul>
30	<p><b>Define NFA.</b></p> <p>A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x. <math>\epsilon</math>- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.</p>
31	<p><b>What is a finite automata?</b></p>

A *recognizer* for a language is a program that takes a string  $x$ , and answers “yes” if  $x$  is a sentence of that language, and “no” otherwise.

- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic (DFA)* or *non-deterministic (NFA)*.

32 **Differentiate NFA and DFA. NOV/DEC 2017**

NFA	DFA
NFA or Non Deterministic Finite Automaton is the one in which there exists many paths for a specific input from current state to next state.	Deterministic Finite Automaton is a FA in which there is only one path for a specific input from current state to next state. There is a unique transition on each input symbol.
Transition Function $\delta : Q \times \Sigma \rightarrow 2^Q$	Transition Function $\delta : Q \times \Sigma \rightarrow Q$

33 **What are the rules that define the regular expression over alphabet? (Or) List the rules that form the BASIS. NOV/DEC 2016**

- $\epsilon$  is a regular expression denoting  $\{ \epsilon \}$ , that is, the language containing only the empty string.
- For each ‘a’ in  $\Sigma$ , is a regular expression denoting  $\{ a \}$ , the language with only one string consisting of the single symbol ‘a’ .
- If R and S are regular expressions, then
  - (R) | (S) means  $L(r) \cup L(s)$
  - R.S means  $L(r).L(s)$   $R^*$  denotes  $L(r^*)$

34 **Construct Regular expression for the language  $L = \{w \in \{0,1\}^* / w$**

	<p><b>consists of odd number of 0's}</b></p> <p>RE = 0(001)*11</p>
35	<p><b>Give the parts of a string?</b></p> <p>Prefix of s, suffix of s, substring of s, proper prefix, proper suffix, proper substring and subsequence of s.</p>
36	<p><b>What are the implementations of lexical analyzer?</b></p> <p>a) Use a lexical analyzer generator, such as Lex compiler, to produce the lexical analyzer from a regular expression based specification.</p> <p>b) Write the lexical analyzer in a conventional systems- programming language using the I/O facilities of that language to read the input.</p> <p>c) Write the lexical analyzer in assembly language and explicitly manage the reading of input.</p>
37	<p><b>Define the length of a string?</b></p> <p>It is the number of occurrences of symbols in string, "s" denoted by  s . Example: s=abc,  s =3.</p>
38	<p><b>Define regular set?</b></p> <p>A language denoted by a regular expression is said to be a regular set.</p>
39	<p><b>Define character class with example.</b></p> <p>The notation [abc] where a, b, c are alphabet symbols denotes the regular expression a/b/c.</p> <p><b>Example:</b> [A-z] = a   b   c   ...   z Regular expression for identifiers using character classes [a - z A - Z] [A - Z a - z 0 - 9] *</p>
40	<p><b>Write the R.E. for the set of statements over {a,b,c} that contain no two consecutive b's</b></p> <p><b>Answer: (B/c) (A/c/ab/cb) *</b></p>
41	<p><b>Describe the language denoted by the R.E. (0/1)*0(0/1)(0/1) Answer:</b></p> <p>The set of all strings of 0's and 1's with the third symbol from the right end is 0.</p>
42	<p><b>What are the tasks in lexical analyzer?</b></p> <ul style="list-style-type: none"> <li>One task is stripping out from the source program comments and white space in the form of blank, tab, new</li> </ul>



AMSCE - 1101

line characters.

- Another task is correlating error messages from the compiler with the source program.

43 **Define parser.**

Hierarchical analysis is one in which the tokens are grouped hierarchically into nested collections with collective meaning. Also termed as Parsing.

44 **Write the R.E. for the set of statements over {a,b,c} that contain an even no of a's.**

**Ans:**  $((b/c)^* a (b/c)^* a)^* (b/c)^*$

45 **Describe the language denoted by the following R.E.  $0(0/1)^*0$  Answer:**

The set of all strings of 0's and 1's starting and ending with 0.

46 **Describe the language denoted by the following R.E.**

$(00/11)^*((01/10)(00/11)^*(01/10)(00/11)^*)$

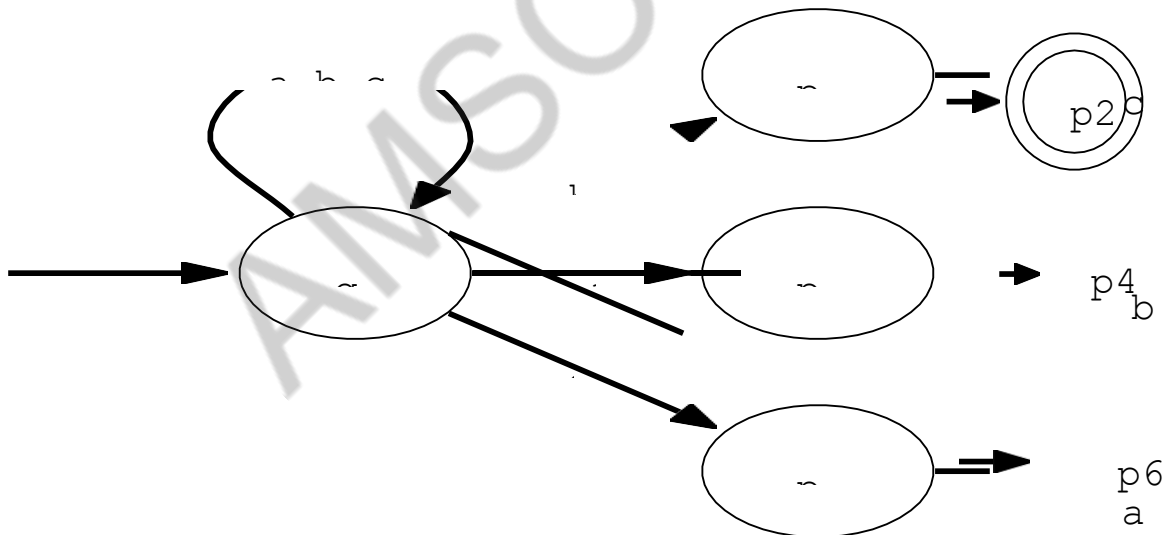
**Answer:**

The set of all strings of 0's and 1's with even number of 0's

47

48 **Draw the NFA for the language of all strings over {a,b,c} that end with one of ab, bc, and ca.**

An NFA for the language of all strings over {a,b,c} that end with one of ab, bc, and ca.



49

50

1 Explain Input Buffering with example. (Page No.88) NOV/DEC 2011

The LA scans the characters of the source pgm one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

3

Explain the specification of tokens. (Page No.92)

MAY/JUNE 2016, MAY/JUNE 2013, APRIL/MAY 2008,

NOV/DEC 2014

### *Specifications of Tokens*

Let us understand how the language theory undertakes the following terms:

#### Alphabets

Any finite set of symbols  $\{0,1\}$  is a set of binary alphabets,  $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$  is a set of Hexadecimal alphabets,  $\{a-z, A-Z\}$  is a set of English language alphabets.

#### Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by  $|\text{tutorialspoint}| = 14$ . A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by  $\epsilon$  (epsilon).

#### Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&,  ,   , !
Shift Operator	>>, >>>, <<, <<<

#### Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer

languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

### *Longest Match Rule*

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

#### **For example:**

```
int int value;
```

While scanning both lexemes till 'int', the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier *int value*.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

4 Elaborate in detail the recognition of tokens. (Page No.98) APRIL/MAY 2012, NOV/DEC 2014

#### **Recognition of tokens:**

We learn how to express pattern using regular expressions. Now, we must study 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.

```
Stmt → if expr then stmt
      | if expr then else stmt
      | e
```

```
Expr → term relop term
      | term
```

```
Term → id
      | number
```

For relop ,we use the comparison operations of languages like Pascal or SQL where = is "equals" and <> is "not equals" because it presents an interesting structure of lexemes.

The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

```
digit      -->[0,9]
digits     -->digit+
number    -->digit(.digit)?(e.[+-]?digits)?
letter     -->[A-Z,a-z]
id         -->letter(letter/digit)*
if         --> if
then       -->then
else       -->else
relop      --></><=>/=/>< >
```

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the “token” we defined by:

$ws \rightarrow (\text{blank/tab/newline})^+$

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

Lexeme	Token Name	Attribute Value
Any ws	$\bar{\quad}$	—
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	ET
<>	relop	NE

## 2.1 TRANSITION DIAGRAM:

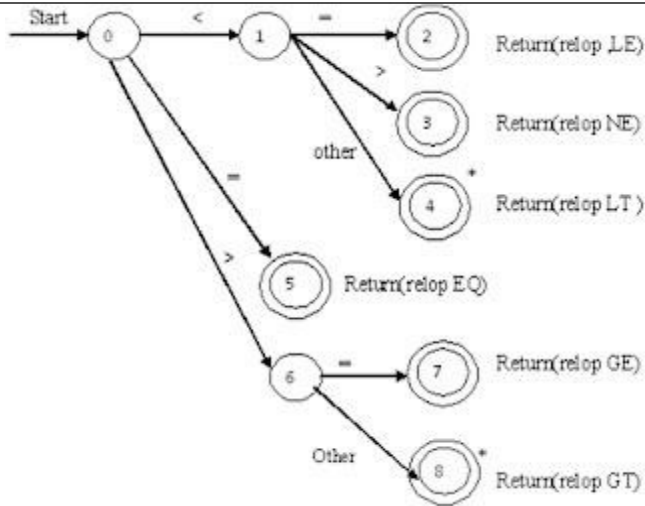
Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns .

Edges are directed from one state of the transition diagram to another. each edge is labeled by a symbol or set of symbols.

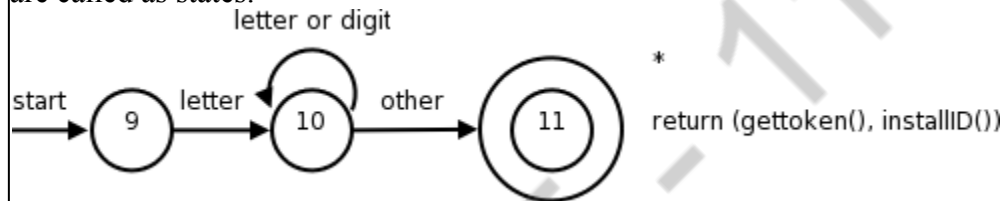
If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

### Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.
2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a \* near that accepting state.
3. One state is designed the state ,or initial state ., it is indicated by an edge labeled “start” entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.



As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.



The above TD for an identifier, defined to be a letter followed by any no of letters or digits. A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code.

```

If      =   if
Then    =   then
Else    =   else
Relop   =   < | <= | = | > | >=
Id      =   letter (letter | digit) *|
Num     = digit |
  
```

5 Write an algorithm to convert NFA to DFA and minimize DFA. Give an example. NOV/DEC 2017  
(Page No.140)

*Non Deterministic Finite Automata to Deterministic Finite Automata (NFA to DFA conversion) :*

The goal is to construct a Deterministic Finite Automata (DFA) from given Non-Deterministic Finite Automata (NFA) machine which is much faster in recognition an input string. The main idea behind this conversion to construct a DFA machine in which each state will corresponds to a set of NFA states.

**Worst Case complexity of (NFA → DFA) conversion:**

The worst case complexity of NFA to DFA conversion is  $O(2^N)$  of sets of states. The DFA can have exponentially many more states than the NFA in rare cases.

*NFA to DFA conversion Algorithm:*

**Input:**

A NFA                       $S = \text{States} = \{s_0, s_1, \dots, s_N\} = S_{\text{NFA}}$   
 $\delta = \text{Move function} = \text{Move}_{\text{NFA}}$   
 $\text{Move}^*(S, a) \rightarrow \text{Set of states}$

**Output:**

A DFA                       $S = \text{States} = \{?, ?, \dots, ?\} = S_{\text{DFA}}$   
 $\delta = \text{Move function} = \text{Move}_{\text{DFA}}$   
 $\text{Move}(s, a) \rightarrow \text{Single state from } S_{\text{DFA}}$

6 What are the issues in Lexical analysis? (Page No.84) MAY/JUNE 2016, APRIL/MAY 2012, MAY/JUNE 2013, MAY/JUNE 2014, APRIL/MAY 2017, NOV/DEC 2017

**Issues in Lexical Analysis**

Lexical analysis is the process of producing tokens from the source program. It has the following issues:

- Lookahead
- Ambiguities

**Lookahead**

*Lookahead* is required to decide when one token will end and the next token will begin. The simple example which has lookahead issues are *i vs. if*, *= vs. ==*. Therefore a way to describe the lexemes of each token is required.

A way needed to resolve ambiguities

- Is *if* it is two variables *i* and *f* or *if*?
- Is *==* is two equal signs *=*, *=* or *==*?
- *arr(5, 4)* vs. *fn(5, 4)* // in Ada (as array reference syntax and function call syntax are similar).

Hence, the number of lookahead to be considered and a way to describe the lexemes of each token is also needed.

Regular expressions are one of the most popular ways of representing tokens.

**Ambiguities**

The lexical analysis programs written with lex accept ambiguous specifications and choose the longest match possible at each input point. Lex can handle ambiguous specifications. When more than one expression can match the current input, lex chooses as follows:

- The longest match is preferred.

- Among rules which matched the same number of characters, the rule given first is preferred.

### **Lexical Errors**

- A character sequence that cannot be scanned into any valid token is a lexical error.
- Lexical errors are uncommon, but they still must be handled by a scanner.
- Misspelling of identifiers, keyword, or operators are considered as lexical errors.

Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

### **Error Recovery Schemes**

- Panic mode recovery
- Local correction
  - o Source text is changed around the error point in order to get a correct text.
  - o Analyzer will be restarted with the resultant new text as input.
- Global correction
  - o It is an enhanced panic mode recovery.
  - o Preferred when local correction fails.

### **Panic mode recovery**

In panic mode recovery, unmatched patterns are deleted from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.

(eg.) For instance the string *fi* is encountered for the first time in a C program in the context:

*fi* (*a*== *f*(*x*))

A lexical analyzer cannot tell whether *fi* is a misspelling of the keyword *if* or an undeclared function identifier.

Since *fi* is a valid lexeme for the token **id**, the lexical analyzer will return the token **id** to the parser.

### **Local correction**

Local correction performs deletion/insertion and/or replacement of any number of symbols in the error detection point.

(eg.) In Pascal, *c[i] '='*; the scanner deletes the first quote because it cannot legally follow the closing bracket and the parser replaces the resulting '=' by an assignment statement.

Most of the errors are corrected by local correction.

(eg.) The effects of lexical error recovery might well create a later syntax error, handled by the parser. Consider

· · · **for Stnight** · · ·

The \$ terminates scanning of *for*. Since no valid token begins with \$, it is deleted. Then *tnight* is scanned as an identifier.

In effect it results,

· · · **fortnight** · · ·

Which will cause a syntax error? Such *false errors* are unavoidable, though a syntactic error-repair may help.

### **Lexical error handling approaches**

Lexical errors can be handled by the following actions:



- Deleting one character from the remaining input.
- Inserting a missing character into the remaining input.
- Replacing a character by another character.
- Transposing two adjacent characters.

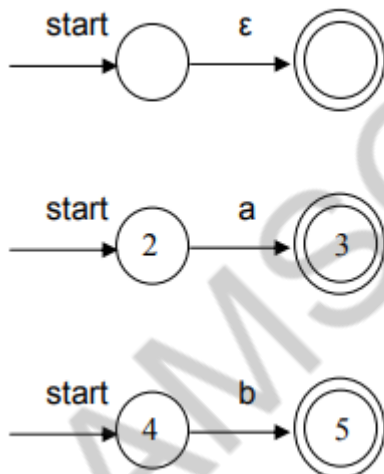
7 (i) Minimize the regular expression  $(a+b)^*abb$ . (NFA toDFA) (Page No.121) MAY/JUNE  
2016,

NOV/DEC 2016

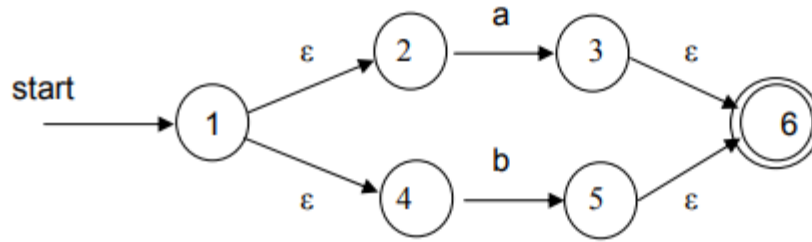
Consider the regular expression  $r = (a|b)^*abb$ , that matches  $\{abb, aabb, babb, aaabb, bbabb, ababb, aababb, \dots\}$ . To construct a NFA from this, use Thompson's construction. This method constructs a regular expression from its components using  $\epsilon$ -transitions. The  $\epsilon$  transitions act as "glue or mortar" for the subcomponent NFA's. An  $\epsilon$ -transition adds nothing since concatenation with the empty string leaves a regular expression unchanged (concatenation with  $\epsilon$  is the identity operation).

Step 1. Parse the regular expression into its subexpressions involving alphabet symbols  $a$  and  $b$  and  $\epsilon$ :  $\epsilon, a, b, a|b, ()^*, ab, abb$ . These describe a. a regular expression for single characters  $\epsilon, a, b$ . b. alternation between  $a$  and  $b$  representing the union of the sets:  $L(a) \cup L(b)$ . c. Kleene star  $()^*$ . d. concatenation of  $a$  and  $b$ :  $ab$ , and also  $abb$ . Subexpressions of these kinds have their own Nondeterministic Finite Automata from which the overall NFA is constructed. Each component NFA has its own start and end accepting states. A Nondeterministic Finite Automata (NFA) has a transition diagram with possibly more than one edge for a symbol (character of the alphabet) that has a start state and an

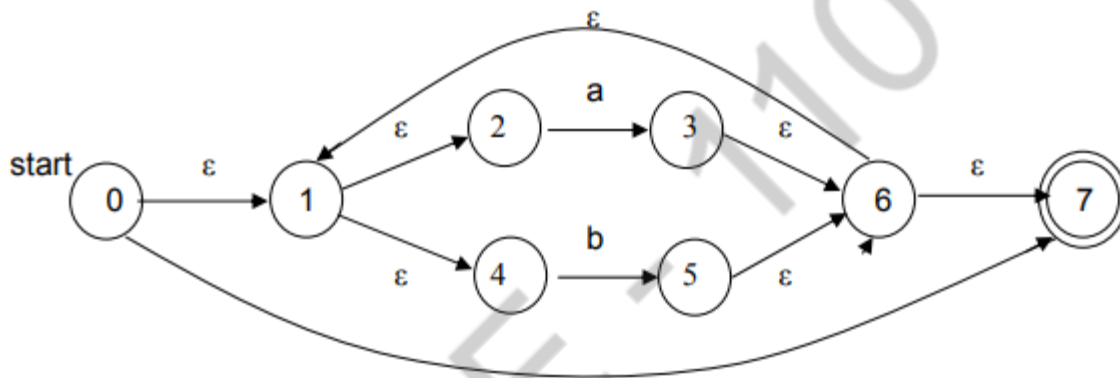
Take these NFA's in turn: a. the NFA's for single character regular expressions  $\epsilon, a, b$



b. the NFA for the union of  $a$  and  $b$ :  $a|b$  is constructed from the individual NFA's using the  $\epsilon$  NFA as "glue". Remove the individual accepting states and replace with the overall accepting state

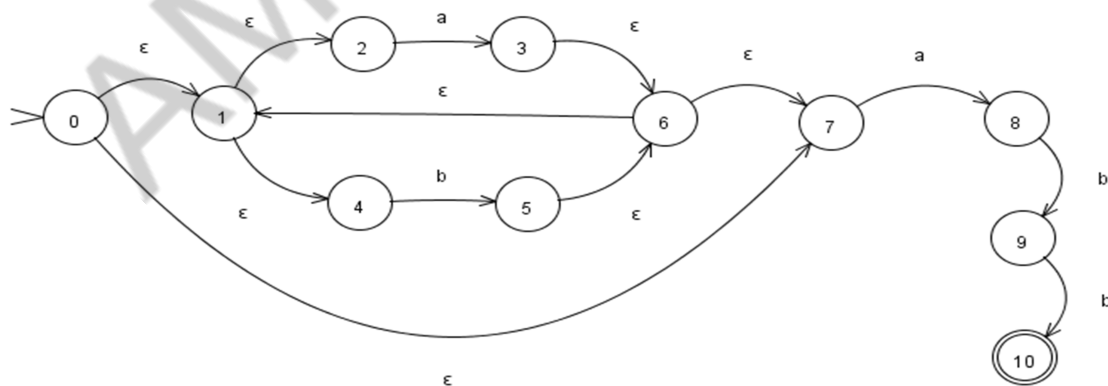


c. Kleene star on  $(a|b)^*$ . The NFA accepts  $\epsilon$  in addition to  $(a|b)^*$



This is the complete NFA. It describes the regular expression  $(a|b)^*abb$ . The problem is that it is not suitable as the basis of a DFA transition table since there are multiple  $\epsilon$  edges leaving many states (0, 1, 6)

(ii) Write an algorithm for minimizing the number of states of a DFA. (Page No.141) NOV/DEC 2016



- $\epsilon$ -closure (0) = {0,1,2,4,7}      Let A
- Move(A,a) = {3,8}

$\epsilon$ -closure (Move(A,a)) = {1,2,3,4,6,7,8}      Let B

$$\text{Move}(A,b) = \{5\}$$

$$\varepsilon - \text{closure}(\text{Move}(A,b)) = \{1,2,4,5,6,7\} \quad \text{Let C}$$

➤  $\text{Move}(B,a) = \{3,8\}$

$$\varepsilon - \text{closure}(\text{Move}(B,a)) = \{1,2,3,4,6,7,8\} \text{ B}$$

$$\text{Move}(B,b) = \{5,9\}$$

$$\varepsilon - \text{closure}(\text{Move}(B,b)) = \{1,2,4,5,6,7,9\} \text{ Let D}$$

➤  $\text{Move}(C,a) = \{3,8\}$

$$\varepsilon - \text{closure}(\text{Move}(C,a)) = \{1,2,3,4,6,7,8\} \text{ B}$$

$$\text{Move}(C,b) = \{5\}$$

$$\varepsilon - \text{closure}(\text{Move}(C,b)) = \{1,2,4,5,6,7\} \text{ C}$$

➤  $\text{Move}(D,a) = \{3,8\}$

$$\varepsilon - \text{closure}(\text{Move}(D,a)) = \{1,2,3,4,6,7,8\} \text{ B}$$

$$\text{Move}(D,b) = \{5,10\}$$

$$\varepsilon - \text{closure}(\text{Move}(D,b)) = \{1,2,4,5,6,7,10\} \quad \text{Let E}$$

➤  $\text{Move}(E,a) = \{3,8\}$

$$\varepsilon - \text{closure}(\text{Move}(E,a)) = \{1,2,3,4,6,7,8\} \text{ B}$$

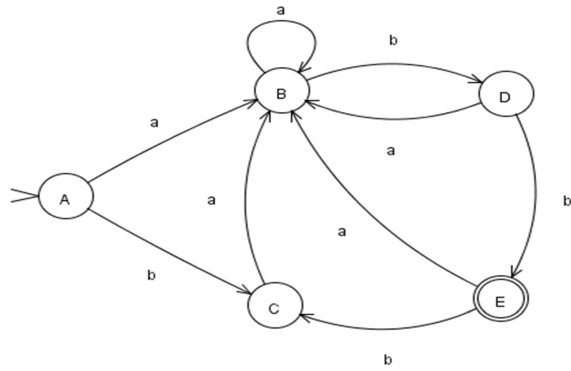
$$\text{Move}(E,b) = \{5\}$$

$$\varepsilon - \text{closure}(\text{Move}(E,b)) = \{1,2,4,5,6,7\} \text{ C}$$

Transition Table:

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

DFA:



8

**What is input buffering? Explain technique of buffer pair.**

MAY/JUNE 2015

The speed of lexical analysis is a concern in compiler design.

1. Buffer pair:

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.

Two pointers to the input are maintained:

1. Pointer *Lexeme Begin*, 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. code to advance forward pointer is given below

*if forward at end of first half then begin*

*reload second half; forward*

*:= forward + 1*

*end*

*else if forward at end of second half then begin*

*reload first half;*

*move forward to beginning of first half*

*end*

*else forward := forward + 1;*

Once the next lexeme is determined, *forward* is set to character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, *Lexeme Begin* is

set to the character immediately after the lexeme just found.

## 2. Sentinels:

If we use the scheme of Buffer pairs 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. Look ahead code with sentinels is given below:

```
forward := forward + 1;
if forward = eof then begin
    if forward at end of first half then begin
        reload second half;
        forward := forward + 1
    end
    else if forward at the second half then begin
        reload first half;
        move forward to beginning of first half
    end
    else terminate lexical analysis
end;
```

9 (i) Differentiate tokens, patterns, lexeme. (Page No.85)

MAY/JUNE 2016, APRIL/MAY 2017

(ii) Write notes on regular expressions. (Page No.94)

### **1) Regular expression & regular definition**

Regular Expression

1

2. 0 or 11 or 111

0+11+111

3. Regular expression over  $\Sigma = \{a,b,c\}$  that represent all string of length

3.  $(a+b+c)(a+b+c)(a+b+c)$

4. String having zero or more a.  
 $a^*$
5. String having one or more a.  
 $a^+$
6. All binary string.  $(0+1)^*$
7. 0 or more occurrence of either a or b or both  
 $(a+b)^*$
8. 1 or more occurrence of either a or b or both  
 $(a+b)^+$
9. Binary no. end with 0  
 $(0+1)^*0$
10. Binary no. end with 1  $(0+1)^*1$
11. Binary no. starts and end with 1.  
 $1(0+1)^*1$
12. String starts and ends with same character.  $0(0+1)^*0$  or  
 $a(a+b)^*a$   
 $1(0+1)^*1$                        $b(a+b)^*b$
13. All string of a and b starting with a  
 $a(a/b)^*$
14. String of 0 and 1 end with 00.  
 $(0+1)^*00$
15. String end with abb.  
 $(a+b)^*abb$
16. String start with 1 and end with 0.  
 $1(0+1)^*0$
17. All binary string with at least 3 characters and 3rd character should be zero.  $(0+1)(0+1)0(0+1)^*$
18. Language which consist of exactly Two b's over the set  
 $\Sigma = \{a, b, \}$   $a^*ba^*ba^*$
19.  $\Sigma = \{a, b, \}$  such that 3rd character from right end of the string is always a.  $(a/b)^*a(a/b)(a/b)$
20. Any no. of a followed by any no. of b followed by any no of c.  $a^*b^*c^*$

21. It should contain at least 3 one.  
 $(0+1)^*1(0+1)^*1(0+1)^*1(0+1)^*$
22. String should contain exactly Two  
 1's  $0^*10^*10^*$
23. Length should be at least be 1 and at most 3.  
 $(0+1)^+(0+1)(0+1)^+(0+1)(0+1)(0+1)$
24. No.of zero should be multiple of 3  
 $(1^*01^*01^*01^*)^*+1^*$
25.  $\Sigma = \{a,b,c\}$  where a are multiple of 3.  
 $((b+c)^*a(b+c)^*a(b+c)^*a(b+c)^*)^*$
26. Even no. of 0.  
 $(1^*01^*01^*)^*$
27. Odd no. of 1.  
 $0^*(10^*10^*)^*10^*$
28. String should have odd length.  
 $(0+1)((0+1)(0+1))^*$
29. String should have even length.  
 $((0+1)(0+1))^*$
30. String start with 0 and has odd length.  
 $0((0+1)(0+1))^*$
31. String start with 1 and has even length.

10

**1) Conversion from Regular Expression to DFA without constructing NFA Example:**

$$a^*(b^*/c^*)(a/c)^*$$

(Page No.121) NOV/DEC 2017

➤ For root node

$$i = \text{lastpos}(C_1) = \{1,2,3,4,5\}$$

$$\text{followpos}(i) = \text{firstpos}(C_2) = \{6\}$$

$$\text{followpos}(1) = \{6\}$$

$$\text{followpos}(2) = \{6\}$$

$$\text{followpos}(3) = \{6\}$$

$$\text{followpos}(4) = \{6\}$$

$$\text{followpos}(5) = \{6\}$$

$$i = \text{lastpos}(C_1) = \{1,2,3\}$$

$$\text{followpos}(i) = \text{firstpos}(C_2) = \{4,5\}$$

$$\text{followpos}(1) = \{4,5\}$$

$$\text{followpos}(2) = \{4,5\}$$

$$\text{followpos}(3) = \{4,5\} \quad i$$

$$= \text{lastpos}(n) = \{4,5\}$$

$$\text{followpos}(i) = \text{firstpos}(n) = \{4,5\}$$

$$\text{followpos}(4) = \{4,5\}$$

$$\text{followpos}(5) = \{4,5\}$$

$$i = \text{lastpos}(C_1) = \{1\}$$

$$\text{followpos}(i) = \text{firstpos}(C_2) = \{2,3\}$$

$$\text{followpos}(1) = \{2,3\}$$

$$i = \text{lastpos}(n) = \{2\}$$

$$\text{followpos}(i) = \text{firstpos}(n) = \{2\}$$

$$\text{followpos}(2) = \{2\}$$

$$i = \text{lastpos}(n) = \{3\}$$

$$\text{followpos}(i) = \text{firstpos}(n) = \{3\}$$

$$\text{followpos}(3) = \{3\}$$

i	Followpos(i)
1	{1,2,3,4,5,6}
2	{2,4,5,6}
3	{3,4,5,6}
4	{4,5,6}
5	{4,5,6}

Construct DFA:

Initial node = firstpos (root node)

$$= \{1,2,3,4,5,6\}$$

$\delta((1,2,3,4,5,6), a) = \text{followpos}(1) \cup \text{followpos}(4)$

$$= \{1,2,3,4,5,6\}$$

$\delta((1,2,3,4,5,6), b) = \text{followpos}(2)$

$$= \{2,4,5,6\}$$



$$\delta((1,2,3,4,5,6), c) = \text{followpos}(3) \cup \text{followpos}(5)$$

$$= \{3,4,5,6\}$$

$$\delta((2,4,5,6), a) = \text{followpos}(4)$$

$$= \{4,5,6\}$$

$$\delta((2,4,5,6), b) = \text{followpos}(2)$$

$$= \{2,4,5,6\}$$

$$\delta((2,4,5,6), c) = \text{followpos}(5)$$

$$= \{4,5,6\}$$

$$\delta((3,4,5,6), a) = \text{followpos}(4)$$

$$= \{4,5,6\} \delta$$

$$((3,4,5,6), b) = \Phi$$

$$\delta((3,4,5,6), c) = \text{followpos}(3) \cup \text{followpos}(5)$$

$$= \{3,4,5,6\}$$

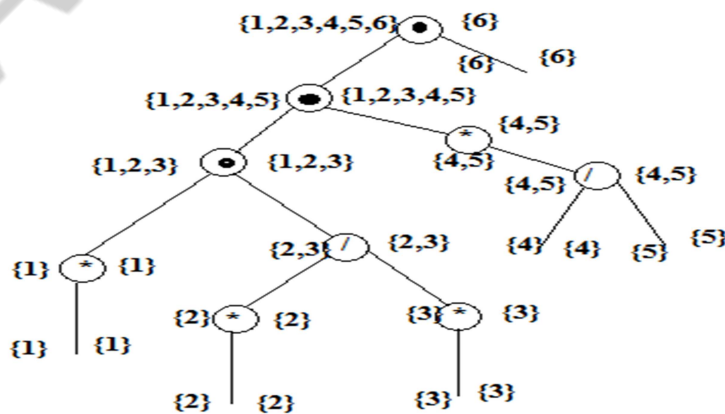
$$\delta((4,5,6), a) = \text{followpos}(4)$$

$$= \{4,5,6\} \delta$$

$$((4,5,6), b) = \Phi$$

$$\delta((4,5,6), c) = \text{followpos}(5)$$

$$= \{4,5,6\}$$



11

2 Write an algorithm for constructing a DFA from a regular expression. Discuss with an example. (Page No.179)

### 2.1 DETERMINISTIC AUTOMATA

A deterministic finite automata has at most one transition from each state on any input. A DFA is a special case of a NFA in which:-

1, it has no transitions on input  $\epsilon$ ,

each input symbol has at most one transition from any state.

DFA formally defined by 5 tuple notation  $M = (Q, \Sigma, \delta, q_0, F)$ , where  $Q$  is a finite 'set of states', which is non empty.

$\Sigma$  is 'input alphabets', indicates input set.

$q_0$  is an 'initial state' and  $q_0$  is in  $Q$  ie,  $q_0 \in \Sigma$ ,

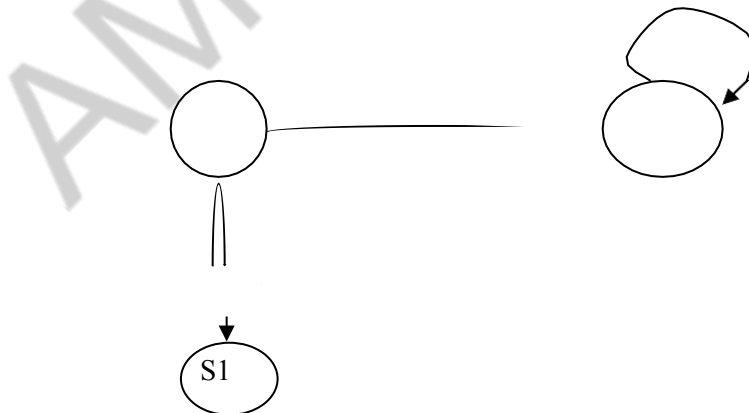
$F$  is a set of 'Final states',

$\delta$  is a 'transmission function' or mapping function, using this function the next state can be determined.

The regular expression is converted into minimized DFA by the following procedure:

**Regular expression  $\rightarrow$  NFA  $\rightarrow$  DFA  $\rightarrow$  Minimized DFA**

The Finite Automata is called DFA if there is only one path for a specific input from current state to next state.



13 i).Solve the following regular expression into minimized DFA.  $(a/b)^*baa$  (Page No.180)  
(ii).Comparison between NFA and DFA

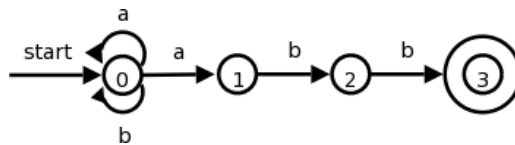
A NFA is a mathematical model that consists of

- A set of states  $S$ .
- A set of input symbols  $\Sigma$ .
- A transition for move from one state to an other.
- A state so that is distinguished as the start (or initial) state.
- A set of states  $F$  distinguished as accepting (or final) state.
- A number of transition to a single symbol.

A NFA can be diagrammatically represented by a labeled directed graph, called a transition graph, In which the nodes are the states and the labeled edges represent the transition function.

This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labeled by the special symbol  $\epsilon$  as well as by input symbols.

The transition graph for an NFA that recognizes the language  $(a | b)^* abb$  is shown



Parameter	NFA	FA
Transition	Non deterministic	Deterministic
No. of states	NFA has a fewer number of states.	More, if NFA contains $Q$ states then the corresponding FA will have $\leq 2^Q$ states.
Power	NFA is as powerful as DFA	FA is powerful as an NFA
Design	Easy to design due to non-determinism	More difficult to design
Acceptance	It is difficult to find whether $w \in L$ as there are several paths. Backtracking is required to explore several parallel paths.	It is easy to find whether $w \in L$ as transition are deterministic.

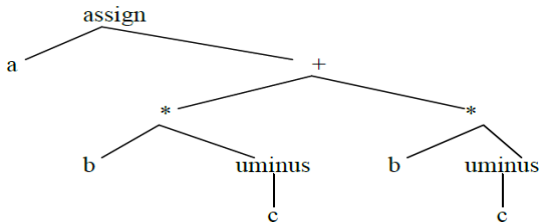
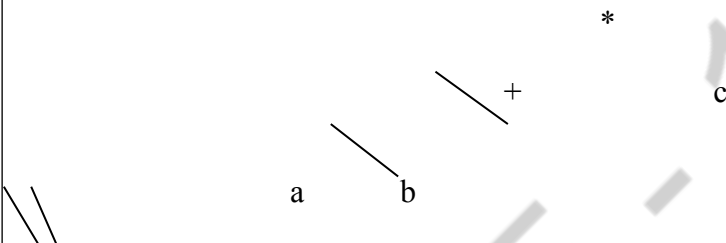
## UNIT II SYNTAX ANALYSIS

Role of Parser – Grammars – Error Handling – Context-free grammars – Writing a grammar – Top Down Parsing - General Strategies Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item Construction of SLR Parsing Table -Introduction to LALR Parser - Error Handling and Recovery in Syntax Analyzer- YACC.

S. No.	Question
1	<p><b>Differentiate Top Down Parser And Bottom Up Parser? Give Example for each. <u>APRIL/MAY 2010</u></b></p> <p><b>Top down Parser</b> are the parsers which constructs the parse tree from the root to the leaves in pre- order for the given input string. Predictive Parser, Recursive Descendent Parser.</p> <p><b>Bottom Up Parser</b> are the parsers which constructs the parse tree from the leaves to the root for the given input string. LR Parser, SLR Parser.</p>
2	<p><b>Compare syntax tree and parse tree. <u>NOV/DEC 2017</u></b></p> <ul style="list-style-type: none"><li>• Syntax tree is a variant of a parse tree in which each leaf represents an operand and each interior node represents an operator.</li><li>• A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order. Each interior node of a parse tree is labeled by some nonterminal A and that the children of the node are labeled from left to right by symbols in the right side of the production by which this A was replaced in the derivation. The leaves of the parse tree are terminal symbols.</li></ul>
3	<p><b>Define Handles. <u>MAY/JUNE 2007</u></b></p> <p>A handle of a string is a substring that matches the right side of a production. This reduction helps in constructing the parse tree or right most derivation.</p>
4	<p><b>Define ambiguous grammar with an example, and specify it demerits. <u>MAY/JUNE 2016</u> <u>MAY/JUNE 2012, APRIL/MAY 2012</u></b></p>

	<p>If a grammar produces more than one parse tree for the given input string then it is called ambiguous grammar. <b>Its demerit is</b> It is difficult to select or determine which parse tree is suitable for an input string.</p> <ul style="list-style-type: none"> <li>• <b>Ex:</b> E E+E / E*E / id</li> </ul>
5	<p><b>Mention the properties of parse tree.</b> <u>NOV/DEC 2012</u></p> <ul style="list-style-type: none"> <li>• The root is labeled by the start symbol.</li> <li>• Each leaf is labeled by a token or by <math>\epsilon</math></li> <li>• Each interior node is labeled by a non terminal</li> <li>• If A is the Non terminal, labeling some interior node and <math>x_1, x_2, x_3 \dots x_n</math> are the labels of the children.</li> </ul>
6	<p><b>What do you mean by a syntax tree?</b> <u>NOV/DEC 2012</u></p> <p>Syntax tree is a variant of a parse tree in which each leaf represents an operand and each interior node represents an operator.</p>
7	<p><b>Define Handle pruning.</b><u>NOV/DEC2011, APRIL/MAY 2011, NOV/DEC 2016</u> <u>APRIL/MAY 2018</u></p> <p>A technique to obtain the rightmost derivation in reverse (called canonical reduction sequence) is known as handle pruning (i.e.) starting with a string of terminals <math>w</math> to be parsed. If <math>w</math> is the sentence of the grammar then <math>\alpha = \alpha_n</math> where <math>\alpha_n</math> is the <math>n</math>th right sentential form of unknown right most derivation.</p>
8	<p><b>How will you define a context free grammar?</b></p> <p>A context free grammar consists of terminals, non-terminals, a start symbol, and productions.</p> <ol style="list-style-type: none"> <li>Terminals are the basic symbols from which strings are formed. "Token" is a synonym for terminal. Ex: <b>if, then, else.</b></li> <li>Nonterminals are syntactic variables that denote sets of strings, which help define the language generated by the grammar. Ex: stmt, expr.</li> <li>Start symbol is one of the nonterminals in a grammar and the set of strings it denotes is the language defined by the grammar. Ex: S.</li> <li>The productions of a grammar specify the manner in</li> </ol>

	which the terminals and non-terminals can be combined to form strings Ex: expr-> id	
9	<b>Differentiate sentence and sentential form.</b>	
	<b>Sentence</b>	<b>Sentential form</b>
	If $S \Rightarrow w$ then the string $w$ is called Sentence of $G$ .	If $S \Rightarrow a$ then $a$ is a sentential form of $G$ .
	Sentence is a string of terminals. Sentence is a sentential form with no nonterminals.	Sentential form may contain non terminals
10	<b>What is left factoring? Give an example. <u>NOV/DEC 2007</u></b>	
	Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.	
11	<b>Derive the string and construct a syntax tree for the input string ceadae using the grammar <math>S \rightarrow SaA A, A \rightarrow AbB B, B \rightarrow cSd e</math> <u>MAY/JUNE 2009</u></b>	
	$S \rightarrow SaA$ $S \rightarrow AaA$ $S \rightarrow cSdaA$ $S \rightarrow cSaAdaA$ $S \rightarrow cAaAdaA$ $S \rightarrow cBaAdaA$ $S \rightarrow ceaBdaA$ $S \rightarrow ceaedaB$ $C \rightarrow ceaedae$	
12	<b>List the factors to be considered for top-down parsing. <u>MAY/JUNE 2009</u></b>	
	<p>We begin with the start symbol and at each step, expand one of the remaining non-terminals by replacing it with the right side of one of its productions. We repeat until only terminals remain.</p> <p>The top-down parse produces a leftmost derivation of the</p>	

	sentence
13	<p><b>Draw syntax tree for the expression <math>a=b*-c+b*-c</math>. <u>NOV/DEC 2017</u></b></p> 
14	<p><b>Construct a parse tree of <math>(a+b)*c</math> for the grammar <math>E \rightarrow E+E/E*(E)/id</math>. (or) grammar <math>-(id+id)</math> <u>APRIL/MAY 2008, NOV/DEC 2016</u></b></p> 
15	<p><b>Eliminate Left Recursion for the grammar <math>A \rightarrow Ac Aad bd</math> <u>APRIL/MAY 2017</u></b></p> <p><math>A \rightarrow bd A'</math></p> <p><math>A' \rightarrow c A' ad A'  \epsilon</math></p>
16	<p><b>What are the various conflicts that occur during shift reduce parsing? <u>APRIL/MAY 2017</u></b></p> <p>Reduce/Reduce conflict</p> <p>Shift/ Reduce conflict</p>
17	<p><b>Eliminate Left Recursion for the given grammar. <u>MAY/JUNE 2007</u></b></p> <p><math>E \rightarrow E + T   T</math></p> <p><math>T \rightarrow T * F   F</math></p> <p><math>F \rightarrow ( E )   id</math></p> <p><math>E \rightarrow TE'</math></p>

	$E' \rightarrow +TE' \mid \epsilon T \rightarrow FT'$ $T' \rightarrow *FT' \mid \epsilon$ $F \rightarrow (E) \mid id$
18	<p><b>Write the algorithm for FIRST and FOLLOW in parser.</b>  <u>MAY/JUNE 2016</u></p> <p>FIRST(<math>\alpha</math>) is the set of terminals that begin strings derived from <math>\alpha</math>.</p> <p>Rules</p> <p>To compute FIRST(X), where X is a grammar symbol</p> <ul style="list-style-type: none"> <li>• If X is a terminal, then FIRST(X)={X}</li> <li>• If <math>X \rightarrow \epsilon</math> is a production, then add <math>\epsilon</math> to FIRST(X)</li> <li>• If X is a non terminal and <math>X \rightarrow Y_1 Y_2 \dots Y_k</math> is a production. Then add FIRST(<math>Y_1</math>) to FIRST (X). If <math>Y_1</math> derives <math>\epsilon</math>. Then add FIRST(<math>Y_2</math>) to FIRST(X)</li> </ul> <p>FOLLOW (A) is the set of terminals <math>\alpha</math> that appear immediately to the right of A. For rightmost sentential form of A, \$ will be in FOLLOW (A).</p> <p><b>Rules</b></p> <ul style="list-style-type: none"> <li>• If \$ is the input end-marker, and S is the start symbol,  <math>\\$ \in FOLLOW(S)</math>.</li> <li>• If there is a production, <math>A \rightarrow \alpha B \beta</math>, then <math>(FIRST(\beta) - \epsilon) \subseteq FOLLOW(B)</math>.</li> <li>• If there is a production, <math>A \rightarrow \alpha B</math>, or a production <math>A \rightarrow \alpha B \beta</math>, where <math>\epsilon \in FIRST(\beta)</math>, then <math>FOLLOW(A) \subseteq FOLLOW(B)</math>.</li> </ul>
19	<p><b>What is dangling reference?</b></p>



	<p><b><u>MAY/JUNE 2012, APRIL/MAY 2012</u></b></p> <p>A dangling reference occurs when there is a reference to storage that has been deallocated. It is a logical error to use dangling references, since the value of deallocated storage is undefined according to the semantics of most languages.</p>
20	<p><b>Write the rule to eliminate left recursion in a grammar.</b></p> <p><b><u>NOV/DEC 2012</u></b></p> <p><math>A \rightarrow A\alpha \beta : A \rightarrow \beta A' ; A' \rightarrow \alpha A' \epsilon</math></p>
21	<p><b>Mention the role of semantic analysis.</b></p> <p><b><u>NOV/DEC 2012</u></b></p> <p>It is used to check the type information of the syntactically verified statements.</p>
22	<p><b>What is the output of syntax analysis phase? What are the three general types of parsers for grammars?</b></p> <p>Parser (or) parse tree is the output of syntax analysis phase General types of parsers:</p> <ol style="list-style-type: none"> <li>1) Universal parsing</li> <li>2) Top-down</li> <li>3) Bottom-up</li> </ol>
23	<p><b>What are the different strategies that a parser can employ to recover from a syntactic error?</b></p> <ul style="list-style-type: none"> <li>• Panic mode</li> <li>• Phrase level</li> <li>• Error productions</li> <li>• Global correction</li> </ul>
24	<p><b>What are the goals of error handler in a parser?</b></p> <p>The error handler in a parser has simple-to-state goals:</p> <ul style="list-style-type: none"> <li>• It should report the presence of errors clearly and accurately</li> </ul>

	<ul style="list-style-type: none"> <li>• It should recover from each error quickly enough to be able to detect subsequent errors.</li> <li>• It should not significantly slow down the processing of correct programs.</li> </ul>
25	<p><b>What is phrase level error recovery?</b></p> <p>On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. This is known as phrase level error recovery.</p>
26	<p><b>Define context free language. When will you say that two CFGs are equal?</b></p> <ul style="list-style-type: none"> <li>• A language that can be generated by a grammar is said to be a context free language.</li> <li>• If two grammars generate the same language, the grammars are said to be equivalent.</li> </ul>
27	<p><b>Give the definition for leftmost and canonical derivations.</b></p> <ul style="list-style-type: none"> <li>• Derivations in which only the leftmost nonterminal in any sentential form is replaced at each step are termed leftmost derivations</li> <li>• Derivations in which the rightmost nonterminal is replaced at each step are termed canonical derivations.</li> </ul>
28	<p><b>What is a parse tree?</b></p> <p>A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order. Each interior node of a parse tree is labeled by some nonterminal A and that the children of the node are labeled from left to right by symbols in the right side of the production by which this A was replaced in the derivation. The leaves of the parse tree are terminal symbols.</p>
29	<p><b>Why do we use regular expressions to define the lexical syntax of a language?</b></p> <p>i. The lexical rules of a language are frequently quite simple,</p>

	<p>and to describe them we do not need a notation as powerful as grammars.</p> <p>ii. Regular expressions generally provide a more concise and easier to understand notation for tokens than grammars.</p> <p>iii. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars</p> <p>iv. Separating the syntactic structure of a language into lexical and non lexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.</p>
30	<p><b>When will you call a grammar as the left recursive one?</b></p> <p>A grammar is a left recursive if it has a nonterminal A such that there is a derivation <math>A \Rightarrow A\alpha</math> for some string <math>\alpha</math>.</p>
31	<p><b>Define left factoring.</b></p> <p>Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal “A”, we may be able to rewrite the “A” productions to refer the decision until we have seen enough of the input to make the right choice.</p>
32	<p><b>Left factor the following grammar: <math>S \rightarrow iEtS \mid iEtSeS \mid aE</math></b>  <math>\rightarrow b</math>.</p> <p>Ans: The left factored grammar is, <math>S \rightarrow iEtSS' \mid a</math></p> <p><math>S' \rightarrow eS \mid \epsilon</math></p> <p><math>E \rightarrow b</math></p>
33	<p><b>Why SLR and LALR are more economical to construct than canonical LR?</b></p> <p>For a comparison of parser size, the SLR and LALR tables for a grammar always have the same number of states, and this number is typically several hundred states for a language like Pascal. The canonical LR table would typically</p>

	<p>have several thousand states for the same size language. Thus, it is much easier and more economical to construct SLR and LALR tables than the canonical LR tables.</p>
34	<p><b>Write the configuration of an LR parser?</b>  A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpanded  input: <math>(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \\$)</math></p>
35	<p><b>What is meant by goto function in LR parser? Give an example</b></p> <ul style="list-style-type: none"> <li>• The function goto takes a state and grammar symbol as arguments and produces a state</li> <li>• The goto function of a parsing table constructed from a grammar G is the transition function of a DFA that recognizes the viable prefixes of G.</li> </ul> <p>Ex: <math>\text{goto}(I, X)</math> Where I is a set of items and X is a grammar symbol to be the closure of the set of all items <math>[A \rightarrow \alpha X \beta]</math> such that <math>[A \rightarrow \alpha \cdot X \beta]</math> is in I</p>
36	<p><b>LR (k) parsing stands for what?</b>  The “L” is for left-to-right scanning of the input, the “R” for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions.</p>
37	<p><b>What do you mean by viable prefixes?</b></p> <ul style="list-style-type: none"> <li>• The set of prefixes of right sentential forms that can appear on the stack of a shiftreduce parser are called viable prefixes.</li> <li>• A viable prefix is that it is a prefix of a right sentential form that does not continue the past the right end of the rightmost handle of that sentential form.</li> </ul>
38	<p><b>What is meant by Predictive parsing? Nov/Dec 2007</b>  A special form of Recursive Descent parsing, in which the look-ahead symbol unambiguously determines the procedure selected for each nonterminal, where no backtracking is required.</p>
39	<p><b>Write the rule to eliminate left recursion in a grammar.</b></p>

	<p><b>Prepare and Eliminate the left recursion for the grammar <math>S \rightarrow Aa \mid b</math></b>  <math>A \rightarrow Ac \mid Sd \mid \epsilon</math> Ans:  Rules <math>\rightarrow A \rightarrow Aa \mid \beta : A \rightarrow \beta A' ; A' \rightarrow \alpha A' \mid \epsilon</math> ILR <math>\rightarrow S \rightarrow Aa \mid b</math>  <math>A \rightarrow SdA' \mid A' A' \rightarrow cA' \mid \epsilon</math></p>
40	<p><b>Define a context free grammar.</b>  A context free grammar G is a collection of the following V is a set of non terminals  T is a set of terminals S is a start symbol  P is a set of production rules  G can be represented as <math>G = (V, T, S, P)</math> Production rules are given in the following form  Non terminal <math>\rightarrow (V \cup T)^*</math></p>
41	<p><b>Define ambiguous grammar.</b>  A grammar G is said to be ambiguous if it generates more than one parse tree for some sentence of language <math>L(G)</math>.</p>
42	<p><b>List the properties of LR parser.</b></p> <ol style="list-style-type: none"> <li>1. LR parsers can be constructed to recognize most of the programming languages for which the context free grammar can be written.</li> <li>2. The class of grammar that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsers.</li> <li>3. LR parsers work using non backtracking shift reduce technique yet it is efficient one.</li> </ol>
43	<p><b>Mention the types of LR parser.</b> SLR parser- simple LR parser  LALR parser- lookahead LR parser  Canonical LR parser</p>
44	<p><b>What are the problems with top down parsing?</b>  The following are the problems associated with top down parsing:  Backtracking  Left recursion  Left factoring  Ambiguity</p>
45	<p><b>Write short notes on YACC.</b>  YACC is an automatic tool for generating the parser program. YACC stands for Yet Another Compiler Compiler which is basically the utility available from UNIX. Basically YACC is LALR parser generator. It can report conflict or ambiguities in the form of error messages.</p>
46	<p><b>Define LR(0) items.</b>  An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. Thus, production <math>A \rightarrow XYZ</math> yields the four items <math>A \rightarrow \cdot XYZ</math>  <math>A \rightarrow X \cdot YZ</math> <math>A \rightarrow XY \cdot Z</math>  <math>A \rightarrow XYZ \cdot</math></p>
47	<p><b>What are kernel &amp; non-kernel items?</b>  <b>Kernel items</b>, which include the initial item, <math>S' \rightarrow \cdot S</math>, and all items whose dots are not</p>

at the left end.

	<b>Non-kernel items, which have their dots at the left end.</b>
48	<b>Solve the following grammar is ambiguous: <math>S \rightarrow aSbS / bSaS / \epsilon</math></b> <b>LMD 1: <math>S \Rightarrow aSbS</math></b> <b><math>\Rightarrow abSaSbS</math></b> <b><math>\Rightarrow abaSbS</math></b> <b><math>\Rightarrow ababS</math></b> <b><math>\Rightarrow abab</math> LMD 2: <math>S \Rightarrow aSbS</math></b> <b><math>\Rightarrow abS</math></b> <b><math>\Rightarrow abaSbS</math></b> <b><math>\Rightarrow ababS</math></b> <b><math>\Rightarrow abab</math></b>
49	<b>Define sentential form?</b> If $G = (V, T, P, S)$ is a CFG, then any string " $\alpha$ " in $(VUT)^*$ such that $S \rightarrow^* \alpha$ is a sentential form.
50	<b>Define yield of the string?</b> A string that is derived from the root variable is called the yield of the tree.
51	<b>Summarize the merits and demerits of LALR parser. APRIL/MAY 2018</b> <ul style="list-style-type: none"><li>• This is the extension of LR(O) items, by introducing the one symbol of lookahead on the input.</li><li>• It supports large class of grammars.</li><li>• The number of states is LALR parser is lesser than that of LR( 1) parser. Hence, LALR is preferable as it can be used with reduced memory.</li><li>• Most syntactic constructs of programming language can be stated conveniently.</li></ul>
52	Draw the activation tree for the following code. <b>APRIL/MAY 2018</b> <pre>int main() {     printf("Enter Your Name"); scanf("%s",username);     int show_data(username);     printf("Press Any Key to Continue...");     ....     int show_data(char *user)     {         printf("Your Name is %s", username); return 0;     } }</pre>

}

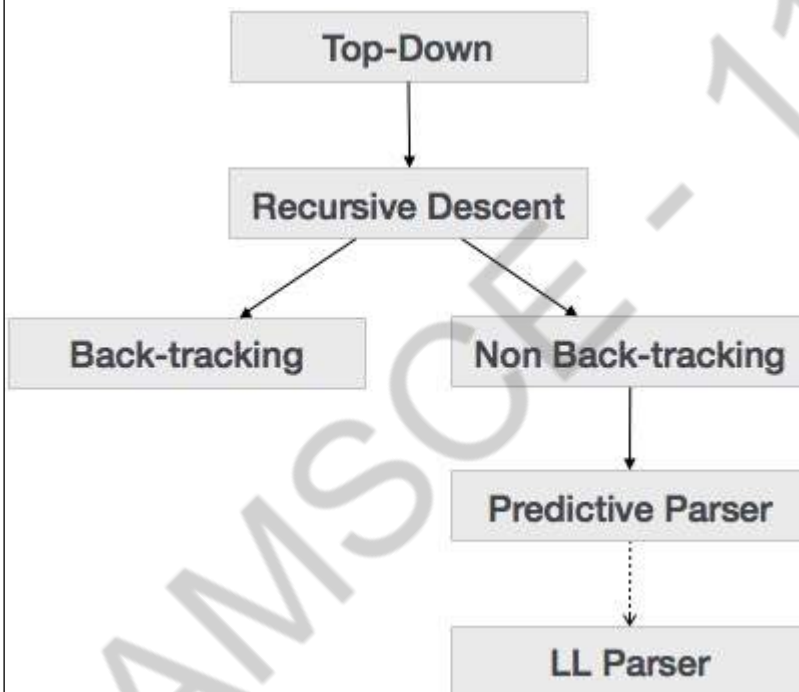
}

**REFER NOTES**

**PART B**

- 1 (i) Explain Top-Down parsing and Bottom up Parsing. (Page No. 181&195)  
MAY/JUNE 2007

The top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. The types of top-down parsing are depicted below:



**Recursive Descent Parsing**

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

## Back-tracking

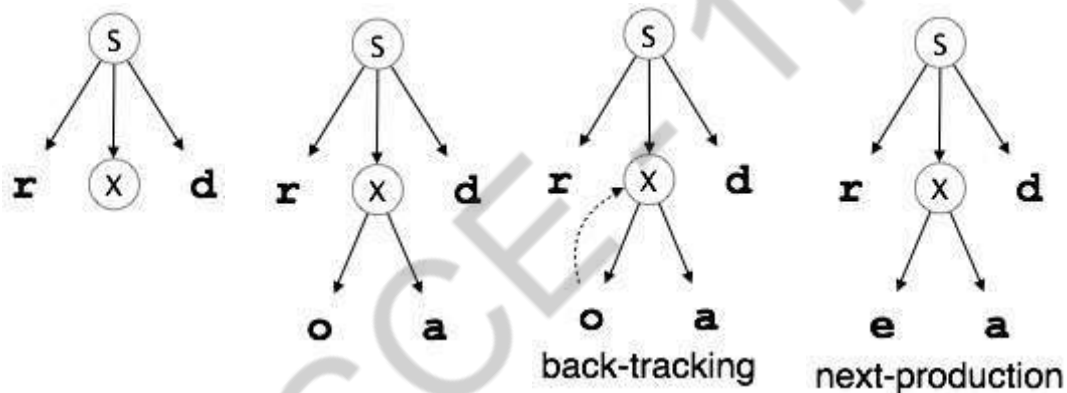
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

$S \rightarrow rXd \mid rZd$   
 $X \rightarrow oa \mid ea$   
 $Z \rightarrow ai$

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S ( $S \rightarrow rXd$ ) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left ( $X \rightarrow oa$ ). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, ( $X \rightarrow ea$ ).

Now the parser matches all the input letters in an ordered manner. The string is accepted.

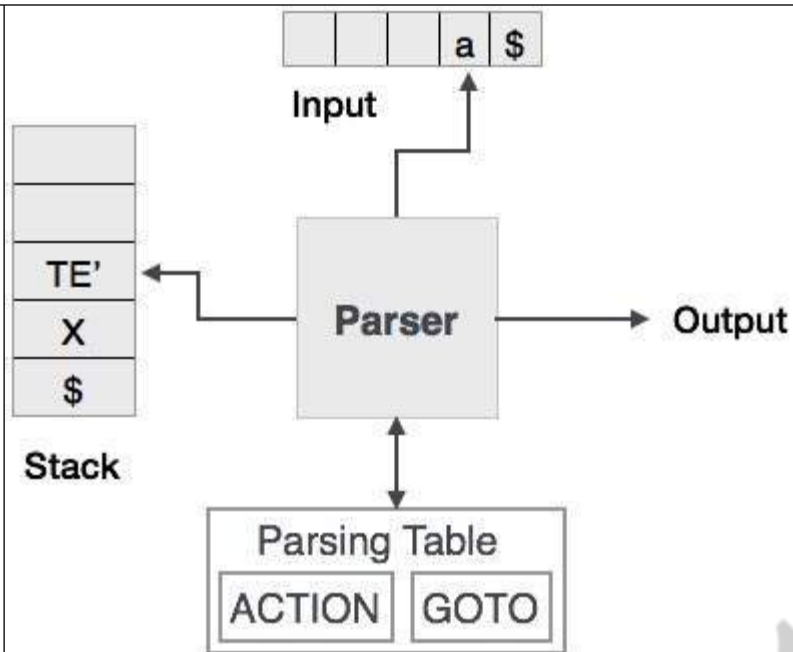


## Predictive Parser

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

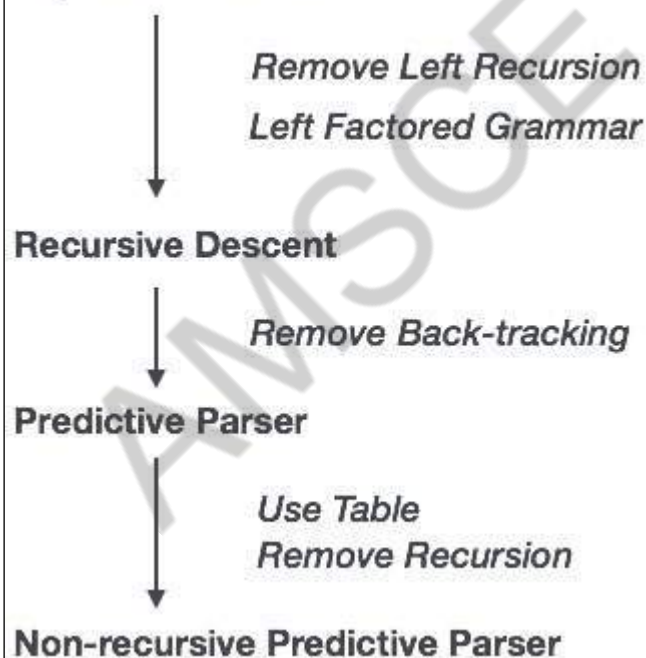
To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.





Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contain an end symbol \$ to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

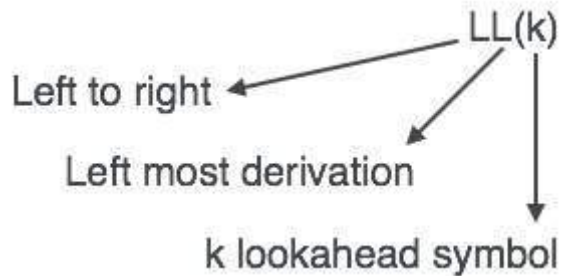
### Top-Bottom Parser



In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally  $k = 1$ , so LL(k) may also be written as LL(1).



### LL Parsing Algorithm

We may stick to deterministic LL(1) for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not LL(1), then usually, it is not LL(k), for any given k.

Given below is an algorithm for LL(1) Parsing:

**Input:**

```
string  $\omega$ 
parsing table M for grammar G
```

**Output:**

```
If  $\omega$  is in L(G) then left-most derivation of  $\omega$ ,
error otherwise.
```

**Initial State :**  $\$S$  on stack (with S being start symbol)

```
 $\omega\$$  in the input buffer
```

SET ip to point the first symbol of  $\omega\$$ .

repeat

```
let X be the top stack symbol and a the symbol pointed by ip.
```

```
if  $X \in V_t$  or  $\$$ 
```

```
if  $X = a$ 
```

```
POP X and advance ip.
```

```
else
```

```
error()
```

```
endif
```

```
else /* X is non-terminal */
```

```
if  $M[X, a] = X \rightarrow Y_1, Y_2, \dots, Y_k$ 
```

```
POP X
```

```
PUSH  $Y_k, Y_{k-1}, \dots, Y_1$  /*  $Y_1$  on top */
```

```

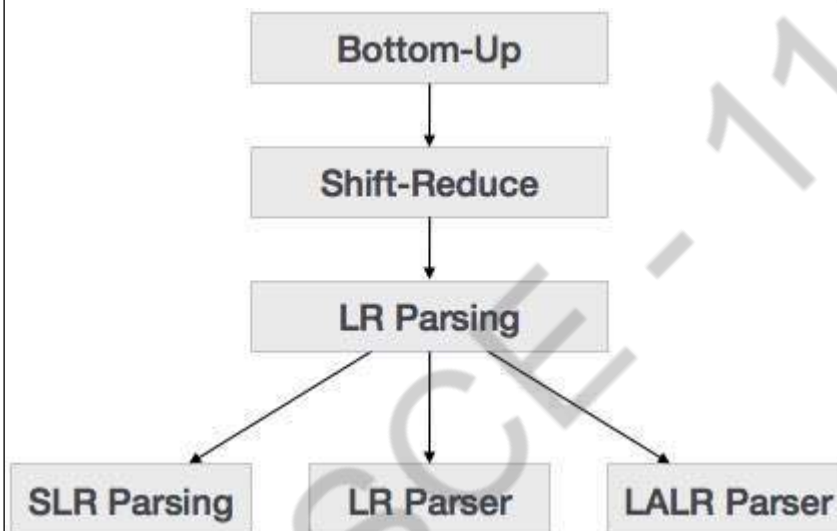
        Output the production  $X \rightarrow Y_1, Y_2, \dots, Y_k$ 
    else
        error()
    endif
endif
until X = $ /* empty stack */

```

A grammar  $G$  is LL(1) if  $A \rightarrow \alpha \mid \beta$  are two distinct productions of  $G$ :

- for no terminal, both  $\alpha$  and  $\beta$  derive strings beginning with  $a$ .
- at most one of  $\alpha$  and  $\beta$  can derive empty string.
- if  $\beta \rightarrow t$ , then  $\alpha$  does not derive any string beginning with a terminal in  $\text{FOLLOW}(A)$ .

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.



### Shift-Reduce Parsing

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step:** The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
- **Reduce step:** When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

### 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

AMSCCE - 1101

**LL**

Does a leftmost derivation.

Starts with the root nonterminal on the stack.

Ends when the stack is empty.

**LR**

Does a rightmost derivation in reverse.

Ends with the root nonterminal on the stack.

Starts with an empty stack.

	<p>Uses the stack for designating what is still to be expected.      Uses the stack for designating what is already seen.</p> <p>Builds the parse tree top-down.      Builds the parse tree bottom-up.</p> <p>Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.      Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.</p> <p>Expands the non-terminals.      Reduces the non-terminals.</p> <p>Reads the terminals when it pops one off the stack.      Reads the terminals while it pushes them on the stack.</p> <p>Pre-order traversal of the parse tree.      Post-order traversal of the parse tree.</p> <p>(ii) Explain Error Recovery in Predictive Parsing. (Page No.192)  <u>MAY/JUNE 2007,</u>  <u>NOV/DEC 2007, APRIL/MAY 2005</u></p> <p style="text-align: center;">BTL5</p>
2	<p>Construct an SLR parsing table for the above grammar. (Page No.218)</p> <p>E -&gt; E + T  E -&gt; T  T -&gt; T * F  T -&gt; F  F -&gt; (E)  F -&gt; id      <u>MAY/JUNE 2009, APR/MAY 2011, APRIL/MAY 2008 ,</u>  <u>MAY/JUNE 2014 NOV/DEC 2012,</u>  <u>MAY/JUNE 2015, NOV/DEC 2016</u></p> <p>The string to be parsed is id + id * id The moves of the LR parser are shown on the next page. For reference: FOLLOW(E) = { + , \$ } FOLLOW(T) = { * , + , \$ } FOLLOW(F) = { * , + , \$ }</p>

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		<b>id * id + id \$</b>	shift
(2)	0 5	<b>id</b>	<b>* id + id \$</b>	reduce by $F \rightarrow \mathbf{id}$
(3)	0 3	$F$	<b>* id + id \$</b>	reduce by $T \rightarrow F$
(4)	0 2	$T$	<b>* id + id \$</b>	shift
(5)	0 2 7	$T *$	<b>id + id \$</b>	shift
(6)	0 2 7 5	$T * \mathbf{id}$	<b>+ id \$</b>	reduce by $F \rightarrow \mathbf{id}$
(7)	0 2 7 10	$T * F$	<b>+ id \$</b>	reduce by $T \rightarrow T * F$
(8)	0 2	$T$	<b>+ id \$</b>	reduce by $E \rightarrow T$
(9)	0 1	$E$	<b>+ id \$</b>	shift
(10)	0 1 6	$E +$	<b>id \$</b>	shift
(11)	0 1 6 5	$E + \mathbf{id}$	<b>\$</b>	reduce by $F \rightarrow \mathbf{id}$
(12)	0 1 6 3	$E + F$	<b>\$</b>	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E + T$	<b>\$</b>	reduce by $E \rightarrow E + T$
(14)	0 1	$E$	<b>\$</b>	accept

If the right-most column is now traversed upwards, and the productions by which the reduce steps occur are arranged in that sequence, then that will constitute a leftmost derivation of the string by this grammar. This highlights the bottom-up nature of SLR parsing.

### III. THE SLR PARSING TABLE

To understand SLR parsing easily, the SLR parsing table is created using the principles discussed above in Item I.

The rows of this table are represented by the state numbers and the columns are divided into two segments – the ACTION function where the columns are represented by the terminals and the GOTO function where the columns are the non-terminals

The SLR parsing table for the previous grammar is shown on the next page, while the notation for the table is specified here for easy reference.

1. Shift  $i$  is represented by “ $si$ ”. (Check the previous notes for the states)
2. Reduction number  $j$  (check Item II for numbering) is represented by “ $rj$ ”
3. Accept is represented by “acc”
4. Error is represented by a blank space in the cell

Using the SLR parsing table, the process of shift-reduce parsing for SLR grammars can be automated and done easily.

STATE	ACTION						GOTO		
	id	+	*	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

#### IV. THINGS TO NOTE:

1. CONFLICT FREE NATURE OF SLR The above procedure worked for the particular grammar because it is an SLR grammar, i.e one that contains no conflicts. An example of a shift/reduce conflict is shown below:  $A \rightarrow B$ .  $A \rightarrow B + C$  If the above two items appear in the same state, then there will be a conflict in the case of ACTION [A , +] about whether to shift and move forward in the input or reduce by "A  $\rightarrow$  B".

Essentially, if each table entry in the parsing table is unique without any duplicity, then there are no conflicts and the grammar in question is SLR.

2. CORRECTNESS OF SLR The stack always contains a set of symbols until the input has been matched and only the start state remains - therefore a right sentential derivation can always be found for the given string, if the table is correctly designed without any conflicts. Hence, SLR parsing is correct

3	<p>Construct the predictive parser or non recursive predictive parsing table for the following grammar:  <math>S \rightarrow (L) \mid a</math> <math>L \rightarrow L, S \mid S</math></p> <p>Construct the behavior of the parser on the sentence (a, a) and (a,(a,(a,a))) using the grammar specified above. <u>APRIL/MAY 2012</u> , <u>MAY/JUNE 2007</u>, <u>APRIL/MAY 2005</u>,<u>NOV/DEC 2012</u>, <u>MAY/JUNE 2012</u> <u>MAY/JUNE 2013</u> , <u>APRIL/MAY 2017</u></p> <p><math>S \rightarrow (L) \mid a</math> with no left recursion <math>S \rightarrow (L) \mid a</math>  <math>L \rightarrow L,S \mid S \rightarrow S L' L' \rightarrow, S L' \mid e</math></p> <p>~~~~~</p> <p>b) recursive descent parser (with lookahead)</p> <pre> tok; // current token match(x) { // matches token if (tok != x) // if wrong token error(); // exit with error tok = getToken(); // get new token } parser() { tok = getToken(); // initialize S(); // start symbol match("\$"); // match EOF } S() { if (tok == "(" ) { // S -&gt; ( L ) match("("); L(); match(")"); } else if (tok == "a") // S -&gt; a match("a"); else error(); } L () { S(); L'(); // L -&gt; S L' } L' () { if (tok == ",") { // L' -&gt; , S L' match(","); S(); L'(); } else // L' -&gt; e ; } } </pre> <p>~~~~~</p> <p>c) example parse of (a,a) Sequence of function calls in parse. Indentation indicates nesting level  S(); // (a,a) match("("); // a,a) L(); // a,a) S(); // a,a) match("a"); // ,a) L'(); // ,a) match(","); // a) S();  // a) match("a"); // ) L'(); // ) match(")"); //</p>
5	<p>Construct Parsing table for the grammar and find moves made by predictive parser on input  <math>id + id * id</math> and find FIRST and FOLLOW.(Page No.186)</p>



NOV/DEC 2016, NOV/DEC 2017

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (E)$

	FIRST	FOLLOW
$E \rightarrow TE'$	{ id, ( }	{ \$, ) }
$E' \rightarrow +TE'/e$	{ +, e }	{ \$, ) }
$T \rightarrow FT'$	{ id, ( }	{ +, \$, ) }
$T' \rightarrow *FT'/e$	{ *, e }	{ +, \$, ) }
$F \rightarrow id/(E)$	{ id, ( }	{ *, +, \$, ) }

Now, the LL(1) Parsing Table is:

	ID	+	*	(	)	\$
<b>E</b>	E →					E →
	TE'					TE'
<b>E'</b>		E' →				E' →
		+TE'				> e
<b>T</b>	T →					T →
	FT'					FT'
<b>T'</b>		T' → e	T' →			T' →
			*FT			> e
<b>F</b>	F →					F →
	id					(E)

As you can see that all the null productions are put under the follow set of that symbol and all the remaining productions are lie under the first of that symbol.

**Note:** Every grammar is not feasible for LL(1) Parsing table. It may be possible that one cell may contain more than one production.

6 Give an algorithm for finding the FIRST and FOLLOW positions for a given non-terminal.

(Page No.188)  
2008

MAY/JUNE 2009 APRIL/MAY

7 Explain Context free grammars with examples (Page No. 165) MAY/JUNE 2016

**Definition** – A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where

- **N** is a set of non-terminal symbols.
- **T** is a set of terminals where **N ∩ T = NULL**.
- **P** is a set of rules, **P: N → (N ∪ T)\***, i.e., the left-hand side of the production rule **P** does have any right context or left context.
- **S** is the start symbol.

**Example**

- The grammar **({A}, {a, b, c}, P, A)**, **P: A → aA, A → abc**.

- The grammar  $(\{S, a, b\}, \{a, b\}, P, S)$ ,  $P: S \rightarrow aSa, S \rightarrow bSb, S \rightarrow \epsilon$
- The grammar  $(\{S, F\}, \{0, 1\}, P, S)$ ,  $P: S \rightarrow 00S \mid 11F, F \rightarrow 00F \mid \epsilon$

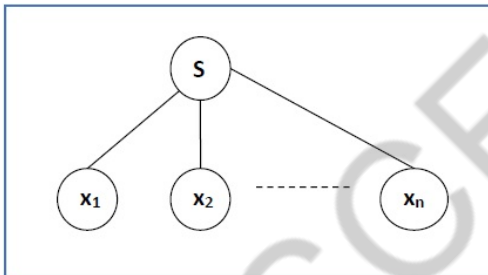
## Generation of Derivation Tree

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

### Representation Technique

- **Root vertex** – Must be labeled by the start symbol.
- **Vertex** – Labeled by a non-terminal symbol.
- **Leaves** – Labeled by a terminal symbol or  $\epsilon$ .

If  $S \rightarrow x_1x_2 \dots x_n$  is a production rule in a CFG, then the parse tree / derivation tree will be as follows –



here are two different approaches to draw a derivation tree –

### Top-down Approach –

- Starts with the starting symbol **S**
- Goes down to tree leaves using productions

### Bottom-up Approach –

- Starts from tree leaves
- Proceeds upward to the root which is the starting symbol **S**

### Derivation or Yield of a Tree

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null,

derivation is Null.

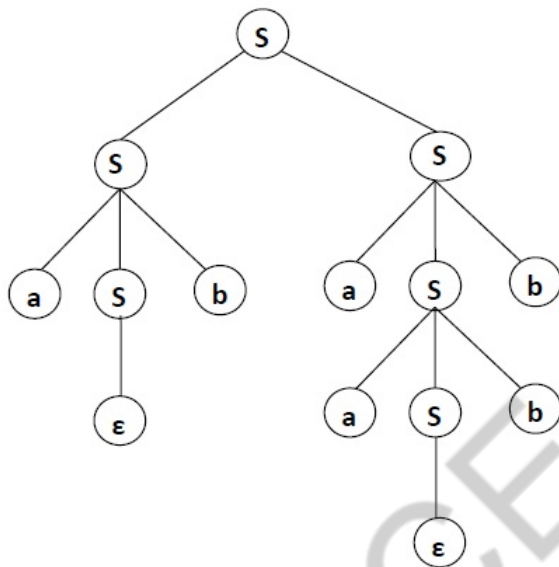
### Example

Let a CFG  $\{N, T, P, S\}$  be

$N = \{S\}$ ,  $T = \{a, b\}$ , Starting symbol = S,  $P = S \rightarrow SS \mid aSb \mid \epsilon$

One derivation from the above CFG is "abaabb"

$S \rightarrow SS \rightarrow aSbS \rightarrow abS \rightarrow abaSb \rightarrow abaaSbb \rightarrow abaabb$



8

Consider the grammar,

$E \rightarrow E + TE \rightarrow T$

$T \rightarrow T * FT \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Construct a LALR parsing table for the grammar given above. Verify whether the input string  $id + id * id$  is accepted by the

APRIL/MAY 2008

Consider the grammar:

$S ::= E \$$

$E ::= E + E$

|  $E * E$

|  $( E )$

|  $id$

|  $num$

and four of its LALR(1) states:

I0:  $S ::= . E \$ ?$

$E ::= . E + E + * \$$     I1:  $S ::= E . \$ ?$     I2:  $E ::= E * .$

$E + * \$ E ::= . E * E + * \$$      $E ::= E . + E + * \$$      $E$

$::= . E + E + * \$ E ::= . ( E ) + * \$$      $E ::= E .$

$* E + * \$$      $E ::= . E$

$* E + * \$$

$E ::= . id + * \$$      $E ::= . ( E ) + * \$$

$E ::= . num + * \$$     I3:  $E ::= E * E . + * \$$      $E ::= . id + * \$$

$E ::= E . + E + * \$$      $E ::= . num + * \$$

$E ::= E . * E + * \$$

Here we have a shift-reduce error. Consider the first two items in I3. If we have  $a*b+c$  and we parsed  $a*b$ , do we reduce using  $E ::= E * E$  or do we shift more symbols? In the former case we get a parse tree  $(a*b)+c$ ; in the latter case we get  $a*(b+c)$ . To resolve this conflict, we can specify that  $*$  has higher precedence than  $+$ . The precedence of a grammar production is equal to the precedence of the rightmost token at the rhs of the production. For example, the precedence of the production  $E ::= E * E$  is equal to the precedence of the operator  $*$ , the precedence of the production  $E ::= ( E )$  is equal to the precedence of the token  $)$ , and the precedence of the production  $E ::= \text{if } E \text{ then } E \text{ else } E$  is equal to the precedence of the token  $\text{else}$ . The idea is that if the look ahead has higher precedence than the production currently used, we shift. For example, if we are parsing  $E + E$  using the production rule  $E ::= E + E$  and the look ahead is  $*$ , we shift  $*$ . If the look ahead has the same precedence as that of the current production and is left associative, we reduce, otherwise we shift. The above grammar is valid if we define the

precedence and associativity of all the operators. Thus, it is very important when you write a parser using CUP or any other LALR(1) parser generator to specify associativities and precedence's for most tokens (especially for those used as operators). Note: you can explicitly define the precedence of a rule in CUP using the %prec directive:

```
E ::= MINUS E %prec UMINUS
```

where UMINUS is a pseudo-token that has higher precedence than TIMES, MINUS etc, so that  $-1*2$  is equal to  $(-1)*2$ , not to  $-(1*2)$ .

Another thing we can do when specifying an LALR(1) grammar for a parser generator is error recovery. All the entries in the ACTION and GOTO tables that have no content correspond to syntax errors. The simplest thing to do in case of error is to report it and stop the parsing. But we would like to continue parsing finding more errors. This is called *error recovery*. Consider the grammar:

```
S ::= L
    = E ;
```

```
| { SL
    } ;
```

```
|
error ;
```

```
SL ::=
```

```
S ;
```

```
| SL S ;
```

The special token error indicates to the parser what to do in case of invalid syntax for S (an invalid statement). In this case, it reads all the tokens from the input stream until it finds the first semicolon. The way the parser handles this is to first push an error state in the stack. In case of an error, the parser pops out elements from the stack until it finds an error state where it can proceed. Then it discards tokens from the input until a restart is possible. Inserting error handling productions in the proper places in a grammar to do good error recovery is considered very hard.

9 Check whether the following grammar is a LL(1) grammar. MAY/JUNE 2016

APRIL/MAY2005

$X \rightarrow Yz \mid a$

$Y \rightarrow bZ \mid \epsilon$

$Z \rightarrow \epsilon$

Also define the FIRST and FOLLOW procedures. (Page No. 191)

To check if a grammar is LL(1), one option is to construct the LL(1) parsing table and check for any conflicts. These conflicts can be

- FIRST/FIRST conflicts, where two different productions would have to be predicted for a nonterminal/terminal pair.
- FIRST/FOLLOW conflicts, where two different productions are predicted, one representing that some production should be taken and expands out to a nonzero number of symbols, and one representing that a production should be used indicating that some nonterminal should be ultimately expanded out to the empty string.
- FOLLOW/FOLLOW conflicts, where two productions indicating that a nonterminal should ultimately be expanded to the empty string conflict with one another.

Let's try this on your grammar by building the FIRST and FOLLOW sets for each of the nonterminals. Here, we get that

FIRST(X) = {a, b, z}

FIRST(Y) = {b, epsilon}

FIRST(Z) = {epsilon}

We also have that the FOLLOW sets are

FOLLOW(X) = {\$}

FOLLOW(Y) = {z}

FOLLOW(Z) = {z}

From this, we can build the following LL(1) parsing table:

	a	b	z	\$
X	a	Yz	Yz	
Y		bZ	eps	
Z			eps	

Since we can build this parsing table with no conflicts, the grammar is LL(1).

To check if a grammar is LR(0) or SLR(1), we begin by building up all of the LR(0) configurating sets for the grammar. In this case, assuming that X is your start symbol, we get the following:

(1)

$X' \rightarrow .X$

$X \rightarrow .Yz$

$X \rightarrow .a$

$Y \rightarrow .$

$Y \rightarrow .bZ$

(2)

$X' \rightarrow X.$

(3)

$X \rightarrow Y.z$

(4)

$X \rightarrow Yz.$

(5)

$X \rightarrow a.$

(6)  
Y → b.Z  
Z → .

(7)  
Y → bZ.

From this, we can see that the grammar is not LR(0) because there are shift/reduce conflicts in states (1) and (6). Specifically, because we have the reduce items  $Z \rightarrow \cdot$  and  $Y \rightarrow \cdot$ , we can't tell whether to reduce the empty string to these symbols or to shift some other symbol. More generally, no grammar with  $\epsilon$ -productions is LR(0).

However, this grammar might be SLR(1). To see this, we augment each reduction with the lookahead set for the particular nonterminals. This gives back this set of SLR(1) configurating sets:

(1)  
X' → .X  
X → .Yz [\$]  
X → .a [\$]  
Y → . [z]  
Y → .bZ [z]

(2)  
X' → X.

(3)  
X → Y.z [\$]

(4)  
X → Yz. [\$]

(5)  
X → a. [\$]

(6)  
Y → b.Z [z]  
Z → . [z]

(7)  
Y → bZ. [z]

Now, we don't have any more shift-reduce conflicts. The conflict in state (1) has been eliminated because we only reduce when the lookahead is z, which doesn't conflict with any of the other items. Similarly, the conflict in (6) is gone for the same reason.

10

Consider the grammar  $E \rightarrow E - E$

$E \rightarrow E \times E$

$E \rightarrow \text{id}$

Show the sequence of moves made by the shift-reduce parser on the input  $\text{id1} + \text{id2} * \text{id3}$  and determine whether the given string is accepted by the parser or not. (Page No. 198)



	<div style="position: absolute; top: 50%; left: 50%; transform: translate(-50%, -50%); opacity: 0.2; font-size: 4em; pointer-events: none;">       MMSCE - 1101     </div> <p style="text-align: center;">\$ E                          \$    Accept</p>
11	<p>What is a shift-reduce parser? Explain in detail the conflicts that may occur during shift-reduce parsing. (Page No.201) <u>MAY/JUNE 2012, APRIL/MAY 2012</u></p> <p><b>Shift Reduce parser</b> attempts for the construction of parse in a similar manner as done in bottom up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of shift reduce parser is LR parser.</p> <p>This parser requires some data structures i.e.</p> <ul style="list-style-type: none"> <li>• A input buffer for storing the input string.</li> <li>• A stack for storing and accessing the production rules.</li> </ul> <p><b>Basic Operations –</b></p> <ul style="list-style-type: none"> <li>• <b>Shift:</b> This involves moving of symbols from input buffer onto the stack.</li> <li>• <b>Reduce:</b> If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.</li> <li>• <b>Accept:</b> If only start symbol is present in the stack and the input buffer is empty then, the parsing</li> </ul>

- action is called accept. When accept action is obtained, it means successful parsing is done.
- Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

There are two kinds of conflicts that can occur in an SLR(1) parsing table.

1. A *shift-reduce* conflict occurs in a state that requests both a shift action and a reduce action.
2. A *reduce-reduce* conflict occurs in a state that requests two or more different reduce actions.

12 Consider the grammar given below.

$E \rightarrow E+E$

$E \rightarrow E^*E$

$E \rightarrow id$

Construct an LR parsing table for the above grammar. Give the moves of LR parser on  $id + id + id$

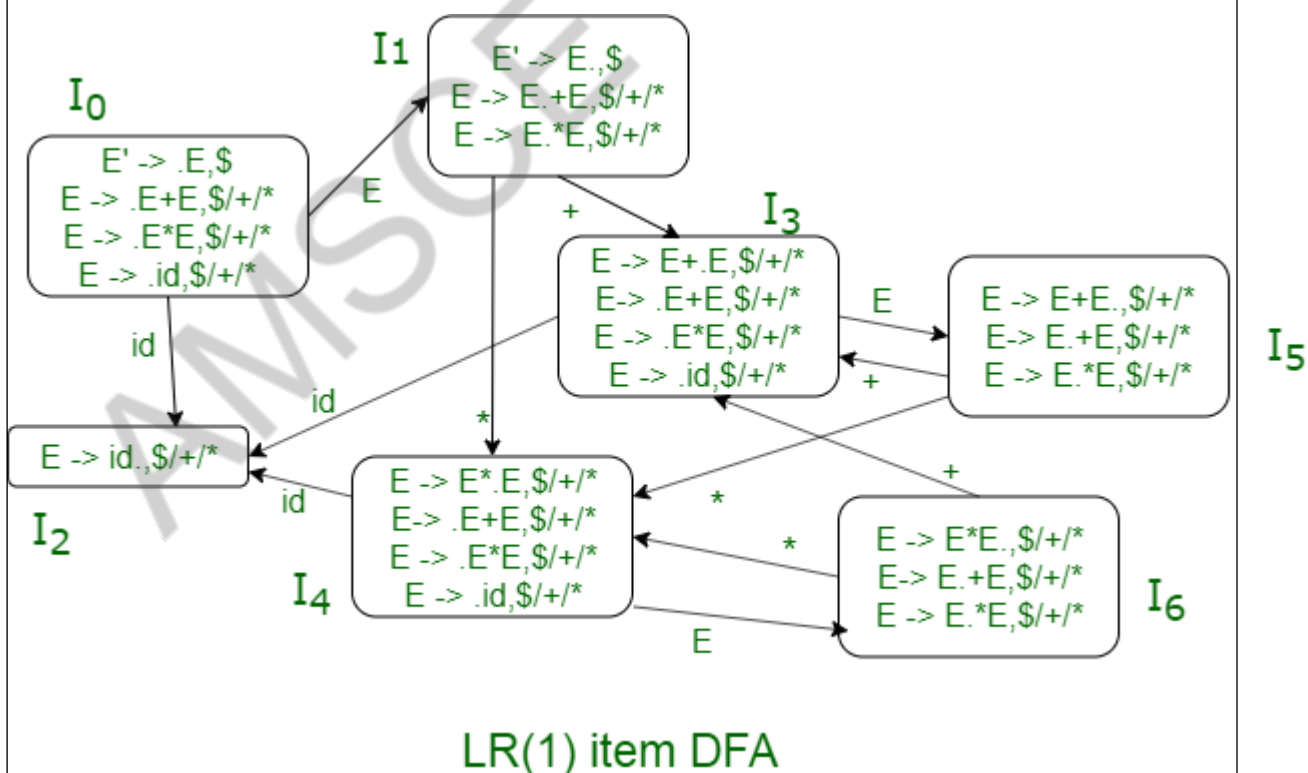
(Page No.218 & 220)

MAY/JUNE 2007

Lets assume, the precedence and associativity of the operators (+ and \*) of the grammar are as follows:

- "+" and "\*" both are left associative,
- Precedence of "\*" is higher than the precedence of "+".

If we use LALR(1) parser, the LR(1) item DFA will be:



From the LR(1) item DFA we can see that there are shift/reduce conflicts in the state  $I_5$  and  $I_6$ . So the

parsing table is as follows:

	id	+	*	\$	E
$I_0$	$S_2$				1
$I_1$		$S_3$	$S_4$	accept	
$I_2$		$r_3$	$r_3$	$r_3$	
$I_3$	$S_2$				5
$I_4$	$S_2$				6
$I_5$		$r_1/S_3$	$r_1/S_4$	$r_1$	
$I_6$		$r_2/S_3$	$r_2/S_4$	$r_2$	

### Parsing Table

There are both shift and reduce moves in  $I_5$  and  $I_6$  on “+” and “\*”. To resolve this conflict, that is to determine which move to keep and which to discard from the table we shall use the precedence and associativity of the operators.

Consider the input string:

Lets look at the parser moves till the conflict state according to the above parsing table.

Stack	Symbol	Input
0		id + id + id \$
		↑
0 2	id	+ id + id \$
		↑
0 1	E	+ id + id \$
		↑
0 1 3	E +	id + id \$
		↑
0 1 3 2	E + id	+id \$
		↑
0 1 3 5	E + E	+id \$
		↑ reduce
0 1	E	+id \$
		↑

### Parser 1

Stack	Symbol	Input
0		id + id + id \$
		↑
0 2	id	+ id + id \$
		↑
0 1	E	+ id + id \$
		↑
0 1 3	E +	id + id \$
		↑
0 1 3 2	E + id	+id \$
		↑
0 1 3 5	E + E	+id \$
		↑ Shift
0 1 3 5 3	E + E +	id \$
		↑

### Parser 2

- If we take the reduce move of  $I_5$  state on symbol “+” as in parser 1, then the left “+” of the input

	<p>string is reduced before the right “+”, which makes “+” left associative.</p> <ul style="list-style-type: none"> <li>• If we take the shift move of I<sub>5</sub> state on symbol “+” as in parser 2, then the right “+” of the input string is reduced before the left “+”, which makes “+” right associative.</li> </ul> <p>Similarly, Taking shift move of I<sub>5</sub> state on symbol “*” will give “*” higher precedence over “+”, as “*” will be reduced before “+”. Taking reduce move of I<sub>5</sub> state on symbol “*” will give “+” higher precedence over “*”, as “+” will be reduced before “*”. Similar to I<sub>5</sub>, conflicts from I<sub>6</sub> can also be resolved.</p> <p>According to the precedence and associativity of our example, the conflict is resolved as follows,</p> <ul style="list-style-type: none"> <li>• The shift/reduce conflict at I<sub>5</sub> on “+” is resolved by keeping the reduce move and discarding the shift move, which makes “+” left associative.</li> <li>• The shift/reduce conflict at I<sub>5</sub> on “*” is resolved by keeping the shift move and discarding the reduce move, which will give “*” higher precedence over “+”.</li> <li>• The shift/reduce conflict at I<sub>6</sub> on “+” is resolved by keeping the reduce move and discarding the shift move, which will give “*” higher precedence over “+”.</li> <li>• The shift/reduce conflict at I<sub>6</sub> on “*” is resolved by keeping the reduce move and discarding the shift move, which makes “*” left associative.</li> </ul>
13	<p>(i) Explain the non-recursive predictive parsing with its algorithm. (Page No.190)  <u>MAY/JUNE 2016, APRIL/MAY 2005, NOV/DEC 2007</u></p> <p>Non-recursive predictive parsing or table-driven is also known as LL(1) parser. This parser follows the leftmost derivation (LMD).</p> <p><b>LL(1):</b>  here, first L is for Left to Right scanning of inputs,  the second L is for left most derivation procedure,  1 = Number of Look Ahead Symbols</p> <p>The main problem during predictive parsing is that of determining the production to be applied for a non-terminal.</p> <p>This non-recursive parser looks up which production to be applied in a parsing table. A LL(1) parser has the following components:</p> <p>buffer: an input buffer which contains the string to be passed</p> <p>(2) stack: a pushdown stack which contains a sequence of grammar symbols</p> <p>(3) A parsing table: a 2d array M[A, a]  where  A → non-terminal, a → terminal or \$</p> <p>(4) output stream:  end of the stack and an end of the input symbols are both denoted with \$</p> <p><b>Algorithm for non recursive Predictive Parsing:</b>  The main Concept → With the help of <b>FIRST()</b> and <b>FOLLOW()</b> sets, this parsing can be done using a just a stack which avoids the recursive calls.  For each rule, A → x in grammar G:</p> <ol style="list-style-type: none"> <li>1. For each terminal ‘a’ contained in <b>FIRST(A)</b> add A → x to M[A, a] in parsing table if x derives ‘a’ as the first symbol.</li> <li>2. If <b>FIRST(A)</b> contain null production for each terminal ‘b’ in <b>FOLLOW(A)</b>, add this production (A → null) to M[A, b] in parsing table.</li> </ol> <p><b>The Procedure:</b></p> <ol style="list-style-type: none"> <li>1. In the beginning, the pushdown stack holds the start symbol of the grammar G.</li> <li>2. At each step a symbol X is popped from the stack:  if X is a terminal then it is matched with the lookahead and lookahead is advanced one step,</li> </ol>

if X is a nonterminal symbol, then using lookahead and a parsing table (implementing the FIRST sets) a production is chosen and its right-hand side is pushed into the stack.

3. This process repeats until the stack and the input string become null (empty).

**Table-driven Parsing algorithm:**

**input:** a string w and a parsing table M for G.

tos top of the stack

```
Stack[tos++] <- $
```

```
Stack[tos++] <- Start Symbol
```

```
token <- next_token()
```

```
X <- Stack[tos]
```

```
repeat
```

```
  if X is a terminal or $ then
```

```
    if X = token then
```

```
      pop X
```

```
      token is next of token()
```

```
      else error()
```

```
    else /* X is a non-terminal */
```

```
      if M[X, token] = X -> y1y2...yk then
```

```
        pop x
```

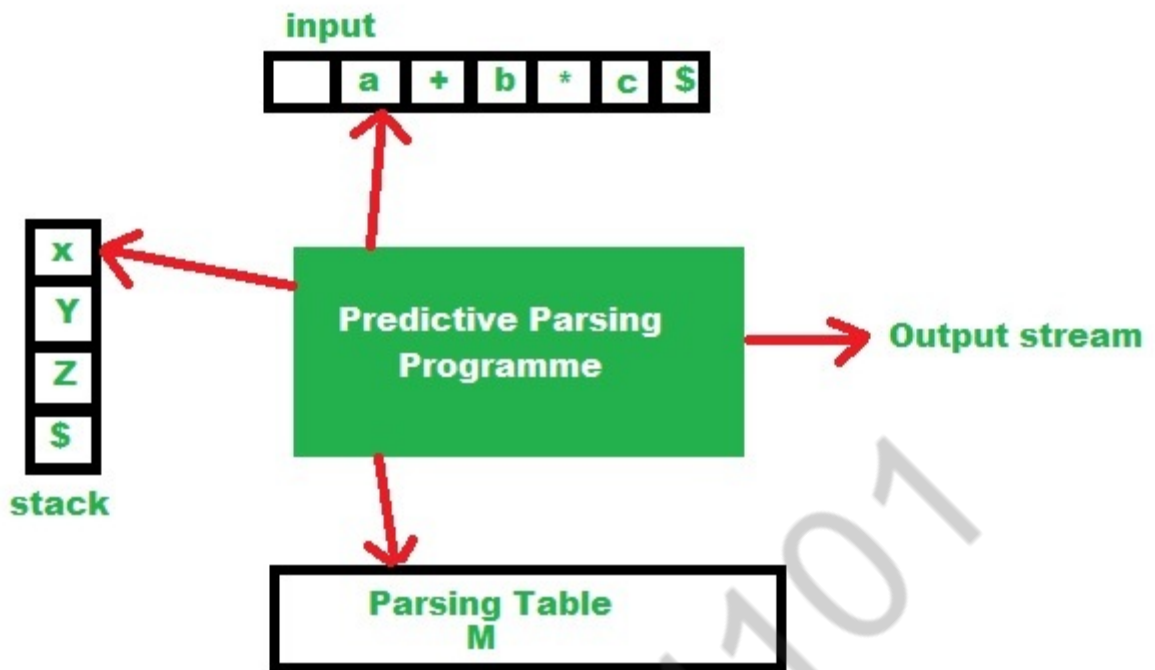
```
        push
```

```
        else error()
```

```
  X Stack[tos]
```

```
until X = $
```

```
// Non-recursive parser model diagram:
```



**fig: Non Recursive Parser Model**

So according to the given diagram the non-recursive parsing algorithm.

**Input:** A input string 'w' and a parsing table('M') for grammar G.

**Output:** If w is in L(G), an LMD of w; otherwise an error indication.

Set input pointer to point to the first symbol of the string \$;

repeat

let X be the symbol pointed by the stack pointer,

and a is the symbol pointed to by input pointer;

if X is a terminal or \$ then

if X=a then

pop X from the stack and increment the input pointer;

else error()

end if

else /\*if X is a non terminal \*/

if then

begin

pop X from the stack;

push onto the stack, with Y1 on top;

output the production

end

else error()

end if

end if

until X=\$ /\* stack is empty \*/

(ii) Explain the LR parsing algorithm in detail. (Page. No. 218) NOV/DEC 2007, APRIL/MAY 2005

14 . (i)What is an ambiguous grammar? Is the following grammar ambiguous? Prove  
 $E \rightarrow E + E \mid E * E \mid (E) \mid id.$   
MAY/JUNE 2014

Consider the grammar:

$S ::= E \$$

$E ::= E + E$

$\mid E * E$

$\mid (E)$

$\mid id$

$\mid num$

and four of its LALR(1) states:

I0:  $S ::= . E \$ ?$

$E ::= . E + E + * \$$     I1:  $S ::= E . \$ ?$     I2:  $E ::= E * .$

$E + * \$ E ::= . E * E + * \$$      $E ::= E . + E + * \$$      $E$

$::= . E + E + * \$ E ::= . ( E ) + * \$$      $E ::= E .$

$* E + * \$$      $E ::= . E$

$* E + * \$$

$E ::= . id + * \$$      $E ::= . ( E ) + * \$$

$E ::= . num + * \$$     I3:  $E ::= E * E . + * \$$      $E ::= . id + * \$$

$E ::= E . + E + * \$$      $E ::= . num + * \$$

$E ::= E . * E + * \$$

Here we have a shift-reduce error. Consider the first two items in I3. If we have  $a*b+c$  and we parsed  $a*b$ , do we reduce using  $E ::= E * E$  or do we shift more symbols? In the former case we get a parse tree  $(a*b)+c$ ; in the latter case we get  $a*(b+c)$ . To resolve this conflict, we can specify that  $*$  has higher precedence than  $+$ . The precedence of a grammar production is equal to the precedence of the rightmost token at the rhs of the production. For example, the precedence of the production  $E ::= E * E$  is equal to the precedence of the operator  $*$ , the precedence of the production  $E ::= ( E )$  is equal to the

precedence of the token `)`, and the precedence of the production  $E ::= \text{if } E \text{ then } E \text{ else } E$  is equal to the precedence of the token `else`. The idea is that if the look ahead has higher precedence than the production currently used, we shift. For example, if we are parsing  $E + E$  using the production rule  $E ::= E + E$  and the look ahead is `*`, we shift `*`. If the look ahead has the same precedence as that of the current production and is left associative, we reduce, otherwise we shift. The above grammar is valid if we define the precedence and associativity of all the operators. Thus, it is very important when you write a parser using CUP or any other LALR(1) parser generator to specify associativities and precedence's for most tokens (especially for those used as operators). Note: you can explicitly define the precedence of a rule in CUP using the `%prec` directive:

```
E ::= MINUS E %prec UMINUS
```

where UMINUS is a pseudo-token that has higher precedence than TIMES, MINUS etc, so that  $-1*2$  is equal to  $(-1)*2$ , not to  $-(1*2)$ .

Another thing we can do when specifying an LALR(1) grammar for a parser generator is

(OR)  $G: E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \text{id}$ . for the sentence  $\text{id}+\text{id}*\text{id}$ . (Refer Notes)  
NOV/DEC 2016

(ii) List all LR(0) items for the following grammar



(Refer Notes) MAY/JUNE 2013

In *LR(0) item* is a production of the grammar with exactly one dot on the right-hand side. For example, production  $T \rightarrow T * F$  leads to four LR(0) items:

$$T \rightarrow \cdot T * F$$

$$T \rightarrow T \cdot * F$$

$$T \rightarrow T * \cdot F$$

$$T \rightarrow T * F \cdot$$

What is to the left of the dot has just been read, and the parser is ready to read the remainder, after the dot.

Two LR(0) items that come from the same production but have the dot in different places are considered different LR(0) items.

### Closures

Suppose that  $S$  is a set of LR(0) items. The following rules tell how to build *closure(S)*, the *closure* of  $S$ . You must add LR(0) items to  $S$  until there are no more to add.

1. All members of  $S$  are in the *closure(S)*.
2. Suppose *closure(S)* contains item  $A \rightarrow \alpha \cdot B \beta$ , where  $B$  is a nonterminal. Find all productions  $B \rightarrow \gamma_1, \dots, B \rightarrow \gamma_n$  with  $B$  on the left-hand side. Add LR(0) items  $B \rightarrow \cdot \gamma_1, \dots, B \rightarrow \cdot \gamma_n$  to *closure(S)*.

For example, let's take the closure of set  $\{E \rightarrow E + \cdot T\}$ .

1. Since there is an item with a dot immediately before nonterminal  $T$ , we add  $T \rightarrow \cdot F$  and  $T \rightarrow \cdot T * F$ .

The set now contains the following LR(0) items.

$$E \rightarrow E + \cdot T$$

$$T \rightarrow \cdot F$$

$$T \rightarrow \cdot T * F$$

2. Now there is an item in the set with a dot immediately followed by  $F$ . So we add items  $F \rightarrow \cdot n$  and  $F \rightarrow \cdot ( E )$ .

The set now contains the following items.

$$E \rightarrow E + \cdot T$$

$$T \rightarrow \cdot F$$

$$T \rightarrow \cdot T * F$$

$$F \rightarrow \cdot n$$

$$F \rightarrow \cdot ( E )$$

3. No more LR(0) items need to be added, so the closure is finished.

What is the point of the closure? LR(0) item  $E \rightarrow E + \cdot T$  indicates that the parser has just finished reading an expression followed by a + sign. In fact,  $E +$  are the top two symbols on the stack.

Now, the parser is looking to see if there is a  $T$  next. (It does not *predict* that there is a  $T$  next. It is just considering that as a possibility.)

But that means it should be looking for something that is the right-hand side of a production for  $T$ . So we add items for  $T$  with the dot at the beginning.

15 Design a syntax rule (YACC) for arithmetic expression. (Page No.257)

16 . Consider the grammar given below.

Construct a CLR parsing table for the above grammar. (Page No.230)

In the SLR method we were working with LR(0) items. In CLR parsing we will be using LR(1) items. LR(k) item is defined to be an item using lookaheads of length k. So , the LR(1) item is comprised of two parts : the LR(0) item and the lookahead associated with the item.

LR(1) parsers are more powerful parser.

For LR(1) items we modify the Closure and GOTO function.

#### Closure Operation

Closure(I)

repeat

for (each item [  $A \rightarrow ?B?, a$  ] in I )  
for (each production  $B \rightarrow ?$  in  $G'$ )  
for (each terminal b in FIRST(?a))  
add [  $B \rightarrow .?, b$  ] to set I;

until no more items are added to I;

return I;

Lets understand it with an example –

$$G = S \rightarrow AA$$

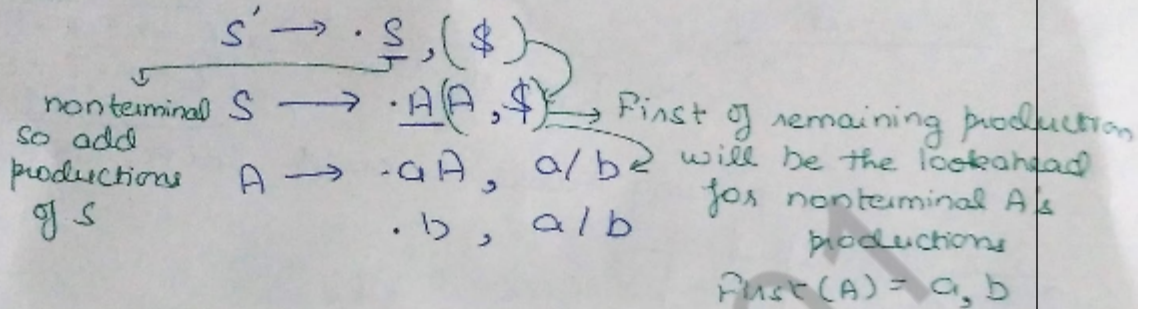
$$A \rightarrow aA / b$$

$$G' = S' \rightarrow S$$

$$S \rightarrow AA$$

$$A \rightarrow aA / b$$

Eg-1 Closure ( $S' \rightarrow \cdot S, \$$ )



Eg-2 Closure ( $A \rightarrow \cdot aA, a/b$ )

$$A \rightarrow \cdot aA, a/b$$

### Goto Operation

Goto(I, X)

Initialise J to be the empty set;

for ( each item  $A \rightarrow ?X?, a ]$  in I )

    Add item  $A \rightarrow ?X?, a ]$  to se J; /\* move the dot one step \*/

return Closure(J); /\* apply closure to the set \*/

Eg-

$$G' = S' \rightarrow S$$

$$S \rightarrow AA$$

$$A \rightarrow aA / b$$

Eg-1  $Goto(\frac{S' \rightarrow \cdot S, \$}{I}, \frac{S}{X}) = S' \rightarrow S \cdot, \$$

Eg-2  $Goto(S \rightarrow \cdot AA, a/b, \frac{A}{X}) =$

$S \rightarrow A \cdot A, (a/b)$  - In closure of  $A$  for lookahead  
 $A \rightarrow \cdot aA, a/b$  see the remaining production,  
 $\cdot b, a/b$   
 apply closure  
 if nothing remaining add the previous lookahead.

### LR(1) items

Void items( $G'$ )

Initialise  $C$  to  $\{ \text{closure}(\{[S' \rightarrow \cdot S, \$]\}) \}$ ;

Repeat

For (each set of items  $I$  in  $C$ )

For (each grammar symbol  $X$ )

if(  $\text{GOTO}(I, X)$  is not empty and not in  $C$ )

Add  $\text{GOTO}(I, X)$  to  $C$ ;

Until no new set of items are added to  $C$ ;

### Construction of GOTO graph

- State  $I_0$  – closure of augmented LR(1) item.
- Using  $I_0$  find all collection of sets of LR(1) items with the help of DFA
- Convert DFA to LR(1) parsing table

### Construction of CLR parsing table-

Input – augmented grammar  $G'$

1. Construct  $C = \{ I_0, I_1, \dots, I_n \}$ , the collection of sets of LR(0) items for  $G'$ .
2. State  $i$  is constructed from  $I_i$ . The parsing actions for state  $i$  are determined as follow :
  - i) If  $[A \rightarrow \cdot a?, b]$  is in  $I_i$  and  $\text{GOTO}(I_i, a) = I_j$ , then set  $\text{ACTION}[i, a]$  to “shift  $j$ ”. Here  $a$  must be terminal.
  - ii) If  $[A \rightarrow \cdot, a]$  is in  $I_i$ ,  $A \neq S$ , then set  $\text{ACTION}[i, a]$  to “reduce  $A \rightarrow ?$ ”.
  - iii) If  $[S \rightarrow \cdot, \$]$  is in  $I_i$ , then set  $\text{action}[i, \$]$  to “accept”.If any conflicting actions are generated by the above rules we say that the grammar is not CLR.
3. The goto transitions for state  $i$  are constructed for all nonterminals  $A$  using the rule: if  $\text{GOTO}(I_i, A) = I_j$  then  $\text{GOTO}[i, A] = j$ .
4. All entries not defined by rules 2 and 3 are made error.

Eg:

Consider the following grammar

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow ?$

$B \rightarrow ?$

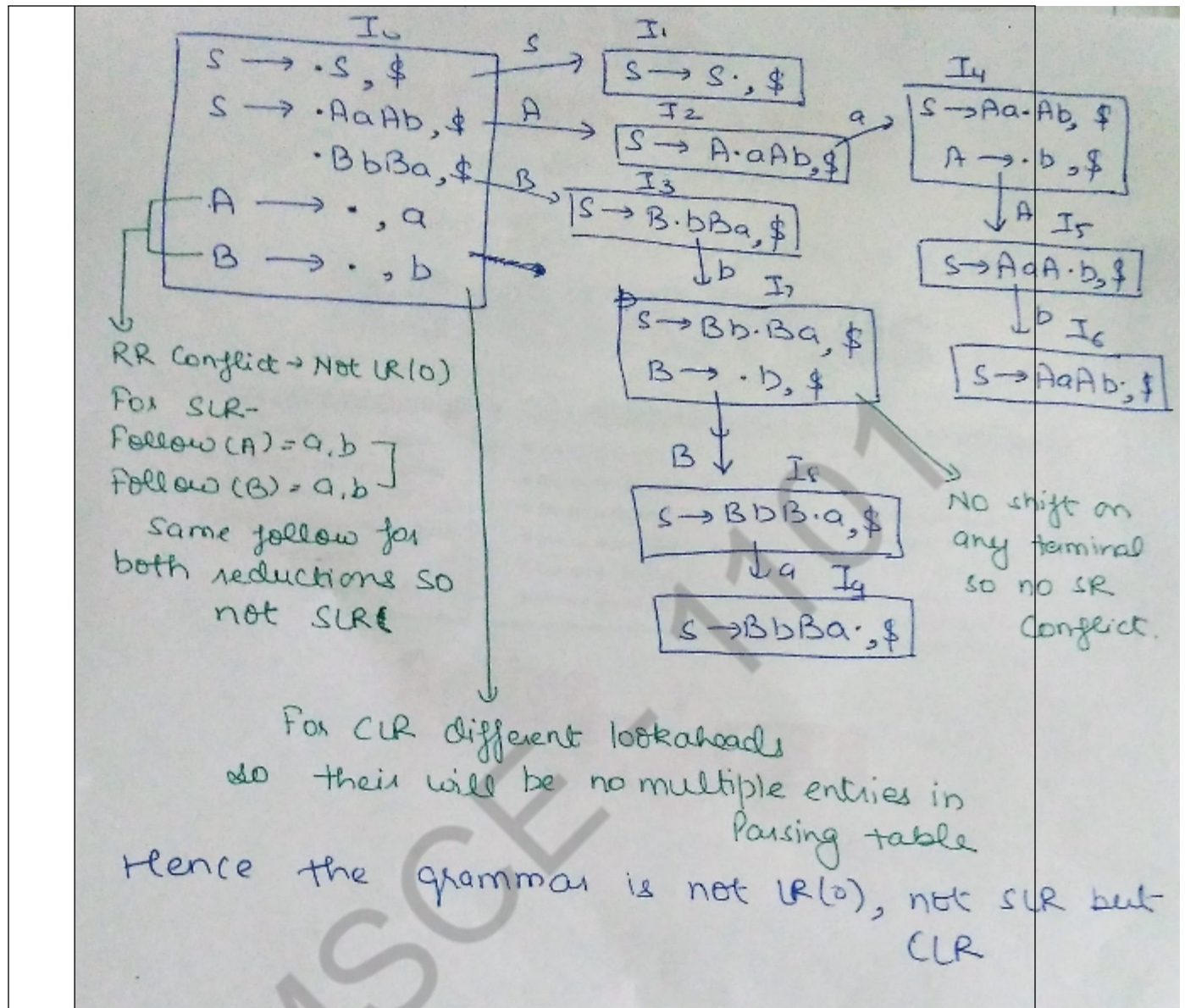
Augmented grammar -  $S' \rightarrow S$

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow ?$

$B \rightarrow ?$

GOTO graph for this grammar will be -



**Note** – if a state has two reductions and both have same lookahead then it will in multiple entries in parsing table thus a conflict. If a state has one reduction and their is a shift from that state on a terminal same as the lookahead of the reduction then it will lead to multiple entries in parsing table thus a conflict.

17 . Construct parse tree for the input string  $w = cad$  using top-down rser. (Page No.181)

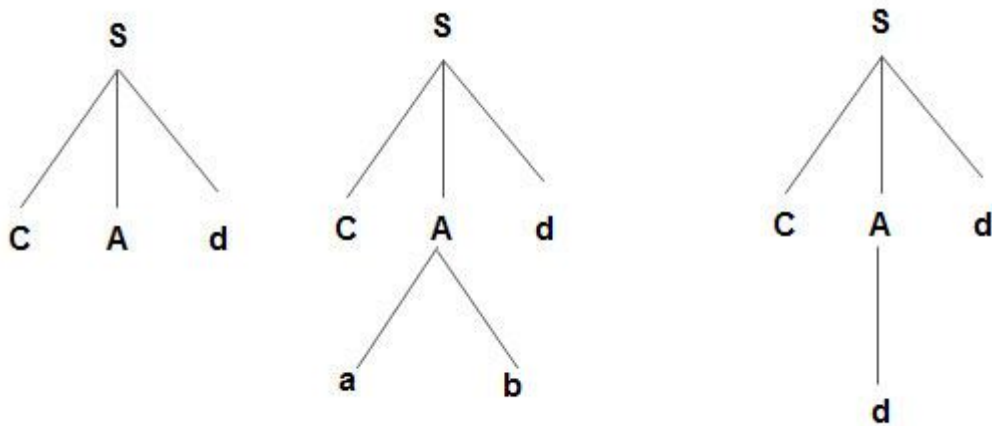
NOV/DEC 2016

eg.) Let grammar G be,

$S \rightarrow cAd$

$A \rightarrow ab \mid d$

$w = cad$



### Explanation

- The root node contains the start symbol which is S.
- The body of production begins with c, which matches with the first symbol of the input string.
- A is a non-terminal which is having two productions  $A \rightarrow ab \mid d$ .
- Apply the first production of A, which results in the string *cabd* that does not match with the given string *cad*.
- Backtrack to the previous step where the production of A gets expanded and try with alternate production of it.
- This produces the string *cad* that matches with the given string.

### Limitation

- If the given grammar has more number of alternatives then the cost of backtracking will be high.

### Recursive descent parser without backtracking

Recursive descent parser without backtracking works in a similar way as that of recursive descent parser with backtracking with the difference that each non-terminal should be expanded by its correct alternative in the first selection itself.

When the correct alternative is not chosen, the parser cannot backtrack and results in syntactic error.

### Advantage

- Overhead associated with backtracking is eliminated.

### Limitation

- When more than one alternative with common prefixes occur, then the selection of the correct alternative is highly difficult.

Hence, this process requires a grammar with no common prefixes for alternatives.

--	--

AMSCE - 1101

## UNIT IV SYNTAX DIRECTED TRANSLATION & RUN TIME ENVIRONMENT

Syntax directed Definitions-Construction of Syntax Tree-Bottom-up Evaluation of S-AttributeDefinitions- Design of predictive translator - Type Systems-Specification of a simple type checker-Equivalence of Type Expressions-Type Conversions.

RUN-TIME ENVIRONMENT: Source Language Issues-Storage Organization-Storage Allocation-Parameter Passing-Symbol Tables-Dynamic Storage Allocation-Storage Allocation in FORTAN.

S. No.	Question
1	<p><b>What are the limitations of static allocation?</b>  <u>APRIL/MAY 2011</u></p> <p>The size of the data object and constraints on its position in memory must be known at compile time.</p> <p>Recursive procedures are restricted, because all activations of a procedure use the same bindings for local names.</p> <p>Data structures cannot be created dynamically, since there is no mechanism for storage allocation at run time.</p>
2	<p><b>Draw the DAG for the statement <math>a = (a*b+c)-(a*b+c)</math>.</b>  <u>NOV/DEC 2017</u></p> <div style="text-align: center;"> </div>
3	<p><b>Define DAG.</b> <u>MAY/JUNE 2016, NOV/DEC 2007, MAY/JUNE 2007</u></p> <p>A DAG for a basic block is a directed acyclic graph with</p>



	<p>the following labels on nodes:</p> <ul style="list-style-type: none"> <li>i) Leaves are labeled by unique identifiers, either variable names or constants.</li> <li>ii) Interior nodes are labeled by an operator symbol.</li> <li>iii) Nodes are also optionally given a sequence of identifiers for labels.</li> </ul>
4	<p><b>When does dangling references occur</b>  <u>MAY/JUNE 2016</u></p> <p>When there is a reference to storage that has been de-allocated, logical error occurs as it uses dangling reference where the value of de-allocated storage is undefined according to the semantics of most languages.</p>
5	<p><b>Mention the two rules for type checking.</b>  <u>NOV/DEC 2011, APRIL/MAY 2017</u></p> <p>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.</p>
6	<p><b>What is syntax directed translation? (or) Write down syntax directed definition of a simple desk calculator.</b>  <u>NOV/DEC 2016</u></p> <p>A syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions</p> <p>Production <math>E \rightarrow E1 + T</math></p> <p>Semantic Rule <math>E.code = E1.code    T.code    '+'</math></p>
7	<p><b>What do you mean by binding of names?</b> <u>APRIL/MAY 2017</u></p> <p>A binding is an association between two entities: Name and memory location  (for variables)</p>

	<p>Name and function</p> <p>Typically a binding is between a name and the object it refers to.</p>													
8	<p><b>What is synthesized attributes?</b></p> <p>A synthesized attribute at node N is defined only in terms of attribute values of children of N and at N</p>													
9	<p><b>What is inherited attributes ?</b></p> <p>An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings</p>													
10	<p><b>What is a syntax tree? Draw the syntax tree for the assignment statement <math>a := b * -c + b * -c.</math></b>  <u>APRIL/MAY 2011, NOV/DEC 2011 NOV/DEC 2012</u>  A syntax tree depicts the natural hierarchical structure of a source program.  Syntax tree:</p> <pre> graph TD     assign[assign] --- a[a]     assign --- plus[+]     plus --- star1[*]     plus --- star2[*]     star1 --- b1[b]     star1 --- uminus1[uminus]     uminus1 --- c1[c]     star2 --- b2[b]     star2 --- uminus2[uminus]     uminus2 --- c2[c] </pre>													
11	<p><b>What are the fields of activation record?</b></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="text-align: center;">Actual parameters</td></tr> <tr><td style="text-align: center;">-----</td></tr> <tr><td style="text-align: center;">Returned values</td></tr> <tr><td style="text-align: center;">-----</td></tr> <tr><td style="text-align: center;">Control link</td></tr> <tr><td style="text-align: center;">-----</td></tr> <tr><td style="text-align: center;">Access link</td></tr> <tr><td style="text-align: center;">-----</td></tr> <tr><td style="text-align: center;">Saved machine status</td></tr> <tr><td style="text-align: center;">-----</td></tr> <tr><td style="text-align: center;">Local data</td></tr> <tr><td style="text-align: center;">-----</td></tr> <tr><td style="text-align: center;">Temporaries</td></tr> </table>	Actual parameters	-----	Returned values	-----	Control link	-----	Access link	-----	Saved machine status	-----	Local data	-----	Temporaries
Actual parameters														
-----														
Returned values														
-----														
Control link														
-----														
Access link														
-----														
Saved machine status														
-----														
Local data														
-----														
Temporaries														

12	<p><b>What is the order of calling sequence ?</b></p> <p>The caller evaluates the actual parameters</p> <p>The caller stores a return address and the old value of <i>top-sp</i> into the callee's activation record.</p> <p>The callee saves the register values and other status information.</p> <p>The callee initializes its local data and begins execution.</p>
13	<p><b>What are the functions and properties of Memory Manager?</b></p> <p>Two basic functions: Allocation Deallocation</p> <p>Properties of memory managers: Space efficiency</p> <p>Program efficiency</p> <p>Low overhead</p>
14	<p><b>What is static checking?</b></p> <p>A compiler must check that the source program follows both syntactic and semantic conversions of the source language. This checking called static checking detects and reports programming errors.</p>
15	<p><b>Give some examples of static checking?</b></p> <p>Type checks:</p> <p>A compiler should report an error if an operator is applied to an incompatible operand.</p> <p>Flow of control checks:</p> <p>Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control.</p>

16	<p><b>What is a Procedure?</b></p> <p>A procedure definition is a declaration that associates an identifier with a statement. The identifier is the procedure name , and the statement is the procedure body.</p>
17	<p><b>What is an Activation tree?</b></p> <p>An activation tree is used to depict the way control enters and leaves activations,</p> <p>i)Each node represents an activation of a procedure. ii)The root represents the activation of the main program.</p> <p>iii) The node for a is the parent of the node for b if and only if control flows from activation a to b.</p> <p>iv) The node for a is to the left of the node for b if and only if the lifetime of a occurs before the lifetime of b.</p>
18	<p><b>What is the use of a control stack?</b></p> <p>A control stack is used to keep track of live procedure activations.The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.</p>
19	<p><b>What are the types of storage allocation strategies? (OR) List Dynamic Storage allocation techniques.</b></p> <p><u>NOV/DEC 2016, NOV/DEC 2017</u></p> <p>Static allocation : Lays out storage for all objects at compile time.</p> <p>Stack allocation : Manages the run-time storage as a stack.</p> <p>Heap allocation : Allocates and deallocates storage as needed at run time from a data area known as heap</p>
20	<p><b>Define dependency graph.</b></p> <p>If an attribute b at a node in a parse tree depends on an attribute c, then the semantic rule for b at the node must be evaluated after the semantic rule that defines</p>

	<p>c. The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called dependency graph.</p>
21	<p><b>What methods have been proposed for evaluating semantic rules?</b></p> <p>Parse – tree methods Rule – based methods Oblivious methods</p>
22	<p><b>What are the functions used to create the nodes of syntax tree?</b></p> <p>mknode(op,left,right) mkleaf(id, entry) mkleaf(num,val)</p>
23	<p><b>What is a topological sort?</b></p> <p>A topological sort of a directed acyclic graph is any ordering <math>m_1, m_2, \dots, m_k</math> of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes.</p>
24	<p><b>What is a type expression?</b></p> <p>The type of a language construct will be denoted by a “type expression” . Informally a type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions.</p>
25	<p><b>What is a type system?</b></p> <p>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.</p>
26	<p><b>What are coercions?</b></p> <p>Conversion from one type to another is said to be implicit if it is to be done automatically by the compiler. Implicit</p>

	type conversions are also called coercions.
27	<p><b>What is an intermediate code?</b></p> <p>Intermediate codes are machine independent codes, but they are close to machine instructions. The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.</p>
28	<p><b>What are quadruples?</b></p> <p>Quadruples are close to machine instructions, but they are not actual machine instructions.</p>
29	<p><b>What is three address code?</b></p> <p>We use the term “three address code” because each statement usually contains three addresses (two for operands, one for the result).</p> <p>General form : <math>X := Y \text{ op } Z</math></p>
30	<p><b>What are the representations of three address code?</b></p> <p>Quadruples Triples</p> <p>Indirect triples.</p>
31	<p><b>What do you mean by strongly typed language?</b></p> <p>A language is strongly typed if its compiler can guarantee that the programs that it accepts will execute without type errors.</p>
32	<p><b>What is sound type system?</b></p> <p>A sound type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when target program runs.</p>
33	<b>Define environment and state.</b>

	<p>The term environment refers to a function that maps a name to a storage location.</p> <p>The term state refers to a function that maps a storage location to the value held there.</p>
34	<p><b>Define symbol table.</b></p> <p>Symbol table is a data structure used by the compiler to keep track of semantics of the variables. It stores information about scope and binding information about names.</p>
35	<p><b>What are the various ways to pass a parameter in a function?</b></p> <p>Call by value Call by reference Copy-restore</p> <p>Call by name</p>
36	<p><b>What are the functions used to create the nodes of syntax trees?</b></p> <p>Mknode (op, left, right) Mkleaf (id,entry)</p> <p>Mkleaf (num, val)</p>
37	<p><b>What are the functions for constructing syntax trees for expressions?</b></p> <p>i) The construction of a syntax tree for an expression is similar to the translation of the expression into postfix form.</p> <p>ii) Each node in a syntax tree can be implemented as a record with several fields</p>
38	<p><b>Give short note about call-by-name?</b></p> <p>Call by name, at every reference to a formal parameter in a procedure body the name of the corresponding actual parameter is evaluated. Access is then made to the</p>

	effective parameter.
39	<p><b>Define an attribute. Give the types of an attribute?</b></p> <p>An attribute may represent any quantity, with each grammar symbol, it associates a set of attributes and with each production, a set of semantic rules for computing values of the attributes associated with the symbols appearing in that production. Example: a type, a value, a memory location etc., i) Synthesized attributes. ii) Inherited attributes.</p>
40	<p><b>Give the 2 attributes of syntax directed translation into 3-addr code?</b></p> <p>i) E.place, the name that will hold the value of E and</p> <p>ii) E.code , the sequence of 3-addr statements evaluating E.</p>
41	<p><b>What are the advantages of generating an intermediate representation?</b></p> <p>Ease of conversion from the source program to the intermediate code. Ease with which subsequent processing can be performed from the intermediate code.</p>
42	<p><b>Define annotated parse tree?</b></p> <p>A parse tree showing the values of attributes at each node is called an annotated parse tree. The process of computing an attribute values at the nodes is called annotating parse tree.</p>
43	<p><b>Define translation scheme?</b></p> <p>A translation scheme is a CFG in which program fragments called semantic action are embedded within the right sides of productions. A translation scheme is like a syntax-directed definition, except that the order of evaluation of the semantic rules is explicitly shown.</p>
44	<p><b>What are the various data structure used for implementing the symbol table?</b></p> <p>Linear list Binary tree</p>
	Hash table
45	<p><b>Write a short note on declarations?</b></p> <p>Declarations in a procedure, for each local name, we create a symbol table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record. The procedure enter (name, type, offset) create a symbol table entry.</p>
46	<p><b>Write the 3-addr code for the statements <math>a = b * -c + b * -c</math>?</b></p> <p>Three address codes are: <math>a = b * -c + b * -c</math></p> <p><math>T1 = -c</math> <math>T2 = b * T1</math> <math>T3 = -c</math> <math>T4 = b * T3</math></p>



	$T5 = T2+T4$ $a:= T5$ .
47	<p><b>List out the two rules for type checking</b></p> <p>Type Synthesis Type inference</p>
48	<p><b>What is S-Attributed Syntax Directed Translation(SDT)?</b></p> <p>If an SDT uses only synthesized attributes, it is called as S-attributed SDT. S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.</p>
49	<p><b>What is L-Attributed Syntax Directed Translation(SDT)?</b></p> <p>If an SDT uses either synthesized attributes or inherited attributes with a restriction that it can inherit values from left siblings only, it is called as L-attributed SDT. Attributes in L-attributed SDTs are evaluated by depth- first and left-to-right parsing manner.</p>
50	<p><b>When stack allocation is not possible ?</b></p> <p>The values of local names must be retained when an activation ends.</p> <p>A called activation outlives the caller.</p>
1	<p>Discuss the various storage allocation strategies in detail. MAY/JUNE 2016, APRIL/MAY 2011, NOV/DEC 2011, NOV/DEC 2007, NOV/DEC 2014, MAY/JUNE 2013, APRIL/MAY 2017 (Page No.401)</p> <h2 style="text-align: center;">Storage Allocation</h2> <p>The different ways to allocate memory are:</p> <ol style="list-style-type: none"> <li>1. Static storage allocation</li> <li>2. Stack storage allocation</li> <li>3. Heap storage allocation</li> </ol> <p><i>Static storage allocation</i></p> <ul style="list-style-type: none"> <li>○ In static allocation, names are bound to storage locations.</li> <li>○ If memory is created at compile time then the memory will be created in static area and only once.</li> <li>○ Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion.</li> <li>○ The drawback with static storage allocation is that the size and position of data objects</li> </ul>

	<p>should be known at compile time.</p> <ul style="list-style-type: none"> <li>Another drawback is restriction of the recursion procedure.</li> </ul> <p><i>Stack Storage Allocation</i></p> <ul style="list-style-type: none"> <li>In static storage allocation, storage is organized as a stack.</li> <li>An activation record is pushed into the stack when activation begins and it is popped when the activation end.</li> <li>Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.</li> <li>It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.</li> </ul> <p><i>Heap Storage Allocation</i></p> <ul style="list-style-type: none"> <li>Heap allocation is the most flexible allocation scheme.</li> <li>Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.</li> <li>Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.</li> <li>Heap storage allocation supports the recursion process.</li> </ul> <p><b>Example:</b></p> <pre> 1. fact (int n) 2. { 3.   if (n&lt;=1) 4.     return 1; 5.   else 6.     return (n * fact(n-1)); 7. } 8. fact (6) </pre>
2	<p>Explain in detail about the specification of a simple type checker. <u>MAY/JUNE 2016, MAY/JUNE 2012, APRIL/MAY 2012, NOV/DEC 2014, MAY/JUNE 2013, NOV/DEC 2016, APRIL/MAY 2017</u> (Page No.348)</p> <p><b>SPECIFICATION OF A SIMPLE TYPE CHECKER</b></p> <p>A type checker for a simple language checks the type of each identifier. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements and functions.</p> <p><b>A Simple Language</b> Consider the following grammar:</p> <pre> P → D ; E D → D ; D   id : T T → char   integer   array [ num ] of T   ↑ T E → literal   num   id   E mod E   E [ E ]   E ↑ </pre>

Translation scheme:

$P \rightarrow D ; E$

$D \rightarrow D ; D$

$D \rightarrow id : T \{ \text{addtype} (id.entry , T.type) \}$

$T \rightarrow char \{ T.type := char \}$

$T \rightarrow integer \{ T.type := integer \}$

$T \rightarrow \uparrow T1 \{ T.type := \text{pointer}(T1.type) \}$

$T \rightarrow \text{array} [ num ] \text{ of } T1 \{ T.type := \text{array} ( 1 \dots num.val , T1.type) \}$

In the above language,

→ There are two basic types : char and integer ; → type\_error is used to signal errors;

→ the prefix operator  $\uparrow$  builds a pointer type. Example ,  $\uparrow integer$  leads to the type expression

pointer ( integer ).

### Type checking of expressions

In the following rules, the attribute type for E gives the type expression assigned to the expression generated by E.

1.  $E \rightarrow \text{literal} \{ E.type := char \} E \rightarrow \text{num} \{ E.type := integer \}$

Here, constants represented by the tokens literal and num have type char and integer.

2.  $E \rightarrow id \{ E.type := \text{lookup} ( id.entry ) \}$

lookup ( e ) is used to fetch the type saved in the symbol table entry pointed to by e.

3.  $E \rightarrow E1 \text{ mod } E2 \{ E.type := \text{if } E1.type = integer \text{ and } E2.type = integer \text{ then } integer \\ \text{else } type\_error \}$

The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is type\_error.

4.  $E \rightarrow E1 [ E2 ] \{ E.type := \text{if } E2.type = integer \text{ and } E1.type = \text{array}(s,t) \text{ then } t \\ \text{else } type\_error \}$

In an array reference  $E1 [ E2 ]$  , the index expression E2 must have type integer. The result is the element type t obtained from the type array(s,t) of E1.

5.  $E \rightarrow E1 \uparrow \{ E.type := \text{if } E1.type = \text{pointer} (t) \text{ then } t \\ \text{else } type\_error \}$

The postfix operator  $\uparrow$  yields the object pointed to by its operand. The type of  $E \uparrow$  is the type t of the object pointed to by the pointer E.

### Type checking of statements

Statements do not have values; hence the basic type void can be assigned to them. If an error is detected within a statement, then type\_error is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:  $S \rightarrow id : = E$

	<p> <math>S \rightarrow id: = E \quad \{ S.type := \text{if } id.type = E.type \text{ then void else type\_error } \}</math> </p> <p> 2. Conditional statement: <math>S \rightarrow \text{if } E \text{ then } S1</math>  <math>S \rightarrow \text{if } E \text{ then } S1 \quad \{ S.type := \text{if } E.type = \text{boolean} \text{ then } S1.type \text{ else type\_error } \}</math> </p> <p> 3. While statement:  <math>S \rightarrow \text{while } E \text{ do } S1</math>  <math>S \rightarrow \text{while } E \text{ do } S1 \quad \{ S.type := \text{if } E.type = \text{boolean} \text{ then } S1.type \text{ else type\_error } \}</math> </p> <p> 4. Sequence of statements:  <math>S \rightarrow S1 ; S2 \quad \{ S.type := \text{if } S1.type = \text{void} \text{ and } S2.type = \text{void} \text{ then void else type\_error } \}</math>  <math>S \rightarrow S1 ; S2 \quad \{ S.type := \text{if } S1.type = \text{void} \text{ and } S2.type = \text{void} \text{ then void else type\_error } \}</math> </p> <p> <b>Type checking of functions</b>  The rule for checking the type of a function application is : <math>E \rightarrow E1 ( E2) \{ E.type := \text{if } E2.type = s \text{ and } E1.type = s \rightarrow t \text{ then } t \text{ else type\_error } \}</math> </p>
3	<p> Explain in detail about the translation of source language details into run time environment. (Page No.473) MAY/JUNE2009 </p> <p> A translation needs to relate the static source text of a program to the dynamic actions that must occur at runtime to implement the program. The program consists of names for procedures, identifiers etc., that require mapping with the actual memory location at runtime. </p> <p> Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system. </p> <p> A program consist of procedures, a procedure definition is a declaration that, in its simplest form, associates an identifier (procedure name) with a statement (body of the procedure). Each execution of procedure is referred to as an activation of the procedure. Lifetime of an activation is the sequence of steps present in the execution of the procedure. If ‘a’ and ‘b’ be two procedures then their activations will be non-overlapping (when one is called after other) or nested (nested procedures). A procedure is recursive if a new activation begins before an earlier activation of the same procedure has ended. An activation tree shows the way control enters and leaves activations. </p> <p> Properties of activation trees are :- </p> <ul style="list-style-type: none"> <li>• Each node represents an activation of a procedure.</li> <li>• The root shows the activation of the main function.</li> <li>• The node for procedure ‘x’ is the parent of node for procedure ‘y’ if and only if the control flows from procedure x to procedure y.</li> </ul>

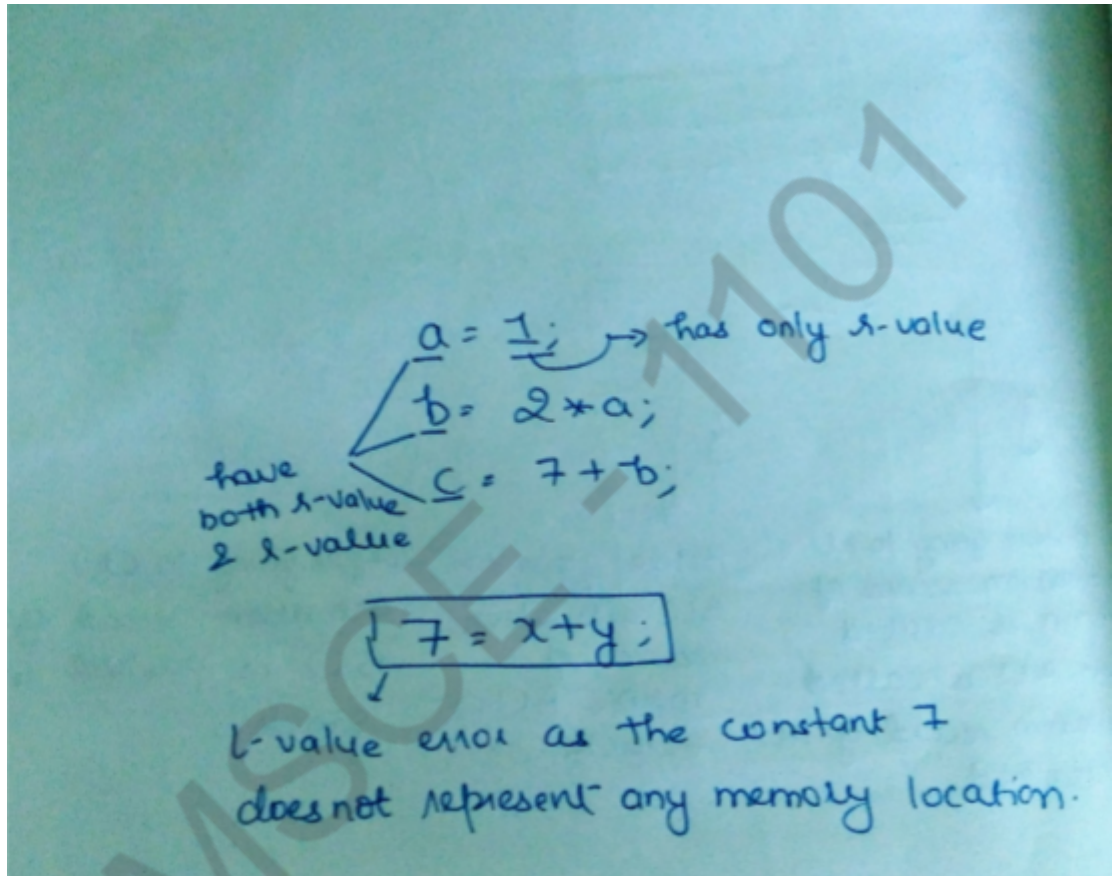
5

Explain about the parameter passing. (Page No.424) APRIL/MAY 2017

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 it appears 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 of the assignment operator.



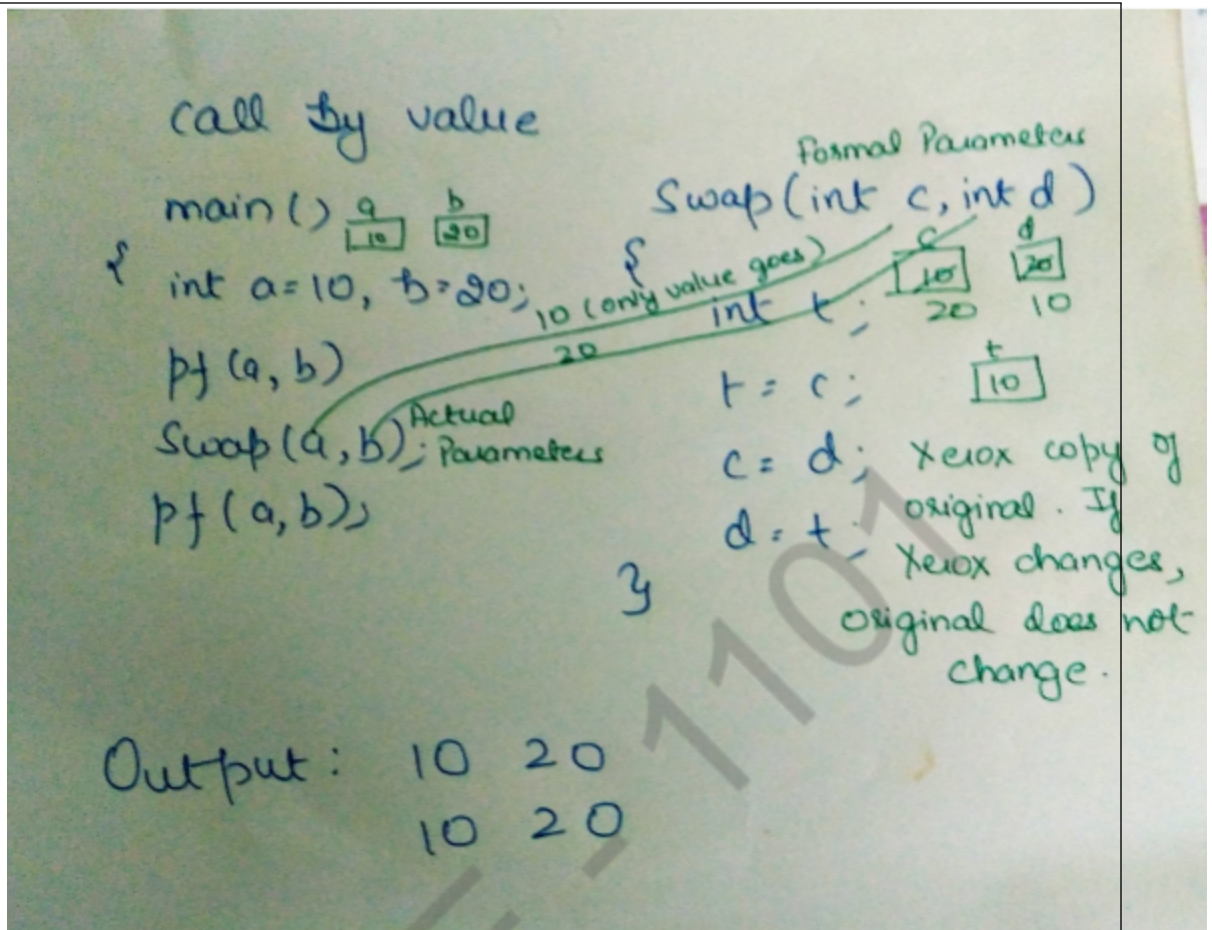
**i. Formal Parameter:** Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

**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**

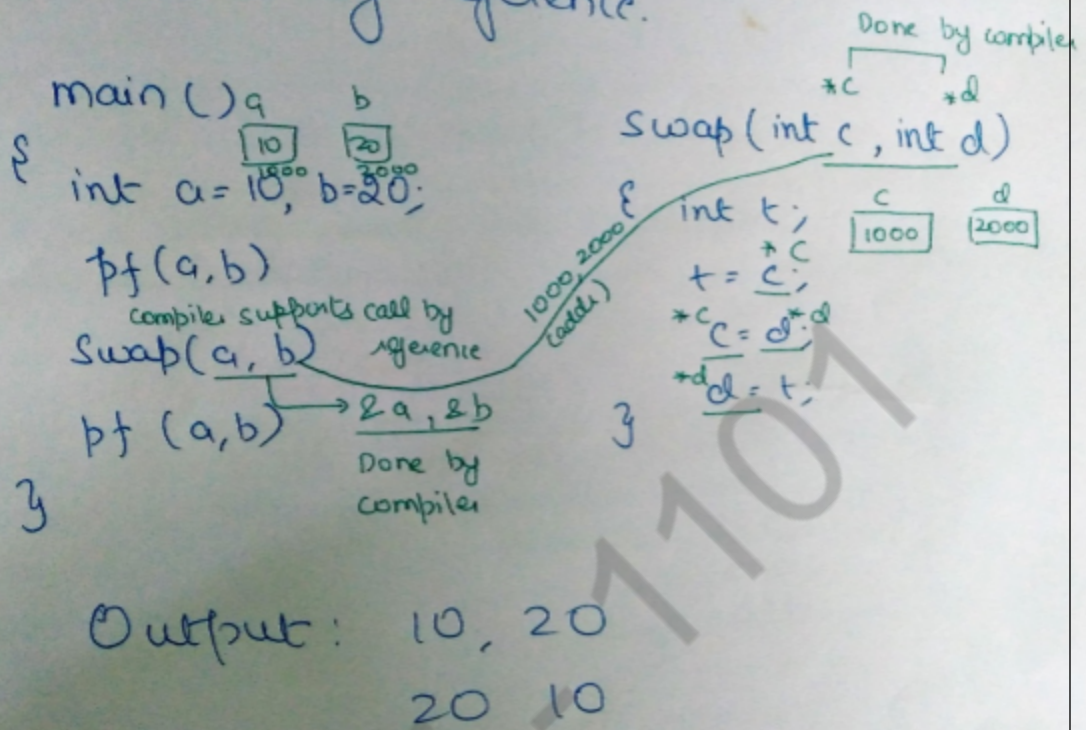
- **Call by Value**

In call by value the calling procedure pass 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 does not affect the actual parameters.



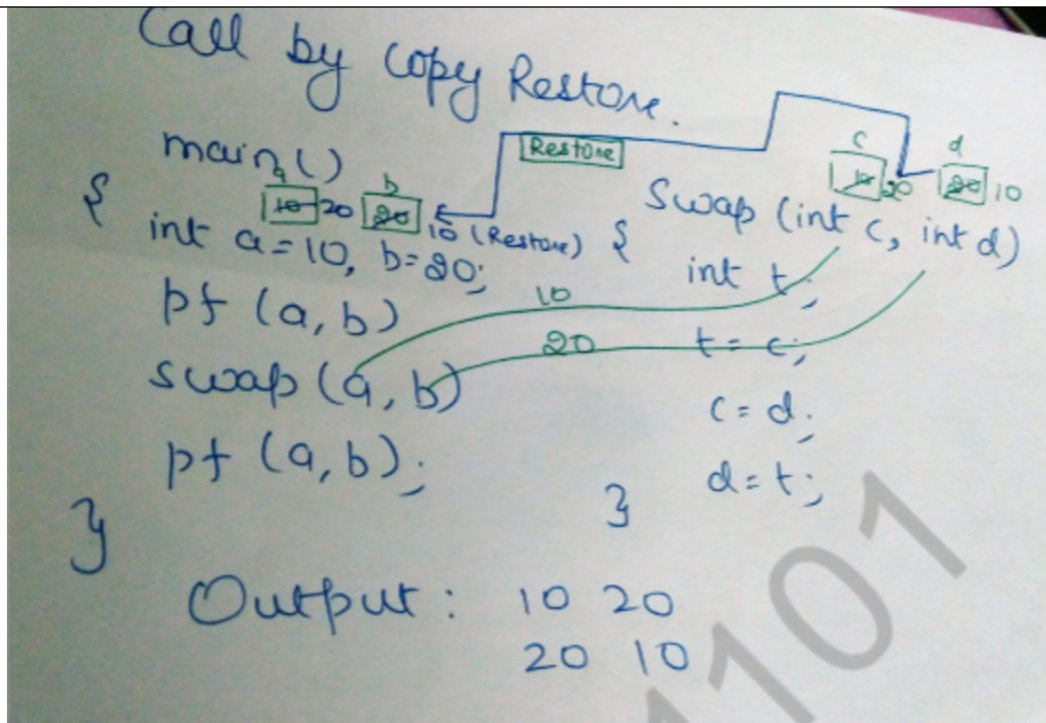
- Call by Reference** In call by reference the formal and actual parameters refers to same memory location. The l-value of actual parameters is copied to the activation record of the called function. Thus the called function has the address of the actual parameters. If the actual parameters does not 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 Reference.



- **Call by Copy Restore**

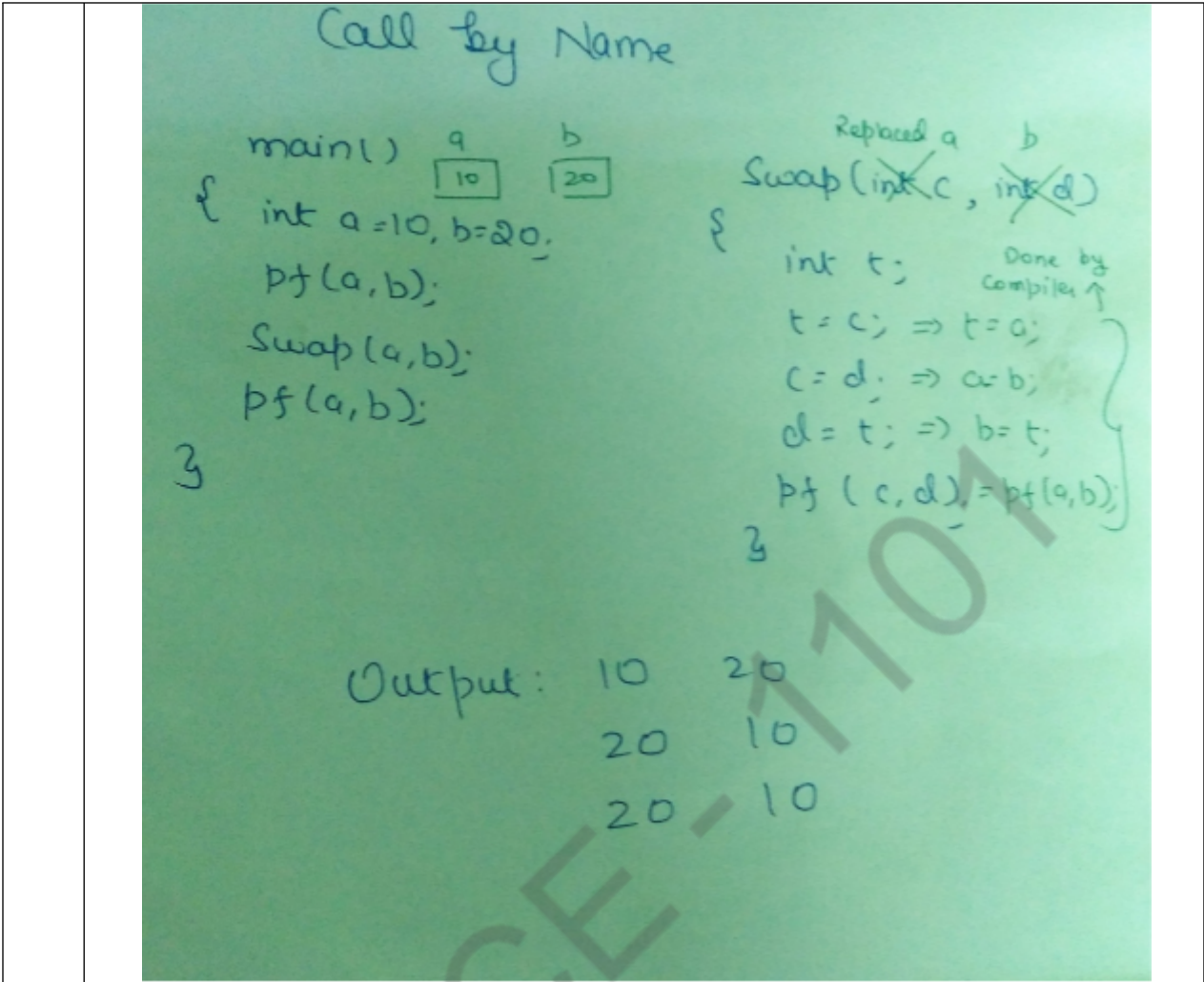
In call by copy restore compiler copies the value in formal parameters when the procedure is called and copy them back in actual parameters when control returns to the called function. The r-values are passed and on return r-value of formals are copied into l-value of actuals.



- **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 Construct a syntax directed definition for constructing a syntax tree for assignment statements  
MAY/JUNE 2016  $S \rightarrow id = E$

$S \rightarrow E$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

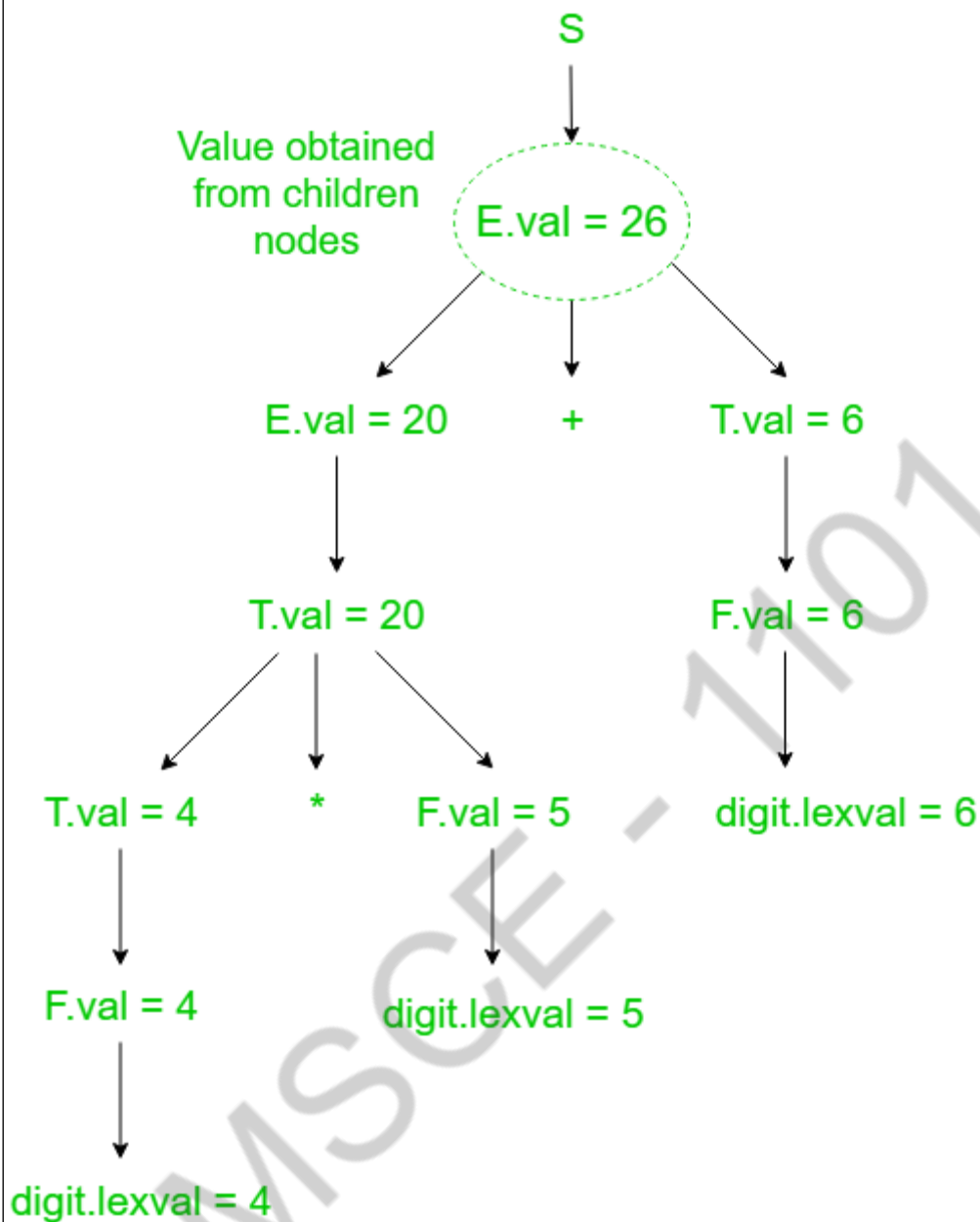
$F \rightarrow \text{digit}$

The SDD for the above grammar can be written as follow

Production	Semantic Actions
$S \rightarrow E$	Print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

Let us assume an input string  $4 * 5 + 6$  for computing synthesized attributes. The annotated parse tree for the input string is

AMSCCE-1101



### Annotated Parse Tree

For computation of attributes we start from leftmost bottom node. The rule  $F \rightarrow digit$  is used to reduce digit to  $F$  and the value of digit is obtained from lexical analyzer which becomes value of  $F$  i.e. from semantic action  $F.val = digit.lexval$ . Hence,  $F.val = 4$  and since  $T$  is parent node of  $F$  so, we get  $T.val = 4$  from semantic action  $T.val = F.val$ . Then, for  $T \rightarrow T_1 * F$  production, the corresponding semantic action is  $T.val = T_1.val * F.val$ . Hence,  $T.val = 4 * 5 = 20$ . Similarly, combination of  $E_1.val + T.val$  becomes  $E.val$  i.e.  $E.val = E_1.val + T.val = 26$ . Then, the production  $S \rightarrow E$  is applied to reduce  $E.val = 26$  and semantic action associated with it prints the result  $E.val$ . Hence, the output will be 26.

**2. Inherited Attributes** – These are the attributes which derive their values from their parent or sibling nodes i.e. value of inherited attributes are computed by value of parent or sibling nodes.

	<p><b>Example:</b>  <math>A \rightarrow BCD \quad \{ C.in = A.in, C.type = B.type \}</math></p> <p><b>Computation of Inherited Attributes –</b></p> <ul style="list-style-type: none"> <li>• Construct the SDD using semantic actions.</li> <li>• The annotated parse tree is generated and attribute values are computed in top down manner.</li> </ul>
7	<p>Write about Bottom-Up evaluation S-Attributed definitions. What is L-attributed definition? Give some example.</p> <p><b>Types of attributes –</b>  Attributes may be of two types – Synthesized or Inherited.</p> <ol style="list-style-type: none"> <li><b>Synthesized attributes –</b>  An Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. The attribute can take value only from its children (Variables in the RHS of the production).  1. For eg. let's say <math>A \rightarrow BC</math> is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes then it will be synthesized attribute.</li> <li><b>Inherited attributes –</b>  An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production).  For example, let's say <math>A \rightarrow BC</math> is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes then it will be inherited attribute.</li> </ol> <p>Now, let's discuss about S-attributed and L-attributed SDT.</p> <ol style="list-style-type: none"> <li><b>S-attributed SDT :</b> <ul style="list-style-type: none"> <li>• If an SDT uses only synthesized attributes, it is called as S-attributed SDT.</li> <li>• S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.</li> <li>• Semantic actions are placed in rightmost place of RHS.</li> </ul> </li> <li><b>L-attributed SDT:</b> <ul style="list-style-type: none"> <li>• If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.</li> <li>• Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.</li> <li>• Semantic actions are placed anywhere in RHS.</li> </ul> </li> </ol> <p>For example,</p> <p><math>A \rightarrow XYZ \quad \{ Y.S = A.S, Y.S = X.S, Y.S = Z.S \}</math></p> <p>is not an L-attributed grammar since <math>Y.S = A.S</math> and <math>Y.S = X.S</math> are allowed but <math>Y.S = Z.S</math> violates the L-attributed SDT definition as attributed is inheriting the value from its right sibling.</p> <p><b>Note –</b> If a definition is S-attributed, then it is also L-attributed but <b>NOT</b> vice-versa.</p>

8

Explain the specification of simple type checker for statements, expressions and functions. (Page No.348) NOV/DEC 2017

A type checker for a simple language checks the type of each identifier. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements and functions.

### A Simple Language

Consider the following grammar:

$$P \rightarrow D ; E$$

$$D \rightarrow D ; D \mid id : T$$

$$T \rightarrow char \mid integer \mid array [ num ] \text{ of } T \mid \uparrow T$$

$$E \rightarrow literal \mid num \mid id \mid E \text{ mod } E \mid E [ E ] \mid E \uparrow$$

Translation scheme:

$$P \rightarrow D ; E$$

$$D \rightarrow D ; D$$

$$D \rightarrow id : T \{ \text{addtype} (id.entry, T.type) \}$$

$$T \rightarrow char \{ T.type := char \}$$

$$T \rightarrow integer \{ T.type := integer \}$$

$$T \rightarrow \uparrow T1 \{ T.type := \text{pointer}(T1.type) \}$$

$$T \rightarrow array [ num ] \text{ of } T1 \{ T.type := \text{array} ( 1 \dots num.val, T1.type) \}$$

In the above language,

→ There are two basic types : char and integer ; → type\_error is used to signal errors;

→ the prefix operator  $\uparrow$  builds a pointer type. Example ,  $\uparrow integer$  leads to the type expression

pointer ( integer ).

### Type checking of expressions

In the following rules, the attribute type for E gives the type expression assigned to the expression generated by E.

1.  $E \rightarrow literal \{ E.type := char \}$   $E \rightarrow num \{ E.type := integer \}$

Here, constants represented by the tokens literal and num have type char and integer.

2.  $E \rightarrow id \{ E.type := \text{lookup} ( id.entry ) \}$

lookup ( e ) is used to fetch the type saved in the symbol table entry pointed to by e.

3.  $E \rightarrow E1 \text{ mod } E2 \{ E.type := \text{if } E1.type = \text{integer and } E2.type = \text{integer}$   
then integer  
else type\_error }

The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is type\_error.

4.  $E \rightarrow E1 [ E2 ] \{ E.type := \text{if } E2.type = \text{integer and } E1.type = \text{array}(s,t)$   
then t  
else type\_error }

In an array reference  $E1 [ E2 ]$ , the index expression  $E2$  must have type integer. The result is the element type  $t$  obtained from the type  $\text{array}(s,t)$  of  $E1$ .

5.  $E \rightarrow E1 \uparrow \{ E.type := \text{if } E1.type = \text{pointer}(t)$  then t  
else type\_error }

The postfix operator  $\uparrow$  yields the object pointed to by its operand. The type of  $E \uparrow$  is the type  $t$  of the object pointed to by the pointer  $E$ .

### **Type checking of statements**

Statements do not have values; hence the basic type void can be assigned to them. If an error is detected within a statement, then type\_error is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:  $S \rightarrow id := E$   
 $S \rightarrow id := E \quad \{ S.type := \text{if } id.type = E.type \text{ then void}$   
else type\_error }

2. Conditional statement:  $S \rightarrow \text{if } E \text{ then } S1$   
 $S \rightarrow \text{if } E \text{ then } S1 \quad \{ S.type := \text{if } E.type = \text{boolean} \text{ then } S1.type$   
else type\_error }

3. While statement:

$S \rightarrow \text{while } E \text{ do } S1$   
 $S \rightarrow \text{while } E \text{ do } S1 \quad \{ S.type := \text{if } E.type = \text{boolean then } S1.type$   
 $\quad \quad \quad \text{else type\_error } \}$

#### 4. Sequence of statements:

$S \rightarrow S1 ; S2 \quad \{ S.type := \text{if } S1.type = \text{void and } S1.type = \text{void then}$   
 $\quad \quad \quad \text{void else type\_error } \}$   
 $S \rightarrow S1 ; S2 \quad \{ S.type := \text{if } S1.type = \text{void and}$   
 $\quad \quad \quad S1.type = \text{void then void}$   
 $\quad \quad \quad \text{else type\_error } \}$

#### Type checking of functions

The rule for checking the type of a function application is :  $E \rightarrow E1 ( E2) \{$   
 $E.type := \text{if } E2.type = s \text{ and}$   
 $\quad \quad \quad E1.type = s \rightarrow t \text{ then } t \text{ else type\_error } \}$

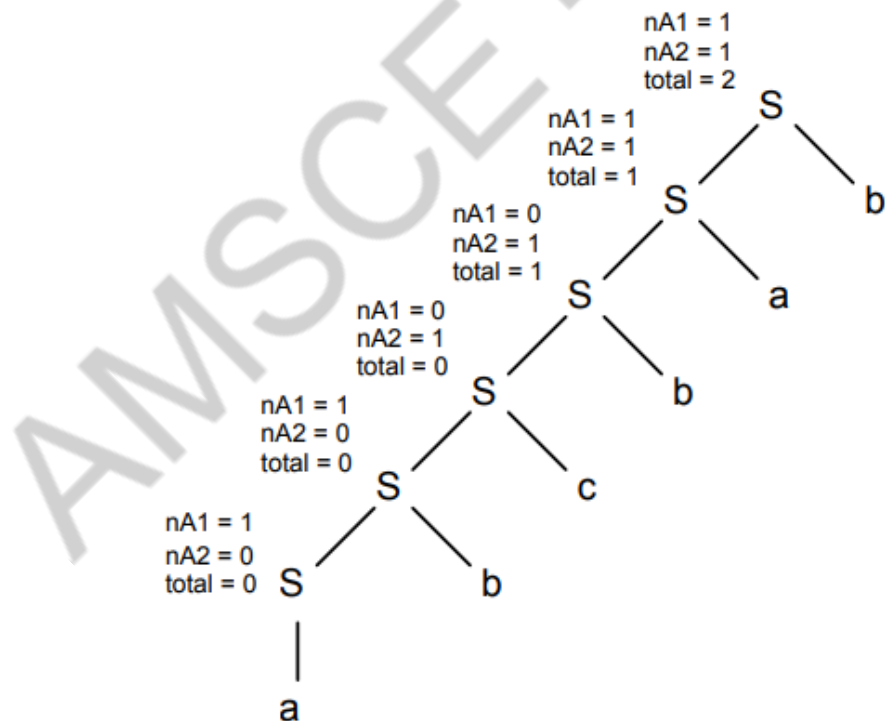
9 A syntax Directed Translation scheme that takes strings of a's , b's and c's as input and produces as output the number of substrings in the input string that corresponds to the pattern  $a(a|b)^*c+(a|b)^*b$ . For example the translation of the input string "abbcabcababc" is "3".

- i) Write a context free grammar that generate all strings of a's, b's and c's.
- ii) Give the semantic attributes for the grammar symbols.
- iii) For each production of the grammar present a set of rules for evaluation of the semantic attributes. (Page No.280) NOV/DEC2016

Your solution should include: a) A context-free grammar that generates all strings of a's, b's and c's b) Semantic attributes for the grammar symbols c) For each production of the grammar a set of rules for evaluation of the semantic attributes d) Justification that your solution is correct. Solution: a) The context-free grammar can be as simple as the one shown below which is essentially a Regular Grammar  $G = \{ \{a,b,c\}, \{S\}, S, P \}$  for all the strings over the alphabet  $\{a,b,c\}$  with P as the set of productions given below.  $S \rightarrow S a$   $S \rightarrow S b$   $S \rightarrow S c$   $S \rightarrow a$   $S \rightarrow b$   $S \rightarrow c$  b) Given the grammar above any string will be parsed and have a parse tree that is left-skewed, i.e., all the branches of the tree are to the left as the grammar is clearly left-recursive. We define three synthesized attributes for the non-

terminal symbol S, namely nA1, nA2 and total. The idea of these attributes is that in the first attribute we will capture the number of a's to the left of a given "c" character, the second attribute, nA2, the number of a's to the right of a given "c" character and the last attributed, total, will accumulate the number of substrings. We need to count the number of a's to the left of a "c" character and to the right of that character so that we can then add the value of nA1 to a running total for each occurrence of a b character to the right of "c" which recording the value of a's to the right of "c" so that when we find a new "c" we copy the value of the "a's" that were to the right of the first "c" and which are now to the left of the second "c".

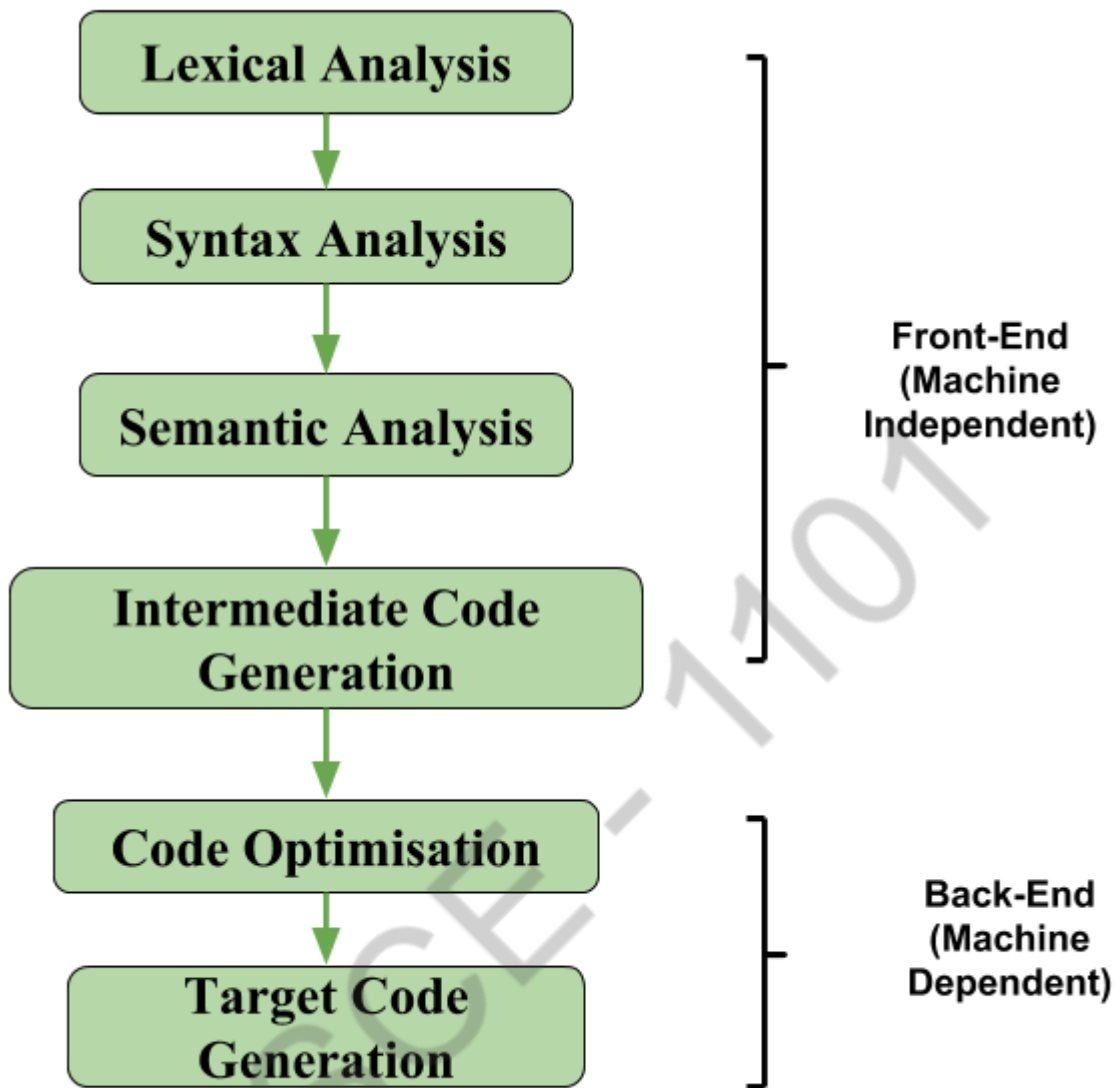
c) As such a set of rules is as follows, here written as semantic actions given that their order of evaluation is done using a bottom-up depth-first search traversal.  $S1 \rightarrow S2$  a {  $S1.nA1 = S2.nA1 + 1$ ;  $S1.nA2 = S2.nA2$ ;  $S1.total = S2.total$ ; }  $S1 \rightarrow S2$  b {  $S1.nA1 = S2.nA1$ ;  $S1.nA2 = S2.nA2$ ;  $S1.total = S2.total + S2.nA2$ ; }  $S1 \rightarrow S2$  c {  $S1.nA1 = 0$ ;  $S1.nA2 = S2.nA1$ ;  $S1.total = S2.total$ ; }  $S1 \rightarrow a$  {  $S1.nA1 = 1$ ;  $S1.nA2 = 0$ ;  $S1.total = 0$ ; }  $S1 \rightarrow b$  {  $S1.nA1 = 0$ ;  $S1.nA2 = 0$ ;  $S1.total = 0$ ; }  $S1 \rightarrow c$  {  $S1.nA1 = 0$ ;  $S1.nA2 = 0$ ;  $S1.total = 0$ ; }



We have two rolling counters for the number of a's one keeping track of the number of a's in the current section of the input string (the sections are delimited by "c" or sequences of "c"s) and the other just saves the number of c's from the previous section. In each section



	<p>we accumulate the number of a's in the previous section for each occurrence of the "b" characters in the current section. At the end of each section we reset one of the counters and save the other counter for the next section.</p>
10	<p>Generate an intermediate code for the following code segment with the required syntax-directed translation scheme.</p> <p>if ( a &gt; b) x = a + b else x = a – b (Page No.303)</p> <p>In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).</p> <p>The benefits of using machine independent intermediate code are:</p> <ul style="list-style-type: none"><li>• Because of the machine independent intermediate code, portability will be enhanced. For example, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.</li><li>• Retargeting is facilitated</li><li>• It is easier to apply source code modification to improve the performance of source code by optimising the intermediate code.</li></ul>



If we generate machine code directly from source code then for  $n$  target machine we will have  $n$  optimisers and  $n$  code generators but if we will have a machine independent intermediate code, we will have only one optimiser. Intermediate code can be either language specific (e.g., Bytecode for Java) or language independent (three-address code).

The following are commonly used intermediate code representation:

1. **Postfix Notation –**

The ordinary (infix) way of writing the sum of  $a$  and  $b$  is with 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 –** The postfix representation of the expression  $(a - b) * (c + d) + (a - b)$  is :  $ab - cd + *ab - +$ .

Read more: [Infix to Postfix](#)

2. **Three-Address Code –**

A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form  $x = y \text{ op } z$ , here  $x, y, z$  will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement.

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

$T_1 = b * c$

$T_2 = a + T_1$

$T_3 = T_2 + d$

$T_1, T_2, T_3$  are temporary variables.

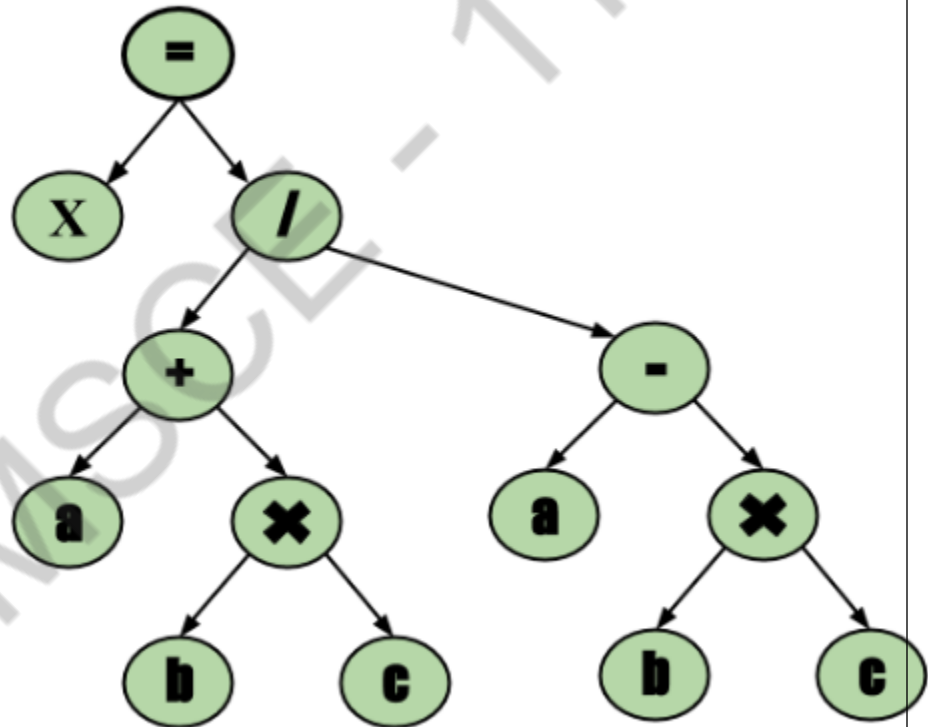
### 3. Syntax Tree –

Syntax tree is nothing more than 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 single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

**Example –**

$x = (a + b * c) / (a - b * c)$

### Operator Root



11 Compare and contrast of static, stack and Heap allocation.

(Page No.401)

Static: Storage can be made by compiler looking only at the text of the program. One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into target code.

Dynamic: Storage can be made by looking at what the program does when the program is

running.

Global constants and other data generated by the compiler(e.g. info to support garbage collection) are allocated static storage. Static variables are bound to memory cells before execution begins and remains bound to the same memory cell throughout execution. E.g., C static variables.

Advantage: efficiency(direct addressing), history-sensitive subprogram support

Disadvantage: lack of flexibility, no recursion if this is the \*only\* kind of variable, as was the case in Fortran

### 3. Heap

Data that may outlive the call to the procedure that created it is usually allocated on a heap. E.g. new to create objects that may be passed from procedure to procedure.

The size of heap can not be determined at compile time. Referenced only through pointers or references, e.g., dynamic objects in C++, all objects in Java

Advantage: provides for dynamic storage management

Disadvantage: inefficient and unreliable

### 4. Stack

Names local to a procedure are allocated space on a stack. The size of stack can not be determined at compile time.

Advantages:

allows recursion

conserves storage

Disadvantages:

Overhead of allocation and deallocation

Subprograms cannot be history sensitive

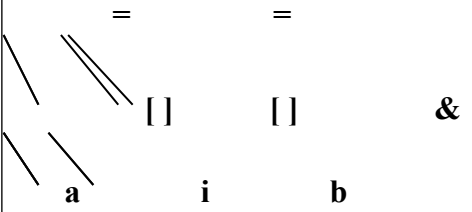
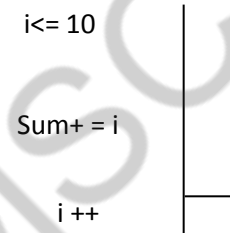
Inefficient references (indirect addressing)

## UNIT V CODE OPTIMIZATION AND CODE GENERATION

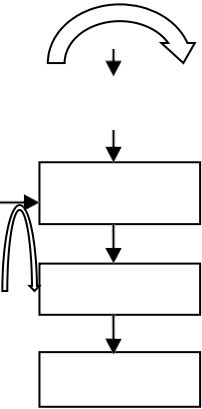
Principal Sources of Optimization-DAG- Optimization of Basic Blocks-Global Data Flow Analysis-Efficient Data Flow Algorithms-Issues in Design of a Code Generator - A Simple Code Generator Algorithm.

S. No.	Question								
1	<p><b>What are basic blocks? <u>NOV/DEC 2011, APRIL/MAY 2005, APRIL/MAY 2010, APRIL/MAY 2008, NOV/DEC 2014 MAY/JUNE 2013, APRIL/MAY 2017</u></b></p> <p>A sequence of consecutive statements which may be entered only at the beginning and when entered are executed in sequence without halt or possibility of branch, are called basic blocks.</p>								
2	<p><b>What do you mean by copy propagation? <u>APRIL/MAY 2017</u></b></p> <p>After the assignment of one variable to another, a reference to one variable may be replaced with the value of the other variable.</p> <p>If <math>w := x</math> appears in a block, all subsequent uses of <math>w</math> can be replaced with uses of <math>x</math>.</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th style="text-align: center;">Before</th> <th style="text-align: center;">After</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;"><math>b := z + y</math></td> <td style="text-align: center;"><math>b := z + y</math></td> </tr> <tr> <td style="text-align: center;"><math>a := b</math></td> <td style="text-align: center;"><math>a := b</math></td> </tr> <tr> <td style="text-align: center;"><math>x := 2 * a</math></td> <td style="text-align: center;"><math>x := 2 * b</math></td> </tr> </tbody> </table>	Before	After	$b := z + y$	$b := z + y$	$a := b$	$a := b$	$x := 2 * a$	$x := 2 * b$
Before	After								
$b := z + y$	$b := z + y$								
$a := b$	$a := b$								
$x := 2 * a$	$x := 2 * b$								
3	<p><b>What is a flow graph? <u>NOV/DEC 2011, MAY/JUNE 2012, NOV/DEC 2014 APRIL/MAY 2008, MAY/JUNE 2013</u></b></p> <p>The basic block and their successor relationships shown</p>								

	by a directed graph is called a flow graph. The nodes of a flow graph are the basic blocks.
4	<p><b>Mention the applications of DAGs. (Or) List the advantages of DAG. <u>NOV/DEC 2012</u></b></p> <p><b><u>MAY/JUNE 2013</u></b></p> <p>We can automatically detect common sub expressions. We can determine the statements that compute the values, which could be used outside the block.</p> <p>We can determine which identifiers have their values used in the block.</p>
5	<p><b>Write the three address code sequence for the assignment statement. <u>MAY/JUNE 2016</u></b></p> <p><math>d := (a-b) + (a-c) + (a-c)</math> <math>t1 = a-b</math>  <math>t2 = a-c</math> <math>t3 = t1 + t2</math> <math>t4 = t3 + t2</math>  <math>d = t4</math></p>
6	<p><b>What is meant by peephole optimization? <u>MAY/JUNE 2007</u></b></p> <p>Peephole optimization is a technique used in many compilers, in connection with the optimization of either intermediate or object code. It is really an attempt to overcome the difficulties encountered in syntax directed generation of code.</p>
7	<p><b>What are the issues in the design of code generators? <u>NOV/DEC 2007</u></b></p> <p>Input to the code generator Target programs  Memory management Instruction selection Register allocation  Choice of evaluation order Approaches to code generation</p>
8	<p><b>What is register descriptor and address descriptor? <u>NOV/DEC 2012</u></b></p> <p>A register descriptor keeps track of what is currently in each register.  An address descriptor keeps track of the location where the current value of the name can be found at run time.</p>
9	<p><b>Define DAG. <u>NOV/DEC 2007, MAY/JUNE 2007</u></b></p> <p>A DAG for a basic block is a directed acyclic graph with the following labels on nodes:</p> <p>Leaves are labeled by unique identifiers, either variable names or constants.</p> <p>Interior nodes are labeled by an operator symbol.</p> <p>Nodes are also optionally given a sequence of identifiers for labels.</p>
10	<p><b>Name the techniques in Loop optimization. <u>MAY/JUNE 2014</u></b></p>

	Code Motion, Induction variable elimination, Reduction in strength
11	<p><b>Draw DAG to represent <math>a[i]=b[i]; a[i]=\&amp;t</math>; <u>NOV/DEC 2014</u></b></p> 
12	<p><b>Represent the following in flow graph <u>NOV/DEC 2014</u></b></p> <pre>i=1; sum=0; while (i&lt;=10){sum+=i;i++;}</pre> <p>i = 1 Sum =</p> 



	
13	<p><b>How to perform register assignment for outer loops? <u>MAY/JUNE 2012</u></b>  Outer loop <math>L_1</math> contains an inner loop <math>L_2</math> names allocated registers in <math>L_2</math> need not be allocated registers in <math>L_1 - L_2</math></p>
14	<p><b>Define local optimization. <u>APRIL/MAY 2011</u></b>  The optimization performed within a block of code is called a local optimization.</p>
15	<p><b>Define constant folding. <u>MAY/JUNE 2013</u></b>  Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.</p>
16	<p><b>What is code motion? <u>APRIL/MAY 2004, MAY/JUNE 2007, APRIL/MAY-2008</u></b>  Code motion is an important modification that decreases the amount of code in a loop.</p>
17	<p><b>What are the properties of optimizing compilers? <u>MAY/JUNE 2016, MAY/JUNE 2013, NOV/DEC 2007, NOV/DEC 2017</u></b>  Transformation must preserve the meaning of programs. Transformation must, on the average, speed up the programs by a measurable amount  A Transformation must be worth the effort.  The source code should be such that it should produce minimum amount of target code.  There should not be any unreachable code.  Dead code should be completely removed from source language.</p>
18	



**Define Local transformation & Global Transformation.**

	<p>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.</p>
19	<p><b>What is meant by Common Sub-expressions?</b> 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.</p>
20	<p><b>What is meant by Dead Code? Or Define Live variable?</b><u>APRIL/MAY 2011, NOV/DEC 2012</u></p> <p>A variable is live at a point in a program if its value can be used subsequently otherwise, it is dead at that point. The statement that computes values that never get used is known Dead code or useless code. .</p>
21	<p><b>What is meant by Reduction in strength?</b> Reduction in strength is the one which replaces an expensive operation by a cheaper one such as a multiplication by an addition</p>
22	<p><b>What is meant by loop invariant computation?</b> An expression that yields the same result independent of the number of times the loop is executed is known as loop invariant computation.</p>
23	<p><b>Define data flow equations.</b> A typical equation has the form <math>Out[S] = gen[S] \cup (In[S] - kill[S])</math> and can be read as, “the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement”. Such equations are called data flow equations.</p>
24	<p><b>When is a flow graph reducible?</b> <u>APRIL/MAY 2012 MAY/JUNE 2012</u></p> <p>A flow graph is reducible if and only if we can partition the edges into two disjoint groups often called the forward edges and back edges.</p>
25	<p><b>What is induction variable?</b> A variable is called an induction variable of a loop if every time the variable changes values, it is incremented or decremented by some constant.</p>
26	<p><b>What is a cross compiler?</b> <u>NOV/DEC 2007, MAY/JUNE 2014</u></p>

	<p><b>A cross compiler</b> is a <a href="#">compiler</a> capable of creating <a href="#">executable</a> code for a <a href="#">platform</a> other than the one on which the compiler is run. (ie). A compiler may run on one machine and produce target code for another machine.</p>
27	<p><b>What is global data flow analysis?</b>  <u>NOV/DEC 2014</u></p> <p>It is a process in which the values are computed using data flow properties. They are available expressions, reaching definition, live variable and busy variable.</p>
28	<p><b>How would you represent the dummy blocks with no statements indicated in global data flow analysis?</b>  <u>MAY/JUNE 2014</u></p> <p>Refer notes</p>
29	<p><b>What is the use of algebraic identities in optimization of basic blocks?</b>  <u>MAY/JUNE 2012</u></p> <p>The algebraic identities are used in Peephole optimization techniques.</p> <p>Simple transformations can be applied on the code in order to optimize it <b>for ex:</b> <math>2*a</math> optimized to <math>a + a</math>.</p>
30	<p><b>List the characteristics of peephole optimization.</b>  <u>NOV/DEC 2016</u></p> <ul style="list-style-type: none"> <li>· Redundant instruction elimination</li> <li>· Flow of control optimization</li> <li>· Algebraic simplification</li> <li>· Use of machine idioms</li> </ul>
31	<p><b>Define code generations?</b></p> <p>It is the final phase in compiler model and it takes as an input an intermediate representation of the source program and output produces as equivalent target programs. Then intermediate instructions are each translated into a sequence of machine instructions that perform the same task.</p>

	41	<b>Define Dead-code elimination with ex.</b> It is defined as the process in which the statement $x=y+z$ appear in a basic block, where $x$ is never subsequently used. Then this statement may be safely removed without changing the value of basic blocks.													
32		<b>Give the variety of forms in target program.</b> Absolute machine language. Relocatable machine language. Assembly language.													
33		<b>Give the factors of instruction selections.</b> Uniformity and completeness of the instruction sets Instruction speed and machine idioms Size of the instruction sets.													
	42	<b>Define Renaming of temporary variables with ex.</b> We have the statement $u:=b+c$ , where $u$ is a new temporary variable, and change all uses of this instance of $t$ to $u$ , then the value of the basic block is not changed.													
34		<b>What are the sub problems in register allocation strategies?</b> During register allocation, we select the set of variables that will reside in register at a point in the program. During a subsequent register assignment phase, we pick the specific register that a variable reside in.													
	43	<b>Prepare the total cost of the following target code. MOV a, R0 ADD b, R0 MOV C, R0 ADD R0,R1 MOV R1,X</b>													
35		<b>Write the steps to partition a sequence of 3 address statements into basic blocks.</b> 1. First determine the set of leaders, the first statement of basic blocks. The rules we can use are the following: The first statement is a leader. Any statement that is the target of a conditional or unconditional goto is a leader. Any statement that immediately follows a goto or conditional goto statement is a leader. 2. For each leader, its basic blocks consists of the leader and all statements up to but not including the next leader or the end of the program.													
	44	<b>Define code optimization and optimizing compiler</b> The term <b>code-optimization</b> refers to techniques a compiler can employ in an attempt to produce a better object language program than the most obvious for a given source program. Compilers that apply code-improving transformations are called <b>Optimizing compilers</b>													
36		<b>Write the code sequence for the <math>d:=(a-b)+(a-c)+(a-c)</math>.</b>													
	45	<table border="1"> <thead> <tr> <th>Statement</th> <th>Code generation</th> <th>Register descriptor</th> <th>Address descriptor</th> </tr> </thead> <tbody> <tr> <td><math>t:=a-b</math></td> <td>MOV a,R0 SUB b,R0</td> <td>R0 contains t</td> <td>t in R0</td> </tr> <tr> <td><math>u:=a-c</math></td> <td>MOV a,R1 SUB c,R1</td> <td>R0 contains t R1 contains u</td> <td>t in R0 u in R1</td> </tr> </tbody> </table> A DAG for a basic block is a directed acyclic graph with the following Labels on nodes: Leaves are labeled by unique identifiers, either variable names or constants. Interior nodes are labeled by an operator symbol. Nodes are also optionally given a sequence of identifiers for labels.	Statement	Code generation	Register descriptor	Address descriptor	$t:=a-b$	MOV a,R0 SUB b,R0	R0 contains t	t in R0	$u:=a-c$	MOV a,R1 SUB c,R1	R0 contains t R1 contains u	t in R0 u in R1	
Statement	Code generation	Register descriptor	Address descriptor												
$t:=a-b$	MOV a,R0 SUB b,R0	R0 contains t	t in R0												
$u:=a-c$	MOV a,R1 SUB c,R1	R0 contains t R1 contains u	t in R0 u in R1												
	46	<b>What are the different data flow properties?</b> Available expressions Reaching definitions Live variables Busy variables													
	47	<b>What do you mean by machine dependent and machine independent optimization?</b>													
		<table border="1"> <tbody> <tr> <td><math>d:=v+u</math></td> <td>ADD R1,R0</td> <td>R0 contains d</td> <td>d in R0</td> </tr> <tr> <td></td> <td>MO V R0,d</td> <td>d</td> <td>R0 and memory</td> </tr> </tbody> </table>	$d:=v+u$	ADD R1,R0	R0 contains d	d in R0		MO V R0,d	d	R0 and memory					
$d:=v+u$	ADD R1,R0	R0 contains d	d in R0												
	MO V R0,d	d	R0 and memory												
37		<b>Write the global data flow equation</b> Data-flow information can be collected by setting up and solving systems of													

	<p>The machine dependent optimization is based on the characteristics of the target machine for the instruction set used and addressing modes used for the instructions to produce the efficient target code.</p> <p>The machine independent optimization is based on the characteristics of the programming languages for appropriate programming structure and usage of efficient arithmetic properties in order to reduce the execution time.</p>						
48	<p><b>What are the basic goals of code movement?</b></p> <ul style="list-style-type: none"> <li>• To reduce the size of the code i.e. to obtain the space complexity.</li> <li>• To reduce the frequency of execution of code i.e. to obtain the time complexity.</li> </ul>						
49	<p><b>How do you calculate the cost of an instruction?</b></p> <p>The cost of an instruction can be computed as one plus cost associated with the source and destination addressing modes given by added cost.</p> <table style="margin-left: 40px;"> <tr> <td>MOV R0,R1</td> <td style="text-align: right;">1</td> </tr> <tr> <td>MOV R1,M</td> <td style="text-align: right;">2</td> </tr> <tr> <td>SUB 5(R0),*10(R1)</td> <td style="text-align: right;">3</td> </tr> </table>	MOV R0,R1	1	MOV R1,M	2	SUB 5(R0),*10(R1)	3
MOV R0,R1	1						
MOV R1,M	2						
SUB 5(R0),*10(R1)	3						
50	<p style="text-align: center;"><b>Identify the constructs for optimization in basic blocks. <u>NOV/DEC 2016</u></b></p> <ul style="list-style-type: none"> <li>➤ It is a linear piece of code.</li> <li>➤ Analyzing and optimizing is easier.</li> <li>➤ Has local scope - and hence effect is limited.</li> <li>➤ Substantial enough, not to ignore it.</li> <li>➤ Can be seen as part of a larger (global) optimization problem.</li> </ul>						
1	<p>i)What are the issues in design of a code generator? Explain in detail. (PageNo:506)  <u>MAY/JUNE 2016, NOV/DEC 2007, Nov/Dec 2011, April/May 2012 , MAY/JUNE 2007</u>  <u>APRIL/MAY 2005 APRIL/MAY 2008,MAY/JUNE 2012, NOV/DEC 2016,</u>  <u>APRIL/MAY 2017, NOV/DEC 2017</u></p> <p><b>Code generator</b> converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.</p> <p><b>The following issue arises during the code generation phase:</b></p> <ol style="list-style-type: none"> <li>1. <b>Input to code generator –</b>  The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects</li> </ol>						

denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

2. **Target program –**

The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.

- Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
- Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.
- Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.

3. **Memory Management –**

Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

4. **Instruction selection –**

Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

For example, the respective three-address statements would be translated into the latter code sequence as shown below:

P:=Q+R

S:=P+T

MOV Q, R0

ADD R, R0

MOV R0, P

MOV P, R0

ADD T, R0

MOV R0, S

Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

5. **Register allocation issues –**

Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:

- During **Register allocation** – we select only those set of variables that will reside in the registers at each point in the program.

- During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example

M a, b

These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

## 2. **Evaluation order –**

The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.

## 3. **Approaches to code generation issues:** Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:

- Correct
- Easily maintainable
- Testable
- Efficient

(ii) Define basic block. Write an algorithm to partition a sequence of three-address statements into basic blocks. (Page No:528)

MAY/JUNE 2012, APRIL/MAY 2011, APRIL/MAY 2012

**Basic Block** is a straight line code sequence which has no branches in and out branches except to the entry and at the end respectively. Basic Block is a set of statements which always executes one after other, in a sequence.

The first task is to partition a sequence of three-address code into basic blocks. A new basic block is begun with the first instruction and instructions are added until a jump or a label is met. In the absence of jump control moves further consecutively from one instruction to another. The idea is standardized in the algorithm below:

### **Algorithm:**

Partitioning three-address code into basic blocks.

**Input:** A sequence of three address instructions.

**Process:** Instructions from intermediate code which are leaders are determined. Following are the rules used for finding leader:

1. The first three-address instruction of the intermediate code is a leader.
2. Instructions which are targets of jump or conditional jump are leaders.
3. Instructions which immediately follows jump are considered as leaders.

For each leader thus determined its basic block contains itself and all instructions up to excluding the next leader.

### **Example:**

Intermediate code to set a 10\*10 matrix to an identity matrix:

- 1)  $i=1$
- 2)  $j=1$
- 3)  $t1 = 10 * i$
- 4)  $t2 = t1 + j$
- 5)  $t3 = 8 * t2$

	<p>6) <math>t4 = t3 - 88</math>  7) <math>a[t4] = 0.0</math>  8) <math>j = j + 1</math>  9) if <math>j \leq</math> goto (3)  10) <math>i = i + 1</math>  11) if <math>i \leq 10</math> goto (2)  12) <math>i = 1</math>  13) <math>t5 = i - 1</math>  14) <math>t6 = 88 * t5</math>  15) <math>a[t6] = 1.0</math>  16) <math>i = i + 1</math>  17) if <math>i \leq 10</math> goto (13)</p> <p>The given algorithm is used to convert a matrix into identity matrix i.e. a matrix with all diagonal elements 0 and all other elements as 1. Steps (3)-(6) are used to make elements 0 and step(14) is used to make an element 1. These steps are used recursively by goto.</p>
2	<p>(i) Explain in the DAG representation of the basic block with example. (Page. No. 598)  <u>APRIL/MAY 2012 APRIL/MAY 2005, APRIL/MAY 2008, MAY/JUNE 2012,</u>  <u>MAY/JUNE 2015,</u>  <u>APRIL/MAY 2017</u></p> <p>(ii) How to generate a code for a basic block from its dag representation? Explain.  (Page No: 546)  <u>APRIL/MAY 2011, NOV/DEC 2011</u></p>
3	<p>(i) Explain the structure-preserving transformations for basic blocks. (Page No:530 )  <u>NOV/DEC 2011</u></p> <p>There are two types of basic block optimizations. They are :</p> <ul style="list-style-type: none"> <li>Ø Structure-Preserving Transformations</li> <li>Ø Algebraic Transformations</li> </ul> <p><b>Structure-Preserving Transformations:</b>  The primary Structure-Preserving Transformation on basic blocks are:</p> <ul style="list-style-type: none"> <li>Ø Common sub-expression elimination</li> <li>Ø Dead code elimination</li> <li>Ø Renaming of temporary variables</li> <li>Ø Interchange of two independent adjacent statements.</li> </ul> <p><b>Common sub-expression elimination:</b></p>

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced.

Example:

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

The 2nd and 4th statements compute the same expression: b+c and a-d

Basic block can be transformed to

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

#### **Dead code elimination:**

It is possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program - once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

#### **Renaming of temporary variables:**

A statement  $t:=b+c$  where  $t$  is a temporary name can be changed to  $u:=b+c$  where  $u$  is another temporary name, and change all uses of  $t$  to  $u$ . In this a basic block is transformed to its equivalent block called normal-form block.

#### **Interchange of two independent adjacent statements:**

- Two statements

```
t1:=b+c  
t2:=x+y
```

can be interchanged or reordered in its computation in the basic block when value of  $t1$  does not affect the value of  $t2$ .



## Algebraic Transformations:

Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength. Another 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.

The relational operators  $\leq$ ,  $\geq$ ,  $<$ ,  $>$ ,  $+$  and  $=$  sometimes generate unexpected common sub expressions. Associative laws may also be applied to expose common sub expressions. For example, 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
```

Example:

$x:=x+0$  can be removed

$x:=y**2$  can be replaced by a cheaper statement  $x:=y*y$

The compiler writer should examine the language specification carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate  $x*y-x*z$  as  $x*(y-z)$  but it may not evaluate  $a+(b-c)$  as  $(a+b)-c$ .

(ii) Explain the simple code generation algorithm in detail. (Page No.535)

APRIL/MAY 2012, MAY/JUNE 2013, MAY/JUNE 2016

APRIL/MAY 2008 April/May 2011, NOV/DEC 2011, NOV/DEC 2012, MAY/JUNE 2012,

Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

Example:

Consider the three address statement  $x:= y + z$ . It can have the following sequence of codes:

- 1.
- 2.
- 3.
- 4.

AMSCCE - 1101

$v := t + u$

ADD R1, R0

R0 contains v

u in R1

	$d := v + u$	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory
--	--------------	-------------------------	---------------	-------------------------------

AMSCSE-1101

4

For the statement given, write three address statements and construct DAG. MAY/JUNE 2013

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

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

#
(0)
(1)
(2)
(3)
(4)
(5)

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

### Triples representation

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

List of pointers to table

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

### Indirect Triples representation

5 Explain the principle sources of code optimization in detail. (Page No:592 )  
MAY/JUNE 2016

NOV/DEC 2011, MAY/JUNE 2012 ,MAY/JUNE 2007

,MAY/JUNE 2009 ,APRIL/MAY 2008, APRIL/MAY 2005, NOV/DEC 2014

MAY/JUNE 2013 MAY/JUNE

2012, NOV/DEC 2017

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

The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

\*\*\*

### **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

```
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:

- Ø Code motion, which moves code outside a loop;
- Ø Induction-variable elimination, which we apply to replace variables from inner loop.
  
- Ø Reduction in strength, which replaces an expensive operation by a cheaper one, such as a multiplication by an addition.

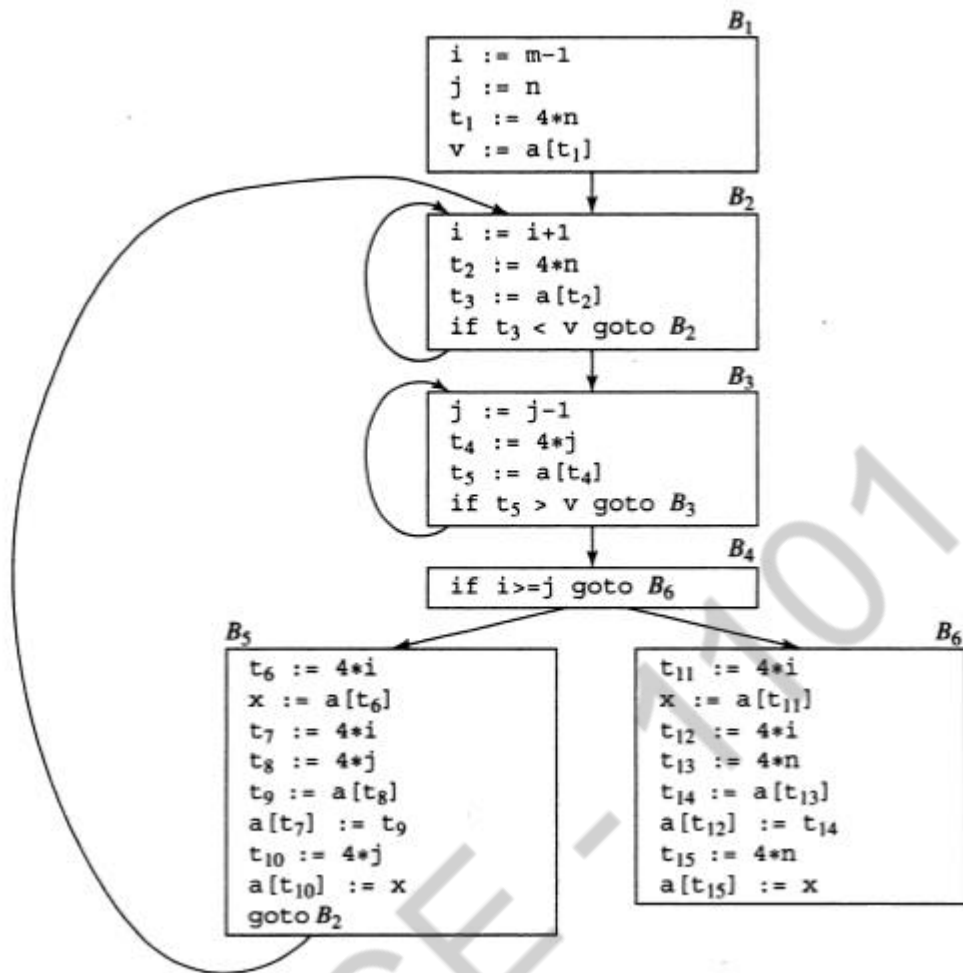


Fig. 5.2 Flow graph

**Code Motion:**

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2) /* statement does not change limit*/
```

Code motion will result in the equivalent of

```
t = limit-2;
while (i <= t) /* statement does not change limit or t */
```

### **Induction Variables :**

Loops are usually processed inside out. For example consider the loop around B3. Note that the values of  $j$  and  $t4$  remain in lock-step; every time the value of  $j$  decreases by 1, that of  $t4$  decreases by 4 because  $4*j$  is assigned to  $t4$ . Such identifiers are called induction variables.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig.5.3 we cannot get rid of either  $j$  or  $t4$  completely;  $t4$  is used in B3 and  $j$  in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually  $j$  will be eliminated when the outer loop of B2- B5 is considered.

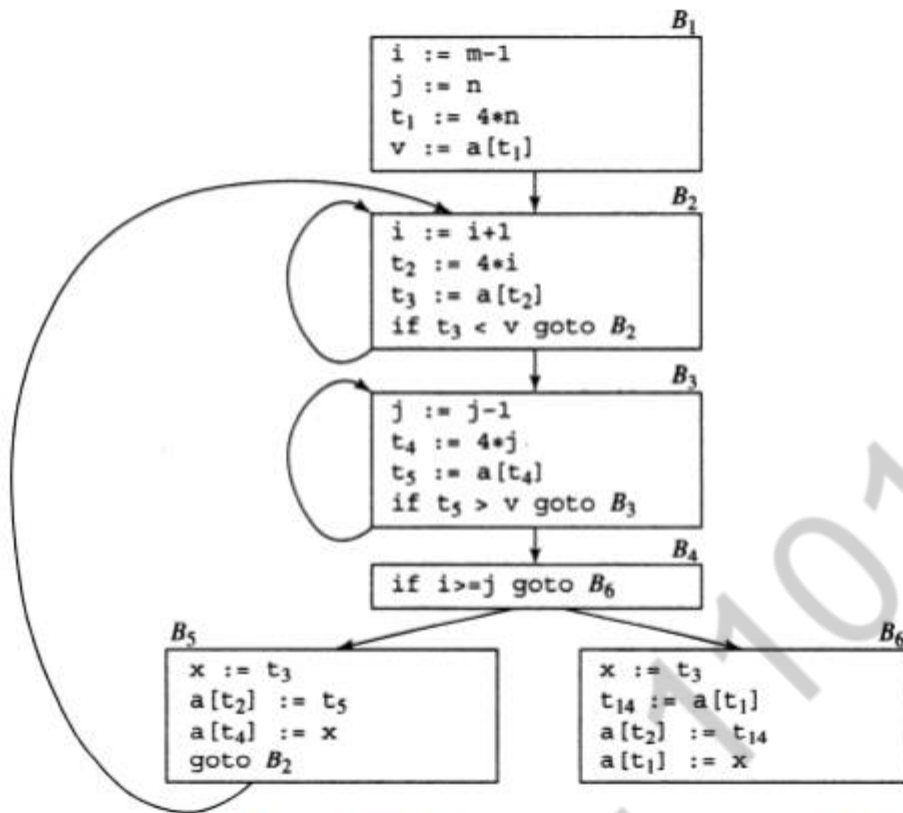
Example:

As the relationship  $t4:=4*j$  surely holds after such an assignment to  $t4$  in Fig. and  $t4$  is not changed elsewhere in the inner loop around B3, it follows that just after the statement  $j:=j-1$  the relationship  $t4:=4*j-4$  must hold. We may therefore replace the assignment  $t4:=4*j$  by  $t4:=t4-4$ . The only problem is that  $t4$  does not have a value when we enter block B3 for the first time. Since we must maintain the relationship  $t4=4*j$  on entry to the block B3, we place an initialization of  $t4$  at the end of the block where  $j$  itself is initialized, shown by the dashed addition to block B1 in Fig.5.3.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

### **Reduction In Strength:**

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example,  $x^2$  is invariably cheaper to implement as  $x*x$  than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.



**Fig. 5.3 B5 and B6 after common subexpression elimination**

**Fig. 5.3 B5 and B6 after common subexpression elimination**

6	<p>(i) Write about Data Flow Analysis of structural programs. (Page No:611 )  <u>NOV/DEC 2011, APRIL/MAY 2012, MAY/JUNE 2014 MAY/JUNE 2013</u>  <u>MAY/JUNE 2012</u></p> <p>(ii) . (Page No.598)  <u>NOV/DEC 2011, MAY/JUNE 2009, MAY/JUNE 2014, NOV/DEC 2014,</u>  <u>APRIL/MAY 2017</u></p>
7	<p>(i) Write an algorithm to construct the natural loop of a back edge. (Page No:604 )  <u>APRIL/MAY 2011</u></p> <p>(ii) Explain in detail about code-improving transformations. (Page No:633)  <u>APRIL/MAY 2011</u></p>

8

(i) Discuss in detail about global data flow analysis. (Page No:608) NOV/DEC 2016

(ii) Explain three techniques for loop optimization with examples.

(Page No:633) NOV/DEC 2012, MAY/JUNE 2013, MAY/JUNE

**Loop Optimization** is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops.

*Loop Optimization is a machine independent optimization.*

Decreasing the number of instructions in an inner loop improves the running time of a program even if the amount of code outside that loop is increased.

**Loop Optimization Techniques:**

1. **Frequency Reduction (Code Motion):**

In frequency reduction, the amount of code in loop is decreased. A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop.

**Example:**

**Initial code:**

```
while(i<100)
{
a = Sin(x)/Cos(x) + i;
i++;
}
```

**Optimized code:**

```
t = Sin(x)/Cos(x);
while(i<100)
{
a = t + i;
i++;
}
```

2. **Loop Unrolling:**

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

**Example:**

**Initial code:**

```
for (int i=0; i<5; i++)
printf("Pankaj\n");
```

**Optimized code:**

```
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
```

```
printf("Pankaj\n");
```

3. **Loop Jamming:**

Loop jamming is the combining the two or more loops in a single loop. It reduces the time taken to compile the many number of loops.

**Example:**

**Initial Code:**

```
for(int i=0; i<5; i++)  
    a = i + 5;  
for(int i=0; i<5; i++)  
    b = i + 10;
```

**Optimized code:**

```
for(int i=0; i<5; i++)  
{  
    a = i + 5;  
    b = i + 10;  
}
```

AMSCCE - 1101

2015

9

(i) Write an algorithm for constructing natural loop of a back edge.

(Page No:604 ) NOV/DEC 2016

(ii) Explain any four issues that crop up when designing a code generator (PageNo:506)

**Code generator** converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.

**The following issue arises during the code generation phase:**

1. **Input to code generator –**

The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

2. **Target program –**

The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.

- Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
- Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.
- Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.

3. **Memory Management –**

Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

4. **Instruction selection –**

Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

For example, the respective three-address statements would be translated into the latter code sequence as shown below:

P:=Q+R

S:=P+T

MOV Q, R0

ADD R, R0

MOV R0, P



MOV P, R0

ADD T, R0

MOV R0, S

Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

5. **Register allocation issues –**

Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:

- During **Register allocation** – we select only those set of variables that will reside in the registers at each point in the program.
- During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example

M a, b

These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

2. **Evaluation order –**

The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.

3. **Approaches to code generation issues:** Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:

- Correct
- Easily maintainable
- Testable
- Efficient

10

(i). Explain in detail about optimization of basic blocks. (Page No.598)

(ii). Construct the DAG for the following Basic block & explain it.

1 Example:  $t1 := a*a$   $t2 := a*b$   $t3 := 2*t2$   $t4 := t1+t3$   $t5 := b*b$   $t6 := t4+t5$

$x := y + z$  • Definition of variables (x) • Use of variables (y, z) • Live variables -- if variable will be used later  
Definition: LEADER instructions are instructions that: 1. First statement in the program 2. Any statement that is a target of a branch 3. Any statement that follows a branch

Algorithm for Partitioning in BB

Input: Program a sequence of 3 address statements Output: List of BB (sequence of instructions that belong to BB) 1. The set of LEADERS (initial instructions in BB) 2. For all x in LEADERS, the set BLOCK(x) of all instructions in the BB

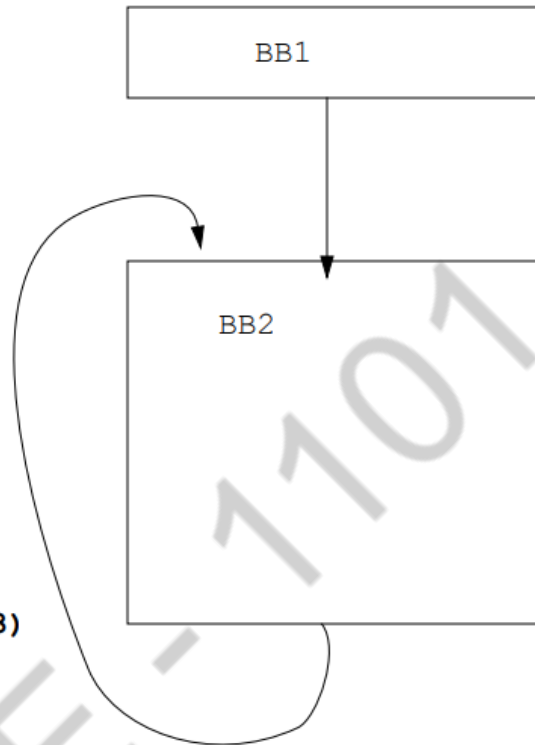
Begin

LEADERS := {1} for j:= 1 to |PROG| do if INST(j) is a branch then add index of branch target to LEADERS endif endfor  
TODO := LEADERS While TODO != O x:= smallest indexed element from TODO TODO := TODO -{x} BLOCK(x):={x} for i:= x+1 to |PROG| and While i!= LEADER do BLOCK(x) := BLOCK(x) +{i} endfor endwhile

---

(1) prod:= 0 -->> L1  
(2) i:=1

(3) t1:= 4\*i --> L2  
(4) t2:= a[t1]  
(5) t3:= 4\*i  
(6) t4:= b[t3]  
(7) t5:= t2\*t4  
(8) t6:= prod+t5  
(9) prod=t6  
(10) t7:= i+1  
(11) i:=y7  
(13) if i<=20 goto (3)



AMSCEN-1101

	<b>RFER NOTES</b>
13	<p>Create DAG and three – address code for the following C program.</p> <pre> i = 1; s = 0; while ( i&lt;= 10) { s = s+ a[i] [i]; i = i + 1; } </pre> <p><b>RFER NOTES</b></p>
14	<p>(i).Identify the optimization techniques applied on procedure calls? Explain with example. (Page No:633)</p> <p>(ii).Describe the concepts of Efficient Data flow algorithms. (Page No:597)</p>
15	<p>(i).Describe the common examples of function preserving transformations and loop optimization process? (Page No:586)</p> <p>(ii).List the types of optimization. (Page No:583)</p>