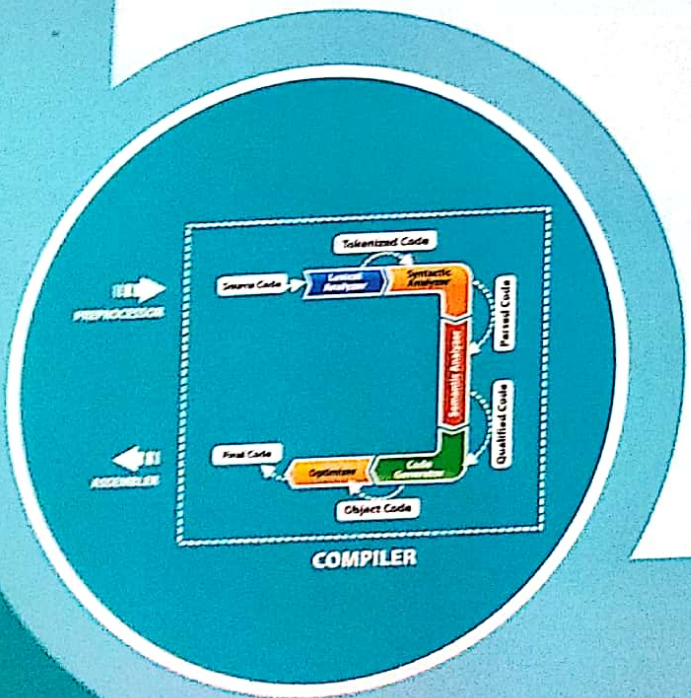
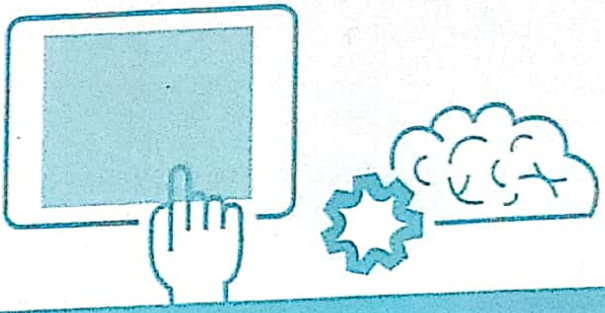
GATE
PSUs

MADE EASY Publications

POSTAL
STUDY PACKAGE

COMPUTER SCIENCE & IT





2020

THEORY BOOK

Compiler Design

Well illustrated theory with solved examples

Contents

Compiler Design

Compiler Design

GOAL OF THE SUBJECT

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space. In order to understand or construct the compiler one must be aware of its design principles. Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

INTRODUCTION

In this book we tried to keep the syllabus of Compiler Design around the GATE syllabus. Each topic required for GATE is crisply covered with illustrative examples and each chapter is provided with Student Assignment at the end of each chapter so that the students get the thorough revision of the topics that he/she had studied. This subject is carefully divided into six chapters as described below.

1. **Introduction to Compiler:** In this chapter we will introduce you to the Analysis-Synthesis model of compilation, various stages of Compilation (namely: lexical, syntax, semantic, intermediate code generation, code optimization and code optimization), grouping of various stages into analysis phase and synthesis phase, passes in compiler and finally we discuss the bootstrapping.
2. **Lexical Analysis:** In this chapter we will study Tokens, lexemes and their patterns and finally we discuss various ways of specifying tokens.
3. **Syntax Analysis (Parser):** In this chapter we introduce you the types of parsers, Top down parser: LL (1) parsing, Bottom up parsers: LR parser, SLR parser, CLR parser, LALR parser and Operator precedence parsing.
4. **Syntax Directed Translation and Intermediate Code Generation:** In this chapter we will study about the Syntax directed definition, attributes (synthesized and inherited), Construction of Syntax trees , of S-Attributed Definitions, L-Attributed Definitions, Bottom up evaluation of inherited Attributes, intermediate code generation , dependency graph using SDT, SDT for intermediate code generation and finally we discuss various representations for intermediate code.
5. **Run Time Environment and Target Code Generation:** In this chapter we will study about the activation trees, control stacks, Storage organization, storage allocation strategies, scope of variables, Symbol table representations, parameter passing and finally we discuss the code generation.

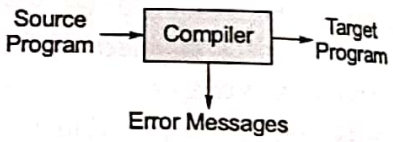
■■■■

CHAPTER 01

Introduction to Compiler

1.1 Compiler

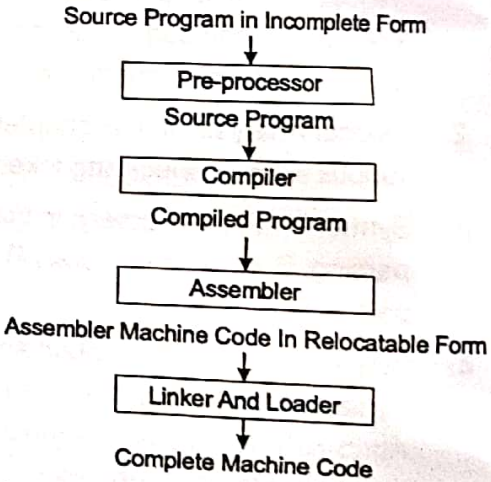
A compiler is a program that reads a program written in one language-the source language-and translates into an equivalent program in another language-the target language. The translation process should also report the presence of errors in the source program.



1.1.1 Interpreter

An interpreter is a program that executes other programs. The interpreter and the underlying machine are combined into a virtual machine and simulate the execution of the input program on the virtual machine.

In general, the source language is any high level language and the object language is an assembly/machine language. We can show the general process of compilation by using following diagram.



Pre-processor is a program that processes its input data to produce output that is used as input to another program (compiler). The output is said to be a preprocessed form of the input data, which is generally used by compiler. Very basic work of pre-processor is to insert the files which are named in source program.

For *example*, let in source program, it is written #include "stdio.h". Pre-processor replace this file by its contents in the produced output.

The basic work of linker is to merge object codes (that has not even connected), produced by compiler, assembler, standard library functions, and operating system resources.

The codes generated by compiler, assembler, and linker are generally re-locatable by its nature, mean to say, the starting location of these codes are not determined, means, they can be anywhere in the computer memory. Thus, the basic task of loader is to find/calculate the exact address of these memory locations.

1.1.2 The Analysis-Synthesis Model of Compilation

There are two parts of compilation:

- **Analysis:** The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- **Synthesis:** The synthesis part constructs the desired target program from the intermediate representation.

1. Analysis of Source Program

In compiling, analysis consists of three phases:

1. **Linear/Lexical Analysis:** 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. The lexical analysis is performed by scanner. The working of the scanner is as follows:
   - The scanner reads characters from the source program.
   - The scanner groups the characters into lexemes (sequences of characters that "go together").
   - Each lexeme corresponds to a token; the scanner returns the next token (plus maybe some additional information) to the parser.
   - The scanner may also discover lexical errors (e.g., erroneous characters).

   The definitions of what is a lexeme, token, or bad character all depend on the source language.
2. **Syntax/Hierarchical Analysis:** It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, the grammatical phrases of the source program are represented by a parse tree.
3. **Semantic Analysis:** The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase. An important component of semantic analysis is type checking. It may also annotate and/or change the abstract syntax tree (e.g., it might annotate each node that represents an expression with its type.

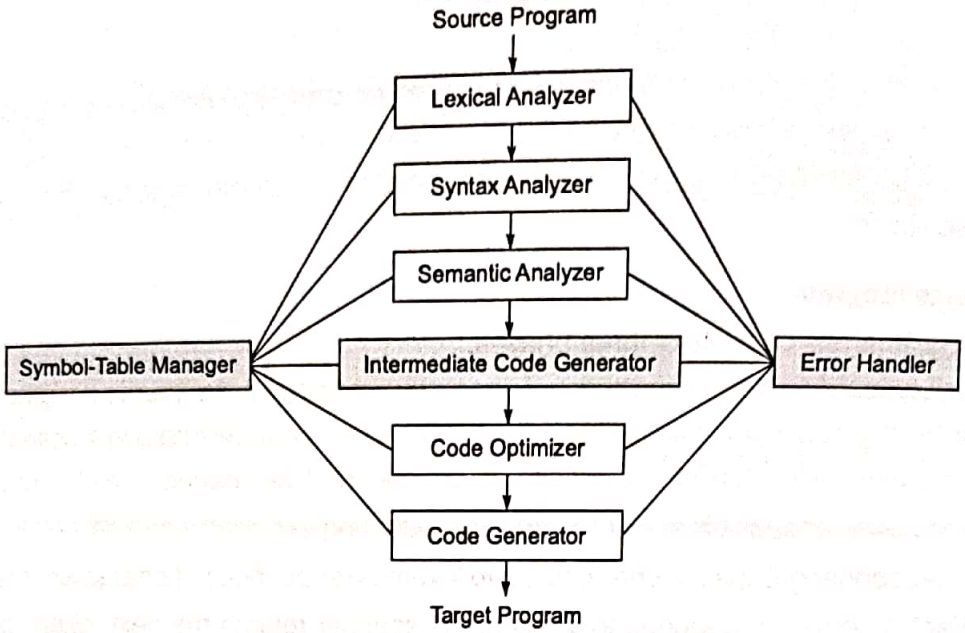
2. Synthesis of Source Program

In compiling, synthesis consists of three phases:

1. **Intermediate Code Generation:** After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This intermediate representation should have two important properties; it should be easy to produce, and easy to translate into the target program. The intermediate code can have a variety of forms.
2. **Code Optimization:** In the optimization phase, the compiler performs various transformations in order to improve the intermediate code. These transformations will result in faster-running machine code.
3. **Code Generation:** The final phase in the compilation process is the generation of target code. This process involves selecting memory locations for each variable used by the program. Then, each intermediate instruction is translated into a sequence of machine instructions that performs the same task.

1.2 Compiler Stages

In general the process of compiler is divided into following six stages/phases. Each phase interact with symbol table administrator and error handling.



The first three phases, forming the bulk of the analysis portion of a compiler and the last three phases, forming the bulk of the synthesis portion of a compiler.

1. Lexical Analysis

It is a program which categorizes character sequence into lexemes and produces a sequence of tokens as an output for parser. Lexeme is a sequence of characters from the input program. Token is just a representative for the bodies that form the text of the program.

LEXEMES —Pattern Rule which tells when a sequence of characters make a token→ TOKEN

For example **integer** is a lexeme, i, n, t, e, g, e, r is the rule, and integer is its token. Similarly **madeeasy** is a lexeme, any sequence of alphanumeric starting with a letter is rule, and finally **madeeasy** becomes a token.

Let us consider the following example.

While (a >= b) a = a – 2;

We can represent it in the form of lexemes and tokens as under.

Lexemes	Tokens	Lexemes	Tokens
while	*WHILE*	*a*	*IDENTIFIER*
(	*LPAREN*	=	*ASSIGNMENT*
a	*IDENTIFIER*	*a*	*IDENTIFIER*
>=	*COMPARISON*	–	*ARITHMETIC*
b	*IDENTIFIER*	2	*INTEGER*
)	*RPAREN*	;	*SEMICOLON*

Let us also consider the following piece of code.

```
/* C program by madeeasy*/
main()
{
printf ("madeeasy/n");
}
```

Let us suppose that the length of words of memory is 4 bytes, and then the starting form of the input code is as follows:

/	*		C		p	r	o	g	r	a	m		b	y		m	a	d	e	e	a	s	y		*	/
	m	a	i	n	(	)		{		p	r	i	n	t	f	(	"	m	a	d	e	e	a	s	y	/
n	"	)	;		}																					

Lexical analyzer reads the input character one by one. Then according to the regulation of lexical grammar of C language, the character stream is grouped into different token and so on, it becomes the stream of tokens. In C language, the tokens can be KEYWORDS, IDENTIFIERS, INTEGERS, REAL NUMBERS, a notation of SINGLE CHARACTER, COMMENTS, and CHARACTER STRINGS, etc. For example, the lexical analyzer starts its work with reading the first word of the input. It reads the "/", it knows that this is not a letter, rather it is an operator. However, as expression with operator is missing, so it continues its reading to the second character to see if it is "*" or not. If it is not "*", then it is wrong. Otherwise it knows that the combination of "/" and "*" forms the identification of the start of comment line, and all the character string before the identification of the end of comment line, i.e., the combination of "*" and "/" is a comment.

Example: The number of tokens in the FORTRAN statement `DO 10 I = 1, 25` is seven.

KEYWORD = 'DO'; STATEMENT-LEBEL= '10'; IDENTIFIER= 'I'; OPERATOR = '=' ; CONSTANT = '1'; COMMA= ',' and CONSTANT = '25'.

2. Syntax Analysis (Parsing)

It is a program that takes sequence of tokens as an input from lexical analysis phase and classify it as an expression with their expression-type. The expression is based on the rules of the source language. For example, consider the following piece of code written in source language.

$$x_1 = x_2 * x_3 + x_4$$

Lexical analyzer converts this code in following frame:

Identifier (x_1) assignment operator (=) identifier (x_2) binary operator (*) identifier (x_3) binary operator (+) identifier (x_4). Let the rules of expression in source language is:

1. CONSTANT and IDENTIFIERS are expressions; and
2. If A and B are two expressions then A BINARY-OPERATOR B is also an expression.

Based on above rules, syntax analyzer performs the following action.

A = IDENTIFIER (2) and B = IDENTIFIER (3) – Both are expressions because from rule (1) IDENTIFIERS are expressions. Its type is "expression".

C = A BINARY-OPERATOR (1)B – It is an expression because from rule (2), if A and B are two expressions then A BINARY-OPERATOR B is also an expression. Its type is "expression".

D = IDENTIFIER (4) – It is expression because from rule (1) IDENTIFIERS are expression. Its type is "expression".

E = C BINARY-OPERATOR (2) D – It is an expression because from rule (2), if A and B are two expressions then A BINARY-OPERATOR B is also an expression. Its type is "expression".

Finally, we get:

id ASSIGNMENT-OPERATOR EXPRESSION – It is an expression and its type is "assignment".

The form of expressions can be best presented by a parse tree or a syntax tree.

NOTE: The main difference between lexical analyzer and syntax analyzer is that; lexical analyzer very rarely uses recursion while syntax analyzer almost always uses recursion.

3. Semantic Analysis

It takes input from syntax analysis phase generally in the form of parse table. It decides that whether the input has properly defined according to the rules of source language. It performs two actions: (1) type checking; and (2) type coercion based on type rules. In general, type rules are of two types: (1) expression-based type rules; and (2) assignment-based type rules.

Expression-based Types Rules		
Expression	**What semantic analyzer determine**	**Type of expression**
Constant	PROPER ACCORDING TO RULE	CONSTANT
Variable	PROPER ACCORDING TO RULE	VARIABLE
Operator	IF BINARY OR UNARY – PROPER ACCORDING TO RULE	OPERATOR

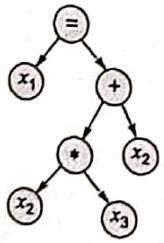
Expression-based Types Rules
Let A is a variable of TYPE(1) and B is properly defined expression of TYPE (2); and if we have TYPE(1) = TYPE(2), where TYPE(1) is an assignable type than A ASSIGN B is a properly defined assignment.

4. Intermediate Code Generation

Combination of lexical analysis, syntax analysis, and semantic analysis is called analysis phase of compiler. After analysis phase, the input source program is fragmented into a parse tree and a symbol table. Using information from these two, compilers starts the procedure of target/object code generation. Generally, code generation is done in two steps: (1) compiler generate code for general machine; and then (2) convert this general machine-based code to the code for particular machine by using a translator. A very general intermediate language is known as three-address code. Generation of code is recursive in nature, and for generating a code we use syntax tree; and therefore the procedure is also called syntax-directed translation.

For example consider the following piece of code: $x_1 = x_2 * x_3 + x_2$

Its equivalent syntax tree is as follows.

Syntax Tree	Equivalent Generated Code
	$t_3 = x_2$ $t_4 = x_3$ $t_1 = t_3 * t_4$ $t_2 = x_2$ $x_1 = t_1 + t_2$

where 't' denotes temporary variable.

5. Code Optimization

The code optimization is a process to improve the code so that the overall run time improves and space requirement reduces. There are lots of methods to improve the code.

6. Code Generation

It is the final stage of compiler process to generate the code for particular machine. Generally in this stage, compiler works on memory management and register allocation.

1.2.1 Symbol-Table Management

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. When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table. However, the attributes of an identifier cannot normally be determined during lexical analysis. The remaining phases enter information about identifiers into the symbol table and then use this information in various ways.

1.2.2 Error Detection and Reporting

Each phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.

Explain all the Necessary Phases and Passes in a Compiler Design?

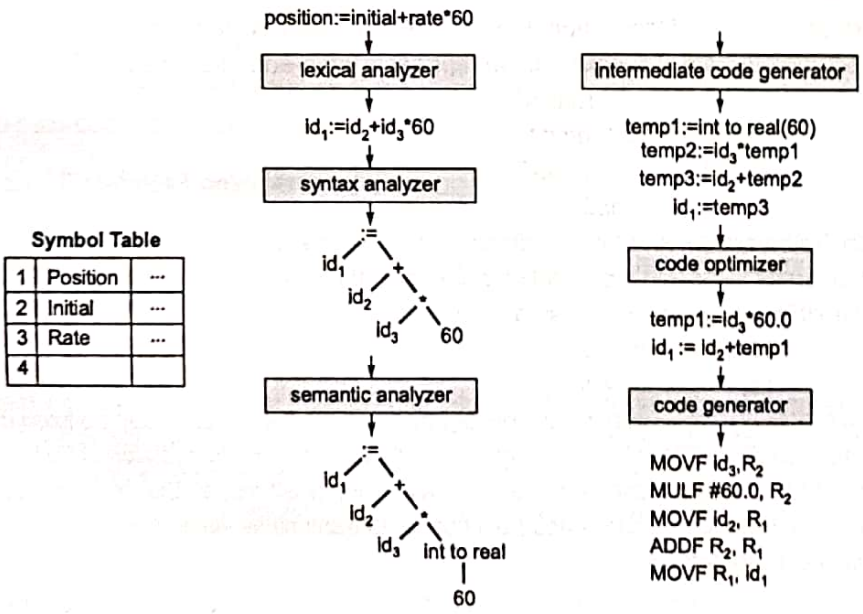
Solution: Since writing a compiler is a non-trivial task, it will be a good idea to structure the work. A representative work of doing this will be to split the compilation into several phases which operate in sequence (though in actual practice, they are often interleaved), each phase takes the result produced by its previous phase as its input.

1. Analysis Phases

As translation progresses, the compiler's internal representation of the source program changes. We illustrate these representations by considering the translation of the statement

position := initial + rate * 60

Following figure shows the representation of this statement after each phase.



- **The Lexical Analysis Phase:** The lexical analysis phase reads the characters in the source program and groups them into streams of tokens; each token represents a logically cohesive sequence of characters, such as identifiers, operators, and keywords. The character sequence that forms a token is called a "lexeme." Certain tokens are augmented by the lexical value; that is, when an identifier like 'position' is found, the lexical analyzer not only returns Id, but it also enters the lexeme position

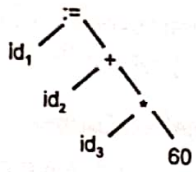
into the symbol table if it does not already exist there. It returns a pointer to this symbol table entry as a lexical value associated with this occurrence of the token id. In our example, tokens will be generated as:

The identifier position, The assignment symbol :=, The identifier initial, The plus sign, The identifier rate, The multiplication sign and The number 60.

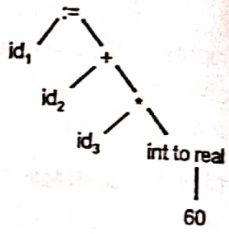
Internal representation of our example after the lexical analysis will be $id_1 := id_2 + id_3 * 60$

The subscripts 1, 2 and 3 are used for convenience; the actual token is id.

- **The Syntax Analysis Phase:** The syntax analysis phase imposes a hierarchical structure on the token stream, shown below



- **The Semantic Analysis Phase:** During the semantic analysis, it is considered that in our example all identifiers have been declared to be reals and that 60 by itself is assumed to be an integer. Type checking of syntax tree reveals that * is applied to a real, 'rate', and an integer, 60. The general approach is to convert the integer into a real. This has been achieved by creating an extra node for the operator int to real that explicitly converts an integer into a real. Alternatively, since the operand of int to real is a constant, the compiler may instead replace the integer constant by an equivalent real constant.



2. Synthesis Phases

- **Intermediate Code Generation:** We consider an intermediate form called "three-address code". The source program of our example will appear in three-address code as

  temp1 := int to real(60)
  temp2 := id_3 * temp1
  temp3 := id_2 + temp2
  id_1 := temp3

- **Code Optimization:** A natural algorithm generates the intermediate code, using an instruction for each operator in the tree representation after semantic analysis, even though there is a better way to perform the same calculation, using the two instructions

  temp1 := id_3 * 60.0
  id_1 := id_2 + temp1

  There is nothing wrong with this simple algorithm, since the problem can be fixed during the code-optimization phase. That is, the compiler can deduce that the conversion of 60 from integer to real representation can be done once and for all at compile time, so the int to real operation can be eliminated. Besides, temp3 is used only once, to transmit its value to id_1. It then becomes safe to substitute id_1 for temp3.

- **Code Generation:** Using registers 1 and 2, the translation of the code of our example might become

  MOVF id_3, R_2
  MULF #60.0, R_2
  MOVF id_2, R_1
  ADDF R_2, R_1
  MOVF R_1, id_1

The first and second operands of each instruction specify a source and destination, respectively. The F is each instruction tells us that instructions deal with floating-point numbers. The # signifies that 60.0 is to be treated as a constant.

1.3 Grouping of Phases

Front End: 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 normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis, and the generation of intermediate code. A certain amount of code optimization can be done by the front end as well. The front end also includes the error handling that goes along with each of these phases.

Back End: The back end includes those portions of the compiler that depend on the target machine, and generally, these portions do not depend on the source language, just the intermediate language. In the back end, we find aspects of the code optimization phases, and we find code generation, along with the necessary error handling and symbol-table operations.

1.4 Passes in a Compiler

A pass is a module in which portions of one or more phases of compiler are combined when a compilers is implemented. A pass reads or scans the instructions constituting the source program or the output produced by its previous pass, makes the necessary transformations specified by its phases, writes the result produced into an intermediate file which may then be read by a subsequent pass. Factors that influence the number of passes are: The structure of the source language and Environment in which compiler will operate.

Mathematical optimally criteria does not affect the number of passes in a compiler. A compiler can explicitly use following types of passes:

1. One-pass compilation technique
2. Two-pass compilation technique
3. Multi-pass compilation technique

1.4.1 Purpose of One-Pass Compiler

A one-pass compiler generates a skeleton of machine instructions the first time it sees a stream of instructions, then appends the machine address for these instructions to a list of instructions to be backpatched once the machine address for it is generated. Backpatching is a technique of merging phases into one pass.

1.4.2 Purpose of Two-Pass Compiler

A two-pass compiler uses its first pass to enter into its symbol table a list of identifiers (as identified in lexical analysis) together with the memory locations to which these identifiers correspond. Then a second pass replaces mnemonic operation codes (mnemonics are keywords as defined by the programming constructs of a language) by their machine-language equivalent and replaced uses of identifiers by their machine addresses or memory locations.

Advantages of Bootstrapping

- It is a non-trivial test of the language being compiled.
- By bootstrapping compiler developers only need to know the language compiled.
- Bootstrapping improves the compiler's back-end which improves not only general purpose programs but also the compiler itself.
- It is a comprehensive consistency check as it is able to reproduce its own object code.

Summary

- Converting a program written in one language into an identical program written in another language is called compilation.
- Phases of compiler:
  1. *Lexical analysis:* Used to generate lexeme (Tokens)
  2. *Syntax analysis:* Used to generate syntax tree.
  3. *Semantics analysis:* Used to check syntax tree semantically correct or not.
  4. *Intermediate code generation:* Used to generate machine independent code.
  5. *Code optimization:* Improve the overall runtime and space of code.
  6. *Code generation:* Generate the code for particular machine.
- The first 3 phases of compiler are called front-end.
- The last 2 phases of compiler are called back-end.
- Pass is a module in which portions of one or more phases of compiler are combined when a compiler is implemented.
- Types of passes:
  1. One-pass compiler
  2. Two-pass compiler
  3. Multi-pass compiler
- In single pass compiler more memory is needed but time to complete compilation is less.
- In multipass compiler memory requirement is less but time to complete compilation is more.

Student's Assignments

Q.1 A C identifier consists of zero or more letters, digits, and underscores, in any order, but may not begin with a digit. Which pattern will match one?

(a) $[A-z]^+[0-9]^*$
(b) $[A-Za-z\_]^+[0-9]^+$
(c) $[a-zA-Z0-9\_]^+$
(d) None of the answers are correct

Q.2 Consider the following statements

1. A compiler is a translator from one language to another.
2. The output of a compiler needs to be in a low-level language.

Which of the following statements are true?

(a) 1 (b) 2
(c) Both (a) and (b) (d) Neither is true

Q.3 Consider the following statements:

1. There is no restriction on compiler that the input and output language must be different.
2. Interpreter takes code written in a particular programming language and executes it directly one statement (or expression, etc.) at a time.
3. Both interpreters and compilers could exists for a given language.

Which of the following statements are true?

(a) 1 and 2 (b) 1, 2 and 3
(c) 2 and 3 (d) Neither is true

Q.4 Consider the following statements:

1. Compiler produce executable binary object file whereas an interpreter produce code, both executable can run many times.
2. Before translation, compiler and interpreter reads all of the input file.

Which of the following statements are true?

(a) 1 (b) 2
(c) Both (a) and (b) (d) Neither is true

Q.5 Consider the following statements:
1. C is a context-free language.
2. C++ is a context-free language.

Which of the following statements are true?
(a) 1 (b) 2
(c) Both a and b (d) Neither is true

Q.6 Consider the following statements:
1. C++ is much harder to parse than C.
2. The backend of the compiler handles program parsing.

Which of the following statements are true?
(a) 1 (b) 2
(c) Both (a) and (b) (d) Neither is true

Q.7 Consider the following statements:
1. Both compiler and interpreter perform syntactic and semantics analysis.
2. Both compiler and interpreter perform optimizations on the input code.
3. Both compilation and interpretation is an off-line process.

Which of the following statements are true?
(a) 1 and 2 (b) 1 and 3
(c) 2 and 3 (d) Neither is true

Q.8 Consider the following statements.
1. Abstract syntax tree made in syntax analysis phase contain cycle.
2. $30 = x * 3$ contains lexical error.

Which of the following statements are true?
(a) 1 (b) 2
(c) Both (a) and (b) (d) Neither is true

Q.9 In a variable declaration, the variable has not previously been declared in the same step.
(a) Compiler handles it in Scanner stage
(b) Compiler handles it in Semantic stage
(c) Compiler handles it in parser stage
(d) None of these

Q.10 Consider the following statements:
S1: A compiler performs code optimization; a preprocessor does not.
S2: A compiler performs full syntactic and semantic analysis; a preprocessor does not.

Which of the following statements is correct?
(a) S1 and S2 are both true
(b) S1 is true, S2 is false
(c) S1 is false, S2 is true
(d) S1 and S2 are both false

Q.11 Consider the following issue:
I. Simplify the phases
II. Compiler efficiency is improved
III. Compiler works faster
IV. Compiler portability is enhanced.

Which is/are true in context of lexical analysis?
(a) I, II, III (b) I, III, IV
(c) I, II, IV (d) All of these

Q.12 The task of the lexical analysis phase is
(a) To parse the source program into the basic elements or token of the language.
(b) To build a literal table and an identifier table
(c) To build a uniform symbol table
(d) All of the above

Q.13 Assembly code data base is associated with
(a) Assembly language version of the program which is created by the code.
(b) A permanent table of decision rules in the form of patterns for matching with the uniform symbol table to discover syntactic structure.
(c) Consists of full or partial list or the token is as they appear in the program. Created by lexical analysis and used for syntax analysis and interpretation.
(d) A permanent table which lists all key words and special symbols of the language in symbolic form.

Q.14 The term "environment" in programming language semantics is said as
(a) Function that maps a name to value held there
(b) Function that maps a storage location to the value held there
(c) Function that maps a name to a storage location
(d) None of the above

Q.15 A linker is given object modules for a set of programs that were compiled separately. What information need to be included in an object module?

(a) Object code
(b) Relocation bits
(c) Name and locations of all external symbols defined in the object module
(d) Absolute addresses of internal symbols.

Q.16 Which of the following is used in syntax analysis?

(a) Finite automata
(b) Push down automata
(c) Regular grammar
(d) None of these

Q.17 Identify the type of error (earliest phase) identified during compilation of the following program.

```
#include <stdio.h>
main( )
{
  int gate, exam, rank;
  gate = exam = rank = 10.3;
  printf("%c", gate);
}
```

(a) Lexical error (b) Syntax error
(c) Semantic error (d) None of these

Q.18 Symbol table is not modified during ____ phase of a compiler.

(a) Lexical
(b) Semantic
(c) Both (a) and (b)
(d) None of these

Answer Key:

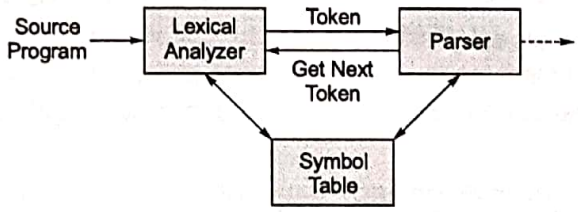
1. (d)	**2.** (a)	**3.** (c)	**4.** (d)	**5.** (d)
6. (a)	**7.** (d)	**8.** (d)	**9.** (b)	**10.** (a)
11. (c)	**12.** (d)	**13.** (a)	**14.** (c)	**15.** (a, c)
16. (b)	**17.** (c)	**18.** (d)		

CHAPTER 02

Lexical Analysis

2.1 Lexical Analyzer

The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.



Typical Tasks of a Scanner (Lexical Analyzer):

- Recognize reserved keywords.
- Find integer and floating-point constants.
- Ignore comments.
- Treat white spaces appropriately in the form of blank, tab and new line characters.
- Counting the number of lines.
- Find string and character constants.
- Find identifiers (variables).
- Reporting errors messages.

2.2 Tokens, Lexemes and Patterns

Token: A token is a set of characters. The following constructs are treated as tokens: keywords, operators, identifiers, constants, literal strings, and punctuation symbols such as parentheses, commas, and semicolons.

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

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example:

Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or > or >=
id	pi, count	letter followed by letters and digits
num	3.14, 286	any numeric constant
literal	"Made Easy"	any characters between " and " except "

2.2.1 Attributes for Tokens

When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler. For example, the pattern num matches both the strings 0 and 1. The lexical analyzer collects information about tokens into their associated attributes. Attributes are used to distinguish different lexemes in a token. Tokens affect syntax analysis and attributes affect semantic analysis.

Example: The tokens and associated attribute-values for the statement A = B * C + 5 are written below as a sequence of pairs:

<id, pointer to symbol-table entry for A>
<assign_op, >
<id, pointer to symbol-table entry for B>
<mult_op, >
<id, pointer to symbol-table entry for C>
<add_op, >
<num, integer value 5 >

2.2.2 Lexical Errors

If none of patterns for tokens matches a prefix of the remaining input, in that case lexical analyzer gets struck and it has to recover from this state to analyze the remaining input.

2.2.3 Recovery Methods

The simplest recovery strategy is "Panic mode error recovery". We delete successive characters from the remaining input until the lexical analyzer can find a well-formed token. Other error recovery methods are:

- Delete an extraneous character
- Insert a missing character
- Replace an incorrect character by a correct character
- Transpose two adjacent characters

Example - 2.1 The C statement if (++x == 5) foo(3);

Find the number of tokens?

Solution:

if	(	++	ID	==	5	)	ID	(	int	)	;

There are 12 tokens.

Example-2.2 Consider the fragment C/C++ code below:

```
01 : If (x == y)
02 : z = z + 1;
03 : else
04 : z = z – (++a);
```

The input string appears to the lexical analyzer as follows:
If (x == y) \n\t z = z + 1;/n else\n\tz = z – (++a);\n
Using the following token classes, write out the lexical tokens that are likely to be returned from the lexical analyzer.
White Space (WS) = '\n', '\t', ' ', Dellmeter (DEL) = ';' ,Keyword (KEY) = 'If', 'else', Binary operator (BOPS) = '+', '–', '*', '/' , Unary operator (UOPS) = '++', '--', Testing operator (TEST) = '==' , Assignment operator (ASGN) = '=' , Identifier (ID) = 'a' – 'z', Number (NUM) = '– 999' – '999' and Parenthesis (PAR) = '(', ')' ?

Solution:

<KEY, 'if'>, <WS, ' '>, <PAR, '('>, <ID, 'x'>, <TEST, '=='>, <ID, 'y'>, <PAR, ')'>, <WS, '\n\t'>, <ID, 'z'>, <ASGN, '='>, <ID, 'z'>, <BOPS, '+'>, <NUM, '1'>, <DEL, ';'>, <WS, '\n'>,
<KEY, 'else'>, <WS, '\n\t'>, <ID, 'z'>, <ASGN, '='>, <ID, 'z'>, <BOPS, '–'>,
<PAR, '('>, <UOPS, '++'>, <ID, 'a'>, <PAR, ')'>, <DEL, ';'>, <WS, '\n'>

Example-2.3 This lexical specification produced the stream of tokens shown below:

IDEN = {'a' – 'z'}, OPER = {'+' + '–' +' * '+'/'}, ASGN = {'='}, TEST = {'==' + '!=' + '<=' + '>='}, PARN = {'(' + ')'}, KEYW = {'if' + 'then' + 'else'}, WTSP = {'\n' + '\t' + ' '}
R = IDEN + OPER + ASGN + TEST + PARN + KEYW+ WTSP... <KEYW,' if'>, <WTSP,' '<, <PARN,'('>, < IDEN,'x'>, <ASGN,'='>, <ASGN,'='>, <IDEN,'y'>, <PARN,')'>,...
The complier is producing the wrong output and you suspect that the language may have been incorrectly specified. Explain what happened and explain how to fix the problem.

Solution:

The rules of precedence will give the ASGN tokens higher priority than the TEST tokens. We except the output to be
<KEYW,'if'>, <WTSP,' '>, <PARN,'('>, <IDEN, 'x'>, <TEST, '=='>, <IDEN, 'y'>, <PARN,')'>
To fix this we must change the order in the language description to :
R = IDEN + OPER + TEST + ASGN + PARN + KEYW + WTSP

2.3 Specification of Tokens

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for sets of strings.

2.3.1 Regular Expressions

A regular expression is a notation to specify a regular set. In Pascal, an identifier is a letter followed by zero or more letters or digits. With regular expression notation, we can define Pascal identifiers as

letter (letter | digit)*

Following are the rules that define the regular expressions over alphabet Σ.

1. ∈ is a regular expression denoting {∈}.
2. If x is a symbol in Σ, then x is a regular expression denoting {x}.

3. Suppose r and s are regular expressions denoting $L(r)$ and $L(s)$
   (*i*) $(r) \mid (s)$ is a regular expression denoting $L(r) \cup L(s)$
   (*ii*) $(r)\,(s)$ is a regular expression denoting $L(r)\,L(s)$
   (*iii*) $(r)^*$ is a regular expression denoting $(L(r))^*$
   (*iv*) (r) is a regular expression denoting $L(r)$

2.3.2 Regular Definition

If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\cdots$$
$$d_n \rightarrow r_n$$

where each d_i is a distinct name, and each r_i is a regular expression over the symbols in $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$, i.e., the basic symbols and the previously defined names. By restricting each r_i to symbols of Σ and the previously defined names, we can construct a regular expression over Σ for any r_i by repeatedly replacing regular-expression names by the expressions they denote. If r_i used d_j for some $j \geq i$, then r_i might be recursively defined, and this substitution process would not terminate.

Example: A regular definition for Pascal identifiers

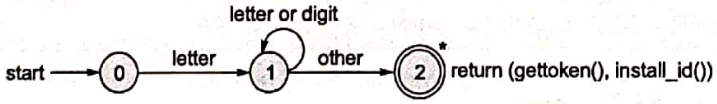
letter → A | B ... | Z | a | b | ... | z
digit → 0 | 1 | ... | 9
id → letter (letter | digit)*

2.3.3 Recognition of Tokens

Lexical analyzer are based on a simple computational model called the finite state automaton. Their functionality is shown with the help of transition diagrams, which consist of states, and arcs showing the transition from one state to another. These machines recognize regular languages.

Transition Diagrams

Transition diagrams depict the actions that take place when a lexical analyzer is called by the parser to get the next token. Positions in a transition diagram are drawn as circles and are called states. The states are connected by arrows, called edges. Edges leaving state '*s*' have labels indicating the input characters that can next appear after the transition diagram has reached state '*s*'. The label 'other' refers to any character that is not indicated by any of the other edges leaving '*s*'. The following transition diagram is to identify Pascal identifiers.



The return statement uses gettoken () and install_id() to obtain the token and attribute-value, respectively to be returned.

Summary

- Lexical analyser also called Scanner.
- When syntax analyser request for token then only lexical analyaser generate token.
- For an invalid token lexical analyser generate token error.
- Sequence of character are called lexemes.
- Set of character are called token.
- Lexical analyser uses finite state automaton to identify different tokens.
- Strings of letters and digits which is started with letter only are called identifier.
- Lexical analyser scan the lexemes always left to right.

Student's Assignments

Q.1 The number of tokens in the following C statement
printf ("i = %d, and i = %x", i, &i);
(a) 3 (b) 26
(c) 10 (d) 21

Q.2 Consider the following statements:
S1. The set of string described by a rule is called pattern associated with token.
S2. A lexeme is a sequence of character in the source program that is matched by pattern for a token.
(a) Both S1 and S2 are true
(b) S1 is true but S2 is false
(c) S2 is true but S1 is false
(d) Both S1 and S2 are false

Q.3 How many tokens are contained in the following FORTRAN statements:
IF(NUMB EQ MAX) GOTO 500
(a) 8 (b) 9
(c) 10 (d) 11

Q.4 If a NFA which has n states, then the maximum number of states the equivalent DFA can have are
(a) n^2 (b) n
(c) 2^n (d) None of these

Q.5 Consider the following language:
$L = \{a^n b^m c^m d^n \mid n, m \geq 1\}$
What is the minimum yield of the above language? (From equivalent parse tree)
(a) abcd
(b) aaabbbcccddd
(c) aabbccdd
(d) aaaabbbbccccdddd

Q.6 In some programming language, an identifier is permitted to be a letter following by any number of letters or digits. If L and D denotes the sets of letters and digits respectively, which of the following expressions defines an identifier?
(a) $(L \cup D)^*$ (b) $L(L \cup D)^*$
(c) $(L \cdot D)^*$ (d) $L \cdot (LD)^*$

Q.7 Which of the following pairs of regular expressions define the same language over {a, b}?
(*i*) $(a^* + b^*)^*$ and $(a + b)^*$
(*ii*) $(a^* + b)^*$ and $(a + b)^*$
(*iii*) $(ab)^*$ a and $a(ba)^*$
(*iv*) $(a^* b)^* a^*$ and $a^*(ba^*)^*$
(a) (*i*), (*iii*) and (*iv*)
(b) (*ii*), (*iii*) and (*iv*)
(c) (*i*), (*ii*), and (*iv*)
(d) (*i*), (*ii*), (*iii*) and (*iv*)

Q.8 Determine the minimum number of states of a DFA that recognizes the language over the alphabet {a, b} consisting of all the strings that contain atleast two 'a's and at least three 'b's.
(a) 6 (b) 12
(c) 18 (d) None of these

Q.9 Match the following groups:

List-I
A. Lexical analyzer
B. Syntax analyzer
C. Type checking
D. Intermediate code generation

List-II
1. Checks the structure of the program.
2. Analysis of entire program by reading each character.
3. High level language is translated to simple machine independent language.
4. Checks the consistency requirements in a context of the program.

Codes:

	A	B	C	D
(a)	1	2	4	3
(b)	2	1	4	3
(c)	2	4	3	1
(d)	1	4	3	2

Q.10 In C programming, which of the following is not used as a token separator during lexical analysis?
(a) White space
(b) Comment
(c) Semicolon
(d) None of these

Q.11 Find the C statement which has a syntax error.
(a) fi (z);
(b) for (a, b, c);
(c) whil (a, b);
(d) All of these

Q.12 Consider the C program

```
main( )
{
    int x = 10;
    x = x + y + z;
}
```

How many tokens are identified by lexical analyzer?

Q.13 How many tokens are generated by the lexical analyzer, if the following program has no lexical error?

```
main( )
{
    int x, y;
    fl/*gate oat z;
    x =/*exam*/10;
    y = 20;
}
```

Q.14 Consider the following C program:

```
1. #include <stdio.h>
2. main( )
3. {
4.       int a = 2, b = 3;
5.       char *x;
6.       x = &a = &b;
7.       a = 1xab;
8.       printf("%d%d", a, *x);
9. }
```

If scanner reads an entire program then find the line number in which lexical error is produced.

Q.15 Find the regular expression that correctly identifies the variable name in C program.
(a) [a – z][a – zA – Z_0 – 9]*
(b) [a – zA – Z_][a – zA–Z_0 – 9]*
(c) [a – zA – Z_][a – zA – Z0 – 9]*
(d) [a – zA – Z][a – zA – Z_0 – 9]*

Q.16 Which of the following is not a token in C language?
(a) Semicolon
(b) Identifier
(c) Keyword
(d) White space

Q.17 Lexical error is _____.
(a) An error produced by lexical analyzer when an illegal character appears.
(b) An error produced by lexical analyzer when a missing left parenthesis in an expression.
(c) An error produced by scanner when both operator and parenthesis appeared consecutively.
(d) All of the above

Answer Key:

1. (c)	**2.** (a)	**3.** (a)	**4.** (c)	**5.** (a)
6. (b)	**7.** (d)	**8.** (b)	**9.** (b)	**10.** (d)
11. (b)	**12.** (18)	**13.** (17)	**14.** (7)	**15.** (b)
16. (d)	**17.** (a)			

CHAPTER 03

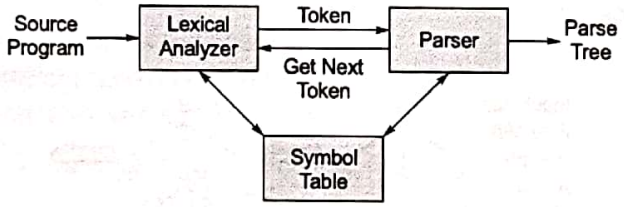
Syntax Analysis (Parser)

3.1 Introduction

In the syntax-analysis phase, a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. Syntax analysis is done by the parser.

The parser obtains a string of tokens from the lexical analyzer, and verifies that the string can be generated by the grammar for the source language.

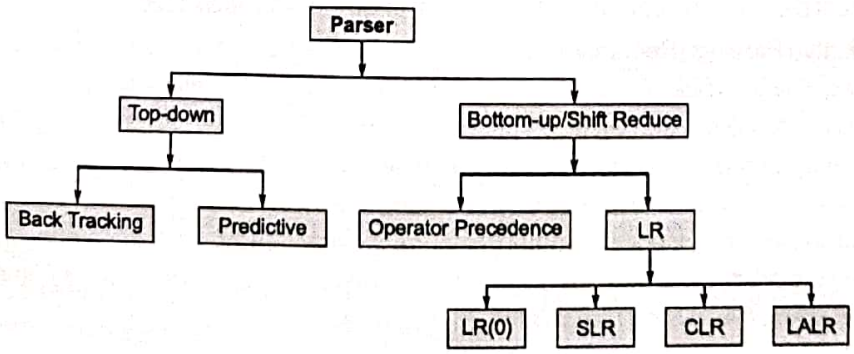
It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated. In other words it determines the syntactic structure of a sentence in some language.



The syntax of a programming language is described by a context-free grammar or BNF (Backus-Naur Form). A grammar gives a precise syntactic specification of a language.

Types of Parser

Their are two types of Parser **(1)** Top-down parser and **(2)** Bottom-up parser.



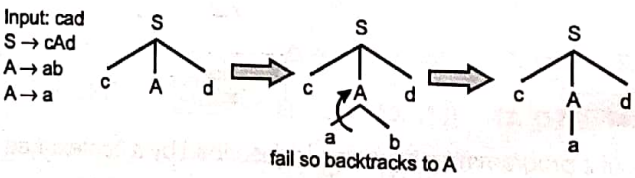
3.2 Top-Down Parsing

Top-down parsing attempts to find the left most derivations for an input string *w*, which is equivalent to constructing a parse tree for the input string *w* that starts from the root and creates the nodes of the parse tree in a predefined order. The reason that top-down parsing seeks the left most derivations for an input string *w* and not the right most derivations is that the input string *w* is scanned by the parser from left to right, one symbol/token at a time, and the left most derivations generate the leaves of the parse tree in left-to-right order, which matches the input scan order.

Since top-down parsing attempts to find the leftmost derivations for an input string *w*, a top-down parser may require backtracking (i.e., repeated scanning of the input); because in the attempt to obtain the leftmost derivation of the input string *w*, a parser may encounter a situation in which a non-terminal A is required to be derived next, and there are multiple A-productions, such as $A \rightarrow a_1 \mid a_2 \mid \ldots \mid a_n$. In such a situation, deciding which A-production to use for the derivation of A is a problem. Therefore, the parse will select one of the A-productions to derive A, and if this derivation finally leads to the derivation of w, then the parser announces the successful completion of parsing. Otherwise, the parser resets the input pointer to where it was when the non-terminal A was derived, and it tries another A-production. The parser will continue this until it either announces the successful completion of the parsing or reports failure after trying all of the alternatives.

1. **Back tracking:** Guess which production is to use next step based on next token. If the guessing is wrong then backtracking. The parser consists of a set of (possibly recursive) procedures. Each procedure is associated with a non-terminal of the grammar that is responsible to derive the productions of that non-terminal. Each procedure should be able to choose a unique production to derive based on the current token. For each terminal in the production, the terminal is matched with the current token. For each non-terminal in the production, the procedure associated with the non-terminal is called. The sequence of matching and procedure calls in processing the input implicitly defines a parse tree for the input.

   Example:



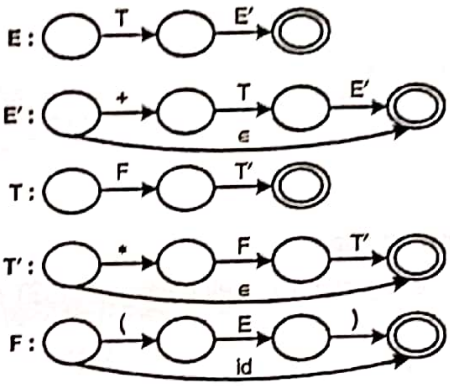
   Recursive descent parsing involves backtracking, that is, making repeated scans of the input. However, backtracking parsers are not seen frequently. One reason is that backtracking is rarely needed to parse programming language constructs. A left recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop.

2. **Predictive Parsing (Recursive):** The special case of recursive descent parsing is called predictive parsing where no backtracking is required. In many cases, by carefully writing a grammar, eliminating left recursion from it, and left factoring the resulting grammar, we can obtain a grammar that can be parsed by a recursive-descent parser that needs no backtracking, i.e., a predictive parser.
   A (recursive) procedure is associated with each non-terminal of a grammar. A production is chosen by looking at the look ahead symbol (the currently scanned input token). A non-terminal in the right side of the production results in a call to the corresponding procedure for the non-terminal. We can model predictive parsing using transition diagrams.

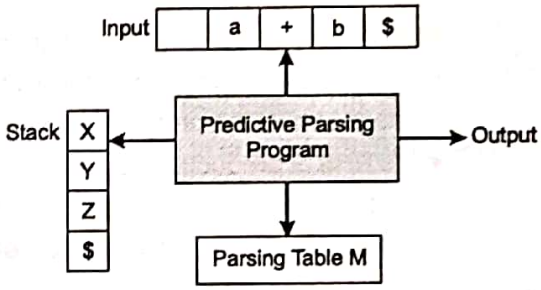
Example:

Before removing left recursion	After removing left recursion
$E \rightarrow E + T \mid T$ $T \rightarrow T * F \mid F$ $F \rightarrow (E) \mid id$	$E \rightarrow TE'$ $E' \rightarrow +TE' \mid \epsilon$ $T \rightarrow FT'$ $T' \rightarrow *FT' \mid \epsilon$ $F \rightarrow (E) \mid id$

The transition diagram for above grammar is as shown.



3. **Non-recursive Predictive Parsing:** It is possible to build a non-recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The non-recursive parser looks up the production to be applied in a parsing table. A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream.



The input buffer contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $. The parsing table is a two-dimensional array M[A, a], where A is a non-terminal, and a is a terminal or the symbol $.

The parser is controlled by a program that behaves as follows. The program considers X, the symbol on top of the stack, and 'a', the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

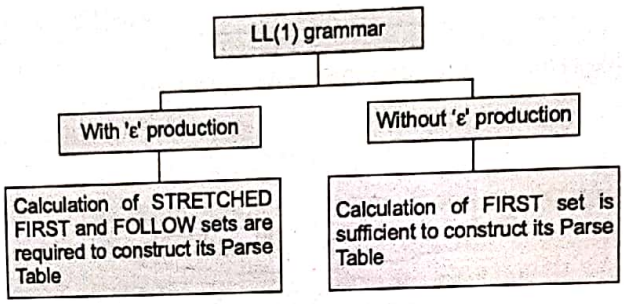
(a) If X = a = $, the parser halts and announces successful completion of parsing.

(b) If (X = a) ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.

(c) If X is a non-terminal, the program consults entry M[X, a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, M[X, a] = {X → UVW}, the parser replaces X on top of the stack by WVU (with U on top). As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X, a] = error, the parser calls an error recover routine.

3.3 LL(1) Parsing

LL(1) means that the grammar allows a deterministic parser that operates from left to right, produces leftmost derivation, using a look ahead of one symbol. To make the parsing deterministic, we could require that each parse table entry contain at most one element. In terms of grammar, this means that all right-hand sides of a non-terminal start with different terminal symbol. A grammar that fulfills this requirement is called a simple LL(1) grammar (SLL(1)), or an s-grammar. Deterministic parsers are much faster than non-deterministic parser.



3.3.1 Algorithm: FIRST Set Computation

FIRST is a set of terminals. The algorithm for calculation of FIRST set is as follows.

Input: Grammar without any ε production in it.

Process: Follow the following Steps.

Step-1: Set the FIRST of all non-terminals to empty.

Step-2: For each productions of grammar, perform the following action.

- If RHS of production begin with any terminal, put that terminal to the FIRST set of non-terminal which is at LHS of the production. For example, let $S \rightarrow aA$, so FIRST(S) = {a}.
- If RHS of production begin with any non-terminal, put all terminals which are in the FIRST set of this non-terminal (which begins the RHS production) to the FIRST set of non-terminal which is at LHS of the production. For example, let $S \rightarrow AB$, so FIRST(S) = FIRST(A); let FIRST(A) = {b, c}, then FIRST(S) = {b, c}.

Step-3: Repeat the previous Step until no more new terminals are added to any of the FIRST set.

3.3.2 Algorithm: FOLLOW Set Computation

FOLLOW is a set of terminals which we calculate when the grammar contains any ε production. In addition to terminals, ε is also allowed as a member of this set and $ is always the member of FOLLOW set of START SYMBOL of given grammar. The Algorithm for calculation of FOLLOW set is as follows:

Input: (1) Grammar with any ε production in it; and (2) FIRST sets of non-terminals

Process: Follow the following Steps:

Step-1: As with the computation of FIRST set, we initialize the FOLLOW set of every non-terminal by empty.

Step-2: We process all RHS, including S$, where S is the start symbol of given grammar. If RHS of any production contain any non-terminal, then we calculate its FOLLOW as under, which is explained by an example. Let $A \rightarrow xBz$ Thus, FOLLOW(B) = FIRST(z). Let $A \rightarrow xBD$ Thus, FOLLOW(B) = FIRST(D) and FOLLOW(D) = FOLLOW(A).

Step-3: Repeat previous Step until no more symbols can be added to any of the FOLLOW set.

3.3.3 Algorithm: STRETCHED-FIRST Set Computation

STRETCHED-FIRST is a set of terminals which we calculate when the grammar contains any ε production. In addition to terminals, ε is also allowed as a member of this set. The algorithm for calculation of STRETCHED-FIRST set is as follows.

Input: Grammar with any ε production in it.

Process: Follow the following Steps.

Step-1: Set the STRETCHED-FIRST of all non-terminals to empty.

Step-2: For each productions of grammar, perform the following action.

- If RHS of production begin with any terminal, put that terminal to the STRETCHED-FIRST set of non-terminal which is at LHS of the production. For example, let S → aA, so FIRST(S) = {a}.
- If RHS of production begin with any non- terminal, put all terminals which are in the STRETCHED-FIRST set of this non-terminal (which begins the RHS production) to the STRETCHED-FIRST set of non-terminal which is at LHS of the production. But, if STRETCHED-FIRST set of non-terminal which is in RHS contains ε then put all terminals (except ε) which are in the STRETCHED-FIRST set of this non-terminal (which begins the RHS production) to the STRETCHED-FIRST set of non-terminal which is at LHS of the production, followed by UNION of rest of RHS production body.

  For *example*, let S → AB, so STRETCHED-FIRST(S) = STRETCHED-FIRST(A); let STRETCHED-FIRST(A) = {ε, b, c} then STRETCHED-FIRST(S) = STRETCHED-FIRST(A), ε-excludes ∪ STRETCHED-FIRST(B) = {b, c} ∪ STRETCHED-FIRST(B). Let STRETCHED-FIRST(B) = {d}, then STRETCHED-FIRST(S) = {b, c} ∪ {d} = {b, c, d}. Again let STRETCHED-FIRST(B) = {ε, d}, then STRETCHED-FIRST(S) = {b, c} ∪ STRETCHED-FIRST(B), ε excludes ∪....= {b, c} ∪ {d} = {b, c, d}. Again let STRETCHED-FIRST(B) = {ε}, then STRETCHED-FIRST(S) = {b, c} ∪ STRETCHED-FIRST(B), ε excludes ∪ ...{b, c}.

Step-3: Repeat the previous Step until no more new terminals are added to any of the STRETCHED-FIRST set.

3.3.4 Algorithm: Parse Table Construction

Parse table is made of rows and columns. The algorithm for constructing parse table for the grammar which does not contain any ε production is as follows.

Step-1: Draw a matrix with – number of rows equivalent to number of non-terminals in the given ε free grammar; and number of columns equivalent to number of terminals +1.

Step-2: For each productions of grammar, perform the following action.

- If RHS of production begin with a terminal symbol, say 'a', add that RHS to (non-terminal at LHS, a) → entry of the parse table.
- If RHS begin with a non-terminal symbol, say 'A', add that RHS to (non-terminal at LHS, a) → entry of the parse table for all symbols 'a' in FIRST(RHS).

3.3.5 Pair-wise Disjoint

A grammar without ε production is LL(1) if for every non-terminal A, the FIRST sets of A are PAIRWISE DISJOINT (no symbol occurs in more than one). For example, consider the grammar.

S → AbC|ad

A → eS|Cr

C → f|p

Consider the set of productions $S \rightarrow AbC|ad$

$FIRST(AbC) \cap FIRST(ad) = FIRST(A) \cap FIRST(a) = \{e, f, p\} \cap \{a\} = \phi$

Consider the set of productions $A \rightarrow eS|Cr$

$FIRST(eS) \cap FIRST(Cr) = FIRST(e) \cap FIRST(C) = \{e\} \cap \{f, p\} = \phi$

Consider the set of productions $C \rightarrow f|p$

$FIRST(f) \cap FIRST(p) = \{f\} \cap \{p\} = \phi$

Therefore, the above grammar is LL(1).

Also consider the following grammar: $A \rightarrow dB|aS|c$

$FIRST(dB) \cap FIRST(aS) = FIRST(d) \cap FIRST(a) = \{d\} \cap \{a\} = \phi$

$FIRST(dB) \cap FIRST(c) = FIRST(d) \cap FIRST(c) = \{d\} \cap \{c\} = \phi$

$FIRST(aS) \cap FIRST(c) = FIRST(a) \cap FIRST(c) = \{a\} \cap \{c\} = \phi$

Therefore, the above grammar is LL(1).

Example-3.1 Consider the following grammar

$S \rightarrow AaAb, S \rightarrow Bb, A \rightarrow \varepsilon$ and $B \rightarrow \varepsilon$

Find its first and follow sets.

Solution:

We initialize the first sets to the empty set.

Initial Pass

First (S)	First (A)	First (B)

1st Pass

First (S)	First (A)	First (B)
	ε	ε

2nd Pass

First (S)	First (A)	First (B)
a, b	ε	ε

3rd Pass

First (S)	First (A)	First (B)
a, b	ε	ε

Set found in 2nd Pass = Set found in 3rd Pass. So we stop the above process.

We initialize the Follow sets to the empty set.

Initial Pass

Follow(S)	Follow (A)	Follow (B)

1st Pass

Follow(S)	Follow (A)	Follow (B)
$	a, b	b

2nd Pass

Follow(S)	Follow (A)	Follow (B)
$	a, b	b

Set found in 1st Pass = Set found in 2nd pass. So we stop the above process.

Thus result is as follows: First (S) = {a, b}, First (A) = {ε}, First (B) = {ε}, Follow (S) = {$}, Follow (A) = {a, b} and Follow (B) = {b}.

Example - 3.2 Construct Parse Table for the grammar

$S \rightarrow AaAb, S \rightarrow Bb, A \rightarrow \varepsilon$ and $B \rightarrow \varepsilon$

Solution:

Since the grammar given contains ε, thus for constructing its Parse Table, we need to calculate First and Follow sets. Parse Table entries are as follows:

1. Consider the production $S \rightarrow AaAb$
   Parsetable [S, First (AaAb)] = Parsetable [S, First(A)]
   Since First(A) contains ε, thus we have to stretch it.
   Thus Parsetable [S, First(A), ε excluded ∪ First (a Ab)] = Parsetable [S, {a}]
   Means, Parse Table [S, a] = $S \rightarrow AaAb$
2. Consider the production $S \rightarrow Bb$
   Parsetable [S, First (Bb)] =Parsetable [S, First (B)]
   Since First (B) contains ε, thus we have to stretch it. Thus:
   Parsetable[S, First (B), ε excluded ∪ First (b)] = Parsetable [S, {b}]
   Means, Parsetable [S, b] = $S \rightarrow Bb$
3. Consider the production $A \rightarrow \varepsilon$
   This production contains ε in its RHS, thus we reduce it as:
   Parsetable [A, Follow (A)] = Parsetable [A, {a, b}]
   Means, Parsetable [A, a] = $A \rightarrow \varepsilon$
   Parsetable {A, b] = $A \rightarrow \varepsilon$
4. Consider the production $B \rightarrow \varepsilon$
   This production contains ε in its RHS, thus we reduce it as:
   Parse Table [B, Follow (B)] = Parsetable [B, {b}]
   Means, Parsetable [B, b] = $B \rightarrow \varepsilon$
   Based on computations 1 to 4, Parsetable is as follows.

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow Bb$	
A	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	
B		$B \rightarrow \varepsilon$	

3.3.6 LL(1) Conflicts and Their Solution

If a parse table entry has more than one element, it is called LL(1) conflict. For solving LL(1) conflicts we have to eliminate left recursion and left factoring from the grammar.

1. LL(1) Left Recursion

A CFG is left recursive if it includes a non-terminal such that $A \overset{+}{\Longrightarrow} A\alpha$. We can eliminate left-recursion in three steps.

Step-1: Eliminate ε-production.

Step-2: Eliminate unit production (also called cycle, i.e., $A \overset{+}{\Longrightarrow} A$)

Step-3: Eliminate left-recursion.

1. **Elimination of ε-Production(s):** Consider the following grammar.
   $S \rightarrow XX \mid Y,\ X \rightarrow aXb \mid \varepsilon,\ Y \rightarrow aYb \mid Z$ and $Z \rightarrow bZa \mid \varepsilon$

We can easily analyze that in given grammar all non-terminals are nullable, i.e. all non-terminals produces ε. Non-terminal S is nullable, but it does not come in RHS of any production. Non-terminal X is nullable and it comes in RHS of productions $S \to XX$ and $X \to aXb$. We can re-write these two productions as $S \to X \mid XX$ and $X \to ab \mid aXb$. Non-terminal Y is nullable and it comes in RHS of production $Y \to aYb$. We can re-write this production as $Y \to ab \mid aYb$. Non-terminal Z is nullable and it comes in RHS of production $Z \to bZa$. We can re-write this production as $Z \to ba \mid bZa$.

The grammar becomes: $S \to XX \mid X \mid Y$, $X \to aXb \mid ab$, $Y \to aYb \mid abZ$ and $Z \to bZa \mid ba$.

2. **Elimination of Unit production(s):** Basic unit pairs are (S, S), (X, X), (Y, Y) and (Z, Z). (S, S) is basic unit pair and $S \to X$ is a unit production thus (S, X) becomes a unit pair. (S, X) is a unit pair and $X \to ab$ is a production, thus we have:

$$S \Rightarrow X \Rightarrow ab$$

Means, we can replace unit production $S \to X$ by a production $S \to ab$.

Similarly, $S \Rightarrow Y \Rightarrow Z \Rightarrow ba$. Thus, we can replace unit productions $S \to Y$ and $Y \to Z$ by a single production $S \to ba$. Similarly, $Y \Rightarrow Z \Rightarrow ba$. Thus, we can replace unit production $Y \to Z$ by a production $Y \to ba$. Thus, grammar becomes.

$$S \to XX \mid ab \mid ba$$
$$X \to aXb \mid ab$$
$$Y \to aYb \mid ab \mid ba$$
$$Z \to bZa \mid ba$$

3. *Elimination of Immediate Left Recursion:* Left recursive productions can cause recursive descent parsers to loop forever. Let us consider the productions $A \to A\alpha \mid \beta$, where α and β are sequence of terminals and non-terminals that do not start with A. The same language can be generated by the productions:

$$A \to \beta R$$
$$R \to \alpha R \mid \varepsilon$$

where R is a new non-terminal. In general, immediate left recursion may be removed as follows. Suppose we have the A-productions

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \ldots \mid b_m$$

where no β_i begins with A. We can replace the A-production by:

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

where A' is new non-terminal.

2. Left Factoring

In general, if $A \to \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, and the input begin with a non-empty string derived from α, we do not know whether to expand to $\alpha\beta_1$ or to $\alpha\beta_2$. Instead, the grammar may be changed. The formal echnique is to change $A \to \alpha\beta_1 \mid \alpha\beta_2$ to:

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

Example-3.3 Consider the following grammar:

$S \rightarrow Sab \mid SaS \mid X, X \rightarrow Xc \mid a \mid b$

Write the equivalent grammar after removal of left recursion?

Solution:

$S \rightarrow Sab \mid SaS \mid X$
$X \rightarrow Xc \mid a \mid b$

$S \rightarrow XT$
$T \rightarrow abT \mid aST \mid \varepsilon$
$X \rightarrow aM \mid bM$
$M \rightarrow cM \mid \varepsilon$

Example-3.4 Eliminate left recursion in the following grammar:

$S \rightarrow A|B, A \rightarrow Aa \mid \varepsilon$ and $B \rightarrow Bb|Sc| \varepsilon$

Solution:

We can remove left-recursion in following steps.

Step 1: Remove the non-immediate left recursion. Note that the labelling of non-terminal is arbitrary. That means we can rearrange the productions before labelling. This is much easier if we swap the second and third productions.

$S \rightarrow A|B, B \rightarrow Bb \mid Ac \mid Bc|\varepsilon$ (The "Sc" was replaced by "Ac | Bc")

$A \rightarrow Aa \mid \varepsilon$

Step 2: Remove immediate left recursion in second production.

$B \rightarrow Bb \mid Ac \mid Bc \mid \varepsilon$ $[\alpha_1 = \beta_1, \alpha_2 = c, \beta_1 = Ac, \beta_2 = \varepsilon]$

Therefore, we get: $B \rightarrow AcB' \mid B', B' \rightarrow bB' \mid cB' \mid \varepsilon$

Step 3: Remove immediate left recursion in third production: $A \rightarrow Aa|\varepsilon$

Therefore, we get: $A \rightarrow A', A' \rightarrow aA' \mid \varepsilon$

These two are same as: $A \rightarrow aA|\varepsilon$

Putting all these together, the new grammar is: $S \rightarrow A|B, A \rightarrow aA| \varepsilon, B \rightarrow AcB' |B', B' \rightarrow bB'|cB'| \varepsilon$

Example-3.5 Consider the following grammar:

$A \rightarrow int \mid int+A \mid int - A \mid A - (A)$

Convert the given grammar in LL grammar.

Solution:

Given grammar can be transformed into LL grammar by two ways:

1. First left-factor and then eliminate left-recursion.
2. First eliminate left-recursion and then left-factor.

Way 1: First left-factor and then eliminate left-recursion. Thus after left-factoring the grammar becomes:

$A \rightarrow int\ A' \mid A - (A), A' \rightarrow \varepsilon \mid +A \mid -A$

Then eliminating left-recursion the grammar becomes

$A \rightarrow int\ A'\ A'', A'' \rightarrow -(A)\ A'' |\varepsilon, A' \rightarrow \varepsilon |+A \mid -A$

Way 2: First eliminate left-recursion and then left-factor. Thus after elimination of left-recursion the grammar becomes: $A \rightarrow int\ A' | int + AA' | int - AA', A' \rightarrow - (A)A' | \varepsilon$

Thus after left factoring the grammar becomes

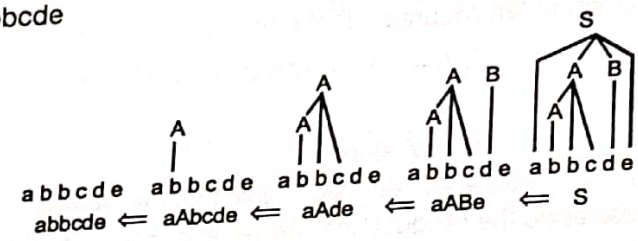
$A \rightarrow int\ A'', A'' \rightarrow A' |+ AA' |-AA', A' \rightarrow - (A)\ A' \mid \varepsilon$

3.4 Bottom-Up Parsing

Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of a grammar by tracing out the right most derivations of *w* in reverse. This is equivalent to constructing a parse tree for the input string *w* by starting with leaves and proceeding toward the root – that is, attempting to construct the parse tree from the bottom up. The reason why bottom-up parsing tries to trace out the right-most derivations of an input string *w* in reverse and not the leftmost derivations is because the parser scans the input string *w* from the left to right, one symbol/ token at a time. And to trace out right-most derivations of an input string *w* in reverse, the tokens of w must be made available in a left-to-right order.

Example: S → a A B e, A → A b c | b and B → d

Input: abbcde



3.5 LR Parser

In computer science, an LR parser is a type of bottom-up parser for context-free grammar that is very commonly used by computer programming language compiler and other associated tools. LR parsers reads their input from left to right and produce a right-most derivation. The term LR (K) parser is also used. Here, the K refers to the number of unconsumed "look-ahead" input symbols that are used in making parser decisions. Usually k is 1 and is often omitted. A context-free grammar is called LR(K) if there exists a LR(K) parser for it.

LR parsing has many benefits:

1. Many programming languages can be parsed using some variation of an LR parser. It should be noted that C++ and Perl are exception of it.
2. LR parsers can be implemented very efficiently.
3. Of all the parsers that scan their input from left to right, LR parsers detect syntactic errors, as soon as possible.

LR parsers are difficult to produce by hand, thus they are usually constructed by a parser generator or a compiler-compiler. Depending on how the parsing table is generated these parsers are called Simple LR parser (SLR, for short), look-ahead LR parser (LALR), and canonical LR parser. These types of parsers can deal with a very large sets of grammars. It should be noted that canonical LR parsers (LR(1)) works on more grammars than LALR parsers, and LALR parsers works on more grammars than SLR parsers. There is a program YACC that produce LALR parsers.

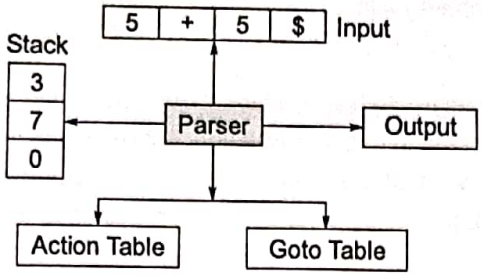
Hierarchy: The grammars for which each of the types of parsers may be constructed is described by the following hierarchy:

$$LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1)$$

Such that every LR(0) grammar is also an SLR(1) grammar, etc. Roughly speaking, the more general the parser type, the more difficult the (manual) construction of the parsing table (i.e., LR(0) parsing tables are the easiest to construct, LR(1) tables are the most difficult). Finally, although the parsing tables for LR(0), SLR(1) and LALR(1) grammars are essentially the same size, LR(1) parsing tables can be extremely large. Because many programming language constructs cannot be described by SLR(1) grammar (and hence neither by LR(0) grammars), but almost all can be described (with effort) by LALR(1) grammars, it is the LALR(1) class of parser that has proven the most popular for bottom-up parsing.

3.5.1 Framework of a Table-based Bottom-up Parser

A table-based bottom-up parser can be schematically presented as:



As shown above, the parser has an input buffer, a stack on which it keeps a list of states it has been in, an action table and a goto table that tell it to what new state it should move.

Action Table of LR(0) Parsing

The action table is indexed by a state of the parser and a terminal including a special terminal $ that indicates the end of the input stream, and contains three types of actions:

1. Shift, which is written as "sn" or "shift n", where n indicates the next state;
2. Reduce, which is written as "rm" or "reduce m" and indicates that a reduction with grammar rule m should be performed.
3. Accepts, which is written as "acc" and indicates that the parser accepts the string in the input stream.

Goto Table of LR(0) Parsing

The goto table is indexed by a state of the parser and a non-terminal and simply indicates "what the next of the parse will be if it has recognized a certain non-terminal".

NOTE: ***Viable Prefixes:*** The viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shift-reduce parser, because they do not extend past the right-most handle.

Formation of Action and Goto table

State	Action						Goto			
	•	•	•	•	•	$	•	•	•	•
•										
•			terminals				non-terminals			
•										
•										
•										

3.5.2 Construction of LR(0) Parsing Table

1. Items

The construction of parsing table is based on the notation of LR(0) items (simply called item) which are grammar rules with a special dot added somewhere in the right-hand side.

For ***example***, the rule $B \rightarrow B + E$ has the following four corresponding items:

$$B \rightarrow \cdot B + E,\ B \rightarrow B \cdot + E,\ B \rightarrow B + \cdot E \text{ and } B \rightarrow B + E \cdot .$$

An exception is rule of the form $A \rightarrow \varepsilon$, whose corresponding item is in the form $A \rightarrow \cdot$.

These rules will be used to denote the state of the parser. The item $B \rightarrow B\bullet + E$, for example, indicates that the parser has recognized a string corresponding with B on the input stream and now expects to read a '+' followed by another string corresponding with E.

2. Item Sets

It is usually not possible to characterize the state of the parser with a single item because it may not be known in advance, which rule it is going to use for reduction. For example, if there is also a rule $B \rightarrow B * E$, then the items $B \rightarrow B\bullet + E$ and $B \rightarrow B\bullet * E$ will both apply after a string corresponding with B has been read. Therefore, we will characterize the state of the parser by a set of items. In this case the set is $\{B \rightarrow B\bullet + E, B \rightarrow B\bullet * E\}$.

3. Closure of Item sets

An item with a dot in front of a non-terminal, such as $B \rightarrow B + \bullet E$, indicates that the parser expect to parse the non-terminal E next. To ensure that the item set contains all possible rules the parser may be in the middle of parsing, it must include all items describing how E itself will be parsed. This means that if there are rules such that $E \rightarrow 1$ and $E \rightarrow 0$ then the item set must include the items $E \rightarrow \bullet 1$ and $E \rightarrow \bullet 0$.

In general, we can formulate it, as under: "If there is an item of the form $A \rightarrow v\bullet Bw$ in an item set and in the grammar if there is a rule of the form $B \rightarrow w'$, then the item $B \rightarrow \bullet w'$ should also be included in the item set."

Any set of items can be extended such that it satisfies the following rule: "Continue to add the appropriate items until all non-terminals preceded by dots".

The minimal extension is called the **closure** of an item set and written as close (I), where I is an item set. These are closed item sets that we will take as the states of the parser, although only the ones that are actually reachable from the start state.

4. Augmented Grammar

In order to determine the transitions between different states, we should always make the grammar augmented with an extra rule: $S \rightarrow S'$, where S is new start symbol and S' is old start symbol. It should be noted that the parser will use this rule for reduction exactly when it has accepted the input string. For example, consider the following grammar and its augmented grammar.

Grammar	Augmented Grammar
$E \rightarrow E * B$ $E \rightarrow E + B$ $E \rightarrow B$ $B \rightarrow 0$ $B \rightarrow 1$	$S \rightarrow E$ $E \rightarrow E * B$ $E \rightarrow E + B$ $B \rightarrow 0$ $B \rightarrow 1$ $E \rightarrow B$

Finding the reachable item sets and the transition between them: the first step of constructing the tables consists of determining the transitions between the closed item sets. These transitions will be determined as if we are considering a finite automaton that can read terminals as well as non-terminals. The start state of this automaton is always the closure of the first item of the added rule: $S \rightarrow S'$, where S is new start state and S' is old start state.

***Example*-3.6** Consider the following grammar

$E \rightarrow E + E$, $E \rightarrow E * E$, $E \rightarrow (E)$ and $E \rightarrow id$

Construct its LR(0) parse table.

Solution:

Augmented grammar is

$S \to E$ rule 0, $E \to E + E$ rule 1
$E \to E * E$ rule 2 $E \to (E)$ rule 3
$E \to id$ rule 4

where S = new start symbol. Now for this augmented grammar, we will determine following item sets.

(a) Thus, calculating item set-0

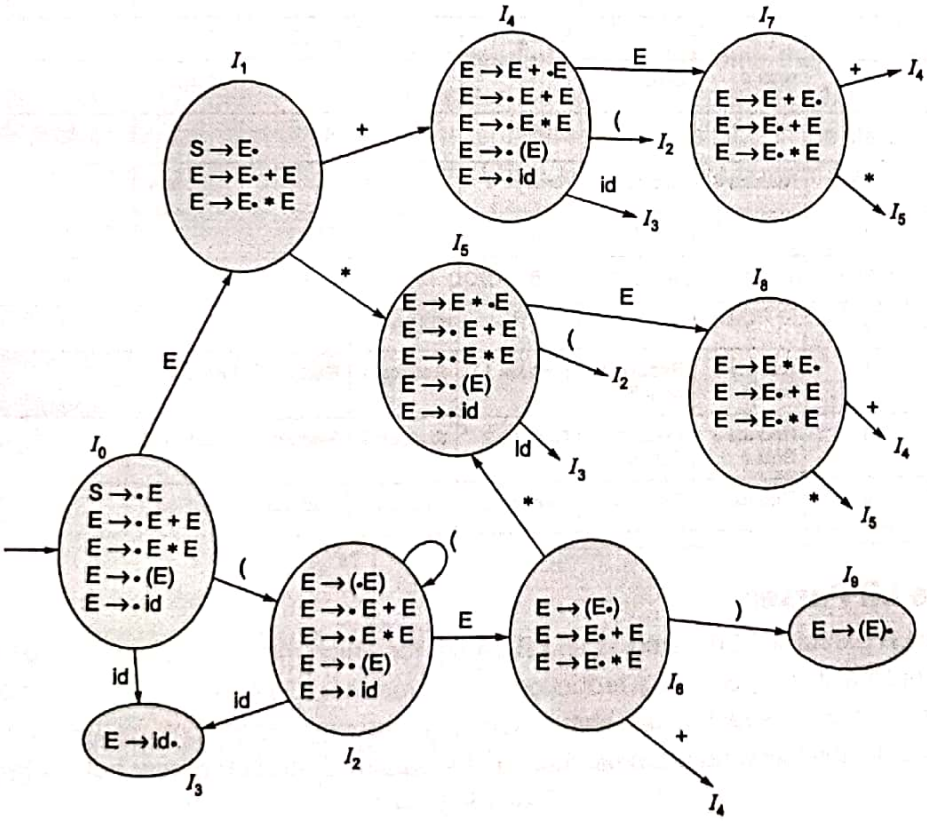
Item set - 0

$S \to . E$ $+ E \to . E + E$
$+ E \to . E * E$ $+ E \to . (E)$
$+ E \to . id$

where denotes closure of E.

(b) Now in item set-0 the symbols just right behind the dot are E, (and Id. For symbol E of item set-0, we have. Thus, calculating Item set-1:

Item Set-1	Item Set-2	Item Set-3	Item Set-4	Item Set-5
$S \to E \bullet$ $E \to E \bullet + E$ $E \to E \bullet * E$ For symbol '(' of item set-0 we have: Thus, calculating item set-2 :	$E \to (\bullet E)$ $E \to \bullet E + E$ $E \to \bullet E * E$ $E \to \bullet (E)$ $E \to \bullet id$ For symbol Id of item set–0 we have: Thus, calculating item set–3	$E \to id \bullet$ Now in Item set–1 the symbols just right behind the dot are + and *. For symbol + of item set–1 we have. Thus, calculating Item set–4	$E \to E + \bullet E$ $E \to \bullet E + E$ $E \to \bullet E * E$ $E \to \bullet (E)$ $E \to \bullet id$ For symbol * of Item set–1, we have:	$E \to E * \bullet E$ $E \to \bullet E + E$ $E \to \bullet E * E$ $E \to \bullet (E)$ $E \to \bullet id$ Similarly all item sets are expanded by reading symbols in each item.



Transition Table

Item Sets	Symbols used in given grammar					
	+	*	(	)	Id	E
0			2		3	1
1	4	5				
2			2		3	6
3						
4			2		3	7
5			2		3	8
6				9		
7	4	5				
8	4	5				
9						

Now we have to construct Action, Goto table

Action GOTO Table

State	Action						Goto
	+	*	(	)	id	$	E
0			Shift 2		Shift 3		1
1	Shift 4	Shift 5				Accept	
2			Shift 2		Shift 3		6
3	Reduce 4	Reduce 4	Reduce 4	Reduce 4	Reduce 4	Reduce 4	
4			Shift 2		Shift 3		7
5			Shift 2		Shift 3		8
6				Shift 9			
7	Reduce 1 Shift 4	Reduce 1 Shift 5	Reduce 1	Reduce 1	Reduce 1	Reduce 1	
8	Reduce 2 Shift 4	Reduce 2 Shift 5	Reduce 2	Reduce 2	Reduce 2	Reduce 2	
9	Reduce 3	Reduce 3	Reduce 3	Reduce 3	Reduce 3	Reduce 3	

3.6 Simple LR Parser

A simple LR parser or SLR parser is an LR parser for which the parsing table are generated as for an LR(0) parser except that it only performs a reduction with a grammar rule $A \rightarrow w$ if the next symbol is in the Follow set of A. Such a parser can prevent certain shift-reduce and reduce-reduce conflicts that occur in LR(0) parser. For example, consider the following grammar that can be parsed by an SLR parser but not by an LR(0) parser.

$$E \rightarrow 1\,E,\ E \rightarrow 1$$

LR(0) parse table which contains shift-reduce conflict is as follows.

State	Action		Goto
	1	$	E
0	shift 2		1
1		accept	
2	shift2, reduce 2	reduce 2	
3	reduce 1	reduce 1	3

Thus, it is not LR(0). Now, the reduction reduce 2 in the Action table of state 2 is due to the item $E \rightarrow 1 \bullet$ and also the reduction reduce 1 in the action table of state 3 is due to the item $E \rightarrow 1\ E \bullet$. Thus, we find the Follow set of E.

$\therefore$ Follow (E) = {$}

Since the follow set of E is {$} so the reduction actions reduce 1 and reduce 2 are only valid in the column for $. Thus, the resultant SLR parse table is as under:

State	Action		Goto
	1	$	E
0	shift 2		1
1		accept	
2	shift 2	reduce 2	3
3		reduce 1	

3.7 Canonical LR Parsing (CLR) and LALR

In the SLR method, we were working with LR(0) items. Therefore, we define an LR(k) item to be an item using lookaheads of length k. So, an LR(1) item is comprised of two parts: the LR(0) item and the lookahead associated with the item. If we work with LR(1) items instead of using LR(0) items, then every state of the parser will correspond to a set of LR(1) items. When the parser looks ahead in the input buffer to decide whether reduction is to be done or not, the information about the terminals will be available in the state of the parser itself, which is not the case with SLR parser state. Hence, with LR(1), we get a more powerful parser.

NOTE: CLR(1) parser is also called as LR(1), sometimes general LR parser.

Therefore, if we modify the closure and the goto functions to work suitably with the LR(1) items, by allowing them to compute the lookaheads, we can obtain the canonical collection of sets of LR(1). And from this we can obtain the parsing Action I GOTO table.

3.7.1 Closure Operation

```
function closure (I);
begin
  repeat
    for each item [A → α·Bβ, a] in I,
      each production B → γ in G′.
        and each terminal b in FIRST(βa)
        such that [B → ·γ, b] is not in I do
          add[B → ·γ, b] to I;
      until no more items can be added to I;
    return I
end;
```

Example: Consider the following augmented grammar.

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC \mid d$

If I is the set of one item {[$S' \rightarrow \cdot S$, \$]}, then closure I contains the items

$S' \rightarrow \cdot S$, \$
$S \rightarrow \cdot CC$, \$
$C \rightarrow \cdot cC$, c
$C \rightarrow \cdot cC$, d
$C \rightarrow \cdot d$, c
$C \rightarrow \cdot d$, d

3.7.2 GOTO Operation

```
function goto (I, X);
begin
  let J be the set of items [A → αX·β, a] such that
    [A → α·Xβ, a] is in I;
  return closure (J)
end;
```

If I is the set of four items {[$S' \rightarrow \cdot S$,\$], [$S \rightarrow \cdot$ CC, \$], [$C \rightarrow \cdot cC$, c | d], [$C \rightarrow \cdot d$, c | d]}, then goto (I, S) consists of $S' \rightarrow S\cdot$, \$ only.

3.7.3 Construction of the Sets of LR(1) Items

Algorithm: Construction of the sets of LR(1) items

Input: An augmented grammar G′.

Output: The sets of LR(1) items that are the set of items valid for one or more viable prefixes of G′.

Method: The procedures closure and goto and the main routine items for constructing the sets of items are used.

```
procedure items (G′);
  begin
    C := {closure ({[S′ → ·S, $]})};
      repeat
        for each set of items I in C and each grammar symbol X
        such that goto(I, X) is not empty and not in C do
        add goto (I, X) to C
      until no more sets of items can be added to C
  end;
```

Example: Consider the following augmented grammar.

$S' \rightarrow S,\ S \rightarrow CC,\ C \rightarrow cC \mid d$

The canonical collection of sets of LR(1) items for grammar are shown below:

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

I_1: $S' \rightarrow S\cdot$, \$

I_2: $S \to C \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

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

I_4: $C \to d \cdot, c|d$

I_5: $S \to CC \cdot, \$$

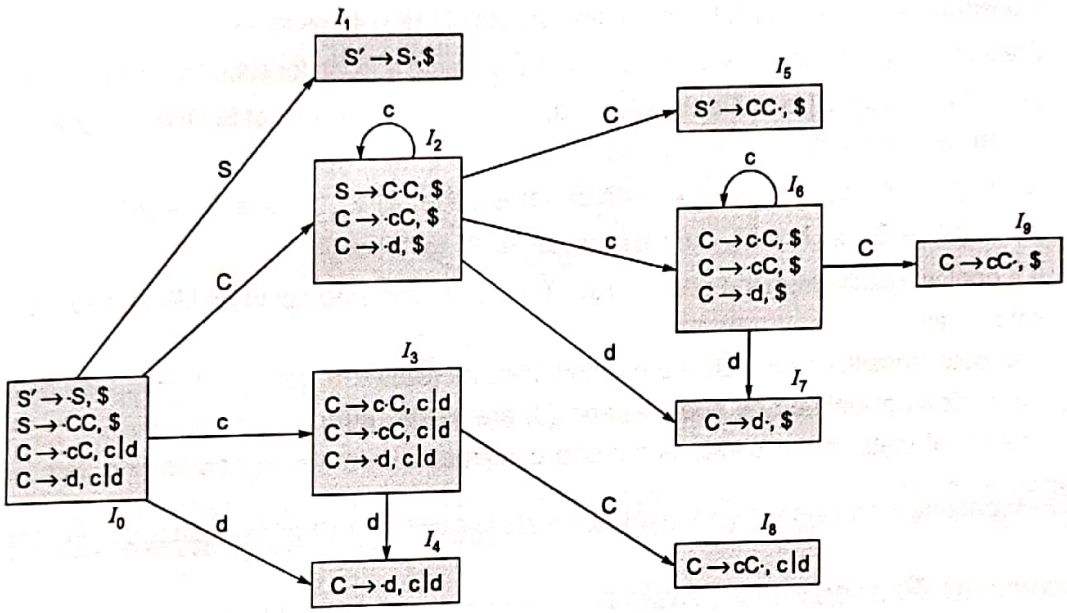
I_6: $C \to c \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

I_7: $C \to d \cdot, \$$

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

I_9: $C \to cC \cdot, \$$

The goto function for this set of items is shown as the transition diagram of a deterministic finite automation in following figure:



NOTE

- Every SLR(1) grammar is an LR(1) grammar, but for an SLR(1) grammar the canonical LR parser may have more states than the SLR parser for the same grammar. But CLR(1) is more powerful than SLR(1).
- By comparing the SLR(1) parser with the CLR(1) parser, we find that the CLR(1) parser is more powerful. But the CLR(1) has a greater number of states than the SLR(1) parser; hence, its storage requirement is also greater than the SLR(1) parser. Therefore, we can devise a parser that is an intermediate between the two; that is, the parser's power will be in between that of SLR(1) and CLR(1), and its storage requirement will be the same as SLR(1)'s. Such a parser, LALR(1), will be much more useful; since each of its states corresponds to the set of LR(1) items, the information about the lookaheads is available in the state itself, making it more powerful than the SLR parser. But a state of the LALR(1) parser is obtained by combining those states of the CLR parser that have identical LR(0) (core) items, but which differ in the lookaheads in their item set representations.

3.7.4 Construction of the Set of LALR Items Table

The steps in constructing an LALR items table are as follow:

1. Obtain the canonical collection of sets of LR(1) items.
2. If more than one set of LR(1) items exists in the canonical collection obtained that have identical cores or LR(0)s, but which have different lookaheads, then combine these sets of LR(1) items to obtain a reduced collection, C_1, of sets of LR(1) items.

3.7.5 Algorithm: Construction of the Canonical LR and LALR Parsing Table

Input: An augmented grammar G′.

Output: The canonical LR parsing table functions action and goto for G′.

Method:

1. Construct C = $\{I_0, I_1, ..., I_n\}$, the collection of sets of LR(1) items for G′.
2. State i of the parser is constructed from I_i. The parsing actions for state i are determined as follows:
   (a) If $[A \to \alpha \cdot a\beta, b]$ is in I_i and goto $(I_i, a) = I_j$, then set action$[i, a]$ to "shift j." Here, a is required to be a terminal
   (b) If $[A \to \alpha \cdot, a]$ is in I_i, $A \neq S'$, then set action$[i, a]$ to "reduce $A \to \alpha$·"
   (c) If $[S' \to S \cdot, \$]$ is in I_i, then set action $[i, \$]$ to "accept."
   If a conflict results from the above rules, the grammar is said not to be LR(1), and the algorithm is said to fail.
3. The goto transitions for state i are determined as follows: If goto $(I_i, A) = I_j$, then goto $[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set containing item $[S' \to \cdot S, \$]$.

NOTE: LALR parsing is less powerful than LR(1), but more powerful than SLR. Yacc uses LALR(1) parsing.

3.8 Operator Precedence Parsing

It is a form of shift-reduce parsing, which is easy to implement. These grammars have the property (among other essential requirements) that no production right side is ∈ or has two adjacent non-terminals. A grammar with these properties is called an operator grammar. In operator precedence parsing, we define three disjoint precedence relations, <·, ≐, and ·>, between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings:

Relation	Meaning
a <· b	a "yield precedence to" b
a ≐ b	a "has the same precedence as" b
a ·> b	a "takes precedence over" b

The common way of determining what precedence relations should hold between a pair of terminals is notions of associativity and precedence of operators.

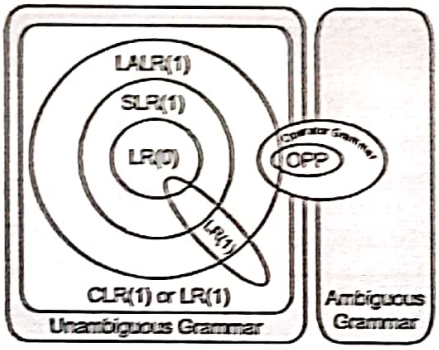
For *example*, if * is to have higher precedence than +, we make + <· * and * ·> +. Consider the following operator grammar: E → E + E|E*E|(E)|id with the assumptions:

1. * is of highest precedence and left-associative.
2. + is of lowest precedence and left-associative.

The operator-precedence relations for this grammar are shown in the following table.

	+	*	(	)	id	$
+	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	
)	$\gtrdot$	$\gtrdot$		$\gtrdot$		$\gtrdot$
id	$\gtrdot$	$\gtrdot$		$\gtrdot$		$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$	

3.9 Hierarchy of Grammar Classes



LL(k): For a grammar to be LL(k), we must be able to recognize the use of a production by seeing only the first k symbols of what its right-hand side derives.

LR(k): For a grammar to be LR(k), we must be able to recognize the use of a production by having seen all of what is derived from its right-hand side with k more symbols of look ahead.

Summary

- Syntax analyzer is also known as parser.
- A parser works on stream of tokens which are generated by lexical analyser.
- For a given input string if there exists 'more than one parse tree' then that grammar is called as "ambiguous grammar".
- A predictive parser (a top down parser without backtracking) insists that the grammar must be left-factored.
- Left factoring avoids backtracking in parser but is not a solution for elimination of ambiguity.
- FIRST(A) is a set of the terminal symbols which occur as first symbols in strings derived from A.
- Follow(A) is the set of terminals which occur immediately after the nonterminal A in the strings derived from the starting symbol.
- Handle pruning: The process of finding the handle and replacing the handle by LHS of corresponding production is called "handle priority".
- Bottom up parsing is also known as shift reduce parser.
- Bottom up parsing can be constructed for both ambiguous and unambiguous grammar.

- For unambiguous grammar, LR parser can be constructed and Operator Precedence Parser (OPP) can be constructed for both ambiguous and unambiguous grammar.
- Bottom up parsing is more powerful than top down parser.
- Bottom up parser simulates the reverse of right most derivation.
- SLR(1) is powerful than LR(0).
- Size of SLR(1) parse table is same as size of LR(0) parse table.
- Every LR(0) grammar is SLR(1) but every SLR(1) need not be LR(0).
- Every LALR(1) grammar is CLR(1) but every CLR(1) grammar need not be LALR(1).
- LALR(1) parsers are often used in practice as parsing tables are smaller than CLR(1).
- OPP < LL(1) < LR(0) < SLR(1) ≤ LALR(1) ≤ CLR(1).
- Number of states (SLR(1)) = Number of states (LALR(1)) ≤ Number of states (CLR(1)).
- LR(0) ⊂ SLR(1) ⊂ LALR(1) ⊂ CLR(1).
- LL(1) ⊂ LALR(1) ⊂ CLR(1).

Student's Assignments

Q.1 Consider the following grammar:

$S \rightarrow Aa \mid b$

$S \rightarrow a$

Which parsing method works more efficient for it?

(a) Top-down (b) Bottom-up

(c) Both (a) and (b) (d) None of these

Q.2 Consider the following statements:

S1: The grammar $S \rightarrow 0 \mid 12 \mid 345$ is LL(1).

S2: This grammar $S \rightarrow 0 \mid T1$

$T \rightarrow 1 \mid S0$ is LL(1).

S3: This grammar $S \rightarrow 0 \mid 11 \mid 01 \mid$ is LL(1).

Which of the following statements are true ?

(a) *S*1 and *S*3 (b) *S*2 and *S*3

(c) *S*1 and *S*2 (d) None of these

Q.3 Consider the following grammar:

$S \rightarrow aS \mid bS \mid abaA$

$A \rightarrow \varepsilon \mid aA \mid bA$

Consider the following statements:

1. Parse table of given grammar contains two rows and three columns.
2. Parse table can be constructed by using FIRST set only.
3. FIRST sets for non-terminals S and A contains 3 elements.
4. FOLLOW(S) = FOLLOW(A) = {$, a}.
5. Given grammar is LL(1).

Which of the following statements are true?

(a) 1, 2, and 3 (b) 3 and 4

(c) 4 and 5 (d) None of these

Q.4 Consider the following grammar:

$E \rightarrow E(T) \mid T$

$T \rightarrow T * F \mid id$

$F \rightarrow (id)$

Which of the following can be correct handle in bottom up parsing for the above grammar?

(a) id * (id) (b) id * F

(c) E(id) (d) (id)

Q.5 Consider the following FIRST and FOLLOWs:

FIRST(X) = {b, d, f}

FIRST(Y) = {b, d}

FIRST(Z) = {c, e}

FOLLOW(d) = {c, e}

FOLLOW(e) = {a}

FOLLOW(f) = {$}

FOLLOW(X) = {$}

FOLLOW(Y) = {c, e}

FOLLOW(Z) = {a}

FOLLOW(a) = {$}
FOLLOW(b) = {b, d}
FOLLOW(c) = {c, e}

Which of the following grammar satisfies the above FIRST and FOLLOW sets?

(a) $X \to Ya \mid f$
$Y \to bY \mid d$
$Z \to cZ \mid e$

(b) $X \to YZa \mid f$
$Y \to bY \mid d$
$Z \to cZ \mid e$

(c) $X \to bY \mid d$
$Y \to Ya \mid f$
$Z \to cZ \mid e$

(d) $X \to bY \mid e$
$Y \to cZ \mid d$
$Z \to Ya \mid f$

Q.6 Consider the following statements:

1. LL(k) parsing uses the next k input tokens.
2. LL(k) parsing is more powerful than LL(1) parsing.
3. LL(k) languages are proper subset of LL(k–1) languages.
4. C programming language is LL(1).
5. LL(k) grammar does not exists for the programming language.

Which of the following statements are true?

(a) 1 and 2 (b) 3 and 4
(c) 3 and 5 (d) Both (a) and (b)

Q.7 Consider a bottom-up parser for a grammar with no ε-production and no single production (i.e., productions with a single symbol on the right-hand side). For an input string with n tokens, what is the maximum number of reduce moves the parser can take?

(a) 2 (b) n
(c) $n-1$ (d) $n+1$

Q.8 Consider the following grammar:

$S' \to S\$$
$S \to xSS$
$S \to y$

Consider the following statements.

S1: The grammar is SLR.
S2: This grammar is not SLR.
S3: The grammar is LR(0).
S4: This grammar is not LR(0).

Which of the following statements are true?

(a) S1 and S3 (b) S2 and S4
(c) S2 and S3 (d) None of these

Q.9 Consider the following grammar:

$exp' \to exp\ \$$
$exp \to id$
$exp \to (exp)$
$exp \to (id)\ exp$ ($ is end marker)

Consider the following statements.

S1: This grammar is SLR.
S2: This grammar is not SLR.
S3: This grammar is not LR (0).
S4: This grammar is not LR (0).

Which of the statements are true?

(a) S1 and S4 (b) S1 and S3
(c) S2 and S4 (d) S2 and S3

Q.10 Consider the following grammar.

$S \to X \mid ay$
$X \to xXy \mid Y$
$Y \to a$

Consider the following statements.

1. Given grammar is SLR(1).
2. FIRST(X) contains 2 elements.

Which of the following statements are true?

(a) 1 (b) 2
(c) Both (a) and (b) (d) None of these

Q.11 Consider the following grammar:

$E \to int \mid int + E \mid int + E \mid int - E \mid int - (E) \mid int * E$

Consider the following statements.

1. Grammar is left factored.
2. Grammar is not left factored.

Which of the following statements are true?

(a) 1
(b) Cannot be determined
(c) 2
(d) Neither is true

Q.12 Consider the following grammar:

$E \to A + B \mid B$
$B \to int \mid (A) \mid B * int$

Consider the following statements:

1. Grammar is left recursive.
2. Grammar is not left recursive.

Which of the following statements are true?

(a) 1
(b) Cannot be determined
(c) 2
(d) Neither is true

Q.13 Consider the following statements:

S1: Left factoring is a technique that can be used to prepare a grammar for use in a recursive descent parser.

S2: Left factoring is a technique that can be used to produce a leftmost derivation of a string from a grammar.

Which of the following statements is correct?

(a) S1 and S2 are both true
(b) S1 is true, S2 is false
(c) S1 is false, S2 is true
(d) S1 and S2 are both false

Q.14 Consider the following statements:

S1: Left factoring is a technique that can be used to remove left recursion from a grammar.

S2: Left factoring is a technique that can be used to factor out left associative operators.

Which of the following statements is correct?

(a) S1 and S2 are both true
(b) S1 is true, S2 is false
(c) S1 is false, S2 is true
(d) S1 and S2 are both false

Q.15 Consider the following statements:

S1: LALR(1) parsers are more powerful than LR(1) parsers.

S2: Every SLR(1) grammar is a LR(1) grammar, but every LR(1) grammar is not necessarily SLR(1).

Which of the following statements is correct?

(a) S1 and S2 are both true
(b) S1 is true, S2 is false
(c) S1 is false, S2 is true
(d) S1 and S2 are both false

Q.16 Consider the following statements:

S1: A LR(1) parser processes the input symbols from left to right.

S2: A LR(1) parser looks ahead at most one input symbol before knowing what action to take.

Which of the following statements is correct?

(a) S1 and S2 are both true
(b) S1 is true, S2 is false
(c) S1 is false, S2 is true
(d) S1 and S2 are both false

Q.17 Consider the following statements:

S1: A LR(1) parser processes a leftmost derivation.

S2: A LR(1) parser takes time proportional to the cube of the number of input symbols.

Which of the following statements is correct?

(a) S1 and S2 are both true
(b) S1 is true, S2 is false
(c) S1 is false, S2 is true
(d) S1 and S2 are both false

Q.18 Consider the following statements:

S1: A grammar with FIRST/FIRST conflict can be made LL(1) by only applying left factoring, that is, no substitution and left-recursion removed are needed.

S2: The following two items: $A \rightarrow \bullet x$ and $B \rightarrow x\bullet$ can coexist in an LR item set.

Which of the following statements is correct?

(a) S1 and S2 are both true
(b) S1 is true, S2 is false
(c) S1 is false, S2 is true
(d) S1 and S2 are both false

Q.19 Consider the following statements:

S1: $S \rightarrow A|xb,\ A \rightarrow aAb|x$

This grammar contains a shift-reduce conflict.

S2: When constructing an LR(1) parser we record for each item exactly in which context it appears, which resolves many conflicts present in SLR(1) parser based on FOLLOW sets. The look ahead set associated with an LR(1) item can contains only one element.

(a) S1 and S2 are both true
(b) S1 is true, S2 is false
(c) S1 is false, S2 is true
(d) S1 and S2 are both false

Q.20 Consider the following statements:

S1: When constructing a top-down parser we need to compute the FIRST sets of all production alternatives. The FIRST set of an alternative α, FIRST(α), contains all terminals α can start with; if α can produce the empty string ε, this ε is included in FIRST(α).

$S \rightarrow AB'$
$A \rightarrow \varepsilon \mid aA$
$B \rightarrow b \mid bB$
First(S) contains 3 elements.

S2: In an attribute grammar each production rule ($N \rightarrow \alpha$) has a corresponding attribute evaluation rule that describes how to compute the values of the inherited attributes of each particular node N in the AST.

Which of the following statements is correct?
(a) S1 and S2 are both true
(b) S1 is true, S2 is false
(c) S1 is false, S2 is true
(d) S1 and S2 are both false

Q.21 Consider the following statements:

S1: $S \rightarrow A \mid xb$, $A \rightarrow aAb \mid x$
This is neither LR(0) nor SLR(1) grammar.

S2: When constructing a top-down parser we need to computer the FIRST sets of all production alternatives. The FIRST set of an alternative α, contains all terminals α can start with: if α can produce the empty string ε, this is included in FIRST (α).

$S \rightarrow AB$
$A \rightarrow \varepsilon \mid aA$
$B \rightarrow b \mid bB$
First (S) contains 2 elements.

Which of the following statements is correct?
(a) S1 and S2 are both true
(b) S1 is true, S2 is false
(c) S1 is false, S2 is true
(d) S1 and S2 are both false

Q.22 Consider the following statements:

S1: $S \rightarrow A \mid x$, $A \rightarrow aAb \mid x$
This grammar do not contains a conflict.

S2: $S \rightarrow aS \mid Sa \mid c$
This grammar is ambiguous.

Which of the following statements is correct?
(a) S1 and S2 are both true
(b) S1 is true, S2 is false
(c) S1 is false, S2 is true
(d) S1 and S2 are both false

Q.23 Consider the following grammar G.

$S \rightarrow AB \mid d$
$A \rightarrow aA \mid b$
$B \rightarrow bB \mid c$

The grammar G is
(a) LL (1) grammar and not LR (0)
(b) LL (1) and LR (0)
(c) Not LL (1) but LR (0)
(d) Neither LL (1) nor LR (0)

Q.24 Which of the following could result shift-reduced conflict in LR (0) parser?
(a) {A → b.c, B → b.}
(b) {A → a.b, B → b.}
(c) {A → ε., B → .a}
(d) All of these

Q.25 Consider the following CFG and LL (1) table.

$S \rightarrow gATe$
$A \rightarrow gA \mid \varepsilon$
$T \rightarrow gA \mid \varepsilon$

	g	e	$
S	S → gATe		
A	E_1	E_2	
T	T → gA		

Find the number of productions entry at E_1.

Q.26 Consider the following CFG:

$S \rightarrow aAb \mid eb$
$A \rightarrow e \mid f$

Find the number of states in LR (0) construction.

Q.27 Consider the following sets of items for some CFG produced while constructing LR (1) parser.

Set 1(State): A → a., {b}
B → ba., {b,c}

Set 2 (State): A → B.a, {$}
B → aB., {b}

Each item has look-a-head part which is computed using CLR (1) parser. Assume the grammar has follow sets as: **Follow (A) = {a, b} and Follow (B) = {b, c}**

Which of the following can be concluded from above two states of CLR (1) construction. [All other states are not in conflict]

(a) Grammar is SLR (1)
(b) Grammar is CLR (1) but not SLR (1)
(c) Grammar is not CLR (1) due to SR conflict
(d) Grammar is not CLR (1) due to RR conflict

Q.28 Consider the following CFG:

S → aAb | aBc | bAd | bBe
A → g
B → g

How many states exist in DFA using LALR (1) constructions for the above grammar?

Answer Key:

1. (b)	**2.** (d)	**3.** (d)	**4.** (d)	**5.** (b)
6. (d)	**7.** (c)	**8.** (a)	**9.** (a)	**10.** (b)
11. (b)	**12.** (a)	**13.** (b)	**14.** (b)	**15.** (c)
16. (a)	**17.** (d)	**18.** (c)	**19.** (b)	**20.** (d)
21. (c)	**22.** (c)	**23.** (b)	**24.** (d)	**25.** (2)
26. (9)	**27.** (d)	**28.** (13)		

CHAPTER 04

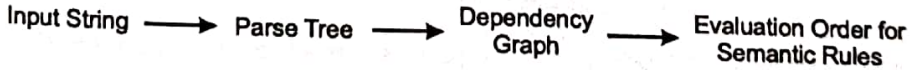
Syntax Directed Translation and Intermediate Code Generation

4.1 Introduction

The translation of languages guided by context-free grammars. We associate information with a programming language construct by attaching attributes to the grammar symbols representing the construct. Values for attributes are computed by "semantic rules" associated with the grammar productions.

There are two notations for associating semantic rules with productions, syntax-directed definitions and translation schemes. Syntax-directed definitions are high-level specifications for translations. They hide many implementation details and free the user from having to specify explicitly the order in which translation takes place. Translation schemes indicate the order in which semantic rules are to be evaluated, so they allow some implementation details to be shown.

Conceptually, with both syntax-directed definitions and translation schemes, we parse the input token stream, build the parse tree, and then traverse the tree as needed to evaluate the semantic rules at the parse-tree nodes. Evaluation of the semantic rules may generate code, save information in a symbol table, issue error messages, or perform any other activities. The translation of the token stream is the result obtained by evaluation of the semantic rules.



4.2 Syntax-Directed Definition

A syntax-directed definition is a generalization of a context-free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called the synthesized and inherited attributes of that grammar symbol. An attribute can represent anything we choose: a string, a number, a type, a memory location, or whatever. The value of an attribute at a parse tree node is defined by a semantic rule associated with the production used at that node.

Semantic rules set up dependencies between attributes that will be represented by a graph. From the dependency graph, we derive an evaluation order for the semantic rules. A parse tree showing the values of attributes at each node is called an annotated parse tree. The process of computing the attribute values at the nodes is called annotating or decorating the parse tree.

Form of a Syntax-Directed Definition

In a syntax-directed definition, each grammar production $A \rightarrow a$ has associated with it a set of semantic rules of the form $b := f(c_1, c_2, \ldots, c_k)$ where f is a function, and either

1. b is a synthesized attribute of A and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of the production, or
2. b is an inherited attribute of one of the grammar symbols on the right side of the production, and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of the production.

In either case, we say that attribute b depends on attributes $c_1, c_2, \ldots, c_k$. An attribute grammar is a syntax-directed definition in which the functions in semantic rules cannot have side effects.

Example:

Production	Semantic Rules
$E \rightarrow E_1 + T$	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
$F \rightarrow id$	F.val := numlexval

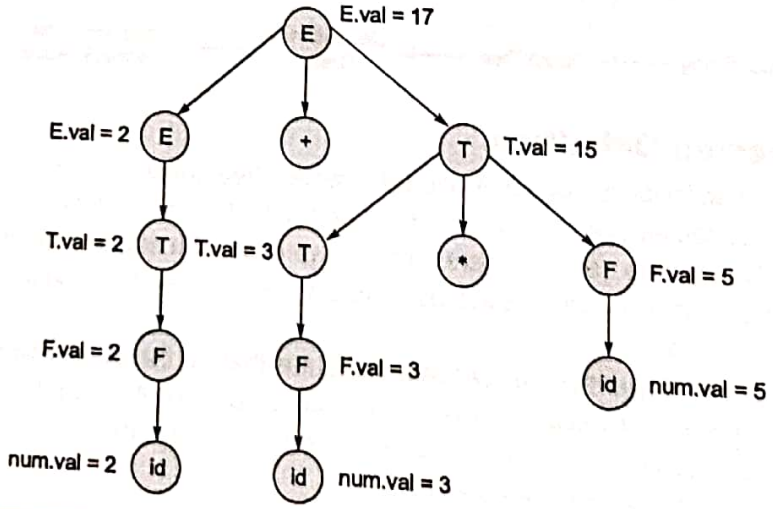
4.2.1 Synthesized Attributes

An attribute is said to be synthesized if its value at a parse tree node is determined by the attribute values at the child nodes. A synthesized attribute has a desirable property; it can be evaluated during a single bottom-up traversal of the parse tree.

Example:

$E \rightarrow E_1 + T$	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
$F \rightarrow id$	F.val := num.lexval

A parse tree, along with the values of the attributes at the nodes, for an expression 2 + 3 * 5 is shown in the following figure. Syntax-directed definitions that use synthesized attributes only are known as "S-attributed" definitions.

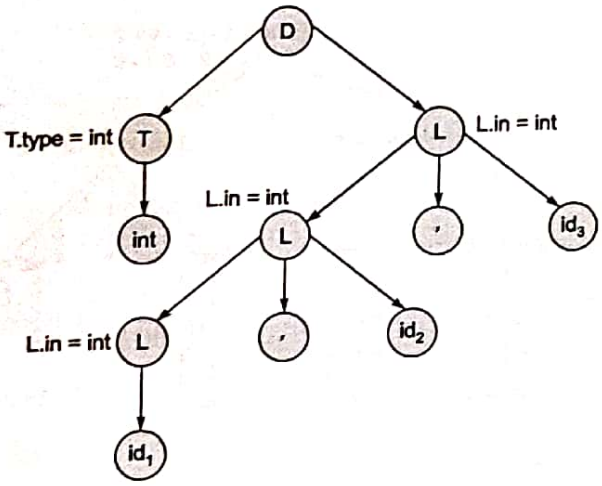


4.2.2 Inherited Attributes

Inherited attributes are those whose initial value at a node in the parse tree is defined in terms of the attributes of the parent and/or siblings of that node. For example, syntax-directed definitions that use inherited attributes are given below:

$D \rightarrow TL$	L.in = T.type
$T \rightarrow$ int	T.type = int
$T \rightarrow$ real	T.type = real
$L \rightarrow L_1$,id	L_1.in = L.in addtype(id.entry, L.in)
$L \rightarrow$ id	addtype(id.entry, L.in)

A parse tree, along with the attributes values at the parse tree nodes, for an input string int id_1,id_2,id_3 is shown in following figure



4.2.3 Comparison between Inherited and Synthesized Attribute

Synthesized Attribute	Inherited Attribute
1. A synthesized attribute for a non-terminal A at a parse-tree node N is defined by a semantic rule associated with the production at N.	1. A inherited attribute for a non-terminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N.
2. The production must have A as its head.	2. The production must have B as a symbol in its body.
3. A synthesized attribute at node N is defined only in terms of attribute values at the children of N itself.	3. An inherited attribute at node N is defined only in terms of attribute values of N's parent, N itself, and N's siblings
4. Ex.: E·val $\rightarrow$ F·val E val ↑ F val val is a synthesized attribute.	3. Ex.: T·avail = F·avail F·code = T·code T avail code ↑ ↓ F avail code avail is synthesized, code is inherited.

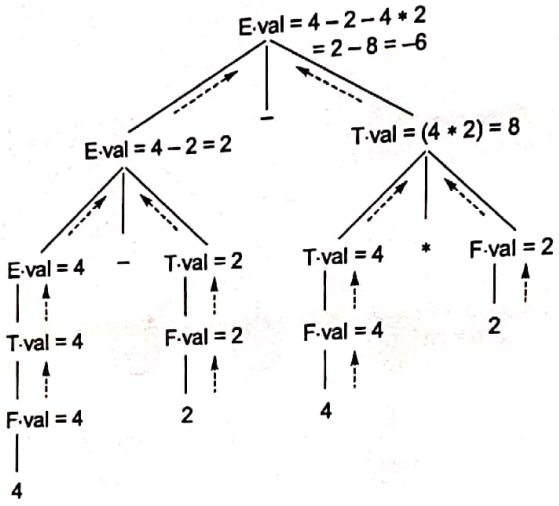
Example-4.1 Consider the following SDTs:

$E \rightarrow E_1 - T$ (E·val = E_1·val – T·val)
$E \rightarrow T$ (E·val = T·val)
$T \rightarrow T_1 * F$ (T·val = T·val * F·val)
$T \rightarrow F$ (T·val = F·val)
$F \rightarrow 2$ (F·val = 2)
$F \rightarrow 4$ (F·val = 4)

Using the above SDTs, construct a parse tree for the expression: 4 – 2 – 4 * 2.

Solution:

Annoted parse tree for the given syntax directed definition with synthesized attribute 'val' is:



Here the dashed arrow show the movement of value from lowest level to the upper levels and the final value calculated of E·val = 4 – 2 – 4 * 2 = –6.

Example-4.2 Consider the following grammar:

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

Associate semantic rules with the productions for construction of syntax tree for an expression using the translation scheme construct the syntax tree for an expression: *a + b * c*.

Solution:

Let us consider, the following three attributes for the calculation of semantics to the productions of given grammar:

- **Avail:** used to denote the first unused temporary compiler generated location which is available by default at the time of execution.
- **Instruct:** it is used to denote the sequence of instructions for an assignment, expression, term etc.
- **mlresult:** it is used to denote the memory location to hold the result of any calculation.

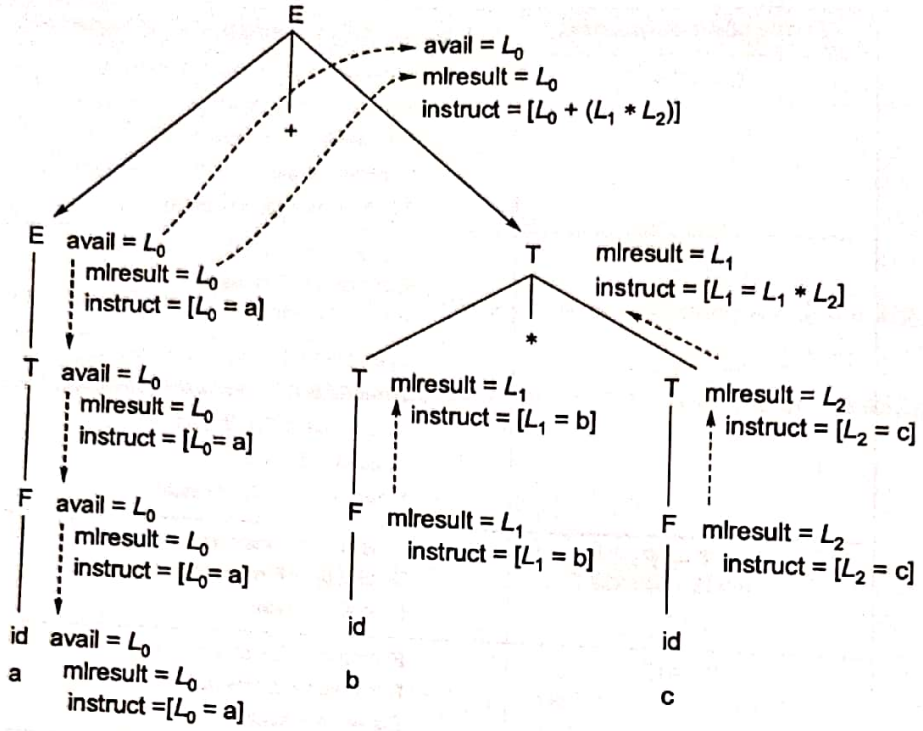
Another compiler dependent function 'next' will be used to allocate the next memory location.
Now, attributed grammar as given:

Production of grammar	Associated semantics with the grammar
$E \rightarrow E + T$	E·instruct = [E_1·instruct, T·instruct] E_1·mlresult = E_1·mlresult + T·mlresult E·mlresult = E_1·mlresult E_1·avail = E·avail T·avail = next (E_1·mlresult)
$E \rightarrow T$	E·instruct = T·instruct E·mlresult = T·mlresult T·avail = E·avail
$T \rightarrow T * F$	T_0·instruct = [T_1·instruct, F·instruct;] T_1·mlresult = T_1·mlresult + F·mlresult T_0·mlresult = T_1·mlresult T_1·avail = T_0·avail F·avail = next (T_1·mlresult)
$T \rightarrow F$	T·instruct = F·instruct T·mlresult = F·mlresult F·avail = T·avail
$F \rightarrow (E)$	F·instruct = E·instruct F·mlresult = E·mlresult E·avail = F·avail
$F \rightarrow id$	F·instruct = F·avail: = id F·mlresult = F·avail

Analysis of Attribute Type:

Production	Attributes added to production	Type/categorisation of attribute used
$E \rightarrow E + T$	instruct mlresult avail	synthesized synthesized inherited
$E \rightarrow T$	instruct mlresult avail	synthesized synthesized inherited
$T \rightarrow T * F$	instruct mlresult avail	synthesized synthesized inherited
$T \rightarrow F$	constant mlresult	synthesized synthesized
$F \rightarrow (E)$	instruct mlresult avail	synthesized synthesized inherited
$F \rightarrow id$	instruct avail mlresult	synthesized inherited synthesized

Construction of syntax tree for $a+b*c$



Example-4.3 Let synthesized attribute 'val', give the value of binary number generated by S in the following grammar on input 101·101, S·val = 5.625

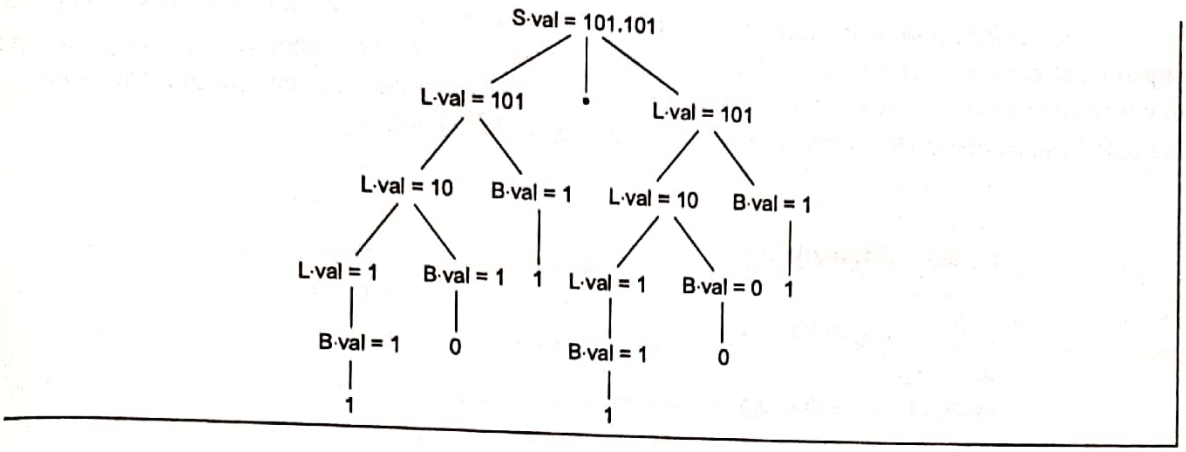
$$S \rightarrow L{\cdot}L \mid L$$
$$L \rightarrow LB \mid B$$
$$B \rightarrow 0 \mid 1$$

Write synthesized attribute value corresponding to each production to determine S·val.

Solution:

Production	Synthesized attribute value
$S \rightarrow L{\cdot}L$	S·val = L·val·L·val
$S \rightarrow L$	S·val = L·val
$L \rightarrow LB$	L·val = L·val B·val = L·val
$L \rightarrow B$	L·val = B·val
$B \rightarrow 0$	B·val = 0·val $\Rightarrow$ B·val = 0
$B \rightarrow 1$	B·val = 1·val $\Rightarrow$ B·val = 1

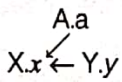
Parse tree for the evaluation of given input string.



4.2.4 Dependency Graph

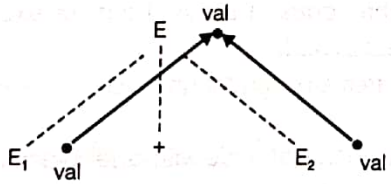
If an attribute 'b' at a node in a parse tree depends on an attribute 'c', then the semantic rule for 'b' at that node must be evaluated after the semantic rule that defines 'c'. The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph. Suppose A.a := f(X.x, Y.y) is a semantic rule for the production A → XY. This rule defines a synthesized attribute A.a that depends on the attributes X.x and Y.y.

If the production A → XY has the semantic rule X.x := g(A.a, Y.y) associated with it, then there will be an edge to X.x from A.a and also an edge to X.x from Y.y, since X.x depends on both A.a and Y.y.



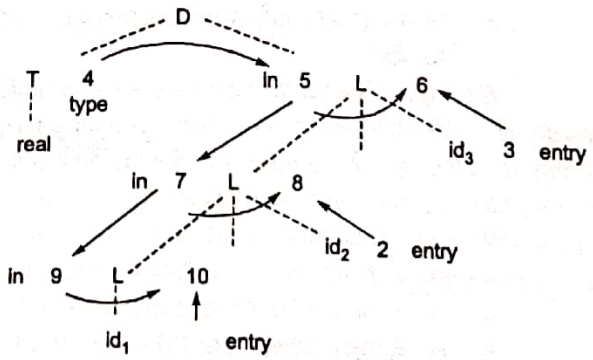
Example:

Production	Semantic Rule
$E \rightarrow E_1 + E_2$	E.val := E_1.val + E_2.val



4.2.5 Evaluation Order

A topological sort of a directed acyclic graph is any ordering m_1, m_2,..., m_k of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes; that is, if $m_i \rightarrow m_j$ is an edge from m_i to m_j, then m_i appears before m_j in the ordering. Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated.



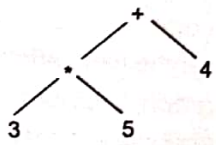
Each of the edges in the dependency graph in the above figure goes from a lower-numbered node to a higher-numbered node. Hence, a topological sort of the dependency graph is obtained by writing down the nodes in the order of their numbers. From this topological sort, we obtain the following program. We write a_n for the attribute associated with the node numbered n in the dependency graph.

a_4 := real;
a_5 := a_4;
addtype (id$_3$.entry, a_5);
a_7 := a_5;
addtype (id$_2$.entry, a_7);
a_9 := a_7;
addtype (id$_1$.entry, a_9);

4.3 Construction of Syntax Trees

4.3.1 Syntax Trees

An (abstract) syntax tree is a condensed form of parse tree useful for representing language constructs. In a syntax tree, operators and keywords do not appear as leaves, but rather are associated with the interior node that would be the parent of those leaves in the parse tree. Another simplification found in syntax trees is that chains of single productions may be collapsed; the syntax tree of 3 * 5 + 4 is as shown in figure.



4.3.2 Constructing Syntax Trees for Expressions

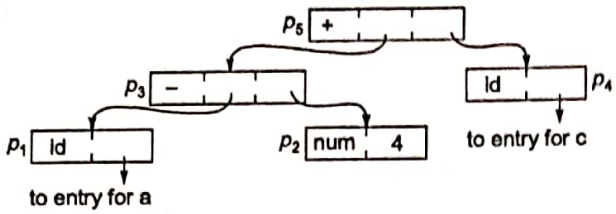
The construction of a syntax tree for an expression is similar to the translation of the expression into postfix form. We construct subtrees for the subexpressions by creating a node for each operator and operand. The children of an operator node are the roots of the nodes representing the subexpressions constituting the operands of that operator.

Each node in a syntax tree can be implemented as a record with several fields. In the node for an operator, one field identifies the operator and the remaining fields contain pointers to the nodes for the operands. The operator is often called the label of the node. When used for translation, the nodes in a syntax tree may have additional fields to hold the values (or pointers to values) of attributes attached to the node. In this section, we use the following functions to create the nodes of syntax trees for expressions with binary operators. Each function returns a pointer to a newly created node.

1. mknode (op, left, right) creates an operator node with label op and two fields containing pointers to left and right.
2. mkleaf (id, entry) creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry for the identifier.
3. mkleaf (num, val) creates a number node with label num and a field containing val, the value of the number.

Example: The following sequence of functions calls creates the syntax tree for the expression $a - 4 + c$ as shown in the following figure. In this sequence, p_1, p_2, ..., p_5 are pointers to nodes, and entrya and entryc are pointers to the symbol table entries for identifiers a and c, respectively.

1. p_1 := mkleaf (id, entrya);
2. p_2 := mkleaf (num, 4);
3. p_3 := mknode ('–', p_1, p_2);
4. p_4 := mkleaf (id, entryc);
5. p_5 := mknode ('+', p_3, p_4);



The tree is constructed bottom up. The function calls mkleaf (id, entrya) and mkleaf (num, 4) construct the leaves for a and 4; the pointers to these nodes are saved using p_1 and p_2. The call mknode ('−', p_1, p_2) then constructs the interior node with leaves for a and 4 as children. After two more steps, p_5 is left pointing to the root.

4.3.3 Syntax Directed Definition for Constructing Syntax Trees

Following table contains an S-attributed definition for constructing a syntax tree for an expression containing the operators + and –. It uses the underlying productions of the grammar to schedule the calls of the functions mknode and mkleaf to construct the tree. The synthesized attribute nptr for E and T keeps track of the pointers returned by the function calls.

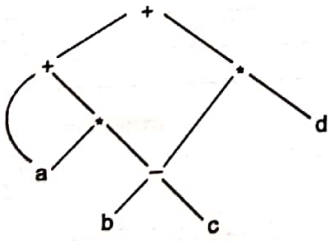
Production	Semantic Rules
$E \rightarrow E_1 + T$	Enptr := mknode ('+', E_1.nptr, T.nptr)
$E \rightarrow E_1 - T$	Enptr := mknode ('−', E_1.nptr, T.nptr)
$E \rightarrow T$	Enptr := T.nptr
$T \rightarrow (E)$	T.nptr := Enptr
$T \rightarrow id$	T.nptr := mkleaf(id, id.entry)
$T \rightarrow num$	T.nptr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

4.3.4 Directed Acyclic Graphs for Expressions

A directed acyclic graph (DAG) for an expression identifies the common subexpressions in the expression. Like a syntax tree, a DAG has a node for every subexpression of the expression; an interior node represents an operator and its children represent its operands. The difference is that a node in a DAG representing a common subexpression has more than one "parent"; in a syntax tree, the common subexpression would be represented as a duplicated subtree.

The DAG for the expression

a + a * (b – c) + (b – c) * d



4.4 Bottom-up Evaluation of S-Attributed Definitions

Synthesized attributes can be evaluated by a bottom-up parser as the input is being parsed. The parser can keep the values of the synthesized attributes associated with the grammar symbols on its stack. Whenever a reduction is made, the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production.

Following figure shows an example of a parser stack with space for one attribute value.

	state	val
	...	...
	X	X.x
	Y	Y.y
top →	Z	Z.z
	...	...

The current top of the stack is indicated by the pointer top. We assume that synthesized attributes are evaluated just before each reduction. Suppose the semantic rule A.a := $f(X.x, Y.y, Z.z)$ is associated with the production A → XYZ. Before XYZ is reduced to A, the value of the attribute Z.z is in val[top], that of Y.y in val[top – 1], and that of X.x in val[top – 2]. If a symbol has no attribute, then the corresponding entry in the val array is undefined. After the reduction, top is decremented by 2, the state covering A is put in state [top] (i.e., where X was), and the value of the synthesized attribute A.a is put in val[top].

Example: Consider the syntax-directed definition of the desk calculator in the following figure

Production	Code Fragment
L → En	print(val[top])
E → E_1 + T	val[ntop] := val[top – 2] + val[top]
E → T	
T → T_1 * F	val[ntop] := val[top – 2] × val[top]
T → F	
F → (E)	val[ntop] := val[top – 1]
F → digit	F.val := digit.lexval

The following figure shows the sequence of moves made by the parser on input 3 * 5 + 4n. The contents of the state and val fields of the parsing stack are shown after each move. Consider the sequence of events on seeing the input symbol 3. In the first move, the parser shifts the state corresponding to the token digit (whose attribute value is 3) onto the stack. On the second move, the parser reduces by the production F → digit and implements the semantic rule F.val := digit.lexval. On the third move the parser reduces by T → F. No code fragment is associated with this production, so the val array is left unchanged. Note that after each reduction the top of the val stack contains the attribute value associated with the left side of the reducing production.

Input	State	Val	Production Used
3*5+4n	–	–	
*5+4n	3	3	
*5+4n	F	3	F → digit
*5+4n	T	3	T → F
5+4n	T*	3 –	
+4n	T*5	3 – 5	
+4n	T*F	3 – 5	F → digit
+4n	T	15	T → T*F
+4n	E	15	E → T
4n	E+	15 –	
n	E+4	15 – 4	
n	E+F	15 – 4	F → digit
n	E+T	15 – 4	T → F
n	E	19	E → E+T
	En	19 –	
	L	19	L → En

4.5 L-Attributed Definitions

When translation takes place during parsing, the order of evaluation of attributes is linked to the order in which nodes of a parse tree are "created" by the parsing method. A natural order that characterizes many top-down and bottom-up translation method is depth-first order. A class of syntax-directed definitions, called L-attributed definitions, whose attributes can always be evaluated in depth-first order.

A syntax-directed definition is L-attributed if each inherited attribute of X_j, $1 \le j \le n$, on the right side of $A \rightarrow X_1 X_2 \ldots X_n$, depends only on

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

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

Translation Schemes

A translation scheme is a context-free grammar in which attributes are associated with the grammar symbols and semantic actions enclosed between braces { } are inserted within the right sides of productions.

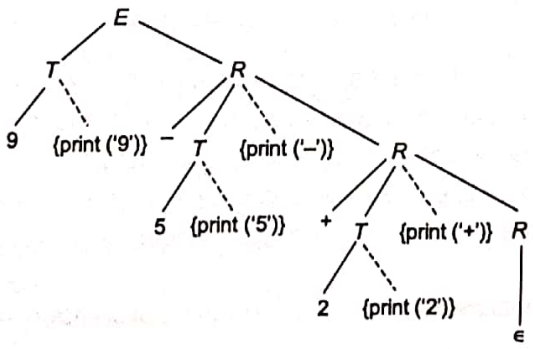
Example: Here is a simple translation scheme that maps infix expressions with addition and subtraction into corresponding postfix expressions.

$E \rightarrow T\ R$

$R \rightarrow$ addop T {print(addop.lexeme)} $R_1 \mid \epsilon$

$T \rightarrow$ num {print(num.val)}

Following figure shows the parse tree for the input 9 – 5 + 2 with each semantic action attached as the appropriate child of the node corresponding to the left side of their production. When performed in depth-first order, the actions print the output 95 – 2 +.



If we have both inherited and synthesized attributes, we must be more careful:

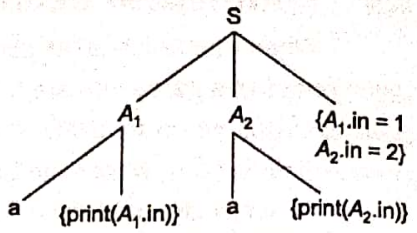
1. An inherited attribute for a symbol on the right side of a production must be computed in an action before that symbol.
2. An action must not refer to a synthesized attribute of a symbol to the right of the action.
3. A synthesized attribute for the nonterminal on the left can only be computed after all attributes it references have been computed. The action computing such attributes can usually be placed at the end of the right side of the production.

Example:

$S \rightarrow A_1\ A_2$ $\quad \{A_1.\text{in} := 1;\ A_2.\text{in} := 2\}$

$A \rightarrow a$ $\quad$ {print(A.in)}

We find that the inherited attribute A.in in the second production is not defined when an attempt is made to print its value during a depth first traversal of the parse tree for the input string 'aa'. That is, a depth first traversal starts at S and visits the subtrees for A_1 and A_2 before the values of A_1.in and A_2.in are set. If the action defining the values of A_1. in and A_2.in is embedded before the A's on the right side of $S \rightarrow A_1A_2$, instead of after, then A.in will be defined each time print(A.in) occurs.



4.6 Bottom-up Evaluation of Inherited Attributes

In the bottom-up translation method we relied upon all translation actions being at the right end of the production. To understand how inherited attributes can be handled bottom up, we introduce a transformation that makes all embedded actions in a translation scheme occur at the right ends of their productions.

The transformation insert new marker nonterminals generating ϵ into the base grammar. We replace each embedded action by a distinct marker nonterminal M and attach the action to the end of the production $M \rightarrow \epsilon$. For example, the translation scheme

$E \rightarrow T\ R$

$R \rightarrow +\ T$ {print('+')} $R \mid -\ T$ {print('–')} $R \mid \epsilon$

$T \rightarrow$ num {print(num.val)}

is transformed using marker nonterminals M and N into

$E \rightarrow T\ R$

$R \rightarrow +\ T\ M\ R \mid -\ T\ N\ R \mid \epsilon$

$T \rightarrow$ num {print(num.val)}

$M \rightarrow \epsilon$ {print('+')}

$N \rightarrow \epsilon$ {print ('–')}

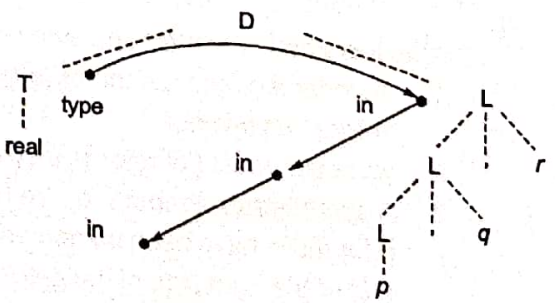
A bottom-up parser reduces the right side of production $A \rightarrow XY$ by removing X and Y from the top of the parser stack and replacing them by A. Suppose X has a synthesized attribute X.*s*, which is kept along with X on the parser stack. Since the value of X.*s* is already on the parser stack before any reductions take place in the subtree below Y, this value can be inherited by Y. That is, if inherited attribute Y.*i* is defined by the copy rule Y.*i* := X.*s*, then the value X.*s* can be used where Y.*i* is called for.

Example: real *p*, *q*, *r*

The moves made by a bottom-up parser on the given input is shown in following figure.

The translation scheme we wish to implement is

D →	T L	{L.in := T.type}
T →	int	{T.type := integer}
T →	real	{T.type := real}
L →		{L_1.in := L.in}
	L_1, id	{addtype (id.entry, L.in)}
L →	id	{addtype (id.entry, L.in)}



If we ignore the actions in the above translation scheme, the sequence of moves made by the parser on the input is as in the following figure.

Input	State	Production Used
real p,q,r	–	
p,q,r	real	
p,q,r	T	T → real
,q,r	T p	
,q,r	T L	L → Id
q,r	T L,	
,r	T L, q	
,r	T L	L → L, Id
r	T L,	
	T L, r	
	T L	L → L, Id
	D	D → T L

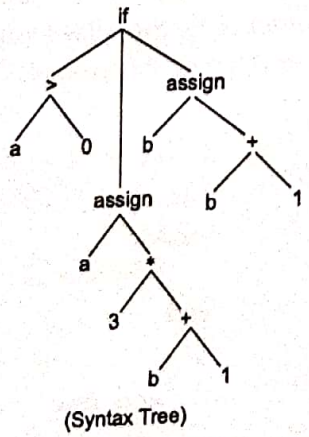
Let the parser stack is implemented as a pair of arrays state and val. If state[i] is for grammar symbol X, then val[i] holds a synthesized attribute X.s. The contents of the state array are shown in above figure. Note that every time the right side of a production for L is reduced, T is in the stack just below the right side. We can use this fact to access the attribute value T.type.

4.7 Intermediate Code Generation

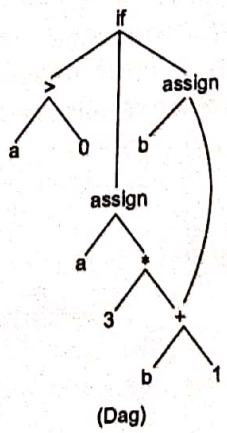
Intermediate code is closer to the target machine than the source language, and hence easier to generate code from. Unlike machine language, intermediate code is more or less machine independent. This makes it easier to retarget the compiler. It allows a lots of optimizations to be performed in a machine independent way. Generally, intermediate code generation can be implemented via syntax-directed translation. There are two types of intermediate languages. (1) High-level and (2) Low-level. High level intermediate language represent high level structure of a program. They are closer to source language (e.g., syntax tree), easy to be generated from input program; and it may not be straight forward. Low-level intermediate language represent low level structure of a program. They are closer to the target machine (e.g., Three-address code); easy to generate code from it; and its generation from input program may involve some work.

There are mainly two types of high-level intermediate language: (1) Syntax tree; and (2) DAGs. (Abstract) syntax tree is a compact form of a parse tree that represents the hierarchical structure of the program: nodes represent operators and the children of a node represent what it operates on. DAG is similar to syntax trees, except that common sub-expressions are represented by a single node.

if ($a > 0$) then $a \leftarrow 3 * (b + 1)$ else $b \leftarrow b + 1$;



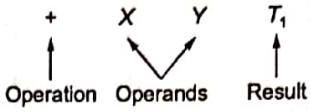
(Syntax Tree)



(Dag)

Low-level intermediate language is mainly represented by Three address code. TAC is of three types: (1) Quadruple; (2) Triple; and (3) Indirect triple.

Quadruple consists of an operation, (upto) two operands, and a result. For example $X + Y$ would be translated int quadruple as:



$M = X + Y * Z$ would be translated into quadruple as:

*	Y	Z	T_1
+	X	T_1	T_2
=	T_2	–	M

An alternative notation for Quadruple is to write them as a sequence of assignment statements. Thus, $M = X + Y * Z$ would be written as:

$$T_1 = Y * Z$$
$$T_2 = X + T_1$$
$$M = T_2$$

If we have GOTO L as an operator and a result, the Quadruple would be | GOTO | – | – | L |

If we have: $X[i] = B$, the quadruple would be

[ ]	X	i	T_1
=	T_1	–	B

If we have: If $X < Y$ then Maximum = Y, the quadruple would be

<	X	Y	Label 1
GOTO			Label 2
			Label 1
=	Y		MAXIMUM
			LABEL 2

Triple do not use an extra temporary variable.

For example, $X + Y * Z$ would be translated into triple as: (1) * YZ and (2) + X (1).

In triple, the execution order is same as the order of the triple itself. But in indirect triple execution order may bot be same. With indirect triples, optimization can change the execution order, rather than the triple itself. Consider the following code and its optimized code.

```
for i = 1 to 10
begin
      x = y * z
      m = i * 3
end
      ⋮
end for
```

⇔ (optimized)

```
x = y * z
for i = 1 to 10
begin
   m = i * 3
end
   ⋮
   end for
```

The triple for the original unoptimized code might be:

(1)	=	1	i
(2)	*	y	z
(3)	=	(2)	x
(4)	*	3	i
(5)	=	(4)	m
(6)	+	1	i
(7)	LE	I	10
(8)	IFT	go	(2)

Execution order = 1, 2, 3, 4, 5, 6, 7, 8
LE = Less than or Equal to
IFT = If True

In optimized code $x = y * z$ moved out of the loop, thus the execution order becomes 2, 3, 1, 4, 5, 6, 7, 8.

4.8 Dependency Graph Generation using Semantic Rules (SDT)

Consider the following assignment statement: $a = 10 * (a + b)$

Let available temporary memory locations are T_0, T_1, T_2,... let us convert the given assignment statement into intermediate code.

$$T_0 = 10$$
$$T_1 = a$$
$$T_2 = b$$
$$T_1 = T_1 + T_2$$
$$T_0 = T_0 * T_1$$
$$a = T_0$$

Let us assume that the attribute grammar uses the following attributes.

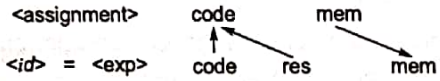
1. mem: to denote the first unused temporary memory location available.
2. res: to denote the result of an expression, term, etc.
3. code: to denote the instructions.

The semantic rules uses the function NEXT, which is applied to temporary memory location to get the result NEXT (T_0) = T_1. Let us write semantic rules as under.

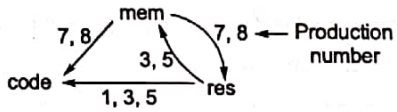
Backus-Naur form	Semantic Rules
1. <assignment> → <id> = <exp>	assignment·code = exp·code id = exp·res assignment·mem = T_0 exp·mem = assignment·mem
2. <exp> → <term>	exp·code = term·code exp·res = term·res term·mem = exp·mem
3. $<exp>_0 \rightarrow <exp>_1 + <term>$	exp_0.code = exp_1.code ll term·code exp_1·res = exp_1·res + term·res exp_0·res = exp_1·res exp_1·mem = exp_0·mem term·mem = next (exp_1·res)

4. <term> → <factor>	term·code = factor·code term·res = factor·res factor·mem = term·mem
5. $<term>_0 \rightarrow <term>_1 * <factor>$	$term_0$·code = $term_1$·code ll factor·code $term_1$·res = $term_1$·res*factor·res $term_0$·res = $term_1$·res $term_1$·mem = $term_0$·mem factor·mem = next ($term_1$·res)
6. <factor> → <(exp)>	factor·code = exp·code factor·res = exp·res exp·mem = factor·mem
7. <factor> → <id>	factor·code = factor·mem = id factor·res = factor·mem
8. <factor> → <const>	factor·code = factor·mem = const factor·res = factor·mem

The attribute depending in this example is difficult, so we explain it in terms of the associated derivation tree nodes. Thus, for production 1, there are the following dependencies where the arrow shows the direction of information flow.



This indicate that code is a synthesized attribute and is dependent on res, and mem is an inherited attribute. We do the above process for every production and then finally find the following dependency graph:



4.9 Syntax-Directed Translation for Intermediate Code Generation

Syntax-directed translation is intermediate code represented as a list of instructions. Instruction sequences are concatenated using the operator ll. Attributes for expression E is E·place : denotes the location that holds the value of E.

E·code : denotes the instruction sequence that evaluates E.

Attribute for statement S is:

S·begin:denotes the first instruction in the code for S.

S·after:denotes the first instruction after the code for S.

S·code:denotes the instruction sequence that represents S.

Functions used in syntax directed translation are:

newtemp( ): returns a new temporary variable each time it is called.

newlabel( ): returns a new label name each time it is called.

Notation used in syntax directed translation is:

gen (x '=' y '+' z) to represent the instruction $x = y + z$

Example - 4.4 Consider the following grammar:

$$E \rightarrow \text{id} \mid (E_1) \mid E_1 + E_2 \mid -E_1$$

Write its semantic rules using SDT.

Solution:

Following is the semantic rules.

Production	Semantic Rule
$E \rightarrow \text{id}$	E·place = id.place
	E·code = ''
$E \rightarrow (E_1)$	E·place = E_1·place
	E·code = E_1·code
$E \rightarrow E_1 + E_2$	E·place = newtemp( )
	E·code = E_1·code ||
	E_2·code ||
	gen(E·place '=' E_1·place '+' E_2·place)
$E \rightarrow -E_1$	E·place = newtemp()
	E·code = E_1·code ||
	gen(E·place '=' '–' E_1·place)

4.9.1 Accessing Array Elements

Array elements can be accessed if the elements are stored in the block of consecutive locations. Let us assume that we want the i^{th} element of an array A whose subscript ranges from low to high. The address of the first element of the array is base. We can avoid address calculation in the intermediate code if we have indexed "addressing modes" at the intermediate code level. In this case, $A[i]$ is the $(i + \text{low})^{th}$ element of the array located at base (starting at element 0). So, a reference $A[i]$ translates to the code

$$t_1 = i + \text{low}$$
$$t_2 = A[t_1]$$

4.9.2 Code Generation for Logical Expressions

The simple format of logical expression is

$$\text{Exp} \rightarrow E_1 \text{ relational-op } E_2$$

Its equivalent code is

t_3 = TRUE
if t_1 relop t_2 goto L
label L

4.9.3 Code Generation for Conditions

The simple format for conditional statement is

$$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$$

we can write its semantic rules using SDT as under.

{

S·begin = newlabel( )
S·after = newlabel( )
S·code = gen ('label' S·begin) ||
E·code ||
gen('if' E·place '=' 0 goto S_2·begin) ||

S_1.code ||
gen('goto' S.after)
S_2.code ||
gen('label' S.after)
}

4.9.4 Code Generation for Loops

The simple format for loop is

$S \rightarrow$ while E do S_1

We can write its semantic rules using SDT as under.

{
S.begin = newlabel( )
S.after = newlabel( )
S.code = gen('label' S.begin) ||
E.code ||
gen('if' E.place '=' 0 'goto' S.after) ||
S_1.code ||
gen ('goto' S.begin) ||
gen('label' S.after)
}

4.9.5 Code Generation for Assignment

The simple format of assignment is

$S \rightarrow$ LHS = RHS

We can write its semantic rules using SDT as under.

{
S.code = LHS.code ||
RHS.code ||
gen (LHS.place = RHS.place)
}

4.10 Intermediate Code Representation

Intermediate code or intermediate text is a language in compilers which is intermediate in complexity between a high-level programming language and machine code. Intermediate code is an intermediate notation so as to generate an optimal or even relatively good code as direct translation from source to machine or assembly language is comparatively a difficult task.

Four types of intermediate code generally used in compilers are:

1. Three-address code (determine of low-level representation):
   Three-address code representations:
   - quadruple
   - triple
   - indirect triple
2. Syntax trees and DAGs (determine high-level intermediate representation)
3. Postfix notation
4. CFG (Control Flow Graph)

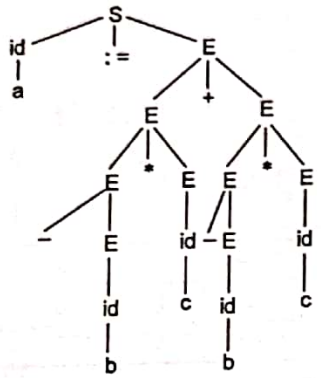
4.10.1 Three-Address Code

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

In a three-address code, there is atmost one operator on the right side of instruction, that is, no build-up arithmetic expressions are permitted.

For instance, consider a source language expression

$a := -b * c + (-b * c)$, the parse tree for this statement can be given as:



where the productions of the grammar for infix algebraic expression are:

$S \rightarrow \text{id} := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow -E_1$

$E \rightarrow (E_1)$

$E \rightarrow \text{id}$

Then three-address code corresponding to interior nodes of syntax tree is

S.No.	Three-address code	Explanation
0.	t_1 = addr a	// t_1 stores address of '*a*'
1.	$t_2 = * t_1$	// t_2 is a pointer t_1 i.e. t_2 stores value of *b*
2.	t_3 = addr *b*	// t_3 stores address of '*b*'
3.	$t_4 = * t_3$	// t_4 points to value at address t_3
4.	t_5 = uminus t_4	// t_5 uminus is a unany operator for multiplication
5.	t_6 = adder *c*	// t_6 stores address of operand *c*
6.	$t_7 = * t_6$	// t_7 points to t_6
7.	$t_8 = t_5$ times t_7	// t_8 stores the result of ($t_5 * t_7$)
8.	t_9 = addr *b*	// t_9 stores address of operand *b*
9.	$t_{10} = * t_9$	// t_{10} points to value at address t_9
10.	t_{11} = uminus t_{10}	// t_{11} uminus is a unany operator
11.	t_{12} = addr *c*	// t_{12} contains address of operand '*c*'
12.	$t_{13} = * t_{12}$	// t_{13} contains value of '*c*'
13.	$t_{14} = t_{11}$ times t_{13}	// t_{14} stores the result of ($t_{11} * t_{13}$)
14.	$t_{15} = t_8$ add t_{14}	// t_{15} stores the result of ($t_8 + t_{14}$)
15.	t_{16} = addr *a*	// t_{16} stores the address of '*a*'
16.	$*t_{16} = t_{15}$	// value of t_{15} is passed to t_{16}, hence assigned to '*a*'

1. **Quads:** The three-address instructions can be implemented as objects or as records with fields for the operator and the operand with quadruple representation. It has the following format.

op	arg1	arg2	result

Where the four fields have their meanings as:
op: contains the internal code of operation
arg1: contains first operand at which operation is performed
arg2: contains second operand at which operation is performed
result: contains the result of operation 'arg1 op arg2'

op	arg1	arg2	result
Uminus	b		t_1
Times	t_1	c	t_2
Uminus	b		t_3
Times	t_3	c	t_4
Add	t_2	t_4	t_5
Assign	t_5		a

2. **Triple:** A representation is of the form:

Position Number	op	arg1	arg2

where the field have their meanings as:
position number : represents pointer into triple structure itself
op : intermediate code for operator
arg1 : first operand
arg2 : argument for second operand

Position Number	op	arg1	arg2
(0)	Uminus	b	
(1)	Times	(0)	c
(2)	Uminus	b	
(3)	Times	(2)	c
(4)	Add	(1)	(3)
(5)	Assign	a	(4)

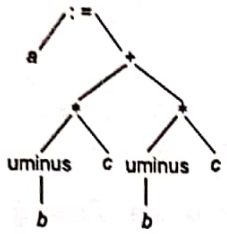
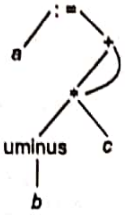
3. **Indirect triple:** It considers in listing pointers to triple itself. A separate array of pointers is maintained that point to instructions.

Pointer	Pointer to position #
(0)	(20)
(1)	(21)
(2)	(22)
(3)	(23)
(4)	(24)

Position #	op	arg1	arg2
(20)	Uminus	b	
(21)	Times	(20)	c
(22)	Uminus	b	
(23)	Times	(22)	c
(24)	Add	(21)	(23)

4.10.2 Syntax Trees and Dags

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

Syntax Tree	DAG
It is a compact form of a parse tree that represents tree like structure of program as shown below: 	It is similar to syntax tree, except that common sub-expressions are represented by a single node to get an optimized code. 

4.10.3 Postfix Notation

For the expression 'E_1 op E_2', where E_1 and E_2 are expressions and 'op' is a binary operator to be applied on E_1 and E_2, the postfix notation is given as:

$$E_1 \text{ op } E_2 \xrightarrow{\text{Postfix transformation}} E_1\ E_2 \text{ op}$$

A postfix notation represents operations on stack.

So, the postfix expression for the expression.

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

can be given as:

$-b$ is written as 'b uminus'

$(-b) * c$ is written as 'b uminus c *'

$-b * c + -b * c$ is written as 'b uminus $c * b$ uminus * +'

$a := -b * c + -b * c$ has the postfix notation as 'ab uminus $c * b$ uminus c * + assign'

Example-4.5 Translate the expression $-(a + b) * (c + d) + (a + b + c)$ into quadruples and triples.

Solution:

Quadruple translation of $-(a + b) * (c + d) + (a + b + c)$ is given as:

op	arg1	arg2	result
Add	a	b	T_1
Uminus	T_1	–	T_2
Add	c	d	T_3
Add	a	b	T_4
Add	T_4	c	T_5
Times	T_2	T_3	T_6
Add	T_6	T_5	T_7

Here, T_1, T_2, T_3, T_4, T_5, T_6 and T_7 are the temporary compiler generated names for storage-address of the calculated value and T_7 has the final value that is to be calculated in the original expression given.

Triple representation of the expression $-(a + b) * (c + d) + (a + b + c)$ is given as under:

Position Number	op	arg1	arg2
(0)	Add	*a*	*b*
(1)	Uminus	(0)	–
(2)	Add	*c*	*d*
(3)	Times	(1)	(2)
(4)	Add	*a*	*b*
(5)	Add	(4)	*c*
(6)	Add	(3)	(5)

Here {(0), (1)...(6)} are the pointers to the triple structure itself and pointer to position number (6), has the final calculated value of expression to be evaluated.

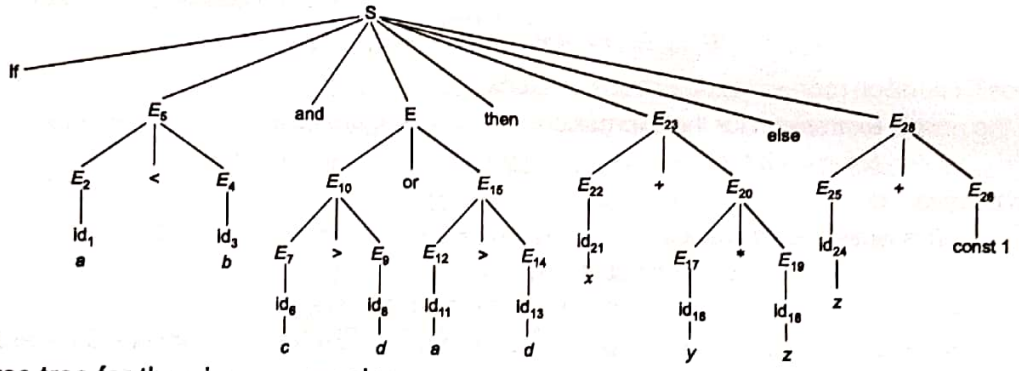
Example - 4.6 Construct three-address code for the following:

if $[(a < b)$ and $((c > d)$ or $(a > d))]$ then

$z = x + y * z$

else, $z = z + 1$

Solution:



Parse tree for the given expression:

L_1: t_1 = add a
$t_2 = * t_1$
t_3 = add b
$t_4 = * t_3$
$t_5 = t_2 < t_4$
If $t_5 == 0$ goto L_2
t_6 = add c
$t_7 = * t_6$
t_8 = add d
$t_9 = * t_8$
$t_{10} = t_7 > t_9$
t_{11} = add a
$t_{12} = * t_{11}$
t_{13} = add d
$t_{14} = * t_{13}$
$t_{15} = t_{11} > t_{14}$

If t_{10} and $t_{15} == 0$ goto L_2
t_{16} = add y
$t_{17} = * t_{16}$
t_{18} = add z
$t_{19} = * t_{17}$
$t_{20} = t_{17} * t_{19}$
t_{21} = add x
$t_{22} = * t_{21}$
$t_{23} = t_{22} + t_{20}$
L_2: t_{24} = add z
$t_{25} = * t_{24}$
t_{26} = const 1
$t_{28} = t_{25} + t_{26}$

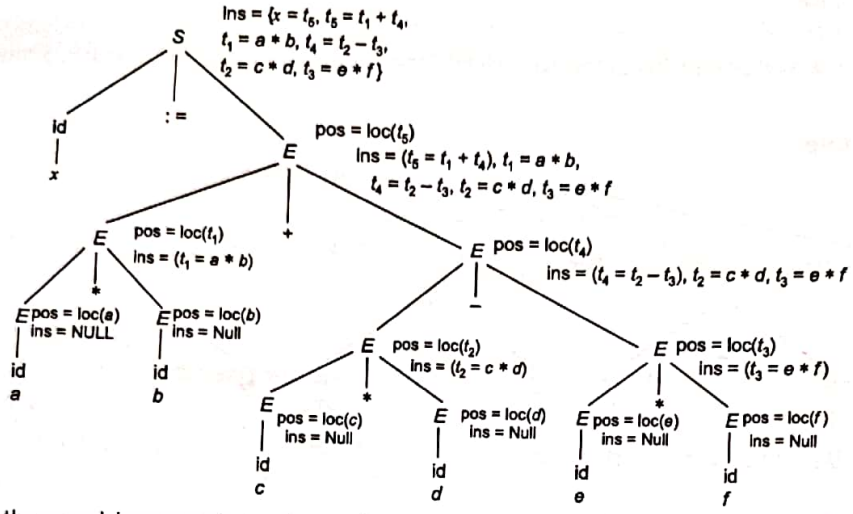
Example - 4.7 For the assignment $x = a * b + c * d - e * f$ perform the following translations:

(a) using the SDT scheme without any effort to minimize the number of register.

(b) using the SDT scheme but now using the slightly modified newtemp functions that attempts to reuse temporary variable as much as possible.

Solution:

To construct the three-address code of the assignment expression given, we first need to build a parse tree as follows:



Therefore, three-address code for the assignment statement is

$$x = t_5 \; ; \; t_5 = t_1 + t_4 \; ; \; t_1 = a * b \; ; \; t_4 = t_2 - t_3 \; ; \; t_2 = c * d \; ; \; t_3 = e * f$$

Now, if we try to minimize the number of registers used then, 3-address code can be given as:

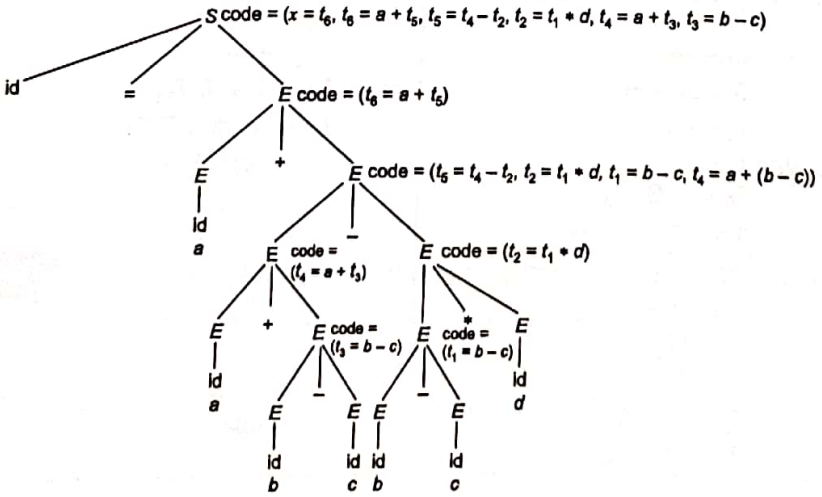
$$t_1 = c * d \; ; \; t_2 = e * f \; ; \; t_1 = t_1 - t_2 \; ; \; t_2 = a * b \; ; \; t_1 = t_2 + t_1 \; ; \; x = t_1$$

Now, this reduced three-address code uses only two registers.

Example - 4.8 For the assignment statement: $x = a + (a + (b - c)) - ((b - c) * d)$.

Solution:

(a) Generate three-address instruction using the SDT scheme and without any minimization of temporary variables. Parse tree for the given expression is



Therefore, three address-code for it can be given as:

$t_1 = b - c\ ;\ t_2 = t_1 * d\ ;\ t_3 = b - c\ ;\ t_4 = a + t_3\ ;\ t_5 = t_4 - t_2\ ;\ t_6 = a + t_5\ ;\ x = t_6$

(b) Redo the code generation above now reusing temporaries.

$t_1 = b - c\ ;\ t_2 = t_1 * d\ ;\ t_1 = b - c\ ;\ t_1 = a + t_1\ ;\ t_1 = t_1 - t_2\ ;\ t_1 = a + t_1\ ;\ x = t_1$

After minimizing the number of temporary registers we found that minimum number of register that can be used is two.

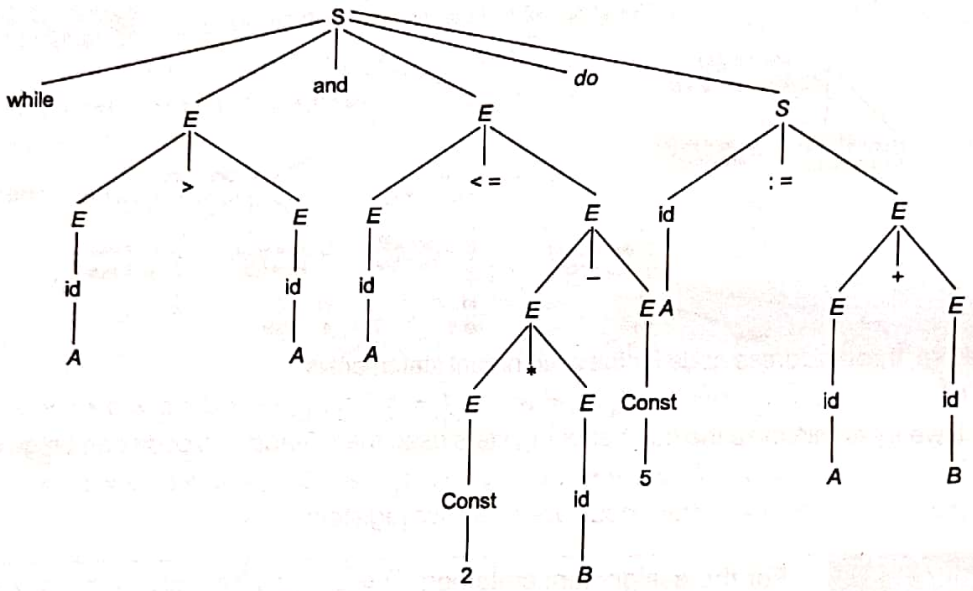
Example-4.9 Consider the following while statement:

While $A > B$ and $A < = 2 * B - 5$ do $A := A + B$

For the above statement: **(*i*)** Construct parse tree and **(*ii*)** Generate intermediate code.

Solution:

(*i*) Parse tree:



Here symbols used in parse tree have their meanings as: S : Statement, E : Expression, id : Identifier, const : Constant, while : a programming construct for while loop and and : Binary AND Operator.

(*ii*) Intermediate or three-address code

T_1 = addr A $T_2 = *T_1$ T_3 = addr B $T_4 = *T_3$ Lable 1: $T_5 = T_2$ GT T_4 if Not T_5 goto END T_6 = const 2 T_{21} = addr B $T_7 = *T_{21}$ $T_8 = T_6 * T_7$ T_9 = const 5 $T_{11} = T_8 - T_9$	T_{12} = addr A $T_{13} = *T_{12}$ $T_{14} = T_{13}$ LE T_{11} if not T_{14} goto END T_{15} = addr A $T_{16} = *T_{15}$ T_{17} = addr B $T_{18} = *T_{17}$ $T_{19} = T_{16} + T_{18}$ T_{20} = addr A $*T_{20} = T_{19}$ goto Lable 1

Example-4.10 Construct syntax tree and give postfix notation for the following expression:

$$(a * b + (b * c)) \uparrow d - e / (f + g)$$

Solution:

Let us build the syntax tree for the given expression and find out the postfix (here mentioned as 'pfix') at each node of the syntax tree.

Now, the postfix notation for the given expression as calculated at the top of tree called root in bottom-up fashion can be given as: $ab * bc * + d \uparrow efg + /-$

Example-4.11 To write the postfix and three address code of $a * - (b + c)$.

Solution:

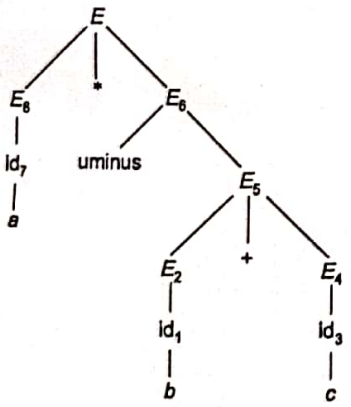
Postfix notation:

$$\begin{aligned} a* - (b + c) &= (a * (-(b + c))) \\ &= (a * (- (bc +))) \\ &= (a * (-bc +)) \quad \text{(since here '–' is unany operator)} \\ &= a - bc + * \end{aligned}$$

Three address code:

To construct the three address code of the given expression, let us first the compute the parse tree for expression: $a* - (b + c)$.

According to the numbering done at parse tree, the three address code for expression $a* - (b + c)$ can be given as:



S.No.	Three address code	Explanation
1.	t_1 = addr a	// passes address of 'b' into t_1
2.	$t_2 = *t_1$	// passes value of t_1 or b into t_2
3.	t_3 = addr c	// passes address of 'c' into t_3
4.	$t_4 = *t_3$	// passes pointer to address of t_3 into t_4
5.	$t_5 = t_2 + t_4$	// adds 'b' and 'c'
6.	t_6 = uminus t_5	// performs $-(b + c)$
7.	t_7 = addr b	// passes address of 'b' into t_7
8.	$t_8 = *t_7$	// passes pointer to address of t_7 into t_8
9.	$t_9 = t_6 * t_8$	// multiplies a and $-(b + c)$

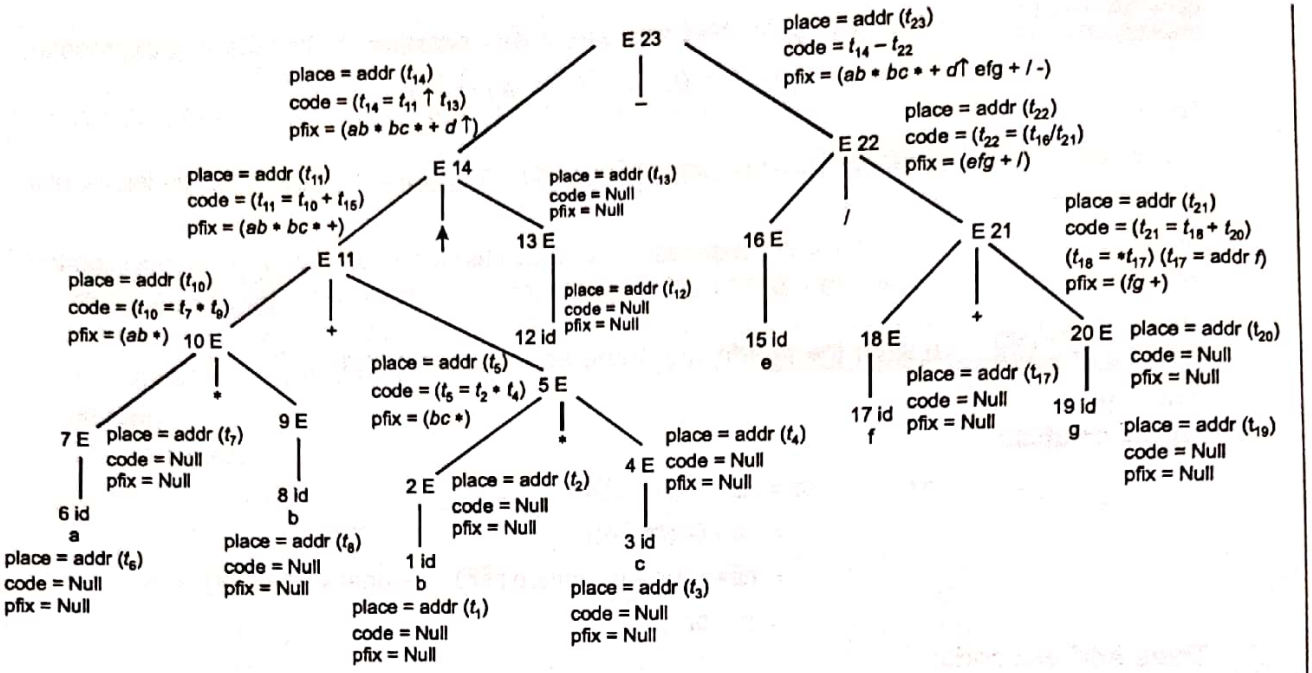
where, $t_1, t_2 \ldots, t_9$ are temporary registers for storage and 'addr' refers to address of operand, 'uminus' is a unany minus (negative) operator.

Example-4.12 Construct syntax tree and give postfix notation for the following expression:

$$(a + b(b * c)) \uparrow d - e / (f + g)$$

Solution:

Let us build the syntax tree for the given expression and find out the postfix (here mentioned as 'pfix') at each node of the syntax tree:



Example-4.13 Translate the expression:

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

(c) $-(x + y) * (z + c) - (x + y + z)$

into: (*i*) Quadruples (*ii*) Triples and (*iii*) Indirect triples.

Solution:

Let us consider the equation: $-(a + b) * (c + d) + (a + b + c)$.

Then, its quadruple representation can be given as:

Pointer to quadruple	Operator	Argument 1	Argument 2	Result
(0)	ADD	*b*	*c*	t_1
(1)	ADD	*a*	t_1	t_2
(2)	ADD	*c*	*d*	t_3
(3)	ADD	*a*	*b*	t_4
(4)	UMINUS	t_4	–	t_5
(5)	TIMES	t_5	t_3	t_6
(6)	ADD	t_6	t_2	t_7

Triple representation is as follows:

Pointer to triple	Operator	Argument 1	Argument 2
(0)	ADD	*a*	*b*
(1)	UMINUS	(0)	–
(2)	ADD	*c*	*d*
(3)	TIMES	(1)	(2)
(4)	ADD	*b*	*c*
(5)	ADD	*a*	(4)
(6)	ADD	(3)	(5)

Indirect triple representation is as follows:

In this representation, if serial number referred to as pointer to triple itself is sequenced other than '(0)', then a reference to it is made in another table as shown:

Pointer to triple	Operator	Argument 1	Argument 2
(16)	ADD	*a*	*b*
(17)	ADD	(16)	c
(18)	ADD	*c*	*d*
(19)	ADD	*a*	*b*
(20)	UMINUS	(19)	–
(21)	TIMES	(20)	(18)
(22)	ADD	(21)	(17)

Sequence no.	Pointers to triple
(0)	(16)
(1)	(17)
(2)	(18)
(3)	(19)
(4)	(20)
(5)	(21)
(6)	(22)

where operator symbol meaning is:

ADD = adds two operators
UMINUS = unary operator for negative multiplication
TIMES = binary operator for multiplication

and t_1, t_2, t_3....are registers for allocation of values calculated. Sequence numbers are pointers to indirect triple itself.

Summary

- Association of semantic rule with the grammar or semantic rules placed on right hand side of production is called syntax directed translation.
- A parse tree which show attribute at each and every node is called as Annotated / Decorated parse tree.
- In synthesized attribute value is evaluated in terms of parents or siblings.
- S-attributed SDD use only synthesized attributes, semantic rules present right most side of production and used bottom up approach.
- L-attributed SDD used both synthesized or inherited attributes, semantic rules present any place of right side of production and used top down approach.
- Every S-attributed SDT is L-attributed SDT but reverse is not true.
- Every SDT follow depth first search notation.
- Some application of SDT are Prefix to Infix conversion, intermediate code conversion, Decimal to Binary conversion etc.
- Generation of machine independent code is called intermediate code generation.
- Three address code (Quadruple, triple and indirect triple), DAG, Postfix notation are types of intermediate code.
- DAG is used to eliminate common sub-expression.

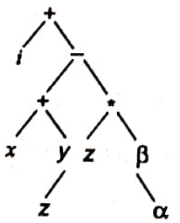
Student's Assignments

Q.1 In a syntax directed translation scheme, if the value of an attribute of a node is a function of the values of the attributes of its children, then it is called a
(a) Synthesized attribute
(b) Inherited attribute
(c) Canonical attribute
(d) None of the above

Q.2 A synthesized attribute is an attribute whose value at a parse tree node depends on
(a) Attributes at the siblings only
(b) Attributes at present node only
(c) Attributes at children node only
(d) None of the above

Q.3 A pictorial representation of the value computed by each statement in the basic block is
(a) Tree (b) DAG
(c) Graph (d) All of these

Q.4 The intermediate code generated for the following syntax tree is



(a) $ixz + z\alpha y + - +$ (b) $ixzy + z\alpha\beta * - +$
(c) $ixzy + z\alpha * - +$ (d) None of these

Q.5 In a bottom-up evaluation of a syntax-directed definition, inherited attributes can
(a) Always be evaluated
(b) Be evaluated only if the definition is L-attributed
(c) Be evaluated only if the definition has synthesized attributes
(d) Never evaluated

Q.6 A shift reduce parser carries out the action specified within braces immediately after reducing with the corresponding rule of grammer.

$S \rightarrow xxW$(print "1")
$S \rightarrow y$(print "2")
$W \rightarrow Sz$(print "3")

What is translation of $xxxxyzz$ using the syntax directed translation scheme described by the above rules?
(a) 23131 (b) 11233
(c) 11231 (d) 33211

Q.7 Consider the following transition rules:

$A \rightarrow BC$
$C \rightarrow + BC \mid A \mid \epsilon$
$B \rightarrow DB$ (print '+') $\mid \epsilon$
$D \rightarrow (A) \mid$ Id{print num.value}

On input '5 + 67' this translation scheme prints
(a) 5 + 67 + (b) 5 + 6 + 7 +
(c) 5 + 6 + 7 (d) 5 + 67 ++

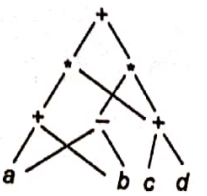
Common Data Questions (8 and 9):

Production Rule	Translation Rule
$E \rightarrow E \# T$ {E·val = E_1·val – T·val}	I
| T {E·val = T·val}	II
$T \rightarrow T \& F$ {_____}	III
| F {T·val = F·val+2}	IV
$F \rightarrow F\%G$ {F.val = F.val +G.val}	V
| G {F.val = G.val –1}	VI
$G \rightarrow$ num {G·val = num.val}	VII

Q.8 If the expression 4 & 2% 3 # 8 is 11 then which one of the following correctly represent the rule III?
(a) T · val = T_1 · val × F · val
(b) T · val = T_1 · val + F · val
(c) T · val = T_1 · val – F · val
(d) None of the above

Q.9 The expression 10 & 4 % 8 # 2 will be evaluated
(a) 117 (b) 118
(c) 119 (d) 121

Q.10 The DAG shown here represents:

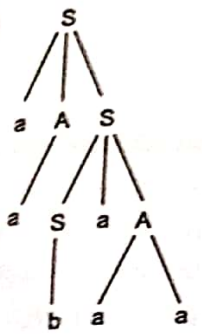


(a) $(a-b)*(a+b)+(c+d)*(c+d)$
(b) $(a-b)*(c+d)+(a-b)*(c-d)$
(c) $(a*b)+(c*d)*(a*b)+(c*d)$
(d) $(a+b)*(c+d)+(a-b)*(c+d)$

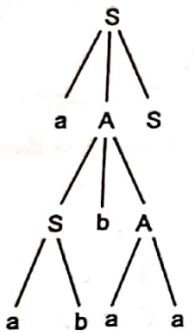
Q.11 For the string aabbaa give the derivation tree using grammer

$G = \{\{S, A\}, \{a, b\}, \{S \to aAS, S \to a, A \to SbA, A \to SS, A \to ba\}, S\}$

(a)

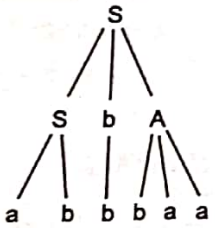


(b)

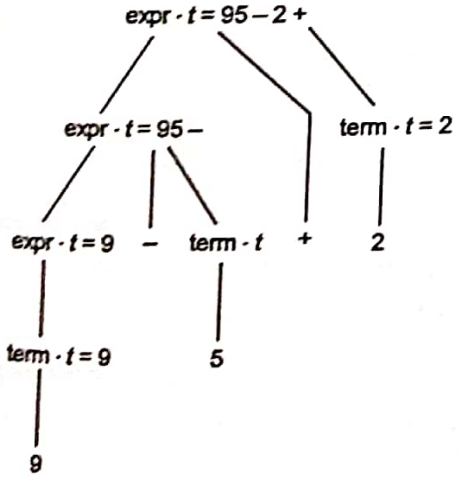


(c)

(d)



Q.12 Consider the annotated parse tree:



Which of the following is correct syntax directed definition of above parse tree?

	Production	Semantic Rule
(a)	expr → expr + term	expr · t = expr · t ‖ term· t ‖ '*'
	expr → expr + term	expr · t = expr · t ‖ term · t‖ '+'
	expr → term	expr · t = term · t
	term → 9	term · t → '9'
	term → 5	term · t → '5'
	term → 2	term · t → '2'
(b)	expr → expr + term	expr · t = expr · t ‖ term · t ‖ '+'
	expr → expr – term	expr · t = expr · t ‖ term · t ‖ '–'
	expr → term	expr = term
	term → 9	term · t = '9'
	term → 5	term · t = '5'
	term → 2	term · t = '2'

(c) Both (a) and (b)
(d) None of the above

Q.13 Let G be a grammar is used for arithmetic expressions. The grammar G is shown below with semantic actions and attribute "sign" can contain either 0 or 1.

$E \to E_1 + E_2$	{E.sign = E_2. sign}
$E \to E_1 \times (E_2)$	{E.sign = E_1. sign × E_2.sign}
$E \to E_1/E_2$	{if (E_1.sign ==0) then E.sign = 1 else E.sign = 0}
$E \to +E_1$	{E.sign = 0}
$E \to -E_1$	{E.sign = 1}
$E \to$ id	{E.sign = 0}

Find the attribute value at the root E for the given input: –id × (–id + id)

[*Note:* $E \to E_1 + E_2$ is same as $E \to E + E$].

Q.14 Consider the following augmented grammar.

$S' \to S$
$S \to A \mid B$
$A \to fAi \mid n$
$B \to fBii \mid d$

Find the number of reduced entries corresponds to the production $B \to d$ in the LR(1) parsing table.

Q.15 Consider the following SDT:

$G \rightarrow A - T$ {Print "–"}
$A \rightarrow Ea$ {Print "*"}
$E \rightarrow Eb$ {Print "+"}
$E \rightarrow \varepsilon$ {Print "0"}
$T \rightarrow Eb$ {Print "1"}

If the SDT uses L-attributed definition, what is the output printed for evaluation of an input string "ba – bb"?

(a) * + 0 – 1 + 0 (b) 0 * + 0 – 1 +
(c) 0 + * 0 + 1 – (d) 0 * + 0 + –1

Q.16 Consider the following SDT:

$E \rightarrow E + T$	$\{E.x = E.x \times T.x\}$
$E \rightarrow T$	$\{E.x = T.x\}$
$T \rightarrow T * F$	$\{T.x = T.x - F.x\}$
$T \rightarrow F$	$\{T.x = F.x - 1\}$
$F \rightarrow id$	$\{F.x = 5\}$

If bottom up parsing uses S-attributed definition then what is the value of attribute evaluated at root node E for an input string "id + id * id"?

Q.17 Consider the following SDT:

$C \rightarrow C \square S$	$\{C.val = C_1.val + S.val\}$
$C \rightarrow S$	$\{C.val = S.val\}$
$S \rightarrow SOE$	$\{S.val = S_1.val \times E.val\}$
$S \rightarrow E$	$\{S.val = E.val\}$
$E \rightarrow id$	$\{E.val = id.num\}$

Find the value of input expression

"2O3□5O3□1O3".

Q.18 Consider the following SDT:

$G \rightarrow A - T$ $\{G.x = A.x - T.x\}$
$A \rightarrow Ea$ $\{A.x = E.x \times 2\}$
$E \rightarrow Eb$ $\{E_1.x = 1 + E_2.x\}$
$E \rightarrow \varepsilon$ $\{E.x = -1\}$
$T \rightarrow Eb$ $\{T.x = E.x - 2\}$

If above SDT uses L-attributed definition then what is the value of an attribute x at root after evaluation for an input string "ba – bb".

Q.19 Consider the following SDT (Syntax Directed Translation):

$A \rightarrow A \# B$	$\{A \cdot val = A \cdot val \times B \cdot val\}$
$A \rightarrow B$	$\{A \cdot val = B \cdot val\}$
$B \rightarrow B \& C$	$\{B \cdot val = B \cdot val - C \cdot val\}$
$B \rightarrow C$	$\{B \cdot val = C \cdot val\}$
$C \rightarrow num$	$\{C \cdot val = num\}$

What will be the output of the following expression?

14 # 13 & 12 & 1 # 10 & 9 # 8 & 2 & 1 # 5

Answer Key:

1. (a)	**2.** (c)	**3.** (b)	**4.** (b)	**5.** (c)
6. (a)	**7.** (d)	**8.** (a)	**9.** (b)	**10.** (d)
11. (c)	**12.** (b)	**13.** (0)	**14.** (2)	**15.** (c)
16. (–4)	**17.** (24)	**18.** (2)	**19.** (0)	

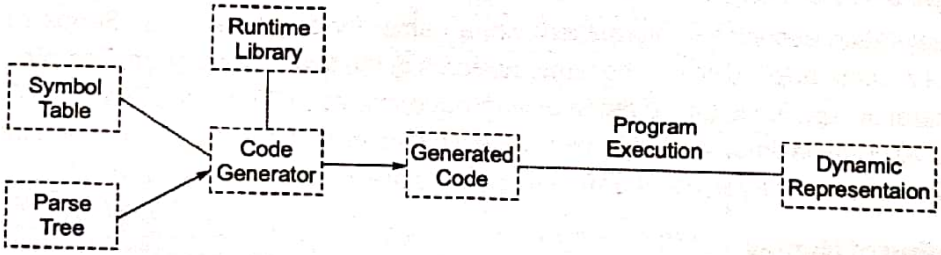
CHAPTER 05

Run-Time Environment and Target Code Generation

5.1 Introduction

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: essentially, the relationship between names and data objects. The runtime support system consists of routines that manage the allocation and de-allocation of data objects. It is important as it bridges the semantic gap between higher level concepts supported by programming language and the lower level concepts supported by target machine.

The complete plan applied by code generation to get the working of every source program on the target machine is known as the run-time environment. It provides the structure for the target machine to reproduce the working behavior of source programs. The run-time library executes this structure. The code generator must produce output code to work within the structure given by the run-time library.



5.1.1 Procedures

A procedure definition is a declaration that, in its simplest form, associates an identifier (procedure name) with a statement (body of the procedure).

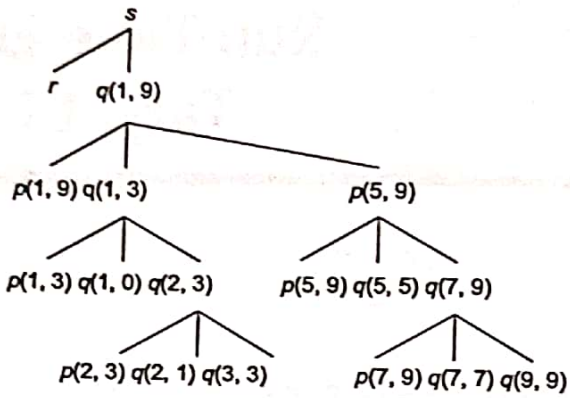
5.1.2 Activation Trees

Each execution of a procedure body is referred to as an activation of the procedure. Life time of an activation of a procedure '*p*' is the sequence of steps during the execution of a procedure body. If '*a*' and '*b*' are procedure activations then their life times are either non-overlapping or are nested. A procedure is recursive if a new activation can begin before an earlier activation of the same procedure has ended. An activation tree depicts the way control enters and leaves activations. Properties of activation trees are:

- Each node represents an activation of a procedure.
- The root shows the activation of the main program.
- The node for '*a*' is the parent of the node for '*b*' if and only if control flows from activation '*a*' to '*b*'.
- 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*'.

The flow of control in a program corresponds to a depth first traversal of the activation tree that starts at the root.

Example:



The above activation tree is for a program to sort numbers. It calls read function (*r*). Then it calls the quick sort function (*q*(1,9)), which in turn calls partition function (*p*(1,9)), which partitions the array and returns index 4 and results in further calls as shown.

5.1.3 Control Stacks

It can be used to keep track of live procedure activations. A procedure name is pushed on to the stack when it is called (activation begins) and it is popped when it returns (activation ends).

5.1.4 Scope of Declaration

A declaration associates information with a name, for example, int a;. Scope rules determine which declaration of a name applies when the name appears in the text of a program. The portion of the program to which a declaration applies is called the scope of that declaration. There are two type of scopes: LOCAL and NON LOCAL. At compile time, symbol table can be used to find the declaration that applies to an occurrence of a name. A symbol table entry is created whenever a declaration is encountered.

5.1.5 Binding of Names

The same name may denote different data objects (storage locations) at runtime. The term environment refers to a function that maps a name to a storage location and the association is referred to as the binding. The term state refers to a function that maps a storage location to the value held there.

An assignment changes the state but not the environment. A binding is the dynamic counterpart of a declaration. A local name in a procedure can have different binding in each activation of a procedure.

NOTE: *Static Notion:* Definition of a procedure, declaration of a name and scope of a declaration.
Dynamic Counterpart: Activations of the procedure, bindings of the name and lifetime of a binding.

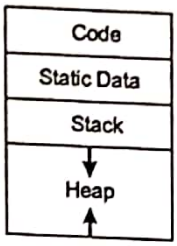
5.2 Storage Organization

Most block-structured languages require manipulation of runtime structures to maintain efficient access to appropriate data and machine resources.

Subdivision of Runtime Memory: Runtime storage can be subdivided to hold:

- **Target code:** Static (as its size can be determined at compile time)
- **Static data objects:** Static
- **Dynamic data objects:** Heap
- **Automatic data objects:** Stack

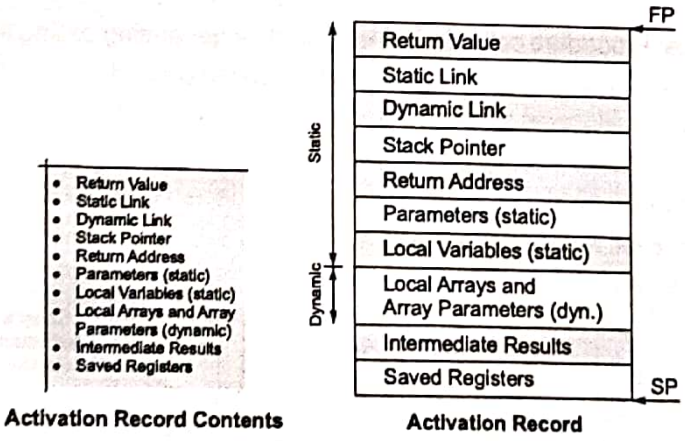
Advantage of statically allocating data objects is their address can be compiled into the target code.



5.3 Activation Records

Information needed by a single execution of a procedure is managed using a contiguous block of storage called "activation record". An activation record is allocated when a procedure is entered and it is deallocated when that procedure is exited. It contains temporary data, local data, machine status, optional access link, optional control link, actual parameters and returned value.

- Program Counter (PC) whose value is the address of the next instruction to be executed.
- Stack pointer (SP) whose value is top of the (top of stack, tos).
- Frame Pointer (FP) which points to the current activation record.



NOTE: Activation records are allocated from one of static area (like Fortran 77), stack area (like C or Pascal) and heap area (like lisp).

The description of the various field in the figure is given as:

- **Temporary data:** stores the values that arise in the evaluation of expressions.
- **Local data:** holds the data that is local to an execution of a procedure.
- **Machine status:** holds the information about the state of the machine just before the procedure call.
- **Optional access link:** refers to non-local data held in other activation records.

- **Optional control link:** points to the activation record of the caller.
- **Actual parameters:** calling procedure may supply parameter to the called procedure through this field.
- **Returned value:** this field may be used by the called procedure to return a value to the calling procedure. The advantage of placing the return value first in a frame is that the caller can find it at a fixed offset from its own frame.

5.4 Storage Allocation Strategies

5.4.1 Static Allocation

The names are bound with the storage at compile time only and hence every time a procedure is invoked its names are bound to the same storage location only. So, values of local names can be retained across activations of a procedure. Here compiler can decide where the activation record goes in respect to the target code and can also fill the addresses in the target code for the data it operates on.

Limitations

(*i*) Size of data should be known at compile time.

(*ii*) Recursive procedures are restricted, as all activations of a procedure use same bindings for local names.

(*iii*) No support for dynamic data structures.

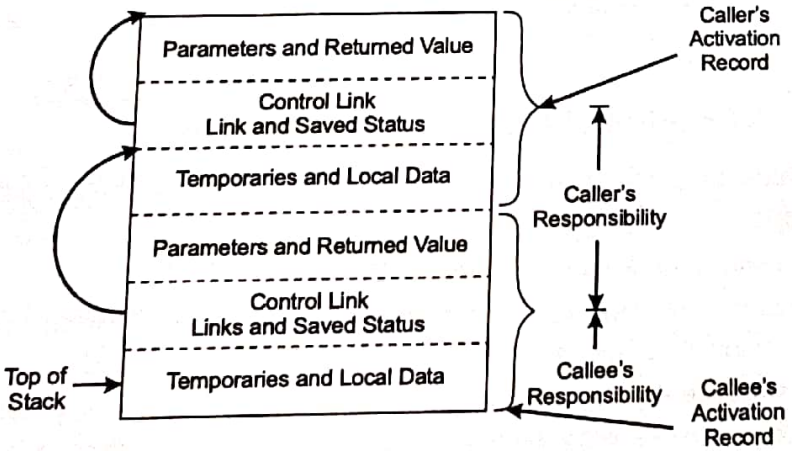
Example: FORTRAN was designed to permit static storage allocation.

5.4.2 Stack Allocation

Storage is organized as a stack, and activation records are pushed and popped as activations begin and end, respectively. Locals are contained in activation records, so they are bound to fresh storage in each activation.

Calling Sequence: Procedure calls are implemented by generating calling sequences in target code. We save space by putting as much code as possible into the called procedure.

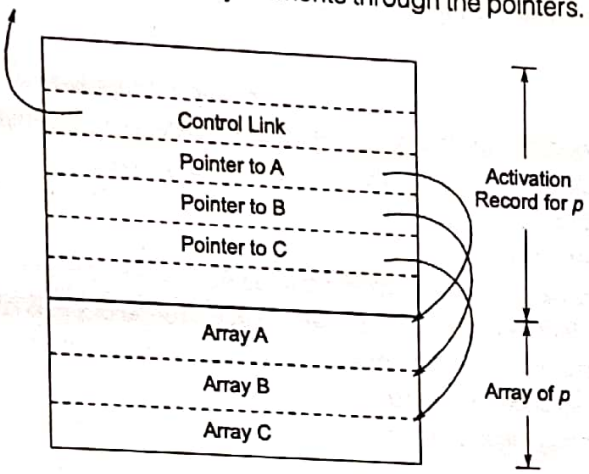
1. **Call sequence:** It allocates an activation record and enters information into its fields.
   - **Caller:** It evaluates actual parameters and it stores a return address and the old value of the top of stack counter into caller's activation record. It increments the top of stack counter.
   - **Callee:** It saves register values and other status information and it initializes its local data and begins execution.



2. **Return sequence:** It restores the state of the machine.
   - **Callee:** It places a return value next to the activation record of the caller and it restores the top of stack counter and other registers and branches to the return address in the caller's code.
   - **Caller:** It copies the return value to its own activation record and use it to evaluate an expression.

Variable Length Data

A common strategy for handling variable-length data is shown in following figure, where procedure *p* has three local arrays. The storage for these arrays is not part of the activation record for *p*; only a pointer to the beginning of each array appears in the activation record. The relative addresses of these pointers are known at compile time, so the target code can access array elements through the pointers.



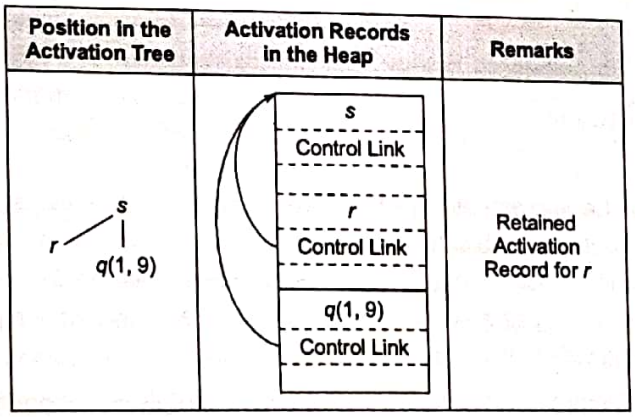
Dangling References

Whenever storage is deallocated, the problem of dangling references arises. A dangling reference occurs when there is a reference to storage that has been deallocated.

5.4.3 Heap Allocation

Heap allocation is required when:

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller. This possibility cannot occur for those languages where activation trees correctly depict the flow of control between procedures.



Heap allocation parcels out pieces of contiguous storage, as needed for activations records or other objects. Pieces may be decollated in any order, so over time the heap will consist of alternate areas that are free

and in use. As shown in the following figure, the record for an activation of procedure *r* is retained when the activation ends. The record for the new activation *q*(1, 9) therefore cannot follow that for *s* physically, as it did in the figure. Now if the retained activation record for *r* is deallocated, there will be free space in the heap between the activation records for *s* and *q*(1, 9). It is left to the heap manager to make use of this space.

5.5 Access to Non-Local Names (Scope of Variables)

Scope rule of the language determines the treatment of references to nonlocal names.

Lexical or Static Scope Rule: Determines the declaration that applies to a name by examines the program text alone. For lexical scope we use displays and access links.

Dynamic Scope Rule: Determines the declaration applicable to a name at run time, by considering the current activations.

Blocks

A block is a statement containing its own local data declarations. Delimiters (e.g., { and }) mark the beginning and end of a block. It is not possible for two blocks to overlap except that one is nested inside the other or one block is independent of other. The scope of declaration in a block-structured language is given by the most closely nested rule:

1. The scope of a declaration in a block B includes B.
2. If a name *x* is not declared in a block B, then an occurrence of *x* in B is in the scope of a declaration of *x* in an enclosing block B′ such that
   (a) B′ has a declaration of *x*, and
   (b) B′ is more closely nested around B than any other block with a declaration of *x*.

```
main( )
{
    int a = 0;
    int b = 0;
    {
        int b = 1;
        {
            int a = 2;
        B2  int a, b;
        }
B0  B1  {
            int b = 3;
        B3  int a, b;
        }
        print a, b;
    }
    print a, b;
}
```

Declaration	Scope
int a = 0;	B0 – B2
int b = 0;	B0 – B2
int b = 1;	B0 – B1
int a = 2;	B1 – B3
int b = 3;	B2
	B3

a = 0
b = 0
b = 1
a = 2, b = 3

Block structure can be implemented using stack allocation. Since the scope of a declaration does not extend outside the block in which it appears, the space for the declared name can be allocated when the block is entered, and deallocated when control leaves the block. This view treats a block as a "parameterless procedure," called only from the point just before the block and returning only to the point just after the block.

An alternative implementation is to allocate storage for a complete procedure body at one time. If there are blocks within the procedure, then allowance is made for the storage needed for declarations within the blocks. For block B0 in above figure, we can allocate storage as shown in figure (above).

5.5.1 Lexical Scope Without Nested Procedures

If there is a non-local reference to a name 'x' in some function, then 'x' must be declared outside any function. The scope of a declaration outside a function consists of the function bodies that follow the declaration, with holes if the name is redeclared within a function. The stack-allocation strategy for local names can be used. Storage for all names declared outside any procedures can be allocated statically.

The position of this storage is known at compile time, so if a name is nonlocal in some procedure body, we simply use the statically determined address. Any other name must be a local of the activation at the top of the stack accessible through the top pointer.

An important benefit of static allocation for nonlocals is that declared procedures can freely be passed as parameters and returned as results. With lexical scope and without nested procedures, any name nonlocal to one procedure is nonlocal to all procedures.

5.5.2 Lexical Scope with Nested Procedures

A nonlocal occurrence of a name x in a Pascal procedure is in the scope of the most closely nested declaration of x in the static program text.

Nesting Depth

The notion of nesting depth of a procedure is used to implement lexical scope. Let the name of the main program be at nesting depth 1; we add 1 to the nesting depth as we go from an enclosing to an enclosed procedure.

Access Links

A direct implementation of lexical scope for nested procedures is obtained by adding a pointer called an access link to each activation record. If procedure p is nested immediately within q in the source text, then the access link in an activation record for p points to the access link in the record for the most recent activation of q. Suppose procedure p at nesting depth n_p refers to a nonlocal a with nesting depth $n_a \leq n_p$. The storage for a can be found as follows:

1. When control is in p, an activation record for p is at the top of the stack. Follow $n_p - n_a$ access links from the record at the top of the stack. The value of $n_p - n_a$ can be precomputed at compile time. If the access link in one record points to the access link in another, then a link can be followed by performing a single indirection operation.
2. After following $n_p - n_a$ links, we reach an activation record for the procedure that a is local too.

Hence, the address of nonlocal a in procedure p is given by the following pair computed at compile time and stored in the symbol table:

($n_p - n_a$, offset within activation record containing a)

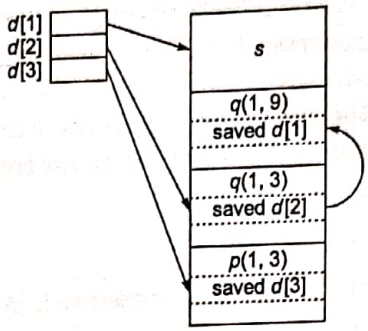
The code to set up access links is part of the calling sequence. Suppose procedure p at nesting depth n_p calls procedure x at nesting depth n_x. The code for setting up the access link in the called procedure depends on whether or not the called procedure is nested within the caller.

Case-1 $n_p < n_x$: Since the called procedure x is nested more deeply than p, it must be declared within p, or it would not be accessible to p. In this case, the access link in the called procedure must point to the access link in the activation record of the caller just below in the stack.

Case-2 $n_p \geq n_x$: From the scope rules, the enclosing procedures at nesting depths 1, 2,..., $n_x - 1$ of the called and calling procedures must be the same. Following $n_p - n_x + 1$ access links from the caller we reach the most recent activation record of the procedure that statically encloses both the called and calling procedures most closely. The access link reached is the one to which the access link in the called procedure must point. Again, $n_p - n_x + 1$ can be computed at compile time.

Procedure Parameters: Lexical scope rules apply even when a nested procedure is passed as a parameter. A nested procedure that is passed as a parameter must take its access link along with it.

Displays: Faster access to nonlocals than with access links can be obtained using an array d of pointers to activation records, called a display. We maintain the display so that storage for a nonlocal a at nesting depth i is in the activation record pointed to by display element $d[i]$.



Using a display is generally faster than following access links because the activation record holding a nonlocal is found by accessing an element of d and then following just one pointer. The display changes when a new activation occurs, and it must be reset when control returns from the new activation. When a new activation record for a procedure at nesting depth i is set up, we

1. Save the value of $d[i]$ in the new activation record and
2. Set $d[i]$ to point to the new activation record.

Just before an activation ends, $d[i]$ is reset to the save value.

5.5.3 Dynamic Scope

Under dynamic scope, a new activation inherits the existing bindings of nonlocal names to storage. A nonlocal name x in the called activation refers to the same storage that it did in the calling activation. New bindings are set up for local names of the called procedure; the names refer to storage in the new activation record. The following two approaches to implementing dynamic scope:

1. **Deep Access:** Use the control link to search into the stack, looking for the first activation record containing storage for the nonlocal name. The term deep access comes from the fact that the search may go "deep" into the stack. The depth to which the search may go depends on the input to the program and cannot be determined at compile time.
2. **Shallow Access:** Hold the current value of each name in statically allocated storage. When a new activation of a procedure p occurs, a local name n in p takes over the storage statically allocated for n. The previous value of n can be saved in the activation record for p and must be restored when the activation of p ends.

5.6 Symbol Table Implementation

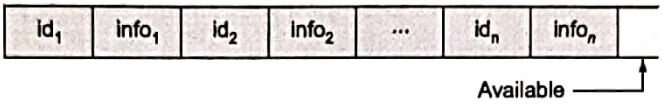
A compiler uses a symbol table to keep track of scope and binding information about names. The symbol table is searched every time a name is encountered in the source text. Changes to the table occur if a new name or new information about an existing name is discovered. The two symbol-table mechanism are linear lists and hash tables. Linear list is the simplest to implement, but its performance is poor when inquiries and entries get large. Hashing schemes provide better performance for somewhat greater programming effort and space overhead. Each entry of symbol table typically includes:

- The basic type of a variable (ptr, int, char, float, struct, etc.)
- Structure layout, pointer specifiers, array information

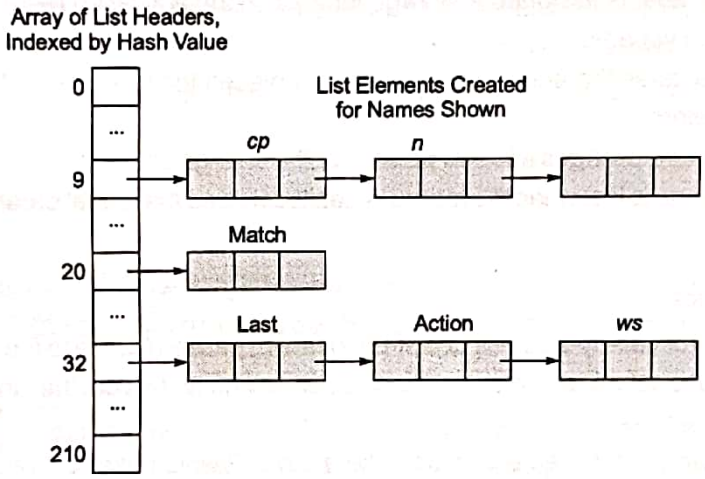
- Initialization values
- Scope information
- Storage specifiers
- Names

5.6.1 List Data Structure for Symbol Table

We use a single array, or equivalently several arrays, to store names and their associated information. New names are added to the list in the order in which they are encountered. The position of the end of the array is marked by the pointer available, pointing to where the next symbol-table entry will go. The search for a name proceeds backwards from the end of the array to the beginning. When the name is located, the associated information can be found in the words following next. If we reach the beginning of the array without finding the name, a fault occurs – an expected name is not in the table.



5.6.2 Hash Table



There are two parts to the data structure:

1. A hash table consisting of a fixed array of *m* pointers to table entries.
2. Table entries organized into m separate linked lists, called buckets (some buckets may be empty). Each record in the symbol table appears on exactly one of these lists.

To determine whether there is an entry for string *s* in the symbol table, we apply a hash function *h* to *s*, such that $h(s)$ returns an integer between 0 and $m-1$. If *s* is in the symbol table, then it is on the list numbered $h(s)$. If *s* is not yet in the symbol table, it is entered by creating a record for *s* that is linked at the front of the list numbered $h(s)$.

Suitable approach for computing hash functions is to proceed as follows:

1. Determine a positive integer *h* from the characters $c_1, c_2, \ldots, c_k$ in string *s*. The conversion of single characters to integers is usually supported by the implementation language.
2. Convert the integer *h* determined above into the number of a list, i.e., an integer between 0 and $m-1$. Simply dividing by *m* and taking the remainder is a reasonable policy. Taking the remainder seems to work better if *m* is a prime.

5.7 Parameter Passing

Let us consider the following PASCAL procedure which exchanges two elements of an integer array, a:

```
procedure exchange (i, j: integer);
var x : integer;
begin
        x : = a[i];
        a[i]: = a[j];
        a[j]: = x;
end;
```

Communication between this procedure and its caller is through the nonlocal a and through the parameters i and j. There are several methods of associating actual and formal parameter : *call – by- value, call-by-reference, copy-restore*, and *call-by-name*. The left value (*l*-value), the right value (*r*-value), or the name of a variable may be passed to the called procedure.

5.7.1 Call-by-value

Usually, *C* and Pascal use call-by-value: the *r*-value of the actual parameter are passed to the called procedure:

- A formal parameter is assigned a storage location in the activation record of the called procedure (just like a local variable)
- The caller evaluates the actual parameters and places their *r*-value in the storage location of the formal parameters.

If the called procedure changes a formal parameter the change only occurs in the activation record of the called procedure. The change is lost when the record is deallocated so the actual parameter in the caller's record is not changed.

5.7.2 Call-by-Reference

Call-by-reference (or call-by-address or call-by-location) passes *l*-value to the called procedure:

- If an actual parameter is a name or an expression having an *l*-value, then that *l*-value is passed to the called procedure.
- If the actual parameter is an expression like $a + b$ or 2 which has no *l*-value, then the expression is evaluated in a new location, and the address of that location is passed to the called procedure.

Let us consider the following PASCAL procedure.

```
procedure swap (var x, y; integer);
    var temp : integer;
    begin
        temp : = x;
        x : = y;
        y : = temp:
    end:
```

In the *swap* procedure, all reads and writes of the formal parameters, x and y, will read and write the associated actual parameter in the caller's activation record. Thus, *swap* will actually swap the actuals.

Example: If the swap procedure above were called with swap (i, $a[i]$) then the following steps occur:

- The *l*-value of i and $a[i]$ would be copied into the activation record of swap.
- temp would be set to the *r*-value of i, say 10.

- *i* would be set to the *r*-value of a [10].
- *a*[10] would be set to the *r*-value of temp or 10.
  Using call-by-reference, the *r*-values of the arguments are swapped correctly.

5.7.3 Copy-Restore

Copy-restore is a hybrid of call-by-value and call-by-reference:

- The actual parameters are evaluated before the call and their *r*-values are passed to the called procedure as in call-by-value. But the caller also remembers the *l*-values of those actuals that have *l*-values.
- When control returns to the caller it copies back the *r*-values of the formal parameters into the *l*-values of the actual.

Usually, copy-restore has the same affect as call-by-reference. It could have a different effect when the called procedure refers to one-or-more of the actual parameters as a non-local.

5.7.4 Call-by-name

Call-by-name is used by ALGOL:

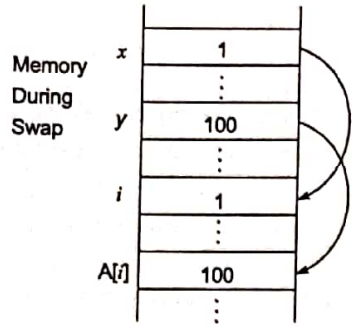
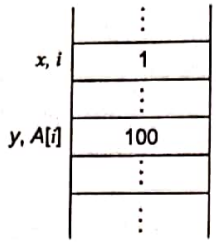
- The procedure is treated as if it were a macro; i.e., its body is substituted for the call in the caller, with the actual parameters literally substituted for the formal.
- Local names in the procedure are kept distinct from names in the caller.
- Actual parameters are surrounded by paranthesis if necessary to preserve their integrity.

Example: Let us suppose that example 1 is repeated by call-by-name instead of call-by-reference:
The swap (*i*, *a*[*i*]) call in the caller is replaced with
temp : = *i*;
i : = *a*[*i*];
a[*i*]: = temp;
The first line copies the r-value of *i*[10] into temp; The 2nd line copies the *r*-value of *a*[10] into *i*; The 3rd line copies the *r*-value of temp [10] into *a*[*a*[10]]. Using call-by-name, the *r*-values of the arguments are not supported correctly.

Example-5.1 Identify the output printed by the following paradigm. Using
1. call by value 2. call by reference 3. call by copy restore

```
Parameter Passing : Swap Paradigm
SWAP (x, y: int)
{
    z = x;
    x = y;
    y = z;
}
main () {
    i = 1;
    A[i] = 100;
    swap (i, A[i]);
    print i, A[i];
}
```

Solution:

Call by Value	Call by Reference	Call by Copy Restore
 A call to SWAP PASSES (x, y) the value of $(i, A[i])$ Prints 1, 100 (Does not swap)	 A call to SWAP makes (x, y) reside at memory location of $(i, A[i])$ Prints : 100, 1 (Swaps)	(x, y) get values of $(i, A[i])$ and upon termination new values of (x, y) are copied to initial memory locations for $(i, A[i])$. Prints : 100, 1 (Swaps)

The formal parameters (x, y) are substituted by $(i, A[i])$

Sequence of Events

$z = i$ $(z = 1)$

$i = A[i]$ $(i = 100)$

$A[i] = z;$ $(A[100] = 1)$

Prints: 100, 1 (Does not swap). This memory location A[100], not $A[i] = 100$.

Example-5.2 Identify the output printed by

1. call by reference 2. call by copy restore 3. macro expansion

```
assign (x, y) {
    i = y;
}
main( ) {
    i = 1; j = 2;
    assign (i, j);
    print (i, j);
}
```

Solution:

1. Call By Reference Prints: 1, 2.
2. Copy Restore Prints: 2, 2 and
3. Call by Macro Expansion

5.8 Code Generation

It is final phase of compiler model

5.8.1 Requirements Imposed on a Code Generator

Following are the requirements imposed on a code generator:

1. Preserving the semantic meaning of the source program and being of high quality.
2. Making effective use of the available resources of the target machine.
3. The code generator itself must run efficiently.

5.8.2 Basic Task of Code Generator

A code generator has three basic tasks:

1. Instruction selection
2. Register Allocation
3. Instruction ordering

5.8.3 Design Issue

Following are some issues in designing of code generator:

1. It should perform all the general tasks, which almost all code generator do, i.e., instruction selection, register allocation and instruction ordering.
2. It should produce correct code
3. It should be designed in such a manner, so that it can be easily implemented, tested and maintained.

5.8.4 Input to the code generator

The input to the code generator is the intermediate representation of the source program produced by the front end along with information in the symbol table that is used to determine the run-time address of the data denoted in IR.

Choices of Instruction Register (IR)

For input to the code generator, we have following three choices:

1. Three-Address codes in either quardruples, triples or indirect triple format.
2. Linear representation such as post fix notation.
3. Graphical representation such as syntax trees and DAG's

5.9 Addresses in the Target Code

Program runs in its own logical address space that was divided into four code and data areas:

1. **CODE:** It is statically determined area that holds the executable target code.
2. **STATIC:** It is statically determined area for holding data objects and other data generated by the compiler.
3. **HEAP:** It is dynamically managed area for holding data objects that are allocated during program execution.
4. **STACK:** It is dynamically managed area for holding activation records as they are created and destroyed during procedure calls and returns.

Example: Consider $x = 0$. Note that the variable in a three-address code is point to a symbol-table entry for that variable name; and any variable name must be replaced by code to access memory storage location.

- Let the symbol table entry for x contains a relative address 12.
- x is statically allocated area begin at address STATIC.
- therefore, the actual run-time address of x is STATIC + 12.
- Finally, the actual assignment is STATIC [12] = 0

Summary

- The runtime environment is the structure of the target computers registers and memory that serves to manage memory and maintain information needed to guide a programs execution process.
- Every procedure will have only one activation record which is allocated before execution.
- Variables are accessed directly via fixed addresses.
- Little book keeping overhead; i.e., at most return address may have to be stored in activation record.
- In this, activation records are allocated (push of the activation records) whenever a function call is made. The neccessary memory is taken from stack portion of the program. When program execution returns from function the memory used by the activation record is deallocated (pop of the activation record). Thus, stack grows and shrinks with the chain of function calls.
- Functional languages such as Lisp, ML, etc. uses this style of call stack management. Sailently, here activation records are de-allocated only when all references to them have disappeared, and this requires that activation records to dynamically freed at arbitrary times during execution. Memory manager (garbage collector) is needed.
- The data structure that handles such a management is heap and thus this is also called as heap management.
- Information needed by a single execution of a procedure is managed using a contiguous block of storage called "activation record".
- An activation record is allocated when a procedure is entered and it is deallocated when that procedure is exited.
- It contains temporary data, local data, machine status, optional access link, optional control link, actual parameters and returned value.
- Program Counter (PC) whose value is the address of the next instruction to be executed.
- Stack pointer (SP) whose value is top of the (top of stack, tos).
- Frame Pointer (FP) which points to the current activation record.
- Activation records are allocated from one of static area (like Fortran 77), stack area (like C or Pascal) and heap area (like lisp).

Student's Assignments

Q.1 Consider the following statements:

S_1 : Static allocation bindings do not change at run time

S_2 : Heap allocation allocates and de-allocates storage an run time.

Which of the above statements is/are true?

(a) S_1 is true and S_2 is false

(b) S_2 is true and S_2 is false

(c) Both S_1 and S_2 are true

(d) Both S_1 and S_2 are false

Q.2 Approach that is used to implement dynamic scope is/are

(a) Shallow access (b) Deep access

(c) Non local access (d) Both (a) and (b)

Q.3 The optional access link in the activation record is used for

(a) Pointing to the activation record of the caller

(b) Referring the non local data held in other activation records

(c) Temporary values, such as those arising in the evaluation of expressions
(d) None of these

Q.4 In static scoping undefined variable
(a) Search in upper block
(b) Search where the function is called
(c) Refers global variable
(d) None of the above

Q.5 Uniform symbol table
(a) Contains all constants in the program
(b) Is a permanent table of decision rules in the form of patterns for matching with the uniform symbol table to discover syntactic structure.
(c) Consists of full or partial list of the token's as they appear in the program. Created by Lexical analysis and used for syntax analysis and interpretation.
(d) A permanent table which lists all key words and special symbols of the language in symbolic form.

Q.6 The following program fragment is written in a programming language that allows variables and nested declarations of functions:

```
global int a = 100, b = 200;
int funct1 (int count);
int funct2 (int c);
void main( ) {
    int count;
    for (count = 1; count < = 5; ++ count)
    print f("%d", funct 1(count));
}
int funct 1(int x)
{   int c, d;
    c = funct 2(x);
    d = (c < 100)? (a + c) : b;
  return (d);
}
int funct 2(int x)
{   static int z = 1;
    z * = x;
    return (z);
}
```

What will be the output of the above program?

(a) 101 102 104 106 124
(b) 101 102 103 104 105
(c) 101 102 106 108 110
(d) 101 102 106 124 200

Q.7 The following program fragment is written in a programming language that allows variables and does not allow nested declarations of functions:

```
global int i = 100, j = 200
void P(x)
{
    int i = 10;
    printf(x + 10);
    i = 200;
    j = 40;
    printf(x);
}
main( ) {
    P(i + j);
}
```

If the programming language used dynamic scoping and call by name parameter passing mechanism, the value printed by the above program are

(a) 310, 240 (b) 325, 240
(c) 215, 150 (d) 240, 315

Common Data for Questions (8 and 9):

The pascal program shown below:

```
Program PARAM (input, output);
    var m, n : integer;
    procedure P(var x, y: integer);
    var m : integer;
    begin
        m : = 1;
        x : = y + 1;
    end;
    Procedure Q(x : integer; var y: integer);
    begin
    x : = y + 1;
    end;
    begin
        m : = 0; P(m, m); write (m);
        n : = 0; Q(n* 1, n); write (n)
    end
```

Q.8 The value of *m*, output by the program PARAM is
(a) 1, because *m* is the a local variable in P
(b) 0, because *m* is the actual parameter that corresponds to the formal parameter *y* in P
(c) 0, because both *x* and *y* are just reference to *m*, and *y* has the value 0
(d) 1, because both *x* and *y* are just references to *m* which gets modified in procedure P

Q.9 The value of *n*, output by the program PARAM is
(a) 0, because *n* is the actual parameter corresponding to *x* in procedure Q
(b) 0, because *n* is the actual parameter to *y* in procedure Q
(c) 1, because *n* is the actual parameter corresponding to *x* in procedure Q
(d) 1, because *n* is the actual parameter corresponding to *y* in procedure Q

Q.10 What will be the output of the following program for call by name and copy restore parameter passing mechanism respectively?

```
int i;
void func(int a, int b) {
    a = a + b;
    b = a + b;
    a = b – a;
    b = a – b;
}
main( )
{
    int j;
    i = 4;
    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    func(i, a[i]);
    for (j = 0; j < 10; j++)
      printf("%d", j);
}
```

(a) Call by name:
–1 2 3 4 5
6 7 8 9 9
Copy restore:
1 2 3 4 –9
6 7 8 9 0

(b) Call by name:
0 2 3 4 5
6 7 8 9 0
Copy restore:
1 2 3 4 –9
6 7 8 9 0

(c) Call by name:
–1 2 3 4 5
6 7 8 9 9
Copy restore:
1 2 3 4 5
6 7 8 9 0

(d) Call by name:
1 2 3 4 5
6 7 8 9 9
Copy restore:
1 2 3 4 –8
6 7 8 9 0

Q.11 Which of the following technique can be used as intermediate code generation?
(a) Postfix notation and three address code
(b) Quadruples
(c) Both (a) and (b)
(d) None of these

Q.12 The output of the code generator is a target program that includes
(a) Assembly language
(b) Relocatable machine language
(c) Absolute machine language
(d) All of the above

Q.13 Which is not a three address code?
(a) If $x < y$ go to *L*
(b) $x = y[I]$
(c) $x = \&y$
(d) None of these

Q.14 Which of the following statements is true?
(a) For optimizing compiler, moving a statement that defines a temporary value requires us to change all reference to that statements. It is an overhead for triples notation.
(b) For optimizing compiler, triples notations has important benefit where statements are often moved around as there are no movement or change.

(c) Quadruples have some disadvantages over triples notation for an optimizing compiler.
(d) Both (a) and (b)

Q.15 Three address code technique for intermediate code generation shows that each statement usually contains three addresses. Three addresses are as follows
(a) Two addresses for operands, one for operator
(b) One for all operator, one for all operands and one for result
(c) One for result, two for operands
(d) None of the above.

Q.16 Code generation can be done by
(a) Binary tree
(b) Labeled tree
(c) DAG tree
(d) Both (b) and (c)
(e) None of the above

Q.17 What will be the optimized code when the expression $p = q * -r + q + -r$ is represented in DAG specification?

(a) $t_1 = -r$
$t_2 = q * t_1$
$t_3 = t_1 + t_2$
$t_4 = t_2 + t_3$
$p = t_3$

(b) $t_1 = -r$
$t_2 = q$
$t_3 = t_1 * t_2$
$t_4 = t_3 + t_3$
$p = t_4$

(c) $t_1 = -r$
$t_2 = q * t_1$
$t_3 = q * t_1$
$t_4 = t_2 + t_3$
$p = t_4$

(d) $t_1 = -r$
$t_2 = q * t_1$
$t_3 = q + t_1$
$t_4 = t_2 + t_3$
$p = t_4$

Q.18 Generation of intermediate code based on an abstract machine model is useful in compilers because
(a) It makes implementation of lexical analysis and syntax analysis easier
(b) Syntax directed translation can be written for intermediate code generation
(c) It enhances the portability of the front end of the compiler
(d) It is not possible to generate code for real machine directly from high level language programs.

Q.19 Which of the following is not a type of three address statements?
(a) Copy statements
(b) Index assignment statements
(c) Pointer assignments
(d) None of the above

Q.20 In a simplified computer the instructions are:

OP R_j, R_i : Performs R_j OP R_i and stores the result in register R_i.

OP m, R_i : Performs val OP R_i and stores the result in R_i. val denotes the content of memory location m.

MCV m, R_i : Moves the content of memory location m to register R_i.

MCV m, R_i, m: Moves the content of register R_i to memory location m.

The computer has only two registers, and OP is either ADD or SUB. Consider the following basic block:

$t_1 = a + b$
$t_2 = c + d$
$t_3 = e - t_2$
$t_4 = t_1 - t_3$

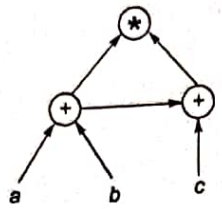
Assume that all operands are initially in memory. The final value of the computation should be memory. What is the minimum number of MOV instructions in the code generated for this basic block?
(a) 2 (b) 3
(c) 5 (d) 6

Q.21 Consider the following 3-address code:

$t_1 = t + e$
$t_2 = g + a$
$t_3 = t_1 * t_2$
$t_4 = t_2 + t_2$
$t_5 = t_4 + t_3$

There are five temporary variables in above code. Find minimum number of temporary variables can be used in the equivalent optimized 3-address code of above code.

Q.22 Consider the directed acyclic graph.



Find the expression that represents above DAG.

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

(c) $a + b* + c$ (d) None of these

Q.23 Consider the following three address code table.

Operation	Operand-1	Operand-2	Result
–	c		t_1
×	b	t_1	t_2
+	a	t_2	t_3
Move	t_3		a

Which of the following expression represents the above three address code (quadruple notation)?

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

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

Answer Key:

1. (c)	**2.** (d)	**3.** (b)	**4.** (c)	**5.** (c)
6. (d)	**7.** (a)	**8.** (d)	**9.** (a)	**10.** (b)
11. (a)	**12.** (d)	**13.** (d)	**14.** (d)	**15.** (c)
16. (c)	**17.** (d)	**18.** (d)	**19.** (d)	**20.** (c)
21. (2)	**22.** (d)	**23.** (c)		