Compilers:

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

$$\begin{tikzcd}\text{Source program} \arrow[r] & \boxed{\text{Compiler}} \arrow[r] & \text{Target Program}\end{tikzcd}$$

The compiler reports to its user the presence of errors in the source program.

The Analysis Synthesis Model for Compilation

There are two parts of compilation:

i. Analysis
ii. Synthesis

i) Analysis: Decomposes the source program into an intermediate representation.

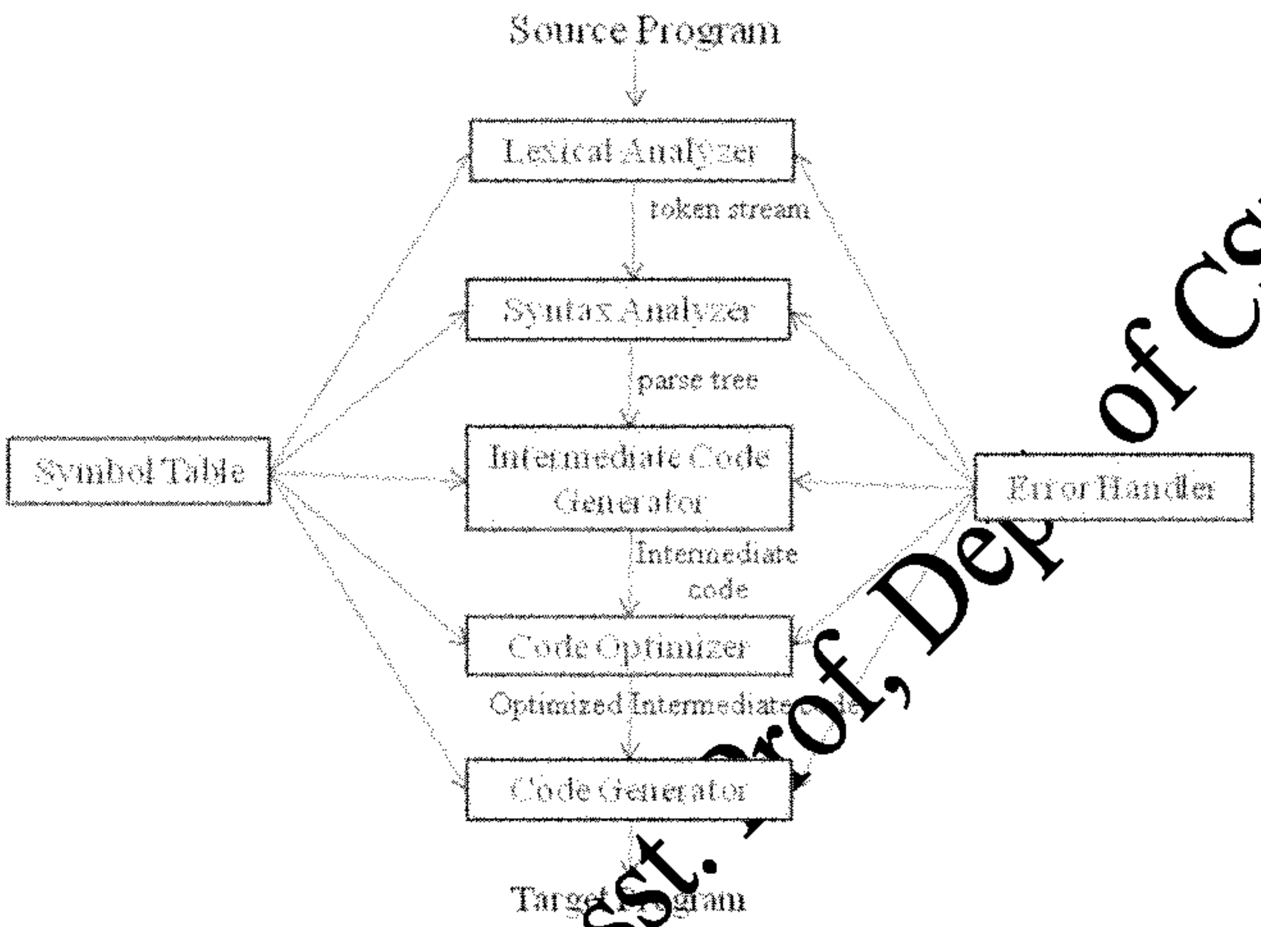
ii) Synthesis: The synthesis part constructs the desired target program from the intermediate representation.

Structure of a compiler:

A compiler takes as input a source program and produces as output an equivalent sequence of machine instructions.

This process is so complex that it is customary to partition the compilation process into a series of sub process called phases.

A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation.



Among these phases Lexical Analysis, Syntax Analysis and Intermediate Code Generator come under the Front – End of compiler, and remaining two phases, the Code Optimization and Code Generation come under Back – End of compiler.

Phase I: Lexical Analyzer (or) Scanner:-

The Lexical Analysis phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequences of character, such as an identifier, a keyword (if, while, etc), a punctuation character. The character sequence forming a token is called the lexeme for the token.

Phase II: Syntax Analyzer / Parsing:

The Syntax Analyzer groups tokens together into hierachical/structures. The hierarchical structure generated in this phase is called parse tree (or) Syntax tree.

Phase III: Intermediate Code Generator:

It uses the structure produced by the syntax Analyzer to create a stream of simple instructions. The intermediate representation can have a variety of forms. One common styles uses instructions with one operator and a small no. of operands. These instructions can be viewed as simple macros where in a instruction will be translated into a sequence of Assembly language statement.

Phase IV: Code Optimization:

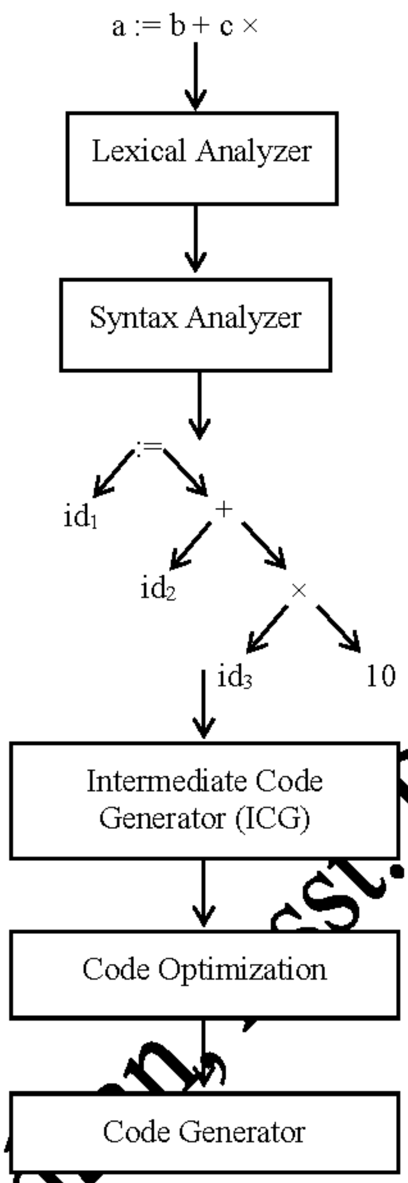
Code optimization is an optional phase designed to improve the intermediate code so that the ultimate object program run faster and/or takes less space.

Phase V: Code Generation:

The final phase of the complier is the generation of target code, consisting normally of relocatable machine code (or) assembly code. Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task.

Example:

Consider the statement "a: = **b + c * 10; "**The following figure shows the representation of this statement after each phase.



ICG:

$temp_1 = 10;$
$temp_2 = id_3 \times temp_1$
$temp_1 = id_2 + temp_2$
$id_1 = temp_3$

Code Optimizer:

$temp_1 = id_3 \times 10$
$id_1 = id_2 + temp_1$

Code Generator:

```
MOV id3 , R1
MUL 10 , R1
MOV id2 , R2
ADD R1 , R2
MOV R2 , id1
```

Translation of a Statement

b) Symbol – Table Management:

A symbol-table is a data structure contains 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 the record quickly.

c) 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.

The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler.

Chapter-I

Lexical Analysis

Lexical Analyzer:

The lexical analyzer is the first phase of a compiler. Whenever the lexical analyzer reads the source program, it performs two tasks:

- The primary task is to read the input characters and produces as output a sequence of tokens that the parser uses for syntax analysis.
- In secondary task it removes comments and white space from the source program.
- It also makes a copy of source program with the error messages in it.

Tokens, Patterns and lexemes:

Tokens: Token is a sequence of characters that can be treated as a single logical entity. It describes the class (or) category of input sting.
Ex: Identifier, Keywords, Constants………………

Patters: Set of rules that describe the token.

Lexeme: Sequence of characters in the source program that are matched with the pattern of the token.
Ex: in Pascal statement **const pi =3.1416;**

Lexeme	Token
Const	Constant
Pi	Identifier
=	Relational operator
3.1416	Number
;	Punctuation symbol

Specification of tokens:

Tokens are specified by regular expressions. Regular expressions are an important notation for specifying patterns. When a pattern is matched by some regular expression then token can be recognized.

Strings and languages:

The term alphabet (or) character class denotes any finite set of symbols denoted by 'Σ'.

String:

A string is a finite sequence of symbol, over an alphabet Σ. It is denoted by 'w'.

w= " IESGATE"

a) **Length of a string ($|w|$):** Is the no. of symbols composing the string.

w = **IESGATE, $|w|$=7**

b) **Empty string 'ε'**: Is a special of zero symbols and zero length. It is denoted by 'ε'.

c) The empty set of strings is denoted by 'ϕ', {}

d) **Power Alphabet(Σ^k):** It is the set of all strings of length 'k' over alphabet .

Ex: Let Σ = {0,1}

Σ^0={ε}, ε is the string having length zero.

Σ^1={0,1}, set of all strings of length 1

Σ^2={00,01,10,11}, set of all string of length 2

Σ^3 = {000,001.....111} , set of all strings of length 3

e) Σ^* : Set of all strings of length zero (or) more.

Ex: Σ = {0,1}

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup$----- = {ε, 0, 1, 00 ,01 ,10 ,11 ,000 ,001 -----}

f) Σ^+ : Set of all non-empty strings.

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup$----- = {0, 1, 00, 01, 10, 11, 000, 001-----}

g) Operations on strings:

i) Prefix

ii) Suffix

iii) Substring

iv)Proper prefix

v) Proper suffix

vi)Proper substring

vii) Subsequence

i) Prefix: A string obtained by removing zero or more leading symbols.

ii) Suffix: A string obtained by removing zero or more trailing symbols

iii) Substring : A sting obtained by removing prefix and suffix of a given string is called substring.

iv)Proper prefix and suffix:

A prefix, substring or suffix, of a string, other than the ε and string itself, is called a proper prefix or suffix, substring.

Ex: *w* = *abc*

SI. No	String operations	Obtained strings	No. of strings
1.	Prefix	ε, a, ab, abc	n+1
2.	Suffix	ε, c, bc, abc	n+1
3.	Substring	ε, a, b, c, ab, bc, abc	$\frac{n(n+1)}{2}+1$
4.	Proper prefix	a, ab	n-1
5.	Proper suffix	c, bc	n-1
6.	Proper substring	a, b, c, ab, bc	$\frac{n(n+1)}{2}-1$
7.	Sub sequence	ε, a, b, c, ab, bc,ac, abc	2^n

❖ Language is a collection of strings. Regular are mathematical symbolism which describe the set of strings of specific language. It produces convenient and useful notation for representing tokens. Each regular expressions ***r*** denotes a language L(r)

- The rules that define the regular expressions over alphabet 'Σ' are:

1) ε is a regular expression that denotes $\{\varepsilon\}$, i.e., the set containing the empty string.
2) Suppose ***r*** and ***s*** are regular expressions denoting the language L(r) and L(s). Then,

   r/s , ***r*** + ***s*** is a regular expression denoting the language L(r) U L(s).
3) If R_1 and R_2 is regular expression then

   $R = R_1 \circ R_2$

   R is also regular expression which represents concatenation operation.
4) If R is a regular expression then $R = R^*$ is also a regular expression which represents *kleen* closure.

A language denoted by regular expressions is said to be a regular set (or) regular language.

Ex: Write a regular expression (R.E) for a language containing the strings of length two, over $\Sigma = \{0,1\}$

Sol: R.E = (0+1) (0+1)

Ex: Write a regular expression for a recognizing identifier.

Sol: R.E = letter (letter + digits *)

Lexical analysis is a process of recognizing tokens from input source program. The lexical analyzer stores the input in a buffer. It builds the regular expressions for corresponding tokens. From these regular expressions, a finite automaton is built. When lexeme matches with the pattern generated by finite automata, the specific token gets recognized.

The transition diagram elaborates the actions to be taken while recognizing the token. The transition diagram is used to keep track of the information about characters that are seen as the forward pointer scans the input.

Construct of lexical Analyzer

L → Σ → R.E → NFA → DFA → min. DFA → code

Types of lexical Errors

i) Un terminated Comments

ii) Nested Comments

iii) Length of longest identifier (≤ 31)

iv) Invalid Identifier

v) Invalid Constant

vi) Illegal Symbols

Exercise Questions

1. Which of the following translation program converts assembly language programs to object program?

   a) Assembler b) Complier c) Macro processor d) Linker

2. The action of parsing the source program into the proper syntactic classes is known as :

   a) lexical Analysis b) Syntactic Analysis c) Semantic Analysis d) None

3. A compiler that runs on a machine and generates code for another machine is called

   a) Cross Compiler b) Incremental Compiler c) Booting Compiler

   d) None of execution

4. A loader is a program that

   a) Program that places programs into memory and prepares them for execution
   b) Program that automates the translation of assembly language into machine language
   c) Program that accepts a program written in a high level language and produces an object program
   d) Program that appears to execute a source program as if it were machine language

5. The output of a lexical Analyzer is

   (a) Machine Code (b) Intermediate Code

   (c) A Stream of tokens (d) a parse tree

6. In a compiler, grouping of characters into tokens is done by

(a) Lexical Analyzer (b) Syntax Analyzer (c) Code Generator

(d) Intermediate Code Optimizer

@ GATE PREVIOUS QUESTIONS

Q_1) In a compiler the module that checks every character of the source text is called

a. The code generator b. The code optimizer **[1987-1M]**

c. The lexical analyzer d. The syntax Analyzer

Q_2) A "Link editor" is a program that: **[1991-1M]**

a. Matches the parameters of the macro-definition with locations of the parameters of the macro cell

b. Matches external names of one program with their location in other program.

c. Acts as a link between test editor and the user

d. Acts as a link between compiler and user program

Q_3) Which of the following strings can definitely be said t be tokens without looking at the next input character while compiling a Pascal program? **[1995-1M]**

I) begin II) program III) < >

a. I b. II c. III d. All of the above.

Q_4) How many substrings of different lengths (non-zero) can be found formed from a character string of length n? **[1998-1M]**

a. n b. n^2 c. 2^n d. $\frac{n(n+1)}{2}$.

Q_5) The number of tokens in the FORTRAN statement DO 10 I = 1.25 is **[1999-1M]**

a. 3 b. 4 c. 5 d. None of the above

Q_6) The number of tokens in the following C statement **[2000-1M]**

printf (" i = %d, &i = %x" , i , & i);

a. 3 b. 26 c. 10 d. 21

Q_7) Which data structure in a compiler is used for managing information about variables and their attributes? **[2010 - 1M]**

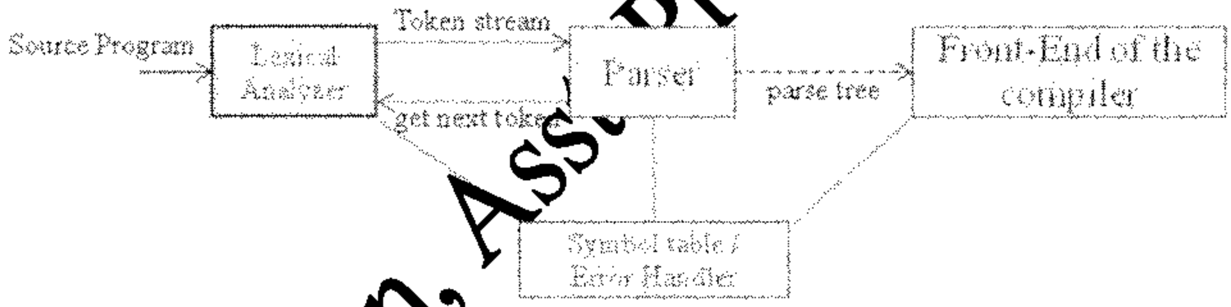
a) Abstract syntax tree b) Symbol table

c) Semantic stack d) Parse table

Chapter – II

Parsing

- A syntax Analyzer is also called as "Parser".
- The syntax Analyzer checks whether a given program satisfies the rules implied by a CFG (or) not. If it satisfies, the syntax Analyzer creates a parse tree for the given program.
- The syntax of a language is specified by a context-free grammar (CFG). The rules in the CFG are mostly recursive.



Position of parser in compiler model

Note: A parser is a program used to check whether the given input expression is in well-formed format (WFF) or not. If it is in WFF, parser generates parse tree else report error message.

- There are three general types of parsers fro grammars:

 i) Universal parsing method

 ii) Top – Down parser method

 iii) Bottom – Up method

 The methods commonly used in compilers are top-down and bottom-up parsers. In both cases, top-down and bottom-up parser the input is scanned from left to right, one symbol at a time.

@ Context – Free Grammar:

Many programming language constructs have an inherently recursive structure that can be defined by context – free grammars. For example, we might have a conditional statement defined by a rule such as

If S_1 and S_2 are statements and E is an expression, then **if E then S_1 else S_2** is a statement. Using the syntactic variables, we can express the above conditional statement by grammar production.

Statement → **if** expression **then** statement **else** statement

A grammar G = (V, T, P, S) is said to be content free if all productions in P of the form :

$A \rightarrow \beta$ where A is a single variable /non-terminal and $\beta \in (V \cup T)^*$

Where **V** is set of Variables, represented as lower-case letters, operator symbols, digits and punctuation symbols.

T is a set of Terminals, represented as Capital letters, derivation… etc.,

S is a start symbol.

- Non-Terminals define sets of strings that help define the language generated by the grammar. They also impose hierarchical structure on the language that is useful for both syntax analysis and translation.
- Context – free grammars derive their name from the fact that the substitution of the variable on the left of a production can be made any time such a variable appears in a sentential form. It does not depend on the symbols in the rest of the sentential form.

@ Derivation:

Derivation from S means generation of string w form S (or) deriving strings(w) in language from start symbols that is a production is treated as a rewriting rule in which the non-terminal on the left is replaced by the string on the right side of the production.

Ex: Let G be a content – free grammar for which the production rules given as below:

$S \rightarrow aSb$

$S \rightarrow ab$

Derive the string *w*: aaabbb using above grammar

Sol:

$S \rightarrow a\underline{S}b$ [$S\rightarrow aSb$]	// Choose arbitrary non-terminal in the RHS of
$\rightarrow aa\underline{S}bb$ [$S\rightarrow ab$]	// production to be replaced for generating string
$\rightarrow aaabbb$	

L(G) is the language generated by G is $\{a^n b^n / n \geq 1\}$

- Derivation of strings can be done in two ways:
  1. A derivation is said to be left most (LMD) if in each step the leftmost variable in the sentential form is replaced.
  2. If in each step the right most variable is replaced, we call the derivation as right most (RMD)

  Strings in a derivation is said to be sentential form

Ex: Consider the grammar with productions

$S \rightarrow aAB$
$A \rightarrow bBb$
$B \rightarrow A / \varepsilon$

Derive the string w : **abbbb** with LMD and RMD

Sol:

Left-Most Derivation(LMD)	Right-Most Derivation(RMD)
$S \rightarrow a\underline{A}B$ [$A\rightarrow bBb$]	$S \rightarrow aA\underline{B}$ [$B\rightarrow\varepsilon$]
$S \rightarrow ab\underline{B}bB$ [$B\rightarrow A$]	$S \rightarrow a\underline{A}$ [$A\rightarrow bBb$]
$S \rightarrow ab\underline{A}bB$ [$A \rightarrow bBb$]	$S \rightarrow ab\underline{B}b$ [$B\rightarrow A$]
$S \rightarrow abb\underline{B}bbB$ [$B \rightarrow \varepsilon$]	$S \rightarrow ab\underline{A}b$ [$A\rightarrow bBb$]
$S \rightarrow abbbb\underline{B}$ [$B\rightarrow\varepsilon$]	$S \rightarrow abb\underline{B}bb$ [$B\rightarrow\varepsilon$]
$S \rightarrow abbbb$	$S \rightarrow abbbb$

@ Derivation Tree (or) Parse Tree:

Its an graphical representation for a derivation.

1. Root is labeled by the start symbol.
2. Each interior node of a parse tree is labeled by some nonterminal.
3. Each leaves of the parse tree are labeled by nonterminals or terminals.
4. All leaves read from left – to – right, they constitute a sentential form.

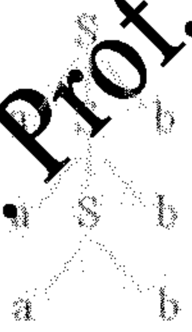
Ex: $S \rightarrow aSb$

$S \rightarrow ab$

Derivation tree/parse tree for the string "aaabbb" is

Derivation **Derivation Tree/ Parse Tree**

$S \rightarrow aSb$ [$S \rightarrow aSb$]
$S \rightarrow aaSbb$ [$S \rightarrow ab$]
$S \rightarrow aaabbb$



Example: Consider the grammar given below

$E \rightarrow E+E|E-E|E*E|E/E|a/b$

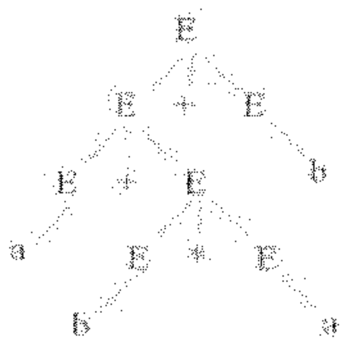
Obtain leftmost and rightmost desiccation for the string "a+b*a+b"

Sol:

LMD

$E \rightarrow E+E$ [$E \rightarrow E+E$]
$E \rightarrow E+E+E$ [$E \rightarrow a$]
$E \rightarrow a+E+E$ [$E \rightarrow E*E$]
$E \rightarrow a+E*E+E$[$E \rightarrow b$]
$E \rightarrow a+b*E+E$ [$E \rightarrow a$]
$E \rightarrow a+b*a+E$ [$E \rightarrow b$]
$E \rightarrow a+b*a+b$

parse tree



@ Ambiguous Grammar:

An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.

Find whether the following grammar is ambiguous or not

$S \rightarrow AB \,/\, aaB$
$A \rightarrow a \,/\, Aa$
$B \rightarrow b$

Sol:

LMD I:
$S \rightarrow \underline{A}B$ [$A \rightarrow Aa$]
$S \rightarrow \underline{A}aB$ [$A \rightarrow a$]
$S \rightarrow aa\underline{B}$ [$B \rightarrow b$]
$S \rightarrow aab$

LMD II:
$S \rightarrow aa\underline{B}$ [$B \rightarrow b$]
$S \rightarrow aab$

The given grammar is ambiguous grammar.

Types of Syntactic Errors:

1. Misspelling of tokens
2. Lack of Tokens
3. Extra Tokens

@ Practice Questions

Q) Find the Leftmost Derivation and Rightmost derivation for the following grammar
Drive the Leftmost, Right Derivation and parse tree for the following grammar G:

i) S → aAS / a
A → SbA / SS / ba   *w: aabbaa*

ii) S → aAb
S → bBA
A → ab / aAB
B → aB / b   *w*$_1$*: aaabbb, w*$_2$*: babab*

iii) S → aB / bA
A → a / aS / bAA
B → b / bS / aBB   *w: aaabbabbba*

Q) Check whether the grammar G is ambiguous or not?

i. stmt → *if* expr *then* stmt   *w: if* E$_1$ *then if* E$_2$ *then* S$_1$ *else* S$_2$
→ *if* expr *then* stmt *else* stmt
→ others

ii. S → AB / aaB
A → a / Aa
B → b

iii. E → EAE
A → + / - / × / /
E → a / b

Q) Consider the context-free grammar

E → E + E

E → (E × E)

E → id

Where E is the starting symbol, the set of terminals is {id, (, +,), *}, and the set of non-terminals is {E}.

a) Which of the following terminal strings has more than one parse tree when parsed according to the above grammar?

i) id + id + id + id   ii) id + (id * (id * id))

iii) (id * (id * id)) + id   iv) ((id * id + id) * id)

b) For the terminal string with more than one parse tree obtained as solution to Question a, how many parse tree are possible?

i) 5 ii) 4 iii) 3 iv) 2

Q) A CFG G is given with the following productions where S is the start symbol, A is a non-terminal and a, b are the terminals.

$S \rightarrow aS \,/\, A$

$A \rightarrow aAb \,/\, bAa \,/\, \varepsilon$

a) Which of the following strings is generated by the above grammar ?

i) aabbaba ii) aabaaba iii) abababb iv) aabbaab

b) For the correct answer in Q.a how many steps are required to drive the string and how many parse tree are there?

i) 6 and 1 b) 6 and 2 c) 7 and 2 d) 4 and 2

@ GATE PREVIOUS QUESTIONS

Q) Consider the following C program **[2005-2M]**

```
int main( ) {
int i,n;
fro(i=0;i<n;i++)
}
```

Identify the compiler response about these lines while creating the object module

(i) No Compilation error (ii) Only a Lexical error

(iii) Only syntactic error (iv) both lexical and syntactic error

Q) Consider the CFG with {S,A,B} as the non-terminals alphabet, {a , b} as the terminal alphabets as the start symbol and the following set of productions are: **[2007-2M]**

$S \rightarrow bA$ $S \rightarrow aB$
$A \rightarrow a$ $B \rightarrow b$
$A \rightarrow aS$ $B \rightarrow bS$
$A \rightarrow bAA$ $B \rightarrow aBB$

(a) Which of the following string is generated by the grammar ?

(i) aaaabb (ii) aabbbb (iii) aabbab (iv) abbbba

(b) for the correct answer string to question (a) how many derivation trees are there ?

(i) 1 (ii) 2 (iii) 3 (iv) 4

Parsing Methods

Parser is a program used to check whether the given input expression is in WFF (satisfies the rules implied by the CFG) or not. If it does generates a parse tree.

Note: In both TDP and BUP, the input of the parser is scanned from left – to – right, one symbol at a time.

- They are no. of task that might be conducted during parsing.
    i. Collecting information about various tokens from symbol table
    ii. Performing type checking
    iii. Other kinds of semantic analysis.

Differences between Top – Down and Bottom- Up Parsers

Top – Down Parser(TDP)		Bottom – Up Parsers(BUP)	
1	It builds parse tree from the top(root) to the bottom(leaves)	1	It builds parse tree from the bottom (leaves) to top(root).
2	It is used for Production/Derivations	2	It is used for Reduction
3	It follows Recursive – Descent Parsing Techniques	3	It follows Shift – Reduction Parsing Techniques
4	TDP Has Classified into Two Types a) With Backtracking b) Without Using Backtracking /Predictive Parsing	4	Shift – Reduce Parsing Classified into Two Types a) Operator Precedence Parser b) Efficient LR Parser i. Sample LR(SLR) ii. Canonical LR(CLR) iii. Look Ahead LR(CALR)
5	It follows Left – Most Derivation	5	It Follows Reverse Inverse Of Right - Most Derivation
6	It accepts only unambiguous types of grammars	6	It accepts both ambiguous and unambiguous types of grammar.

A general form of top-down parsing, called recursive descent, that may involve backtracking, that is making repeated scans of the input. However, backtracking parsers are not seen frequently. The special case of recursive – descent parsing, called predictive parsing, where no backtracking is required.

Difficulties with Recursive – Descent Parser are:

i. Left – Recursion
ii. Backtracking

i) Left – Recursion: A grammar G is said to be left – recursive if it has a non-terminal A such that there is a derivation A ⇒ Aα, for some string α.

- Note: A left – recursive grammar can cause a recursive – descent parser, even one with backtracking, to go into an infinite loop.

ii) Backtracking: If we make a sequence of incorrect expansions and subsequently discover a mismatch. We may have to undo all these incorrect expansions.

Ex: $\begin{array}{l} S \to cAd \\ A \to ab / b \end{array}$ derive the string w: ***cabd***

I $S \rightarrow c\underline{A}d$ [$A \rightarrow b$] **II** $S \rightarrow c\underline{A}d$ [$A \rightarrow ab$]
$S \rightarrow cbd$ $S \rightarrow cabd$
w^1: $cbd \neq w$ w^{11}: $cabd = w$

In derivation **I** , the obtained string(w^1) is not equivalent to the desired string (w). where as in derivation **II** , undo in correct expansions as [$A \rightarrow ab$] subsequently obtain the desired string as (w^{11} is equivalent to w)

1. Eliminate Left – Recursion:

The Left-Recursion is of two forms:

i) Immediate Left Recursion

ii) General Left Recursion

i) The Immediate Left-Recursion of the form is shown below:

$A \rightarrow A\alpha / \beta$

The productions after elimination of the left – recursion is:

$A \rightarrow \beta A^1$

$A^1 \rightarrow \alpha A^1 / \varepsilon$

Ex:

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$ (After eliminating left recursion)

$F \rightarrow id$

The production $E \rightarrow E+T / T$ is in the form of $A \rightarrow A\alpha / \beta$, where A is E , α is $\underline{+T}$ and β is T

After rewriting in the production rules

$E \rightarrow T E^1$

$E^1 \rightarrow +TE^1 / \varepsilon$

And the remaining productions after eliminating the left recursion the productions are

$T \rightarrow F T^1$

$T^1 \rightarrow *F T^1 / \varepsilon$

$F \rightarrow id$

The rule $A \rightarrow A\alpha_1 / A\alpha_2 / \ldots\ldots./A\alpha_m / \beta_1 / \beta_2 /\ldots/ \beta_n$ can be modified as,

$$\boxed{\begin{array}{l} A \rightarrow \beta_1A^1 / \beta_2A^1 /\ldots\ldots/ \beta_nA^1 \\ A^1 \rightarrow \alpha_1A^1 / \alpha_2A^1 /\ldots\ldots/ \alpha_nA^1 / \varepsilon \end{array}}$$

ii. General Left-Recursion:

Even if there may be no immediate left recursion, a number of production rules may act together to give a general left recursion.

Ex: $S \rightarrow Aa / b$
$A \rightarrow Ac / Sd / \varepsilon$

Here, S is left recursive, because $S \rightarrow Aa \rightarrow Sda$. The rule $S \rightarrow Aa / b$ has no immediate left recursion. The production $A \rightarrow Ac / Sd / \varepsilon$ is modified as $A \rightarrow Ac / Aad / bd / \varepsilon$, which has immediate left recursion and left recursion is eliminated as :

$A \rightarrow dbA^1 / A^1$
$A^1 \rightarrow adA^1 / cA^1 / \varepsilon$

2. Elimination of Left factoring

The left factoring is of the form: $A \rightarrow \alpha\beta_1 / \alpha\beta_2$

After elimination of left factoring the productions are:

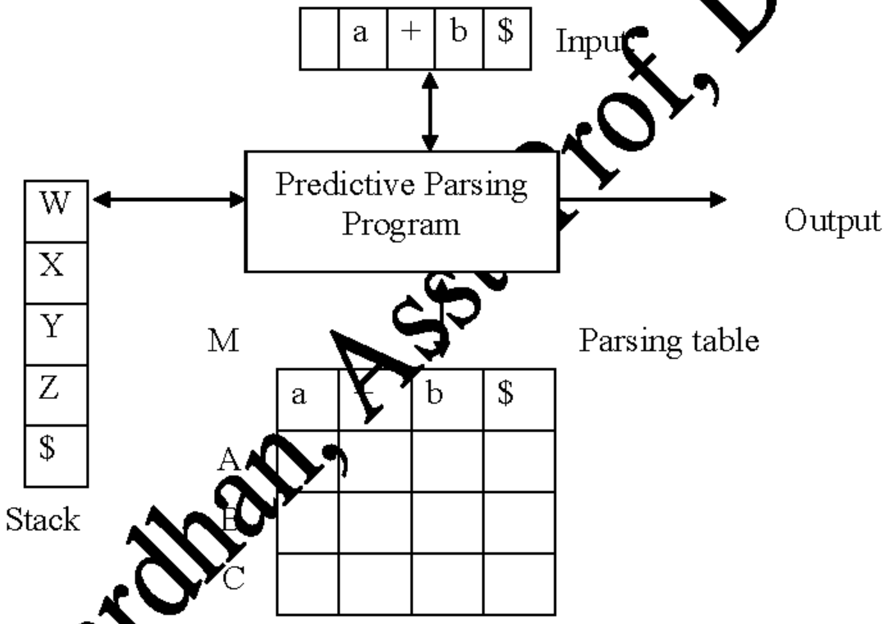
$A \rightarrow \alpha A^1$

$A^1 \rightarrow \beta_1 / \beta_2$

Ex: $S \rightarrow iEtS$ After elimination of left factoring the productions are
$S \rightarrow iEtSeS / a$ **$S \rightarrow iEtSS^1 / a$**
$E \rightarrow b$ **$S^1 \rightarrow eS / \varepsilon$, $E \rightarrow b$**

@ <u>Construction of LL(K) Parser</u>

- The first 'L' in LL(K) is stands for scanning the input from left – to – right and second 'L' stands for left – most Derivation.
- Where 'k' stands for lookahead value.
- Look ahead is a buffer which can hold zero or more numbers of symbols (tokens) ahead.
- - A Grammar whose parsing table has no multiple entries is said to be LL(1) Grammar.
- The grammar which is ambiguous and left recursion is not an LL(1) grammar.



Model of LL(1) parser.

The table-driven predictive parser has an:

1) Input Buffer
2) A Stack
3) A Parsing Table
4) Output Stream

- The input buffer contains the strings 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 nonterminal, and **a** is a terminal (or) the symbol \$.

- The parser program consider **X**, the symbol on the top of the stack, and **a**, the current input symbol. These two symbols determine the action of the parser.

There are three possibilities:

1) If X=a=\$ (if the top of the stack and current i/p symbol in \$) the parser halts and announces successful completion of parsing.
2) If X=a=\$ the parser pops X off and the stack and advances input pointer to the next input symbol.
3) If X is a nonterminal, 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.

→

Ex: If M[X,a] = {X → UVW}, the parser replaces X on top of the stack by WVU (with U on top). If M[X,a] = error, the parser calls the error recovery routine.

<u>FIRST and Follow</u>

The construction of predictive parser is aided by two functions associated with a grammar G. The functions FIRST and FOLLOW allows us to fill in the entries of a predictive parsing table for G.

<u>FIRST Function</u>

Let A → α, then FIRST(α) is a set of beginning terminal symbols of α .

Apply the following rules until on more terminals or ε can be added to any FIRST set:

1) If $X \rightarrow \varepsilon$ is an production, then add ε to FIRST(X) i.e. FIRST(X) = {ε}

2) If $X \rightarrow a$, then FIRST(X) = {a}, where 'a' is terminal

3) If $X \rightarrow a\alpha$, then FIRST(X) = {a}

4) If $X \rightarrow A\alpha$, then FIRST(X) = FIRST(A).

5) If $X \rightarrow A\alpha$ and $A \overset{*}{\Rightarrow} \varepsilon$, then FIRST(X) = {FIRST(A) - ε} $\cup$ FIRST(α)

6) If $X \rightarrow Y_1 Y_2 \ldots\ldots Y_k$ a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_1) ---- FIRST(Y_{i-1})

i.e., FIRST(X) = {FIRST(Y_1) $\cup$ FIRST(Y_2)------ FIRST(Y_k)}

FOLLOW ( ) function

FOLLOW(A) is defined as the set of terminal symbols that appear immediately to the right of A in some sentential form.

i.e., FOLLOW(A) = {a}.

There exists a derivation of the form $S \Rightarrow \alpha A a \beta$, where α and β are some grammar symbols.

The rules for computing FOLLOW function are:

1) FOLLOW(S) = {\$}, where S is start symbol.

2) If $A \rightarrow \alpha B \beta$, then FOLLOW(B) = FIRST(β)

3) If $A \rightarrow \alpha B$, then FOLLOW(B) contains FOLLOW(A)

4) If $A \rightarrow \alpha B \beta$, $\beta \overset{*}{\Rightarrow} \varepsilon$, then FOLLOW(B) ={ FIRST(β) - ε} $\cup$FOLLOW(A).

Ex: Construct FIRST and FOLLOW factions for the given grammars

1. E → E + T / T

T → T * F / F

F → (E) / id

Sol:

FIRST( )

- The production E → E+T , is of the form $X \rightarrow A\alpha$, then FIRST(X)=FIRST(A) //Rule4

Where X is E, A is E and α is +T.

FISRT(E)=FIRST(E) // In this production we are unable to derive terminal in FIRST(E)

- The production E → T is follows Rule$_4$.

  FIRST(E)=FIRST(T) =

- The production T→T*F / F , Rule$_4$ is applicable to this production.

  FIRST(T)=FISRT(F) =

- The production F → (E) is of the form X → aα, then FIRST(X)={a} //R$_3$

  Where **X** is F , **a** is (and **α** is E)

  FIRST(F)={ **(** }.

- The production F → id is of the form X → a, then FIRST(X)={a} //R$_2$

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

• FISRT(F)={ **(, id** }

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

FOLLOW( )

1. FOLLOW(E) = { \$ } // Where E is the start symbol

The production E → E + T , is of the form A → αBβ, then FOLLOW(B)=FIRST(β)

Where A is E, α is ε, B is E and + T is β. //R$_2$

FOLLOW(E)=FIRST(+T)= { **+** }

▪ The production F → (E), is follows R$_2$, FOLLOW(E)={ **)** }

FOLLOW(E)={ +,) , \$ }

2. FOLLOW(T)

• The production E → E + T is of the form A → αB, then FOLLOW(B)=FOLLOW(A)

  Where A is E, α is E + and B is T.

FOLLOW(T) = FOLLOW(E) = { + ,) , $ }

- The production E → T is follows R_3, where **α** is **ε** and **β** is **T**.

FOLLOW(T) = FOLLOW(E) = { + ,) , $ }

- The production T → T * F is of the form R_1.

FOLLOW(T) = FIRST(* F) = { * }

FOLLOW(T)={ + , * ,) , $ }

3. FOLLOW(F)

- The production rule T → T * F / F is of the form R_3.

FOLLOW(F) = FOLLOW(T)

3. After eliminating left recursion, the productions are:

$E \rightarrow TE^1$
$E^1 \rightarrow +TE^1 / \in$
$T \rightarrow FT^1$
$T^1 \rightarrow *FT^1 / \in$
$F \rightarrow (E) / id$

FIRST and FOLLOW function for above Grammar is

- **FIRST( )**
- $E \rightarrow TE^1$

  FIRST(E) = FIRST(T) // Rule(4)
- $T \rightarrow FT^1$

  FIRST(T) = FIRST(F) // Rule (4)
- $F \rightarrow (E) / id$

  FIRST(F) = FIRST(**(E)**) ∪ FIRST(id) Rule (6)

  = {**(** } ∪ {id}

  = {**(, id**}

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

- $E^1 \rightarrow +TE^1 / \in$

  FIRST(E^1) = FIRST($+TE^1$) ∪ FIRST(∈) = { + , ε }

- $T^1 \rightarrow *FT^1 / \in$

  FIRST(T^1) = FIRST($*FT^1$)$\cup$FIRST($\in$)

  $\boxed{\text{First}(T^1) = \{*, \in\}}$

- **FOLLOW(E)**

$\boxed{\text{Follow(E)} = \{\$\}}$

- FOLLOW(E^1)

$E \to TE^1 \Rightarrow$ $\boxed{\text{FOLLOW}(E^1) = \text{FOLLOW}(E) = \{\$\}}$ // Rule (3)

$E^1 \to +TE^1 \Rightarrow$

$\boxed{\text{FOLLOW}(E^1) = \text{FOLLOW}(E^1)}$ // Rule (3)

- FOLLOW(T)

$E \to TE^1 \Rightarrow$	FOLLOW(T)={ FIRST(E^1) – ε } ∪ FOLLOW(E)
	 = {+, $}
	 FOLLOW(T) = {+, $}

$E^1 \to +TE^1 \Rightarrow$	Follow(T) = { FIRST (E^1) - ε } ∪ FOLLOW(E^1)
	 ={+, $}

FOLLOW(F)

- $T \to FT^1$

FOLLOW(F) = { FIRST (T^1) – ε } ∪ FOLLOW(T)
 = {(*,∈)-∈} ∪ {+ , $ } = { +,* , $ }
	// Rule(4)

- $T^1 \to *FT^1$

FOLLOW(F) = { FIRST(T^1)-∈ } ∪ FOLLOW(T^1) = {+,*, $}	// Rule(4)

FOLLOW(T^1)

- $T \to FT^1$

FOLLOW (T^1) = FOLLOW(T) = {+, $}

- $T^1 \to *FT^1 / \in$

FOLLOW (T^1) = FOLLOW(T^1) = {+, $}

@ Construction of a predictive parsing table

1. For each production A→α of the grammar, do step2 and step3
2. For each terminal **a** is FIRST(α), add A→α to M[A, a]
3. If ε is in FIRST(α), add A → α to M[A, b] for each terminal **b** in FOLLOW(A)

   If ε is in FIRST(α) and $ is in FOLLOW(A), add A → α to M[A, $]
4. Make each undefined entry of M be error.

@ LL(1) Grammar

Q1) Check whether the following grammar is LL(1) or not

$S \rightarrow iEtS$
$S \rightarrow iEtSeS \,/\, a$
$E \rightarrow b$

Sol:

Check whether the given grammar contains left recursion, and left factoring. If so, eliminate it.

After eliminating left factoring it becomes

$S \rightarrow iEtSS^1 \,/\, a$
$S^1 \rightarrow eS \,/\, \epsilon$
$E \rightarrow b$

$$\boxed{\begin{array}{l} A \rightarrow \alpha \;\; \beta_1 \\ A^1 \rightarrow \beta_1 \,/\, \beta_2 \end{array}}$$

Construct FIRST and FOLLOW function for filling the predictive parser table.

- FIRST(S) = FIRST($\underline{i}EtSS^1$) // $S \rightarrow iEtSS^1$

  = {i} // Rule 3
- FIRST(S) = FIRST(a) // $S \rightarrow a$

  = {a} // Rule 2

  FIRST(S) = {i, a} // Rule 6

- FIRST(S^1) = FIRST(eS) // $S^1 \rightarrow eS$

  ={e} // Rule 3
- FIRST(S^1) = FIRST($\in$) // $S^1 \rightarrow \varepsilon$

  = {$\in$} // Rule 1

  FIRST(S) = {e , $\in$}

- FIRST(E) = FIRST(b) // $E \to b$

  FIRST(E) = {b} // Rule 2

<u>FOLLOW(S)</u>

1. $S \to iEt\underline{S}S^1$ $\boxed{\text{FOLLOW}(S) = \{\$\}}$

FOLLOW(S) = {FIRST (S^1) - ε} ∪ FOLLOW(S) //Apply Rule (4)

= {(e, ε) - ε} ∪ { $ }

= {e , $} $\boxed{\text{FOLLOW}(S) = \{e, \$\}}$

2. $S^1 \to e\underline{S}$

FOLLOW(S) = FOLLOW (S^1) // Rule (3)

= {e, $} // from FOLLOW($S^1$)

<u>FOLLOW (S^1)</u>

1. $S \to iEtS\underline{S^1}$

FOLLOW(S^1) = FOLLOW(S) // Rule 3

$\boxed{\text{FOLLOW}(S^1) = \{e, \$\}}$

<u>FOLLOW (E)</u>

1.$S \to i\underline{E}tSS^1$

FOLLOW (E) = FIRST(tSS^1) // Rule 2

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

FIRST( ) and FOLLOW( ) for grammar 1.2 is

Non – Terminals	FIRST (X)	FOLLOW(X)
S	{i, a}	{e, $}
S^1	{e, ∈}	{e, $}
E	{b}	{t}

Construction of predictive parser table

1. $S \to iEtSS^1$, then M[S, FIRST($iEtSS^1$)] = M [S , i] add $S \to iEtSS^1$

2. $S \to a$,then M[S, FIRST(a)] = M [S , a] add $S \to A$

3. $S^1 \to eS$, then M[S^1, FIRST(eS)] = M [S^1 , e] add $S^1 \to eS$

4. $S^1 \to \in$, then M[S^1, FOLLOW(S^1)] = M [S^1 , {e,$}] add $S^1 \to \varepsilon$

5. $E \to b$, then M[E, FIRST(b)] = M [E, b] add $E \to b$

Parsing table M

Non-terminal	Input symbols					
	i	t	a	e	b	$
S	$S \to iEtSS^1$		$S \to a$			
S^1				$S^1 \to eS$ $S^1 \to \epsilon$		$S^1 \to \in$
E					$E \to b$	

The given grammar is not LL(1) because the M[S^1,e] is having multiple entries.

@ **Practice Questions:**

Q) Eliminate Left Recursion in the following grammar G:

i) $S \rightarrow Aab$
$A \rightarrow AC$
$\rightarrow Sd / \varepsilon$

ii) $S \rightarrow L + L / L$
$L \rightarrow LB / B$
$B \rightarrow 0 / 1$

iii) $S \rightarrow Aa / b$
$A \rightarrow Ac / Sd / \varepsilon$

iv) $S \rightarrow aAcBe$
$A \rightarrow Ab / b$
$B \rightarrow d$

Q) Eliminate Left Factoring in the following grammar G:

i) $S \rightarrow aAd / aB$
$A \rightarrow b / c$
$B \rightarrow cba / cbe$

ii) $B \rightarrow Abc / c$
$A \rightarrow aC / abc$

iii) $S \rightarrow abA$
$A \rightarrow cd / c / \varepsilon$

Q) Compute the FIRST( ) and FOLLOW( ) for the following grammar G:

a) $S \rightarrow iEtS / a$ $S \rightarrow iEtSeS$ $E \rightarrow b$	b) $S \rightarrow Ad$ $A \rightarrow x / y / \varepsilon$	c) $S \rightarrow 1AB / \varepsilon$ $A \rightarrow 1AC / 0C$ $B \rightarrow 0S,\ C \rightarrow 1$	d) $E \rightarrow E + T / T$ $T \rightarrow T \times F / F$ $F \rightarrow id$

Q) Check whether the following grammar G is LL(1) or not?

i) $S \rightarrow iEtSS^1 / a$
$S^1 \rightarrow eS / \varepsilon$
$E \rightarrow b$

ii) $S \rightarrow ACB / CdB / Ba$
$A \rightarrow da / BC,\ B \rightarrow g / \varepsilon$
$C \rightarrow b / \varepsilon$

iii) $S \rightarrow AaAb / BbBa$
$A \rightarrow \varepsilon\ ,\ B \rightarrow \varepsilon$

iv) $S \rightarrow aAd / bBd / aBe / bAe\ ,\ A \rightarrow c\ ,\ B \rightarrow c$

v) $S \rightarrow ABC,\ A \rightarrow a / \varepsilon,\ B \rightarrow b / \varepsilon,\ C \rightarrow c / \varepsilon$

@ GATE PREVIOUS QUESTIONS

Q) Consider the following grammar for arithmetic expression: [1991 – 2M]

$E \rightarrow E - T \;/\; T$
$T \rightarrow T * F \;/\; F$
$F \rightarrow (E) \;/\; id$

I. a. Is Ambiguous
b. Is Unambiguous
c. Information is not sufficient
d. Left factoring

II. After eliminating the left-factoring from the above grammar is:
a. $E \rightarrow TE^1, E^1 \rightarrow - TE^1 / \varepsilon, T \rightarrow FT^1, T^1 \rightarrow *FT^1 / \varepsilon, F \rightarrow (E) / id$
b. $E \rightarrow TE, E^1 \rightarrow - TE^1 / \varepsilon, T \rightarrow FT, T^1 \rightarrow *FT^1 / \varepsilon, F \rightarrow (E) / id$
c. $E \rightarrow TE, E \rightarrow - TE / \varepsilon, T \rightarrow FT, T \rightarrow *FT / \varepsilon, F \rightarrow (E) / id$
d. None of these

Q) For a context-free grammar, FOLLOW(A) is the set of terminals that can appear immediately to the right of non-terminal A in some "sentential" form. We define two sets LFOLLOW(A) and RFOLLOW(A) by replacing the word "sentential" by "left sentential" and "right sentential" respectively in the definition of FOLLOW(A).
Which of the following statements is/are true? [1992 – 1M]
a. FOLLOW(A) and FOLLOW(A) may be different
b. FOLLOW(A) and FOLLOW(A) are always the same
c. All the three sets are identical
d. All the three sets are different

Q) The grammar whose productions are [1996 – 1M]

‹stmt› → *if* id *then* ‹stmt›
→ *if* id *then* ‹stmt› *else* ‹stmt ›
→ id := id

is ambiguous because
(a) The sentence ***if* a *then if* b *then* c:= d**
(b) The left most and right most derivations of the sentence ***if* a *then if* b *then* c:= d *else* c := f** give rise two different parse trees
(c) The sentence ***if* a *then if* b *then* c:= d *else* c := f** has more than two parse trees
(d) The sentence ***if* a *then if* b *then* c:= d *else* c := f** has two parse trees

Q) Type checking is normally done during [1998 – 1M]
(a) Lexical analysis (b) Syntax analysis
(c) Syntax directed translation (d) Code optimization

Q) Consider the grammar

S → Aa / b

A → Ac / Sd / ε

An equivalent grammar with no left recursion and with minimum number of production rules are:

(a) S → Aa / b , A → Ac / Aad / bd / ε

(b) S → Aa / b , A → dbA^1 / A^1 , A^1 → cA^1 / adA^1 / ε

(c) S → Aa / b , A → dbA / A^1 , A^1 → cA^1 / adA^1 / ε

(d) None of these

Q) A grammar that is both left and right recursive for a non-terminal, is [1999-1M]

(a) Ambiguous (b) Unambiguous

(c) Information is not sufficient to decide whether it is ambiguous or unambiguous

(d) None of the above

Q) Consider the following grammar with terminal alphabet Σ {a, (,) , +, *} and start symbol E.

The production rules of the grammar are: [2001-5M]

E → aA

E → (E)

A→ +E

A→ *E

A→ ε

(a) Compute the FIRST and FOLLOW sets for E and A

(b) Compute the LL(1) parse table for the grammar

Q) (a) Remove left-recursion from the following grammar: S → Sa / Sb / a / b [2001-5M]

(b) Consider the following grammar: S → aSbS / bSaS / ε

Construct all possible parse tree for the string *abab* . Is the grammar ambiguous?

Q) Which of the following is a Top-Down Parser ? [2007-2M]

(i) Recursive Descent Parser (ii) Operator-Precedence Parser

(iii) An LR(k) parser (iv) An LALR(k) parser.

Q) Which of the following suffices to convert an arbitrary CFG to an LL(1) grammar ?

i) Removing left-recursive alone [2003-1M]

ii) Factoring the grammar alone

iii) Removing left-recursion and factoring the grammar

iv) None of the above.

Q) The grammar A→AA / (A) / ε is not suitable for predictive parsing because the grammar is: [2005-1M]

a) Ambiguous b) Un ambiguous c) Left-recursive d) Right –recursive

Q) Consider the grammar shown below [2003-2M]

$S \rightarrow iEtSS^1 / \alpha$

$S^1 \rightarrow eS / \varepsilon$

$E \rightarrow b$

In the predictive parser table M, of this grammar, the entries $M[S^1,e]$ and $M[S^1, \$]$ respectively are:

(a) $\{S^1 \rightarrow eS\}$ and $\{S^1 \rightarrow \varepsilon\}$ (b) $\{S^1 \rightarrow eS\}$ and $\{\ \}$

(c) $\{S^1 \rightarrow \varepsilon\}$ and $\{S^1 \rightarrow \varepsilon\}$ (d) $\{S^1 \rightarrow eS, S^1 \rightarrow \varepsilon\}$ and $\{S^1 \rightarrow \varepsilon\}$

Q) Consider the grammar with non-terminals $N = \{S, C, S_1\}$, terminals [2007-2M]
$T = \{a, b, i, t, e\}$,with S as the start symbol, and the following set of rules:

$S \rightarrow iCtSS_1 / a$

$S_1 \rightarrow eS / \varepsilon$

$C \rightarrow b$

The grammar is *not* LL(1) because:

(a) It is left recursive (b) It is Right Recursive

(c) It is Ambiguous (d) It is not Context-Free

Q) Consider the following grammar [2006-2M]

$S \rightarrow FR$

$R \rightarrow \times S / \varepsilon$

$F \rightarrow id$

In the predicate parse table M, of the grammar the entries M[S , id] and M[R , \$] respectively

(a) $\{S \rightarrow FR\}$ and $\{R \rightarrow \varepsilon\}$ (b) $\{S \rightarrow FR\}$ and $\{\ \}$

(c) $\{S \rightarrow FR\}$ and $\{R \rightarrow \times S\}$ (d) $\{F \rightarrow id\}$ and $\{R \rightarrow \varepsilon\}$

Bottom – Up Parsing

General styles of bottom – up syntax analysis, known as shift – reduce parsing.

Shit – reduce parsing attempts construct a parse tree for a input string beginning at he leaves (the bottom) and working up towards the root (top).

The process of "reducing" a string *w* to the start symbol of a grammar. At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left of that production, and if the substring is chosen correctly each step, a rightmost derivation is traced out in render.

Ex: Consider the grammar,

$S \rightarrow aABe$

$A \rightarrow Abc \,/\, b$

$B \rightarrow d$

Given the input string "abbcde", which has to be checked for its acceptance by the language.

a$\underline{\text{b}}$bcde [A → b]

a$\underline{\text{Abc}}$de [A → Abc]

aA$\underline{\text{d}}$e [B → d]

$\underline{\text{aABe}}$ [S → aAbe]

S

This process is similar to the right – most derivation in reverse.

S → aA$\underline{\text{B}}$e [B→d]

→ a$\underline{\text{A}}$de [A→Abc]

→ a$\underline{\text{A}}$bcde [A→b]

→ abbcde

Handle

A "handle" of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production represents one step of the reverse of a rightmost derivation.

If there is a production $A \rightarrow \beta$, than β is said to handle, since it can to reduce to A, in the string $\alpha\beta\omega$. Reducing β to A in $\alpha\beta\omega$ is said to be " pruning the handle ".

Ex: Consider the following grammar

$1. E \rightarrow E + E$
$2. E \rightarrow E * E$
$3. E \rightarrow (E)$
$4. E \rightarrow id$

The string **id + id * id** is reduced to start symbol E

$\underline{\text{id}}$ + id * id [E→id]

E + $\underline{\text{id}}$ * id [E→id]

$\underline{\text{E + E}}$ * id [E→E+E]

E * $\underline{\text{id}}$ [E →id]

$\underline{\text{E * E}}$ [E →E*E]

E

Note:

1. If a grammar is unambiguous, then every right – sentential form of the grammar has exactly one handle.
2. The string appearing to the right of handle contains only terminal symbols
3. A rightmost derivation in reverse can be obtained by "handle pruning". That is, we start with a string of terminals w and replace the handle with non-terminal. We repeat this procedure until start symbol.

@ Stack Implementation of Shift – Reduce Parsing:

There are two problems that must be solved if we are to parse by handling pruning.

First, to locate the substring to be reduced in a right – sentential form.

Second, to determine what productions to choose in case that is more than one production with the substring on the right side

@ A convenient way to implement a shift – reduce parsers is to uses:

- ❖ A stack, to hold grammar symbols
- ❖ An input buffer, to hold the string to be parsed.

- We use $ to mark the bottom of the stack and also the right end of the input.
- Initially, the stack is empty, and the string *w* is on the input tape, as follows:

STACK	INPUT
$	w $

The parser operates by shifting zero or more input symbols onto the stack until a handle β is on top of the stack. The parser then reduces β to the left side of the appropriate production. The parser repeat this cycle until it has detected an error or until the stack contains the start symbol and input is empty:

STACK	INPUT
$S	$

EX:

$$\begin{aligned} E &\to E+E \\ &\to E*E \\ &\to id \end{aligned}$$

STACK	INPUT	ACTION
$	$id_1+id_2*id_3\$$	Shift id_1
$\$id_1$	$+id_2*id_3\$$	Reduce by $E \rightarrow id$
$E	$+id_2*id_3\$$	Shift +
$E+	$id_2*id_3\$$	Shift id_2
$\$E+id_2$	$*id_3\$$	Reduce by $E \rightarrow id$
$E+E	$*id_3\$$	Shift *
$E+E*	$id_3\$$	Shift id_3
$\$E+E*id_3$	$	Reduce by $E \rightarrow id$
$E+E*E	$	Reduce by $E \rightarrow E*E$
$E+E	$	Reduce by $E \rightarrow E+E$
$E	$	Accept

There are four possible actions of shift – reduce parser:

1. Shift
2. Reduce
3. Accept
4. Error

1. In a shift operation, the next input symbol is shifted onto the top of the stack[TOS].
2. In a reduce action, the parser knows the right end of the handle is at the TOS.
3. In an accept action, the parser announces successful completion of parsing.
4. In an error action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts During Shift – Reduce Parsing:

- There are context – free grammars for which shift-reduce parsing cannot be used. Even after knowing the entire stack contents and the next input symbol. Shift-reduce parser for such a grammar,
  - ❖ Cannot decide whether to shift (or) to reduce (a shift/reduce conflict)
  - ❖ Cannot decide which of several reductions to make (a reduce/reduce conflict)

Operator – Precedence Parsing

The operator grammar has the property

1. That no production right side is ε
2. Has no two adjacent nonterminals

Ex: $E \rightarrow EAE$ / (E) / - E/id

$A \rightarrow + / - / * /$

The above grammar is not a operator grammar, because the right side EAE has two consecutive nonterminals.

However, if we substitute for *A* each of its alternative, it will become operator grammar

$E \rightarrow E+E$ / E-E / E*E / E/E

This technique was described as a manipulation on tokens without any references to an underlying grammar.

❖ The operator precedence parser has number of disadvantages.

Ex: It is hard to handle tokens like the minus sign, which has two different precedence's.

❖ Only a small class of grammars can be parsed.

Often these operators – precedence parsers use recursive – descent for statements and high – level constructs.

In operator – precedence parsing, there are three disjoint precedence relations, between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings:

Relation	Meaning
$a \prec \cdot b$	a "yields precedence to" b
$a \doteq b$	a " has the same precedence as" b
$a \cdot\succ b$	a " takes precedence over" b

These are two common ways to determining what precedence relations should hold between a pair of terminals.

❖ First Method, is based on the traditional notions of associativity and precedence of operators. This approach will resolve the ambiguity's of grammar.
❖ Second Method, is to first construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.

First method - Using Operator – Precedence Parser

The intention of the precedence relations is delimit the handle of a right-sentential form. We must have exactly one relations $\lessdot, \doteq$ *and* $\gtrdot$ between a_i and a_{i+1} and we use $ to mark end of the string, we remove the nonterminals from the string and place the current relation $\lessdot, \doteq$ *or* $\gtrdot$ between each pair of terminals and between the end terminals and the $ is marking the ends of the string.

Ex: The Right – Sentential form id + id * id and the precedence relation is shown below:

	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

Operator – precedence relations

The string with the precedence relations inserted is:

$$\$ \lessdot id \gtrdot + \lessdot id \gtrdot * \lessdot id \gtrdot \$$$

The handle can be found by the following process.

1. Scan the string from the left end until the first $\gtrdot$ is encountered.
2. Then scan back words (to the left) over any $\doteq$'s until a $\lessdot$ is encountered.
3. The handle contains everything to the left of the first $\gtrdot$ and to the right of the $\lessdot$.

LR Parsers

The Bottom – Up syntax analysis technique can be used to parse a large class of context free grammars. The techniques is called LR(k) parsing.

The "L" is for Left to Right scanning of the i/p, the "R" for constructing a rightmost derivation in reverse and '*k*' for the no. of input symbols of lookahead that are used in making parsing decision.

There are three techniques for constructing an LR parsing table for a grammar.
First method, called **simple LR**(SLR for short), is the easiest to implement.
Second method, called **canonical LR**(CLR,LR(1)) is the most powerful, and expensive.
Third method, called **lookahead LR**(LA LR), is intermediate in power and less cost than other two.

The LR parsing algorithm

The LR parser consists of input, an output, a stack, a driver program, and a parsing table that has two parts (Action and Goto).

- The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form $S_0 X_1 S_1 X_2$ --X_n S_n, where S_n is on top. Each X_i is a grammar symbol and S_i is a symbol called a state.

- The program driving the LR parser behaves as follows:

It determines S_m, the state currently on top of the stack, and a_i, the current input symbol. It then consults action[S_m, a_i], the parsing action table entry for state S_m and a_i where can have one of four values:

1. Shift S, where S is a state
2. Reduce by a grammar production $A \rightarrow \beta$
3. Accept, and
4. Error

- The next move of the parser is determined by reading a_i, the current input symbol, and S_m, the state on top of the stack, and then consulting the parsing action table entry action [S_m, a_i].
- The configuration resulting after each of the four types of move are as follows:

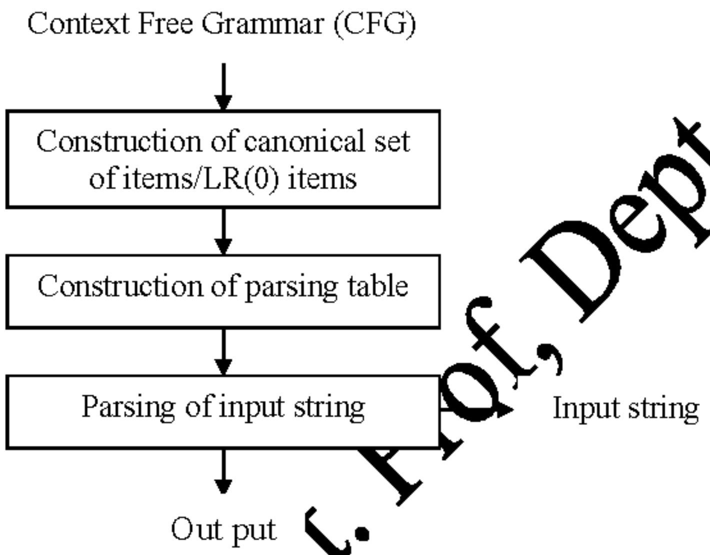
1. If action [S_m, a_i] = Shift s, the parser executes a shift move. Then the parser shifts the current input symbol a_i, and the next state S, which is given in action [S_m, a_i], onto the stack; a_{i+1} becomes the current input symbol.
2. If action [S_m, a_i] = reduce $A \rightarrow \beta$, then the parser executes a reduce move. The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reduced production.
3. If action [S_m, a_i] = accept, parsing is completed
4. If action [S_m, a_i] = error, the parser has discovered an error and calls an error recovery routine.

Types LR Parser

- The CLR Parser is most powerful parser among others (SLR, LALR and LL(i))
- The no. of states in SLR are equal to LALR parser

Constructing SLR Parsing Tables

- SLR method is very easy to implement and weakest in terms of the no. of grammars for which it succeeds.
- The parsing can be done as follows:



Context Free Grammar (CFG)
Construction of canonical set of items/LR(0) items
Construction of parsing table
Parsing of input string
Input string
Out put

Step 1

Augment Grammar

If G is a grammar with start symbol S, then G^1 the augment grammar for G, is G with a new start symbol S^1 and production $S^1 \rightarrow S$. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by $S^1 \rightarrow S$.

Step 2

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. Thus, production $A \rightarrow XYZ$ yields the four items

$$\left.\begin{array}{l} A \rightarrow .XYZ \\ A \rightarrow X.YZ \\ A \rightarrow XY.Z \end{array}\right\} \textit{Shift item}$$

$$A \rightarrow XYZ. - \text{Re}\,\textit{duce item}$$

The production $A \rightarrow \in$ generates only one item, $A \rightarrow .$

Step 3

The Closure Operation

It I is a set of items for a grammar G, then Closure (I) is the set of items constructed from I by the two rules:

1. Initially, every item in I is added to Closure(I)
2. If A→ α . Bβ is in closure (I) and $B \rightarrow \gamma$ is a production then add the item B→.γ to I, if it is not already there.

   Repeat this process until no more new items can be added to closure (I).

Step 4

Construct SLR(1) parsing table

I. If [S^1→S.] is in I_i ,then set Action[I_i , $]=Accept

II. If [A→ α . Bβ] is in I_i , then set Goto[I_i, B] = I_j

III. If [A→ α . aβ] is in I_i , then set Action[I_i , a] = Shift

IV. If [A→ α .] is in I_i , then set Action[I_i , Follow(A)] = Reduce $A \rightarrow \alpha$

Step 5

The given grammar G is said to be SLR(1). If the parsing table does not contain

. Shift /Reduce

(or)

. Reduce / Reduce, entries.

Q$_1$) Check whether the following grammar G is SLR(1)

$1. E \rightarrow E + T$

$2. E \rightarrow T$

$3. T \rightarrow T * F$

$4. F \rightarrow (E)$

$5. F \rightarrow id$

Sol:

Step 1:

$E^1 \rightarrow E$
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

Step 2:

$E^1 \rightarrow .E$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Step 3:

I$_0$:

$E^1 \rightarrow .E$	Goto[I_0 , E]=1
$E \rightarrow .E + T$	Goto[I_0 , E]=1
$E \rightarrow .T$	Goto[I_0 , T]=2
$T \rightarrow .T*F$	Goto[I_0 , T]=2
$T \rightarrow .F$	Goto[I_0 , F]=3
$F \rightarrow .(E)$	Action[I_0 , ( ]= S_4
$F \rightarrow .id$	Action[I_0 , id]=S_5

I$_1$:	$E^1 \rightarrow E.$	Action[I_1 , \$]=Accept
	$E \rightarrow E. + T$	Action[I_1 , +]=S_6

I_2:	$E \rightarrow T.$	Action[I_2 , Follow(E)]=Action[I_2 , {+,), \$}]=R_2
	$T \rightarrow T. * F$	Action[I_2 , *]=S_7
I_3:	$T \rightarrow F.$	Action[I_3, Follow(T)]=Action[I_3 , {+,*,),\$}]=R_4
I_4:	$F \rightarrow$ **(.E)**	Goto[I_4 , E]=8
	$E \rightarrow .E + T$	Goto[I_4 ,E]=8
	$E \rightarrow .T$	Goto[I_4 ,T]=2
	$T \rightarrow .T*F$	Goto[I_4 ,T]=2
	$T \rightarrow .F$	Goto[I_4 ,F]=3
	$F \rightarrow$ **.(E)**	Action[I_4 ,**(**]=S_4
	$F \rightarrow .id$	Action[I_4 ,id]=S_5
I_5:	$F \rightarrow id.$	Action[I_5, Follow(F)]=Action[I_5 , {+,*,),\$}]=R_4
I_6:	$E \rightarrow E + .T$	Goto[I_6 ,T]=9
	$T \rightarrow .T * F$	Goto[I_6 ,T]=9
	$T \rightarrow .F$	Goto[I_6 ,F]=3
	$F \rightarrow$ **. (E)**	Action[I_6 ,**(**]=S_4
	$F \rightarrow .id$	Action[I_6 ,id]=S_5
I_7:	$T \rightarrow T * .F$	Goto[I_7 , F]=10
	$F \rightarrow$ **. (E)**	Action[I_7 ,**(**]=S_4
	$F \rightarrow .id$	Action[I_7 ,id]=S_5
I_8:	$F \rightarrow$ **(E.)**	Action[I_8 , **)**]=S_{11}
	$E \rightarrow E .+ T$	Action[I_8, +] =S_6
I_9:	$E \rightarrow E + T.$	Action[I_9,Follow(E)]= Action[I_9 , {+,), \$}]=R_1
	$T \rightarrow T.* F$	Action[I_9, *]=S_7
I_{10}:	$T \rightarrow T * F.$	Action[I_{10}, Follow(T)]=Action[I_{10} , {+,*,),\$}]=R_3
I_{11}:	$F \rightarrow (E) .$	Action[I_{11}, Follow(F)]=Action[I_{11} , {+,*,),\$}]=R_5

First (E) = First (T) = First (F) = {(, id}

Follow (E) = {+,), \$}

Follow (T) = {+,*,), \$}

Follow (F) = {+,*,), \$}

Items	Action						Go to		
	+	*****	**(**	**)**	**$**	**id**	**E**	**T**	**F**
0			S_4			S_5	1	2	3
1					Accept		6		
2	R_2	S_7		R_2	R_2				
3	R_4	R_4		R_4	R_4				
4			S_4			S_5	8	2	3
5	R_6	R_6		R_6	R_6				
6			S_4			S_5		9	3
7			S_4			S_5			10
8	S_6			S_{11}					
9	R_1			R_1		R_1			
10	R_3	R_3		R_3	R_3				
11	R_5	R_5		R_5	R_5				

The given grammar is SLR, because the predictive parser table doesn't contain multiple define entries.

Q$_2$): Check whether the following grammar G is SLR?

1. S→ Aa
2. S → bAc
3. S → dc
4. S → bda
5. A →d

Solution:

Step 1: Augment Grammar

S^1→ S

S →Aa

S→bAc

S→dc

S→bda

A→ d

Step 2:

$S^1 \rightarrow .S$

$S \rightarrow .Aa$

$S \rightarrow .bAc$

$S \rightarrow .dc$

$S \rightarrow .bda$

$A \rightarrow .d$

Step 3:

I_0:	$S^1 \rightarrow .S$	Goto[I_0, 3] = 1
	$S \rightarrow .Aa$	Goto[I_0, A] = 2
	$S \rightarrow .bAc$	Action[I_0, b] = S_3
	$S \rightarrow .dc$	Action[I_0, d] = S_4
	$S \rightarrow .bda$	Action[I_0, b] = S_3
	$A \rightarrow .d$	Action[I_0, d] = S_4
I_1:	$S^1 \rightarrow S.$	Action[I_1, \$] = Accept
I_2:	$S \rightarrow A.a$	Aciton[I_2, a] = S_5
I_3:	$S \rightarrow b.Ac$	Goto[I_3, A] = 6
	$S \rightarrow b.da$	Action[I_3, d] = S_7
I_4:	$S \rightarrow d.c$	Action[I_4, c] = S_8
	$A \rightarrow d.$	Action[I_4, Follow(A)] = Action[I_4, {a,c}] = R_5
I_5:	$S \rightarrow Aa.$	Action[I_5, Follow (S)] = Action[I_5, {\$}] = R_1
I_6:	$S \rightarrow bA.c$	Action[I_6, c] = S_9
I_7:	$S \rightarrow bd.a$	Action[I_7, a] = S_{10}
	$A \rightarrow d.$	Action[I_7, Follow (A)] = Action[I_7, {a,c}] = R_7
I_8:	$S \rightarrow dc.$	Action[I_8, follow(S)] = Action[I_8, {\$}] = R_3
I_9:	$S \rightarrow bAc.$	Action[I_9, follow(S)] = Action[I_9, {\$}] = R_2
I_9:	$S \rightarrow bda.$	Action[I_{10}, follow(S)] = Action[I_{10}, {\$}] = R_4

- First(S) = {b,d}
- First(A) ={d}
- Follow(S) ={\$}
- Follow(A) = {a,c}

Item	Action					Go to	
	a	**b**	**c**	**d**	**\$**	**A**	**S**
0		S_3		S_4		2	1
1					Accept		
2	S_5						
3				S_7		6	
4	R_5		S_8, R_5				
5					R[illegible]		
6			S_9				
7	S_{10}, R_7		R_7				
8					R_3		
9					R_2		
10					R_4		

The give grammar is not SLR(1). Because the SLR(1) passing table has shift / Reduce conflict.

Q.3) Check whether the following grammar is SLR?

S → AaAb

S → BbBa

A → ε

B → ε

Solution:

Step 1: Augment Grammar

S^1 → S

S → Aa Ab

S → BbBa

A → ε

B → ε

Step 2: LR(0) items

$S^1 \rightarrow .S$

$S \rightarrow .AaAb$

$S \rightarrow .BbBa$

$A \rightarrow .$

$B \rightarrow .$

Step 3: Construct closure (I) Items

I_0:	$S^1 \rightarrow .S$	Goto[I_0, S] = 1
	$S \rightarrow .AaAb$	Goto[I_0, A] = 2
	$S \rightarrow .BbBa$	Goto[I_0, B] = 3
	$A \rightarrow .$	Aciton[I_0, Follow (A)] = Action [I_0, {a,b}] = R_3
	$B \rightarrow .$	Action[I_0, Follow (B)] = Action [I_0, {a,b}] = R_4
I_1:	$S^1 \rightarrow S.$	Action[I_1, \$] = Accept
I_2:	$S \rightarrow A.aAb$	Action[I_2, a] = S_4
I_3:	$S \rightarrow B.bBa$	Action[I_3, b] = S_5
I_4:	$S \rightarrow Aa.Ab$	Goto[I_4, A] = 6
	$A \rightarrow .$	Action[I_4, Follow (A)] = [I_4, {a,b}] = R_3
I_5:	$S \rightarrow Bb.Ba$	Goto[I_5, B] = 7
	$B \rightarrow .$	Action[I_5, follow (B)] = [I_5, {a,b}] = R_4
I_6:	$S \rightarrow AaA.b$	[I_6, b] = S_8
I_7:	$S \rightarrow BbB.a$	[I_7, a] = S_9
I_8:	$S \rightarrow AaAb.$	[I_8, follow(s)] = [I_8, {a,b}] = R_1
I_9:	$S \rightarrow BbBa.$	[I_9, follow(s)] = [I_9, {a,b}] = R_2

Follow(S) = {\$}

Follow (A) = {a,b}

Follow (B) = {a,b}

Step 4:

Items	Action			Go to		
	a	b	$	S	A	B
0	R_3, R_4	R_3, R_4		1	2	3
1			Accept			
2	S_4					
3		S_5			6	
4	R_3	R_3				7
5	R_4	R_4				
6		S_8				
7	S_9					
8	R_1	R_1				
9	R_2	R_2				

The given grammar G is not SLR.

Construction Canonical LR parsing table

The CLR /LR(1) parsing table similar to SLR parsing table comprised of action and go to parts.

Step 1: Construct Augment Grammar

Step 2: Find Closure (I)

The item ($A \rightarrow \alpha.B\beta$, a) and $B \rightarrow .\boldsymbol{\delta}$, then add [$B \rightarrow .\boldsymbol{\delta}$, First($\beta$a)] to I, if not in I.
Repeat the process until no more items can be added.

Step 3: Construct CLR parsing table.

The canonical LR parsing table functions action and go to for augment grammar.

State **I** of the parser is constructed from **I**. The parsing actions for State I are determined as follows:

a) If [$S^1 \rightarrow S.$,\$] is in I_i **,** then set Action [I_i , \$] = Accept

b) If [$A \rightarrow \alpha.B\beta$, a] is in I_i , then set Goto [I_i , A] = I_j

c) If [$A \rightarrow \alpha.b\beta$, a] is in I_i , then set Action[I_i , b] = Shift j. where b is terminal.

d) If [$A \rightarrow \alpha.$, a] is in I_i , then set Action[I_i, a] = $A \rightarrow \alpha$

e) If the entries are not defined, they made as "error".

Note:

- If the parsing action function has no multiply defined entries, then the grammar is called an LR (1) grammar. Every SLR (1) grammar is an LR (1) grammar.
- The canonical LR parser may have more state than the SLR parser for the same grammar.

Q_1) Consider the following grammar.

1. S → CC
3. C → cC
2. C → d

Construct the CLR parsing table for this grammar.

Solution:

Step 1: Augment Grammar

S^1 → S

S → CC

C → cC

C → d

Step 2: Closure Items

I_0: S^1 → .S,$

S → .CC, first (ε.$) = S → .CC, first ($) = S → .CC,$ // first (C)= {c,d}

C → .cC, first (C$)

C → .d, first (C$)

C → .cC, c/d,

C → .d, c/d

Step 3: Construct closure (I) Items

I_0:	**S^1 → .S, $**	**Goto[I_0, S] = 1**
	S → .CC, $	**Goto[I_0, C] = 2**
	C → .cC, c/d	**Action[I_0, c] = S_3**
	C → .d, c/d	**Action[I_0, d] = S_4**
I_1:	**S1→ S. ,$**	**Action[I_1, $] = Accept**
I_2:	**S → C.C, $**	**Goto[I_2,C]=5**

C → .cC, first (ε.$), C → .d, first (ε.$),

C → .cC, first ($), C → .d, first ($)

	C → .cC, $	**Action[I_0, c] = S_3**
	C → .d, $	**Action[I_0, d] = S_4**

I_3:	**C → c.C, c/d**	**Goto[I_3, C] = 8**

C → .cC, first (ε.c/d), C → .d, first (ε.c/d)

$C \rightarrow .cC$, first $(\varepsilon.C, \varepsilon.d)$, $C \rightarrow .d$, first $(\varepsilon.C, \varepsilon.d)$

$C \rightarrow .cC$, {first (C), first (d)}, $C \rightarrow .d$, {first (C), first (d)}

	$C \rightarrow c.C$, c/d	**Action[I_3, c] = S_3**
	$C \rightarrow .d$, c/d	**Action[I_3, d] = S_4**
I_4:	**$C \rightarrow d.$, c/d**	**Action[I_4, {c,d}] = r_3**
I_5:	**$S \rightarrow CC.$, \$**	**Action[I_5, {\$}]** = r_1
I_6:	**$C \rightarrow .cC$, \$**	**Action[I_6, c] = 9**

$C \rightarrow .cC$, first $(\varepsilon.\$)$, $C \rightarrow .d$, first $(\varepsilon.\$)$

$C \rightarrow .cC$, first (\$), $C \rightarrow .d$, first (\$)

	$C \rightarrow .cC$, \$,	**Action[I_6, c] = S_6**
	$C \rightarrow .d$, \$	**Action[I_6, d] = S_7**
I_7:	**$C \rightarrow d.$, \$**	**Action[I_7, \$] = r_3**
I_8:	**$C \rightarrow cC.$, c/d**	**Action[I_8, {c,d}]** = r_2
I_9:	**$C \rightarrow cC.$, \$**	**Action[I_9, {\$}] = r_2**

Item	**Action**			**Go to**	
	c	**d**	**\$**	**S**	**C**
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

The given grammar G is CLR. Since, it doesn't contain multiply-defined entries.

Q.2): Check whether the following grammar is CLR.

1. $S \rightarrow AaAb$

2. $S \rightarrow BbBa$

3. $A \rightarrow \varepsilon$

4. $B \rightarrow \varepsilon$

Solution:

Step 1: Augment Grammar

$S^1 \rightarrow .S$
$S \rightarrow .AaAb$
$S \rightarrow .BbBa$
$A \rightarrow .$
$B \rightarrow .$

Step2:

I_0:	$S^1 \rightarrow .S,\$$	Goto $[I_0,S]=1$
	$S \rightarrow .AaAb, \$$	Goto $[I_0,A]=2$
	$S \rightarrow .BbBa, \$$	Goto $[I_0,B]=3$
	$A \rightarrow . , a$	Action$[I_0,a]=R_3$
	$B \rightarrow . , b$	Action$[I_0,b]=R_4$
I_1:	$S \rightarrow S. , \$$	Goto$[I_1,\$]$=Accept
I_2:	$S \rightarrow A.aAb,\$$	Action$[I_2,a]=S_4$
I_3:	$S \rightarrow B.bBa,\$$	Action$[I_3,b]=S_5$
I_4:	$S \rightarrow Aa.Ab,\$$	Goto$[I_4,A]=6$
	$A \rightarrow ., b$	Action$[I_4,b]=R_3$
I_5:	$S \rightarrow Bb.Ba,\$$	Goto$[I_5,B]=7$
	$B \rightarrow ., a$	Action$[I,a]=R_4$
I_6:	$S \rightarrow AaA.b,\$$	Aciton$[I,b]=S_8$
I_7:	$S \rightarrow BbB.a, \$$	Action$[I,a]=S_9$
I_8:	$S \rightarrow AaAb., \$$	Action$[I,\$]=R_1$
I_9:	$S \rightarrow BbBa., \$$	Action$[I,\$]=R_2$

Item	**Action**			**Go to**		
	a	**b**	**$**	**S**	**A**	**B**
0	R_3	R_4		1	2	3
1			Accept			
2	S_4					
3		S_5				
4		R_3			6	
5	R_4					7
6	S_9	S_8				
7	S_9					
8			R_1			
9			R_2			

The give grammar is CLR, because the parsing table doesn't contain multiply – defined entries.

@ Construction of LALR Parsing Table

- LALR Stands for **L**ook **A**head **LR** technique. The LALR parsing table is small than the CLR parsing table. In practice LALR parsing method are mostly used.

Step 1: Construct augments grammar

Step 2: Closure (I) is similar to CLR

Step 3: In LALR(1) collection of items, the LR(1) items are combined based on the first term of the items, without any concern with the look ahead symbols.

Ex: If $I_0 = A \rightarrow .b$, c/d and $I_1 = A \rightarrow .b$, \$

Then the LALR item will be $I_{01} = A \rightarrow .b$, {c,d,\$}

Q) Consider the following grammar G

1. $S \rightarrow CC$
2. $C \rightarrow cC$
3. $C \rightarrow d$

Construct the LALR(1) parsing table

Solution:

Step 1: Augment Grammar

$S^1 \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC$
$C \rightarrow d$

Step 2: Closure Items

I_0: $S^1 \rightarrow .S,\$$
$C \rightarrow .cC$, c/d,
$C \rightarrow .d$, c/d

Step 3: Construct closure (I) Items

I_0:	$S^1 \rightarrow .S$, \$	Goto[I_0, S] = 1
	$S \rightarrow .CC$, \$	Goto[I_0, C] = 2
	$C \rightarrow .cC$, c/d	Action[I_0, c] = S_3
	$C \rightarrow .cC$, c/d	Action[I_0, d] = S_4

I_1:	$S1 \rightarrow S.\ ,\$$	Action[I_1, \$] = Accept
I_2:	$S \rightarrow C.C, \$$	Goto[I_2,C]=5
	$C \rightarrow .cC, \$$	Action[I_0, c] = S_3
	$C \rightarrow .d, \$$	Action[I_0, d] = S_4
I_3:	$C \rightarrow c.C, c/d$	Goto[I_3, C] = 8
	$C \rightarrow c.C, c/d$	Action[I_3, c] = S_3
	$C \rightarrow .d, c/d$	Action[I_3, d] = S_4
I_4:	$C \rightarrow d., c/d$	Action[I_4, {c,d}] = r_3
I_5:	$S \rightarrow CC., \$$	Action[I_5, {\$}] = r_1
I_6:	$C \rightarrow .cC, \$$	Action[I_6, c] = 9
	$C \rightarrow .cC, \$,$	Action[I_6, c] = S_6
	$C \rightarrow .d, \$$	Action[I_6, d] = S_7
I_7:	$C \rightarrow d., \$$	Action[I_7, \$] = r_3
I_8:	$C \rightarrow cC., c/d$	Action[I_8, {c,d}] = r_2
I_9:	$C \rightarrow cC., \$$	Action[I_9, {\$}] = r_2

Step 3: Merge common core item.

Common core items are [I_3, I_6], [I_4, I_7], [I_8, I_9]

I_0:	$S^1 \rightarrow .S, \$$	Goto[I_0, S] = 1
	$S \rightarrow .CC, \$$	Goto[I_0, C] = 2
	$C \rightarrow .cC, c/d$	Action[I_0, c] = S_{36}
	$C \rightarrow .d, c/d$	Action[I_0, d] = S_{47}
I_1:	$S^1 \rightarrow S., \$$	Action[I_1, \$] = Accept
I_2:	$S \rightarrow C.C, \$$	Goto[I_2, C] = 5
	$C \rightarrow .cC, c/d$	Action[I_2, C] = S_2
	$C \rightarrow .d, \$$	Action[I_2, d] = S_{47}
I_{36}:	$C \rightarrow c.C,\ \{\$, c/d\}$	Goto[I_{36}, C] = 89
	$C \rightarrow .cC, \{\$, c/d\}$	Action[I_{36}, c] = S_{36}
	$C \rightarrow .d, \{\$, c/d\}$	Action[I_{36}, d] = S_{47}
I_{47}:	$C \rightarrow d., \{c,d,\$\}$	Action[I_{47}, {c,d,\$}] = R_3
I_5:	$S \rightarrow CC., \$$	Action[I_5, \$] = R_1
I_{89}:	$C \rightarrow cC., \{c,d, \$\}$	Action[I_{89}, {c,d \$}] = R_2

Item	Action			Go to	
	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			Accept		
2	S_2	S_{47}			5
36	S_{36}	S_{47}			89
47	R_3	R_3	R_3		
5			R_1		
89	R_2	R_2	R_2		

The given grammar is LALR. The parsing table dosen't contain multiply defined entries.

Q): Consider the following grammar.

1. $S \to Aa$
2. $S \to bAc$
3. $S \to Bc$
4. $S \to bBa$
5. $A \to d$
6. $B \to d$

is LR(1) but not LALR(1)

Solution:

Step 1: Augment Grammar

$S^1 \to S$
$S \to Aa$
$S \to bAc$
$S \to Bc$
$S \to bBa$
$A \to d$
$B \to d$

Step 2:

I_0:	$S^1 \to .S, \$$	$Goto[I_0, S] = 1$
	$S \to .Aa, \$$	$Goto[I_0, A] = 2$
	$S \to .bAc. \$$	$Action[I_0, b] = S_3$
	$S \to .Bc, \$$	$Goto[I_0, B] = 5$
	$S \to .bBa, \$$	$Action[I_0, b] = S_3$
	$A \to .d, a$	$Action[I_0, d] = S_4$
	$B \to .d, c$	$Action[I_0, d] = S_4$
I_1:	$S^1 \to S. \$$	$Action[I_1, \$] = Accept$
I_2:	$S \to A.a, \$$	$Action[I_2, a] = S_6$

I_3:	$S \rightarrow b.Ac, \$$	$Goto[I_3, A] = 7$
	$S \rightarrow b.Ba, \$$	$Goto[I_3, B] = 8$
	$A \rightarrow .d, c$	$Action[I_3, d] = S_4$
	$B \rightarrow .d, a$	$Action[I_3, d] = S_4$
I_4:	$A \rightarrow d., a$	$Action[I_4, \{a,c\}] = R_5$
	$B \rightarrow d., c$	$Action[I_4,\{a,c\}] = R_6$
I_5:	$S \rightarrow B.c, \$$	$Action[I_5, c] = S_{10}$
I_6:	$S \rightarrow Aa., \$$	$Action[I_6, \{\$\}] = R_1$
I_7:	$S \rightarrow bA.c, \$$	$Action[I_7, c] = S_{11}$
I_8:	$S \rightarrow bB.a, \$$	$Action[I_8, a] = S_{12}$
I_9:	$A \rightarrow d., c$	$Action[I_9, \{a,c\}] = R_5$
	$B \rightarrow d., a$	$Action[I_9, \{a,c\}] = R_6$
I_{10}:	$S \rightarrow Bc., \$$	$Action[I_{10}, \{\$\}] = R_3$
I_{11}:	$S \rightarrow bAc., \$$	$Action[I_{11}, \{\$\}] = R_2$
I_{12}:	$S \rightarrow bBa., \$$	$Action[I_{12}, \{\$\}] = R_4$

Combine the common core item $[I_4, I_9]$

$A \rightarrow d., \{a\} \text{ u } \{c\} \Rightarrow A \rightarrow d., \{a,c\}$

$B \rightarrow d., \{c\} \text{ u } \{a\} \Rightarrow B \rightarrow d., \{a,c\}$

Item	Action					Go to		
	a	**b**	**c**	**d**	**$**	**S**	**A**	**B**
I_0		S_3		S_4		1	2	5
I_1					Accept			
I_2	S_6							
I_3				S_{49}			7	8
I_{49}	R_5, R_6		R_5, R_6					
I_5			S_{10}					
I_6					R_1			
I_7			S_{11}					
I_8	S_{12}							
I_{10}					R_3			
I_{11}					R_2			
I_{12}					R_4			

The given grammar is not LALR because the parsing table contains multiples entries.

@ Practice Questions

Q) Consider the grammar

S → S × B ⁄ A
A → A – B ⁄ A
A → A + D ⁄ D
D → a

Which has the high-test precedence

a) + b) × c) + ═ × d) Both a and b

Q) The grammar S → S **,** S⁄ S ↑ S ⁄A
A → B × B⁄ B|B ⁄ B
B → i

a) **,** has highest precedence than ×
b) ↑ has highest precedence than **,**
c) ⁄ has highest precedence than ×
d) × has highest precedence than ↑

Q) Find out the highest, least precedence in the following grammar

E → E × T⁄T
T → T – F⁄F
F → G + H ⁄H
H → id

Q) Check whether the following grammar G is SLR (1) , LR(1) and LALR (1)

i) S → AaAb / BbBa , A → ε , B → ε
ii) S → L=R/R , L → *R / id , R → L
iii) S → SA / A , A → a
iv) S → AS / b , A → SA / a
v) S→aAd /bBd / aBe / bAe , A → c , B → c
vi) S → Aa / bAc / dc / dba , A → d
vii) S → Aa / bAc / Bc / bBa , A → d , B → d
vii) S → aA / e , A → bAb / a

@ Gate Previous Questions

Q) Consider the following expression grammar: [2000-2M]

$E \rightarrow E \times F$

$E \rightarrow F + E$

$E \rightarrow F$, $F \rightarrow F -$, $F \rightarrow id$

Which of the following is true?

a) $\times$ has higher precedence than +

b)- has higher precedence than $\times$

c) + has equal precedence with -

d) + has higher precedence than $\times$

Q) Which of the following grammar rules violate the requirements of an operator grammar ? P , Q , R are non-terminals , and r ,s ,t are terminals ? [2004- 1M]

(i) $P \rightarrow QR$ (ii) $P \rightarrow QsR$ (iii) $P \rightarrow \varepsilon$ (iv) $P \rightarrow QtRr$

a) i only b) i and iii only c) ii and iii only d) iii and iv only

Q) In the following grammar $X \rightarrow X \# Y$, $Y \rightarrow Z \times Y$

$X \rightarrow Y$, $Y \rightarrow Z$, $Z \rightarrow id$

Which of the following is true?

a) # has left association , $\times$ has right associative

b) Both # and $\times$ has Left associative

c) Has R $\rightarrow$ L, $\times$ has left associative

d) None of the above.

Q) (a) Construct all the parse tree corresponding to $i + j * k$ for the grammar [2002-5M]

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

(b) In this grammar, what is the precedence of the two operators * and + ?

(c) If only one parse tree is desired for any string in the same language, what changes are to be made so that the resulting LALR(1) grammar is non-ambiguous?

Q) Consider the grammar $E \rightarrow E + n \,/\, E \times n \,/\, n$ [2005- 2M]

For a sentence $n + n \times n$, the handles in the right-sentential form of the reduction are:

a) n , E + n and E + n × n b) n , E + n and E + E × n

c) n, n + n and n + n × n d) n , E + n and E × n

Q) Consider the following grammar [2006- 1M]

$S \rightarrow S \times E$ $E \rightarrow F + E$

$S \rightarrow E$ $E \rightarrow F$ $F \rightarrow id$

Consider the following LR(0) items corresponding to the grammar above

(i) $S \rightarrow S \times . E$ (ii) $E \rightarrow F . + E$ (iii) $E \rightarrow F + . E$

Given the items above, which two of them will appear in the same set in the canonical set-of-items for the grammar ?

a) i and ii b) ii and iii c) i and ii d) None of these

Q) Consider the grammar [2005-2M]

$S \rightarrow (S) / a$

Let the number of states in SLR(1), LR(1) and LALR(1) parser for the grammar be n1 , n2 and n3 respectively. The following relationships holds good.

a) $n1 < n2 < n3$ b) $n1 = n3 < n2$ c) $n1 = n2 = n3$ d) $n1 \geq n2 \geq n3$

Q) Assume that the SLR parsers for a grammar G has n, states and the LALR parser for G has n2 states. The relationship between n1 and n2 is: [2003-1M]

a) n1 is necessarily less than n2 b) n1 is necessarily equal to n2
c) n1 is necessarily greater than n2 d) None of the above.

Q) Consider the grammar shown below: [2003-2M]

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

The grammar is

a) LL(1)
b) SLR(1) but not LL(1)
c) LALR(1) but not SLR(1)
d) LR(1) but not LALR(1)

Q) The grammar $S \rightarrow aSa \mid bS \mid c$ is [2010 - 2M]

a) LL(1) but not LR(1)
b) LR (1) but not LL(1)
c) Both LL(1) and LR(1)
d) Neither LL(1) nor LR(1)

Q) An LALR(1) parser for a grammar G can have Shift-Reduce (S-R) conflicts if and only if

a) The SLR(1) parser for G has S-R Conflicts [2008-2M]
b) The LR(1) parser for G has S-R Conflicts
c) The LR(0) parser for G has S-R Conflicts
d) The LALR(1) parser for G has R-R Conflicts

Q) The grammar is not suitable for predictive - parsing because the grammar is : [2005-1M]

a) Ambiguous b) Left-recursive c) Right-Recursive d) An operator-grammar

Q) Consider the following expression grammar. The semantic rules for expression calculation are stated next to each grammar production. [2005-2M]

$E \rightarrow$ number	E.val = number.val
$E \rightarrow E$ '+' E	$E^{(1)}$.val = $E^{(2)}$.val + $E^{(3)}$.val
$E \rightarrow E$ '×' E	$E^{(1)}$.val = $E^{(2)}$.val × $E^{(3)}$.val

1) The above grammar and the semantic rules are fed to a *yacc* tool (which is an LALR(1) parser generator) for parsing and evaluating arithmetic expressions. Which one of the following is true about the action of *yacc* for the given grammar?

a) It detects recursion and eliminates recursion.
b) It detects reduce-reduce conflict, and resolves
c) It detects shift-reduce conflict, and resolves the conflict in favor of a shift over a reduce action.
d) It detects shift-reduce conflict, and resolves the conflict in favor of a reduce over a shift action.

II) Assume the conflicts in Part(I) of this question are resolved and an LALR(1) parser is generated for parsing arithmetic expressions as per the given grammar. Consider an expression $3 \times 2 + 1$. What precedence and associativity properties does the generated parser realize?

a) Equal precedence and Left associativity; expression is evaluated to 7
b) Equal precedence and Right associativity; expression is evaluated to 9
c) Precedence of '×' is higher than that of '+' , and both operators are left associative; expression is evaluated to 7.
d) Precedence of '+' is higher than that of '×' , and both operators are left associative; expression is evaluated to 9.

Q) Match the following items [1995]

(i) Backus-Naur Form	(a) Regular expression
(ii) Lexical Analysis	(b) LALR(1) grammar
(iii) YACC	(c) LL(1) grammar
(iv) Recursive descent parsing	(d) General context-free grammar

Chapter - III

SYNTAX – DIRECTED TRANSLATION

Syntax directed translation associates information with a programming language construct by attaching attributes to the grammar symbols. Values for attributes are computed by "Semantics Rules" associated with the grammar productions.

There are two notations for associating semantics rules with productions.

i) Syntax – directed definitions

ii) Syntax – Translation Schemes.

i) Syntax – Directed Definition

Syntax – Directed definition is a generalization of context-free grammar in which each grammar production $X \rightarrow \alpha$ is associated with it a set of semantic rules of the form $a := f(b_1, b_2, \ldots b_k)$, where "$a$" is an attribute obtained from the function f. The attribute can be a string, number, a type, a memory location and anything else.

Input String $\rightarrow$ Parse tree $\rightarrow$ Dependency Graph $\rightarrow$ Evaluation Order for semantic rules

Conceptual view of syntax – directed translation

- There are two types of attributes:
  - Synthesized attributes
  - Inherited attributes

Semantic rules are used for computing values of the attributes associated with the symbols appearing in the grammar production.

- The value of an attribute at a parse tree is defined by a semantic rules associated with the production used at that node.
- The value of a **synthesized attribute** at a node is computed from the values of attributes at the children of that node in the parse tree.
- The value of an **inherited attribute** is computed from the values of attributes at the siblings and parent of that node.
- A syntax-directed definition that uses synthesized attributes exclusively is said to be an **S-attributed definition**.

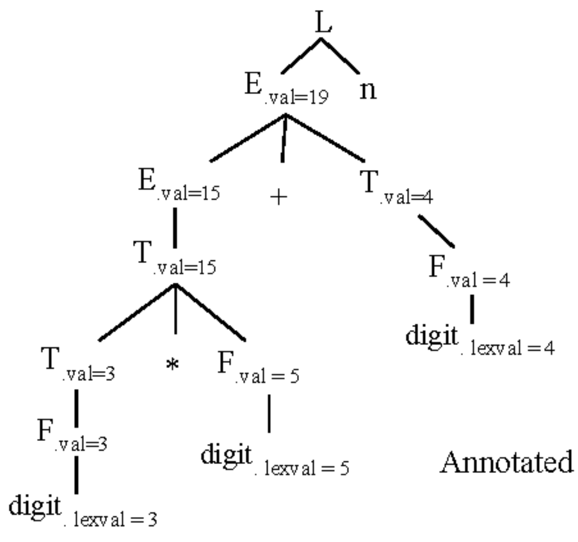
- A parse tree for an S-Attributed definition can be evaluating the semantic rules for the attributes at each node in Bottom-Up Fashion (or) Postorder Traversal
- An attribute grammar is a syntax-directed definition in which the functions in the semantic rules cannot have side effects.
- If 'X' is a symbol and 'a' is one of its attributes then we write $X._a$ to denote the value of **a** at parse tree node labeled X.
- The parse tree that shows value of attributes at each node is called **annotated parse tree**.

Ex: The following syntax-directed definition shows the calculator program. The definition associates an integer-valued synthesized attribute called value with each of the non-terminals E, T, and F. The token digit has a synthesized attribute lexical whose value is assumed to be supplied by the Lexical Analyzer.

Production	**Semantic Rules**
$L \rightarrow E_n$	Print (E.val)
$E \rightarrow E_1 + T$	E.val: $=E_1$.val + T.val
$E \rightarrow T$	E.val: = T.val
$T \rightarrow T * F$	T.val: = T_1. val x F.val
$T \rightarrow F$	T.val: = F.val
$F \rightarrow (E)$	F.val: = E.val
$F \rightarrow$ digit	F.val: = digit. lexval
$N \rightarrow ;$	Can be ignored by lexical analyzer as ;

Construct parser tree for the string *w* := 3*5+4n

Parse Tree:



Annotated parse tree

<u>Inherited Attributes</u>

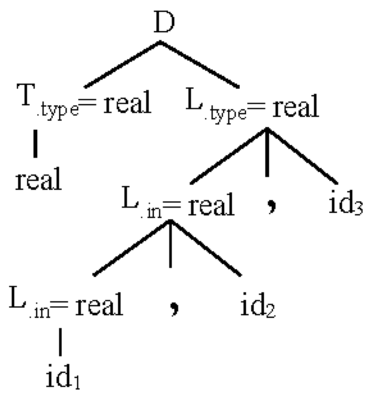
An inherited attribute is one whose value at a node in a parse tree is defined in terms of attributes at the parent and / or siblings of that node. Inherited attributes are convenient for expressing the dependence of a programming language construct on the context in which it appears. Example, An inherited attribute distributes types information to the various identifiers in a declaration.

In the following syntax-directed definition, the declaration generated by the non-terminal D consists of the keywords int. or real followed by a list of identifiers.

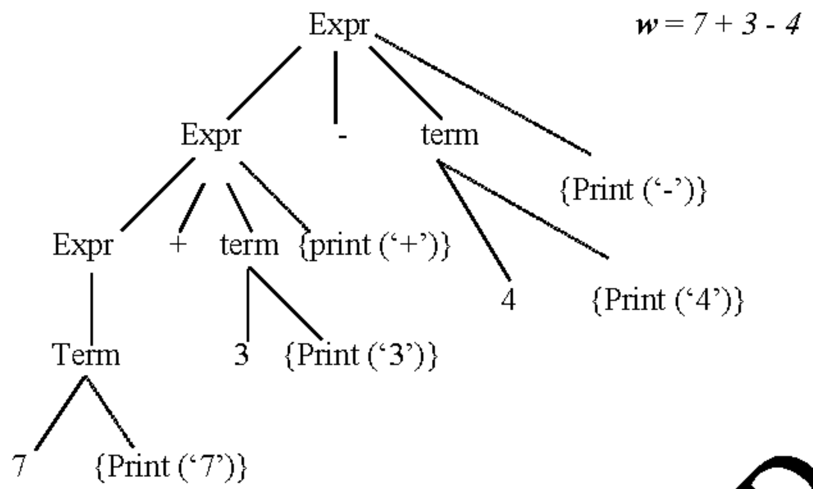
- The non-terminal T has a synthesized attribute type, whose value is determined by the keyword in the declaration.
- The semantic rule $L_{.in} = T_{.type}$, associated with production $D \rightarrow TL$, sets inherited attribute $L_{.in}$ to the type in the declaration.

Production	**Semantic Rules**
$D \rightarrow TL$	$L_{.in} := T_{.type}$
$T \rightarrow int$	$T_{.type} :=$ integer
$T \rightarrow real$	$T_{.type} :=$ real
$L \rightarrow L_1 , id$	$L_{1.in} := L_{.in}$ add type ($id_{.entry}$, $L_{.in}$)
$L \rightarrow id$	add type ($id_{.entry}$, $L_{.in}$)

The following annotated parse tree represents the sentence ***real id₁, id₂, id₃***



Parse tree with inherited attribute in at each node. The value of L.in at the three L-nodes gives the type of the identifiers id_1, id_2, and id_3. These values are determined by computing the value of the attribute T-type at the left child of the root and then evaluating L in top-down at the three L-nodes in the right sub tree as the root.



The Parse translated the sentence of "7+3-4" into "7 3+4 -"

Dependency Graph

The directed graph represents the interdependencies between synthesized and inherited attribute at node in the parse tree is called dependency graph.
For the rule X → YZ the semantic action is given by X.x := *f* (Y.y , Z.z) then synthesized attribute is X.x depends upon attribute Y.y and Z.z.

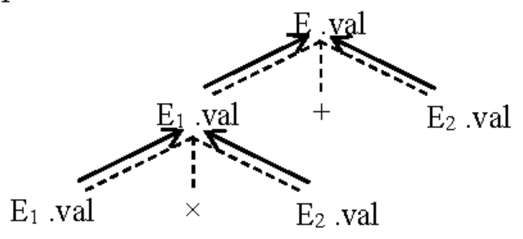
Ex:-
Design the dependency graph for the following grammar:

$$E \rightarrow E_1 + E_2$$
$$E \rightarrow E_1 \times E_2$$

The semantic rules for grammar production are:

Production Rules	Semantic Rules
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$
$E \rightarrow E_1 \times E_2$	$E.val = E_1.val \times E_2.val$

The dependency graph is:



The synthesized attributes can be represented by .val. Hence the synthesized attribute are given by E.val, E_1 .val and E_2 .val. The dependencies among the nodes are given by solid arrows. The arrows form E_1 and E_2 shown that value of E depends upon E_1 and E_2. Parse tree is represented using dotted lines.

Construction of Syntax Tree:

The syntax tree is an abstract representation of the language constructs. The syntax tree is used to write the translation routine using syntax directed definitions.

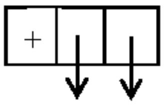
Ex: Consider the grammar for expression is

$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow E \times T$
$E \rightarrow T$
$T \rightarrow id$
$T \rightarrow num$

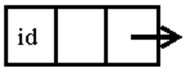
construct syntax for the above expression.

Constructing syntax tree for an expression means translation of expression into postfix form. Each node can be implemented as a record with multiple fields.

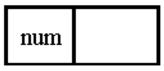
Following are the functions used in syntax tree for expression:



1. mknode(op, left, right) this function creates a node with the field operator having **operator** as a label, and the two pointers to left and right.



2. mkleaf(id, entry) This function creates an identifier node with label *id* and a pointer to symbol table is given by 'entry'.



3. mkleaf(num, val) This function creates node for number with label *num* and *val* is for value of that number.

Q) Construct the syntax tree for the expression : x × y - 5 + z

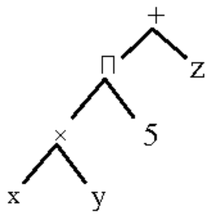
Sol:

Step 1: Convert the expression from infix to postfix as x y × 5 – z +
Step 2: Make use of the function mknode(op, left, right), mkleaf(id, ptr) and mkleaf(num, val).
Step 3: The sequence of function calls are:

Postfix expression is: x y × 5 – z +

Symbol	Operation
x	P_1 = mkleaf(id, ptr to entry x)
y	P_2 = mkleaf(id, ptr to entry y)
×	P_3 = mknode(×, P_1, P_2)
5	P_4 = mkleaf(num, 5)
–	P_5 = mknode(□ , P_3, P_4)
z	P_6 = mkleaf(id, ptr to entry z)
+	P_7 = mknode(+, P_5, P_6)

Parse tree for the string " x × y - 5 + z " is



Directed Acyclic Graph for Expression

The directed acyclic graph usually referred as DAG. It is drawn by identifying the common subexpressions. Like syntax tree, DAG has nodes representing the subexpressions in the expression. These nodes have operand$_1$, operand$_2$ and operator, where operands are the children of that node.

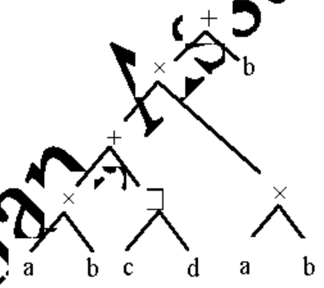
The difference between DAG and syntax tree is that common subexpression has more than one parent and syntax tree the common subexpression would be represented as duplicated subtree.

Draw the syntax tree and DAG for the expression: (a × b) + (c − d) × (a × b) + b

Sol: This expression can be evaluated as: $\big(\big(((a \times b) + (c - d)\big) \times (a \times b)\big) + b\big)$

The postorder traversal is a b × c d - + a b × × b +

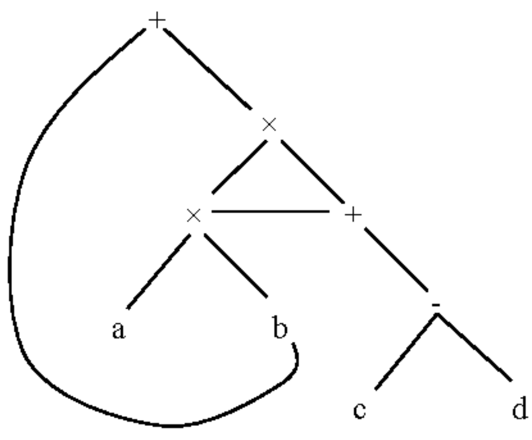
From this postorder sequence the syntax tree and DAG can be generated as follows:



Syntax tree

The sequence of operations for syntax tree is:

P_1 = mkleaf(id, a)
P_2 = mkleaf(id, b)
P_3 = mknode(×, P_1, P_2)
P_4 = mkleaf(id, c)
P_5 = mkleaf(id, d)
P_6 = mknode(-, P_4, P_5)
P_7 = mknode(+, P_3, P_6)
P_8 = mkleaf(id, a)
P_9 = mkleaf(id, b)
P_{10} = mknode(×, P_8, P_9)
P_{11} = mknode(×, P_7, P_{10})
P_{12} = mkleaf(id, b)
P_{13} = mknode(+, P_{11}, P_{12})



DAG

The sequence of operations for DAG is:

P_1 = mkleaf(id, a)
P_2 = mkleaf(id, b)
P_3 = mknode(×, P_1, P_2)
P_4 = mkleaf(id, c)
P_5 = mkleaf(id, d)
P_6 = mknode(-, P_4, P_5)
P_7 = mknode(+, P_3, P_6)
P_8 = mknode(×, P_7, P_3)
P_9 = mknode(+, P_8 ,P_2)

L-Attribute Definition

The S-attribute definitions are for synthesized attribute. There is another class of attribute which is L-attribute. The class of L-attribute can be evaluated in depth first order.

The syntax directed definition can be defined as the L-attribute for the production rule $A \rightarrow X_1 X_2 \ldots X_n$. The production $A \rightarrow X_1 X_2 \ldots X_n$ is such that

1) It depends upon the attributes of the symbol $X_1 X_2 \ldots X_{j-1}$ to the left of X_j.
2) It also depends upon the inherited attribute A

Note that because of these two conditions every S-attributed is also L-attributed definition.

Ex:

Check whether the given SDD (Syntax Directed Definition) is L-attributed or not.

$A \rightarrow PQ$	$P._{in} := p(A._{in})$
	$Q._{in} := q(P._{sy})$
	$A._{sy} := f(q._{sy})$
$A \rightarrow XY$	$Y._{in} := y(A._{in})$
	$X._{in} := x(Y._{sy})$
	$A._{sy} := f(X._{sy})$

<u>Sol:</u> The attribute *in* and *sy* represent the inherited and synthesized attribute respectively.

Production	Semantic Action	Class of Attribute
$A \rightarrow PQ$	$P._{in} := p(A._{in})$	L-attribute
	$Q._{in} := q(P._{sy})$	L-attribute
	$A._{sy} := f(q._{sy})$	L-attribute
$A \rightarrow XY$	$Y._{in} := y(A._{in})$	L-attribute
	$X._{in} := x(Y._{sy})$	Not L-attribute
	$A._{sy} := f(X._{sy})$	L-attribute

The given syntax directed definition is not L-attribute definition because the value of left symbol X is dependent upon value of right symbol (i.e., Y).

Q) Check whether the syntax-directed definition is L-attributed or not.

Production	**Semantic Rules**
$D \rightarrow TL$	$L._{in} := T._{type}$
$T \rightarrow int$	$T._{type} :=$ integer
$T \rightarrow real$	$T._{type} :=$ real
$T \rightarrow L_1, I$	$L_1._{in} := L._{in}$
	$I._{in} := L._{in}$
$L \rightarrow I$	$I._{in} := L._{in}$
$I \rightarrow I_1[num]$	$I_1._{in} :=$ array($num._{val}$, $I._{in}$)
$I \rightarrow id$	addtype($id._{entry}$, $I._{in}$)

Solution:

It is not L-attributed because $I_1._{in}$ depends on $num._{val}$ and *num* is to the right of I_1 in the production $I \rightarrow I_1$ [*num*].

ii) Syntax - Directed Translation Scheme

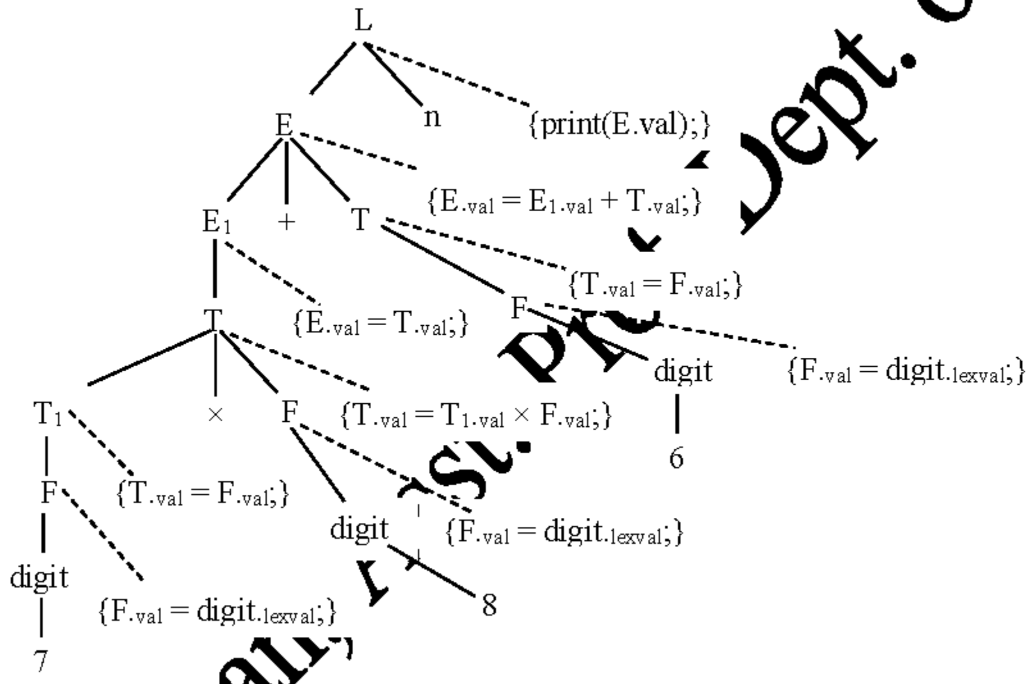
The syntax directed translations (SDT) Schemes are a complementary notation to syntax-directed definition. Syntax directed translation scheme is a context-free grammar in which program fragments (semantic actions) embedded within production bodies and can be appear at any position within a production body.

During the process of parsing the evaluation of attribute takes place by consulting the semantic action enclosed by { } at the right of the grammar symbol. The process of executing the semantic actions from the syntax directed definition is called Syntax-Directed Translation

For the given translation scheme, generate the annotated parse tree for the input $7 \times 8 + 6$

$L \rightarrow E_n$	{print(E.val);}
$E \rightarrow E_1 + T$	$\{E._{val} = E_{1}._{val} + T._{val};\}$
$E \rightarrow T$	$\{E._{val} = T._{val};\}$
$T \rightarrow T_1 \times F$	$\{T.val = T_{1}._{val} \times F._{val};\}$
$T \rightarrow F$	$\{T._{val} = F._{val};\}$
$F \rightarrow (E)$	$\{F._{val} = E._{val};\}$
$F \rightarrow digit$	$\{F._{val} = digit._{lexval};\}$

Sol: *w*: $7 \times 8 + 6$



The SDT prints 62 because $E._{val}$ is containing 62.

@ Practice Questions

Q) Consider the following production rules and their syntax directed definition:

Production	Semantic Rules
expr → expr 1 + term	expr. t := expr1. t || term. t || '+'
expr → expr 1 + term	expr. t := expr1. t || term. t || '- '
expr → term	expr. t := term. t
term → 0	term. t := '0'
term → 1	term. t := '1'
….	….
term → 9	term. t := '9'

What will be the translation of expression: ' 8 – 5 + 2 '

i) 8-5+2 ii) 852-+ iii) -+852 iv) 85-2+

Q) Consider the grammar with the following translation rules and E as the start symbol [2004-2M]

$E \rightarrow E_1$ # T { E. value = E_1. value * T. value }
E → T { E. value = T. value }
$T \rightarrow T_1$ & F{ T. value = T_1 .value + F. value }
T → F { T. value = F. value }
F → num { F. value = num. value }

Compute E. value for the root of the parse tree for the expression: 2 & 3 # 5& 6 # 4

a) 220 b) 180 c) 160 d) 40

@ Gate Previous Questions

Q) A shift reduce parser carries out the actions specified within braces immediately after reducing with the corresponding rule of grammar [1995]

S → xxW { print "1" }
S → y { print "2" }
W → Sz { print "3" }

What is the translation of xxxxyzz using the syntax directed translation scheme described by the above rules ?

(a) 23131 (b) 11233 (c) 11231 (d) 33211

Q) Consider the syntax directed translation scheme (SDTS) given in the following. Assume attribute evaluation with bottom-up parsing, i.e., attributes are evaluated immediately after a reduction. [2000-5M]

$E \rightarrow E_1$ * T { E. value = E_1. value * T.value }
E → T { E. value = T. value }
$T \rightarrow F - T_1$ { T. value = F. value - T_1 .value }
T → F { T. value = F. value }
F → 2 { F. value = 2 }
F → 4 { F. value = 4 }

(a) Using this SDTS, construct a parse tree for the expression “ 4 – 2 – 4 * 2 “ and also compute its $E_{.val}$

Q) Consider the translation scheme shown below: [2003-2M]

$S \rightarrow TR$
$R \rightarrow +T$ { *print* (‘+‘); } R | ε
$T \rightarrow$ *num* { *print* (*num.val*); }

Here *num* is a token that represents an integer and *num.val* represents the corresponding integer
value. For an input string ‘9+5+2’ , this translation scheme will print:

i) 9 + 5 + 2 ii) 9 5 + 2 + iii) 9 5 2 + + iv) + + 9 5 2

Q) Consider the grammar with the following translation rules and E as the start symbol

$E \rightarrow E_1$ # T { E. value = E_1. value * T. value } [2004-2M]
$E \rightarrow T$ { E. value = T. value }
$T \rightarrow T_1$ & F{ T. value = T_1 .value + F. value }
$T \rightarrow F$ { T. value = F. value }
$F \rightarrow$ num { F. value = num. value }

Compute E. value for the root of the parse tree for the expression: 2 # 3 & 5 # 6&4

a) 200 b) 180 c) 160 d) 40

Q) Consider the following translation scheme [2006-2M]

$S \rightarrow ER$
$R \rightarrow$* E { *print* (‘*‘); } R | ε
$E \rightarrow F + E$ { *print* (‘+’); } | F
$F \rightarrow$ (S) | id { *print* (*id. value*); }

Here *id* is a token that represents an integer and *id. Value* represents the corresponding integer value. For an input string ‘2*3+4’ , this translation scheme prints

a) 2*3+4 b) 2*+34 c) 23*4+ d) 234+*

CHAPTER – IV

RUN – TIME ENVIRONMENT

INTRODUCTION

The allocation and deallocation of data objects is managed by the run-time support package, consisting of routines loaded with the generated target code. The design of the run-time support package is influenced by the semantics of procedures. Each execution of a procedure is referred as an activation of the procedure. The representation of a data objects at run-time determined by its type.

Source Language Issues:

Procedures:

A procedure definition is a declaration that associates an identifier with a statement when a procedure name appears within an executable statement. We say that the procedure is called at that point.

- The identifiers (parameters) appearing in the procedure definition are called formal parameters.
- The arguments that are passed to a procedure are called actual parameters which are substituted for the formal parameters in the body.

Activation Tree:

The execution of a procedure body is referred as an activation of the procedure. The lifetime of an activation of a procedure is the sequence of steps between the first and last steps in the execution of the procedure body.

- The scope of a variable is the range of statements in which the variable is visible. A variable is visible in a statement if it can be referenced in that statement.
- A variable is **local** in a program unit or block if it is **declared there**.
- The **non local variables** of a program unit or block are those that are **visible** within the program unit or block but are **not declared there**.
- In an Activation tree:
  - The root represents the activation of the main program.

- Each node represents an activation of a procedure.
- The node for ***a*** is the parent of the node for ***b*** if and only if control follows from activation ***a*** to ***b***, and
- The node for ***a*** is to the left of the node for ***b*** if and only if the life time of ***a*** occurs before the lifetime of ***b***.

Control stack:

Control stack is used to keep track of live procedure activation. The content of the control stack are related to paths to the right of the activation tree.

Bindings of Names:

Even, if a name is declared only once in a program, the same name may denote different data objects at run time. The informal term "data object" corresponds to a storage location that can hold values. The term environment refers to a function that maps a name to a storage location, and the term state refers to a function that maps a storage location to the value held there.

$$\begin{tikzcd}
\text{Name} \arrow[r, bend left, "\text{Environment}"] & \text{Storage} \arrow[r, bend left, "\text{State}"] & \text{Value}
\end{tikzcd}$$

Two-stage mapping from *names* to *values*.

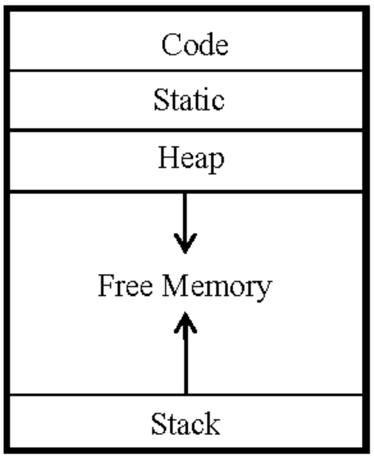
Environment map names to l-values, whereas states map l-values to r-values.

@Storage Organization:

It deals with the division of memory into different areas. The run-time storage might be subdivided hold:

1. The generated target code
2. Data objects, and
3. A counterpart of the control stack to keep track of procedure environment.

- The size of the generated target code is fined at compile times so the compiler can place it in a statically determined area.
- Similarly, the size of the some of the data objects may be known at compile time, they can be placed in statically determined area.



Typical subdivision of run-time memory into code and data areas.

Code Area: It contains the generated code area.

Static Area: It contains data whose absolute address can be determined at compile time.

Stack and *Heap* area utilizes the storage during the run time. These areas are opp. ends of the remainder of the address space and are dynamic. The stack is used to store data structures called activation records that get generated during procedure calls.

Activation Record:

Procedure calls and returns are usually managed by a run-time stack called the *control stack*. Each live activation has an *activation record / frames* on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.

The field of an activation record is as follows:

1. *Temporary Values*, such as those arising in the evaluation of expressions.
2. *Local data* belonging to the local environment of currently activated procedure.

Returned Value
Actual Parameter
Control Link
Access Link
Saved Machine Status
Local Data
Temporaries

A general activation records

3. The field for *saved machine status* holds information about the state of the machine just before the call to the procedure.
4. The *Access link* may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation.
5. The *Control link* pointing to the activation record of the caller.
6. The field for *actual parameters* is used by the calling procedure to supply parameters to the called procedure.
7. The field for the *Returned value* is used by the called procedure to return a value to the calling procedure.

Activation record is a conceptual aggregate of data which contains all information required for a single activation of a procedure.

Storage Allocation Strategies:

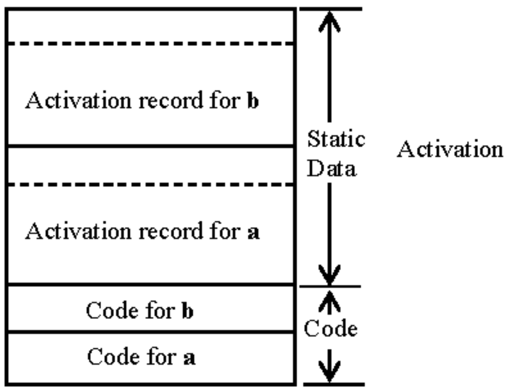
A different storage allocations schemes are followed:

1. Static allocation lays out storage for all data objects at compile time
2. Stack allocation manages the run-time storage as a stack.
3. Heap allocation allocates and de allocates storage as needed at runtime from a data area known as a heap.

I) Static Allocation:

In static allocation, names are bound to storage at compile time. Bindings do not change at run time, every time a procedure is activated; its names are bound to the same storage locations. This property allows the values of local names to be retained across activations of a procedure i.e., when control returns to a procedure, the values of the locals are the same as they were when centered left the last time.

Example:



Static Storage Allocation.

The limitations of static storage allocations are:

1. The size required must be known at compile time.
2. Recursive procedure cannot be implemented.
3. Data structure cannot be created dynamically.

II) <u>Stack Allocation:</u>

The storage is organized as a stack. Activation records are pushed and popped form the stack. Storage for the locals in each call of a procedure is contained in the activation record for that calls.

- Thus locals are bound to storage for activation. A new activation record is pushed into the stack when a call is made. The values are locals are deleted when the activation ends or when it popped from the stack.

<u>Calling Sequence</u>:

The calling sequence for a procedure allocates an activation and fills its fields with appropriate values. The return sequence restores the machine state to allow execution of the calling procedure to continue. The code in the calling sequence is often divided between the calling procedure (Caller) and the procedure it calls (Callee).

In the activation record, the control link, access link and machine status fields appear in the middle. The values in these fields are fixed at compile-construction time. Even though sizes of the fields for the temporaries are fixed at compile time, the size may not be known to the front end due to changes occurring during code optimization.

<u>During a procedure call:</u>

I. Callers Activities:

a. Makes space on the stack for a return value.

b. Loads actual parameters on to the stack.

c. Sets the static link.

d. Jumps to the called procedure. Returns address is saved on the stack.

II. Callee's Activities:

a. Sets the dynamic link.
b. Sets the base of the new activation.
c. Stores the registers on stack.
d. Make space for local data on the stack.

During a return form a procedure

Callee's activities

1. Restores register
2. Sets the base pointer to the activation record of the calling procedure.
3. Returns to the caller.

Dangling References

Whenever storage can be deallocated, the problem of dangling references arises. A dangling reference occurs when there is a reference to storage that has been deallocated. (It is a logical error to use dangling references, since the value of deallocated storage is undefined according to the semantics of most languages).

Heap Allocation

Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects. Pieces may be de allocated in any order, so over time the heap will consist of alternate areas that are free and in use. It allows the programmer explicit control over the allocation/deallocation of storage to the program variable.

Static Scope

The scope of a variable can be statically determined, that is prior to execution.

- A block is a statement containing its own local data declaration.

Dynamic Scope

Under Dynamic Scope, a new activation inherits the existing bindings of non local names to storage.

- A non local name a 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 refers to storage in the new activation record.

Passing Parameter

When one procedure calls another, the usual method of communication between them is through non local names and through parameters of the called procedure. Several common methods for associating actual and formal parameters are:

1. Call – by – value
2. Call – by – reference
3. Copy – restore
4. Call – by – name

Differences between parameter-passing methods are based primarily on whether an actual parameter represents an r-value, an l-value, or the text of the actual parameter itself.

i) Call – by - Value

The actual parameters are evaluated and their r-values are passed to the called procedure.

Call- by – value can be implemented as follows:

1. A formal parameter is treated just like a local name, so the storage for the formals is in the activation record of the called procedure.
2. The caller evaluates the actual parameters and places their r-values in the storage for the formals.

- As distinguishing feature of call-by-value is that operations on the formal parameters do not affect values in the activation record of the caller.

ii) Call-by-Reference

When parameters are passed by reference (also known as call-by-address or call-by-location), the caller passes to the called procedure a pointer to storage address of each actual parameter.

A reference to a formal parameter in the called procedure becomes, in the target code, an indirect reference through the pointer passed to the called procedure.

iii) Copy-Restore

A hybrid between call-by-value and call-by-reference is copy-restore linkage (also known as copy-in-copy-out, or value-result).

1. Before control flows to the called procedure, the actual parameters are evaluated.
2. When control returns, the current r-values of the formal parameters are copied back into the l-values of the actual, using the l-values computed before the call.

iv) Call-by-Name

Call-by-Name is traditionally defined by the "Copy-rule".

1. The procedure is treated as if it were a macro; that is, its body is substituted for the call in the caller, with the actual parameters literally substituted for the formals. Such a literal substitution is called macro-expansion or in-line expansion.
2. The local names of the called procedure are kept distinct from the names of the calling procedure. We can think of each local of the called procedure being systematically renamed into a distinct new name before the macro-expansion is done.
3. The actual parameters are surrounded by parentheses if necessary to preserve their integrity.

@ Previous Gate Questions

Q) A language L allows declaration of arrays whose size are not known during compilation. It is required to make efficient use of memory. Which one of the following is true?

a) A compiler using static memory allocation can be written for L [1997-1M]

b) A compiler cannot be written for L, an interpreter must be used

c) A compiler using dynamic memory allocation can be written for L

d) None of the above

Q) Heap allocation is required for languages [1997-1M]

a) That support recursion

b) That support dynamic data structures

c) That use dynamic scope rule

d) None of the above

Q) Consider a program P that consists of two source modules M_1 and M_2 contained in two different files. If M contains a reference to a function defined in M , the reference will be resolved at [2004-2M]

a) Edit-time b) Compile-time c) Link-time d) Load-time

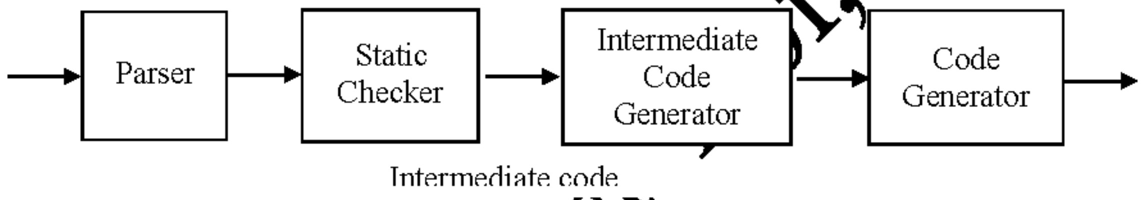
CHAPTER – V

INTERMEDIATE AND TARGET CODE GENERATION

In the compiler, the front end translates a source program into an intermediate representation from which the back end generates target code.

Benefits of using a machine-independent intermediate form are

1. Retargeting Facilities: A compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.



Portion of Intermediate Code Generator

Intermediate Language

Syntax trees and prefix notations are two kinds of intermediate representations. A third, called "three-address code" was introduced. The semantic roles for generating three-address code from common programming language constructs are similar to those for constructing syntax tree or for generating prefix notation.

Three-Address Code

Three address codes is a sequence of statements of the general form

x: = y opz

Where x,y and z are names, constants, or compiler-generated temporaries; op stands for any operator.

Note that no built-up arithmetic expressions are permitted, as there is only one operator on the right side of a statement.

Ex: Thus a source language expression like x+y*z might be translated into a sequence.

$t_1 := y*z$

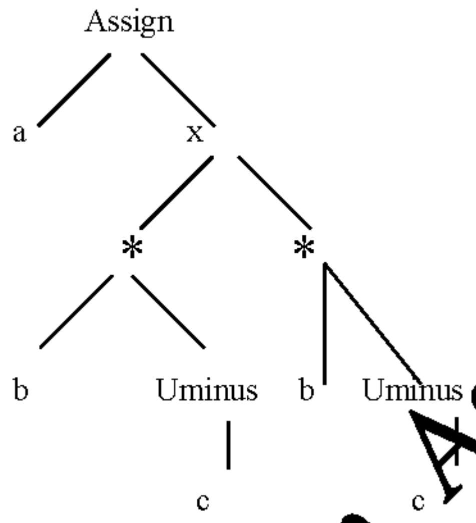
$t_2 := x+t_1$

Where t_1 and t_2 are compiler-generated temporary names.

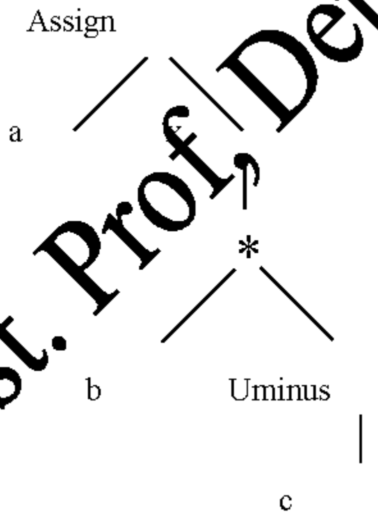
Three-address code is a linearized representation of syntax tree or dag in which explicit names corresponds to the interior nodes of the graph.

A syntax tree or dag for the assignment statement

A: = b*-c+b*-c



a) Parse Tree



b) Dag

Production	Semantic Rules
$t_1 := -c$ $t_2 := b*t_1$ $t_3 := -c$ $t_4 := b*t_3$ $t_5 := t_2 + t_4$ $a := t_5$ *a) Code for the syntax tree*	$t_1 = -c$ $t_2 := b*t_1$ $t_3 := t_2 + t_2$ $a := t_5$ *b) Code for the dag*

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

Implementation of Three-Address Statement

A three-address statement is an abstract from of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are

i) Quadruples
ii) Triples
iii) Indirect Triples

i) Quadruples

A quadruple is a record structure with four fields, which we call

Op, arg1, arg 2, and result

The op field contains on internal code for the operator.

Ex: The three address statement x: =y opz is represented by placing y in arg1, z in arg2, and x in result.

- The statements with unary operators like x: =-y or x: =y do not use arg2.
- The contents of fields arg1, arg2, and result are normally pointers to the symbol-table entries for the names represented by these fields.

- The quadruples m following table, are for the assignment a: =b*-c+b*-c.

	OP	**Arg$_1$**	**Arg$_2$**	**Result**
(0)	Uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	Uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	T_5		a

Quadruples

ii) Triples

To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.

The three-address statements can be represented by records with only three fields: op, arg1 and arg2.

	OP	Arg_1	Arg_2
(0)	Uminus	c	
(1)	*	B	(0)
(2)	Uminus	C	
(3)	*	B	(2)
(4)	+	(1)	(3)
(5)	Assign	A	(4)

Triples correspond to the representation of a syntax tree or dag by an array of nodes.

iii) Indirect Triples

Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves.

Ex: Let us use an array statement to list pointers to triples in the desired order.

Indirect triples representation of three-address statement.

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

	OP	Arg_1	Arg_2
(0)	Uminus	c	
(1)	*	B	(0)
(2)	Uminus	C	
(3)	*	B	(2)
(4)	+	(1)	(3)
(5)	Assign	A	(4)

- Using the quadruple notation, a three address statement defining or using a temporary can immediately access the location for that temporary via the symbol table.
  - Benefit of quadruples appears in an optimizing compiler, where statements are often moved around.
  - Using quadruples notation, the symbol table interposes an extra degree of indirection between the computation of a value and its use.
- In the triples notation, moving a statement that defines a temporary value requires us to change all references to that statement. This problem makes triples difficult to use in an optimizing compiler.
- In direct triples present no such problem. A statement can be moved by recording the statement list.
  - It can save space compared with quadruples if the same temporary value is used more than once.
  - Indirect triples look very much like quadruples as for as utility is concerned. The two notations requires about the same amount of space and they are equally efficient for reordering code.

@ GATE Previous Questions

Q) Generating of intermediate code based on an abstract machine model is useful in compiler because

a. It makes implementation of lexical analysis and syntax analysis easier [1994]
b. Syntax-directed translations 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 machines directly form high level language programs.

Q) Consider the syntax directed definition shown below: [2003-2M]

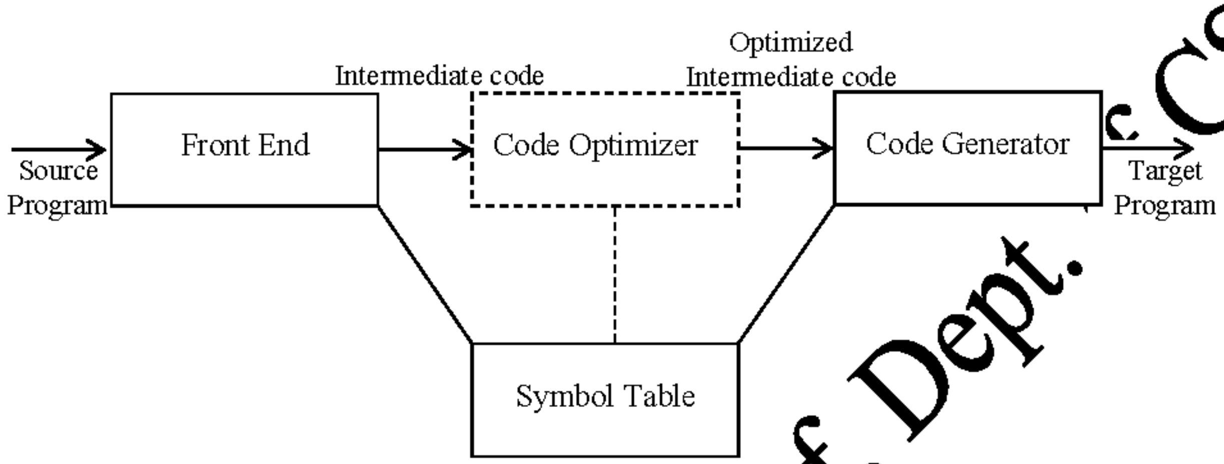
$S \rightarrow id := E$ { gen(id.place=E.place) }
$E \rightarrow E_1 + E_2$ { t = newtemp( ); gen (t=E_1 .place + E_2 .place); E.place =t }
$E \rightarrow id$ { E . place = id. place}

Here, *gen* is a function that generates the output code, and new temp is a function that returns the name of a new temporary variable on every call. Assume that t's are the temporary variable names generated by *newtemp* . for the statement ' **X := Y + Z** ',
the 3-address code sequence generated by this definition is :

a) X=Y+Z
b) $t_1 = Y + Z$; $X = t_1$
c) $t_1 = Y$; $t_2 = t_1 + Z$; $X=t_2$
d) $t_1=Y$; $t_2=Z$; $t=t_1+t_2$; $X=t_3$

CODE GENERATION

The code generation phase takes an input an intermediate representation of the source program and produces as output an equivalent target program.



Position of Code Generator

The output code must be make effective use of the resources of the target machine.

Issues in the design of a Code Generator

While the details are dependent on the target language and the operating system, issues such as:

1. Input to the code generator
2. Target programs
3. Memory Management
4. Instruction Selection
5. Register Allocation
6. Choice of evaluation order
7. Approaches to code generation

1. Input to the Code Generator

The input to the code generator consists of the intermediate representation of source program, together with the information in the symbol table that is used to determine run-time address of the data objects denoted by the names in the intermediate representation.

2. **The Target Program**

The output of the code generator is the target program. The target program may take a variety of forms i.e.,

- Absolute machine language
- Relocated machine language
- Assembly language.

3. **Memory Management**

Mapping names in the source program to address of data objects in run time memory is done by the code generator. To generate the machine code, labels in three-address statements have to be converted to addresses of instruction.

4. **Instruction Selection**

The nature of the instruction set of the target machine determines the difficulty of instruction selection. Deciding which machine code sequence is best for a given three address construct may require knowledge about the context in which that construct appears.

5. **Register Allocation**

Instructions involving register operands are usually shorter and faster than those involving operands in memory. The use of registers is sub divided into two sub programs:

i) During register allocation, we select the set of variable that will reside in registers at a point in the program.

ii) During a subsequent register assignment phase, we select the specific register that avoidable will reside in.

6. **Evaluation Order**

The order in which computations are performed can affect the efficiency of the target code. An efficient order of computation is also an important issue in code generation. Reducing the number of LOAD and STORE instruction improve the speed of code generation.

7. Approaches to Code Generation

The most important factor that should be satisfied in the generation of the target code is that it should be error free and easy to test and maintain.

Machine Dependent Code Generation

Target computer is a byte-addressable machine with four bytes to a word and n general-purpose registers, R_0, R_1……., R_{n-1}. It has two-address instruction of the form,

Op source, destination.

In which op is an operation code (op code), source and destination are data fields.

- The source and destination fields are not long enough to hold memory address, so contain bit patterns in these fields specify that words following on instruction contain operands and / or addresses.
- The source and destination of an instruction are specified by combining registers and memory locations with address modes.

Different Address Modes

Mode	Form	Address	Added Cost
Absolute	M	M	1
Register	R	R	0
Indexed	C (R)	C + Contents (R)	1
Indirect Register	* R	Contents (R)	0
Indirect Indexed	* C (R)	Contents (C+Contents (R))	1
Literal	# C	(Source to be a Constant)	1

Instruction Costs:

The cost of an instruction to be one plus the costs associated with the source and destination address modes. This cost corresponds to the length of the instruction.

The time taken to fetch an instruction from memory exceeds the time spent executing the instruction. Therefore, by minimizing the instruction length we also tend to minimize the time taken to perform the instruction as well.

Some of the difficulties in generating code for this machine can seen by considering what code to generate for a three-address statement of the form:

A = b + c

This statement can be implemented by many different instruction sequences.

Ex:

		1 + Address Mode Cost	**Instruction Cost**
1)	MOV b, R_0	1+1	2
	ADD c, R_0	1+1	2
	MOV R_0, a	1+1	2
		Total Cost	6
2)	MOV b, a	1+2	3
	ADD c, a	1+2	3
		Total Cost	6

Assuming R_0, R_1 and R_2 contain the address of a, b and c respectively.

		1 + Address Mode Cost	**Instruction Code**
1)	MOV *R_1, *R_0	1+0	1
	ADD *R_2, *R_0	1+0	1
		Total Cost	2
2)	ADD R_2, R_1	1+0	1
	MOV R_1, a	1+1	2
		Total Cost	3

In order to generate good code for this machine, we must utilize its addressing capabilities efficiently.

@ GATE Previous Questions

Q) In a simplified computer the instructions are: [2007-2M]

OP R_j, R_i – Performs R_j 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 R_i, m – Moves the content of register Ri to memory location m.

The computer has only two registers, and OP is either ADD or SUB. Consider the following basic block:

$t_1 = a + b$

$t_2 = c + d$

$t_3 = e - d$

$t_4 = t_1 - t_3$

Assume that all operands are initially in memory. The final value of the computation should be in 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

CHAPTER – VI

CODE OPTIMIZATION

The code produced by straight forward compiling algorithms can often be made to run faster or take less space or both compilers that apply code-improving transformations are called optimizing compilers.

Optimization can be done on:

- Machine-Independent
- Machine-Dependent

- Machine – Independent optimizations, program transformations that improve the target code without taking into consideration any properties of the target machine.
- Machine-Dependent optimizations, such as register allocation and utilization of special machine – instruction sequences.
- As compiler does not have the benefit of sample input data, so it must make its best guess as to where the program hot spots are. The program inner loops are good candidates for improvement.
- In general, a process called "control-flow analysis" identifies loops in the flow graph of a program.
- Several algorithms for collecting information using data-flow analysis and for effectively using this information in optimization.

Data-flow analysis, a process of collecting information about the way variables are used in a program. The information collected at various points in a program can be relegated using simple set equations.

Criteria for Code-Improving Transformations

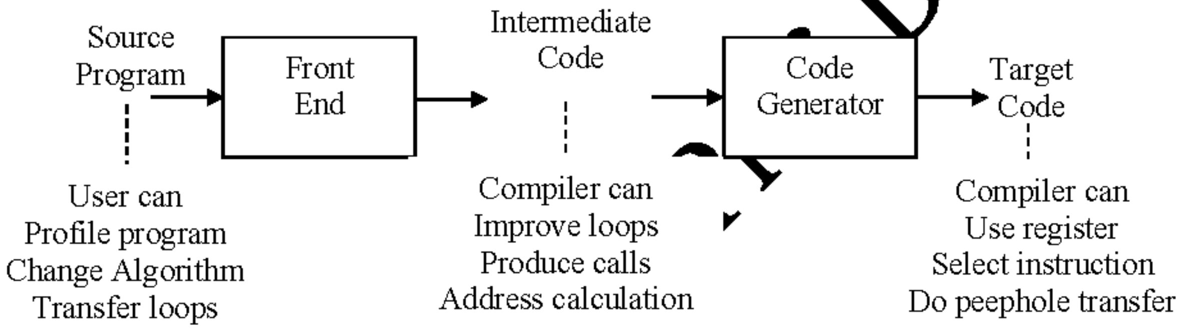
The transformations provided by an optimizing compiler should have several properties:

1. First: A transformation must preserve the meaning of programs. That is, an "optimization" must not change the output produced by a program for a given input, or cause an error.

2. Second: A transformation must, on the average, speed op programs by a measurable amount.
3. Third: A transformation must be worth the effort. Certain local or 'peephole' transformations of the kind are simple enough and beneficial enough to be included in any compiler.

Getting Better Performance

The running time of a program is improved at levels, from source level to the target level in the compiles as shown in figure below. At each level, the available options fall between the two extremes of finding a better algorithm and of implementing a given algorithm so that fewer operations are performed.



Places for potential improvements by the user and the compiler.

The Principal Sources of Optimization

A compiler optimization must preserve the semantics of the original program. Once a programmer chooses and implements a particular algorithm, the compiler cannot understand enough about the program to replace and more efficient algorithm. A complier knows only how to apply relatively low-level semantic transformation or program semantics such as the fact that performing the same operation on the same values yields the same result.

Semantics-Preserving (Function-Preserving) Transformations:

There are number of ways in which a compiler can improve a program without changing the function it computes.

- ✓ Global Common Subexpressions
- ✓ Copy propagation
- ✓ Dead-code elimination
- ✓ Constant Folding

A program will include several calculations of the same value, such as an offset in an array.

Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

<u>Common Sub expression</u>

An occurrence of an expression E is called a common sub expression if E was previously computed, and the values of variables in E have not changed since the previous compotation. We can avoid recomputing E if we can use the previously computed value.

Example:

B_5

$t_6 = 4 \times i;$
$x\ = a[t_6];$
$t_7 = 4 \times i$
$t_8 = 4 \times j$
$t_9 = a[t_8];$
$a[t_7] = t_9;$
$t_{10} = 4 \times j;$
$a[t_{10}] = x;$
goto B_2

a) Before

B_5

$t_6 = 4 \times i;$
$x\ = a[t_6];$
$t_8 = 4 \times j$
$t_9 = a[t_8];$
$a[t_6] = t_9;$
$a[t_8] = x;$
goto B_2

b) After

Local common-subexpression elimination.

The assignment to t_7 and t_{10} in above fig. compute the common subexpressions $4 \times i$ and $4 \times j$, respectively. These steps have eliminated in fig (b), which uses t_6 instead of t_7 & t_8 instead of t_{10}.

<u>Copy Propagation</u>

The idea behind the copy-propagation transformation is to use *v* for *u*, wherever possible after the copy statement u = v.

Example: in fig(a) the assignment x t3 in that block is copy. Copy propagation applied to B5 yields the code in fig(b). This change may not appear to be an improvement, but it gives us the opportunity to eliminate the assignment to x.

B_5	B_5

x =t_3;
a[t_2]=t_5;
a[t_4]=x;
goto B_2
	x =t_3;
	a[t_2]=t_5;
	a[t_4]=t_3;
	goto B_2

a) Before	b) After

Basic block B_5 after copy propagation

Dead-Code Elimination

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

Ex: The use of 'debug' that is set to true or false at various points in the program, and used in statement like

if (debug) print…..

By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement, the value of "debug is false".

If a copy-propagation replaces debug by false, then the print statement is dead because it cannot be reached.

- Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
- One advantage of copy-propagation is that it often turns the copy statement into dead code.

Loop Optimizations

Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

i) Code-Motion
ii) Induction-Variable Elimination
iii) Reduction in Strength.

i) Code- Motion

By some modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed(a *loop-invariant computation*) and evaluates the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop, that is, one basic block to which all jumps from outside the loop go.

Example:

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

Code motion will result in the equivalent code

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

Now, the computation of *limit-2* is performed once, before we enter the loop. Previously, there would be *n+1* calculations of *limit-2* if we iterated the body of the loop *n* times.

ii) & iii) Induction-variable and reduction in strength

A variable *x* is called an *induction variable* if there is a positive or negative constant c such that each time *x* is assigned, its value increased by *c*.

Flow graph relevant to transformations is shown below:

B_3

$j = j - 1$
$t_4 = 4 * j$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3

Fig. a

B_3

$j = j - 1$
$t_4 = t_4 - 4$
$t_5 = a[t_4]$
if $t_5 > v$ goto B_3

Fig. b

Strength reduction applied to 4 * j in the block B_3

Induction variables can be computed with a single increment (addition or subtraction) per loop iteration. The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as *strength reduction.*

In the B_3 values of *j* and t_4 remain in lock step shown in Fig. a. Every time the value of *j* decreases by 1, the value of t_4 decreases by 4, because 4 * j is assigned to t_4. The strength of the instruction is reduced by replacing multiplication by subtraction shown in Fig.b .

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

Peephole Optimization

A statement-by-statement code-generation strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying “optimizing” transformations to the target code.

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

Characteristics of peephole optimizations:

- Redundant-Instruction Elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Redundant Loads and stores

Ex: If the instruction sequences as

1) MOV R_0, a
2) MOV a, R_0

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

Note that if (2) had a label.

MOV R_0, a

Label: MOV a, R_0

We could not be sure that (1) was always executed immediately before (2) and so we could not remove (2). Put another way (1) and (2) have to be in the same basic block for this transformation to be safe.

Unreachable Code

An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be eliminate a sequence of instruction.

Ex: In C, the source code might look like:

```
# define debug O
- - - -
if (debug) {
print debugging information
}
```

In the intermediate representation the if statement may be translated as:

If debug = 1 goto L_1

Goto L_2

L_1: print debugging information

L_2:

Peephole optimization is to eliminate jump over jumps. Thus, no matter what the value of debug, can be replaced by:

If debug # 1 goto L_2

Print debugging information

L_2:

As the argument of the first statement evaluates to a constant true, it can be replaced by goto L_2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flow-of-Control Optimization:

The intermediate code generation algorithms produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.

We can replace the sequence

goto L_1

…..

L_1: goto L_2

By the sequence

goto L_2

…..

L_1: goto L_2

There are no jmps to L_1, then it may be possible to eliminate the statement L_1: goto L_2 provided it is provided it is preceded by an unconditional jump.

Algebraic Simplification

Peephole optimizations are often applied to algebraic identities.

Ex: x:=x+0;

Or

X:x*1;

(Statements often produced by straight forward intermediate code-generation algorithms, can be eliminated easily through peephole optimization).

Reduction in Strength

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.

Ex: X^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine.

Use of Machine Idioms

The target machine may have hardware instructions to implement certain specific operations efficiently.

Ex: The use auto-increment and auto-decrement has greatly improved the quality of code when pushing or poping a stack, as in parameter passing.

DAG Representation for Basic Blocks

The direct a cyclic graph is used to apply transformations on the basic block. To apply the transformations on the basic block DAG is constructed from three address statement.

A DAG can be constructed for the following types of tables on nodes.

1) Leaf nodes are labeled by identifiers or variable names or constants. Generally leaves represent r-value.
2) Interior nodes store operator value.

The DAG and flow graphs are two different pictorial representations. Each node of the flow graph can be represented by DAG because each node of the flow graph is a basic block.

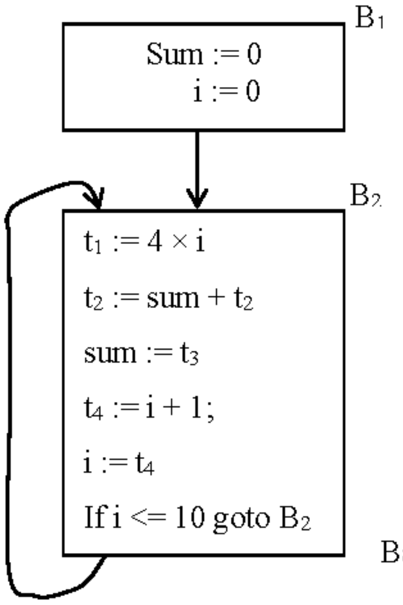
Ex: consider

```
sum = 0;
for (i=0; i <= 10; i++)
sum = sum + a[i];
```

The three address code for above code is:

1. sum := 0
2. i := 0
3. $t_1 := 4 \times i$
4. $t_2 :=$ sum $+ t_2$
5. sum := t_3
6. $t_4 :=$ i + 1;
7. i := t_4
8. If i <= 10 goto (3)

The above code partitioned into basic block as follows:

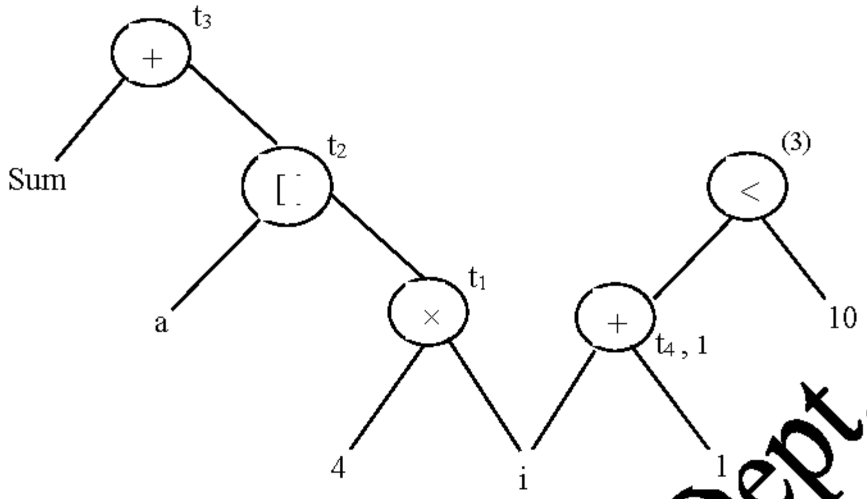


```
Sum := 0
    i := 0
```

```
t1 := 4 × i
t2 := sum + t2
sum := t3
t4 := i + 1;
i := t4
If i <= 10 goto B2
```

Basic Blocks

DAG representation for block 2 as follows:



DAG Based Local Optimization

A DAG can be constructed for a block and certain transformations such as common sub expression elimination: dead code elimination can be applied for performing the local optimization.

Ex: Construct the DAG for the following block

a := b * c

d := b

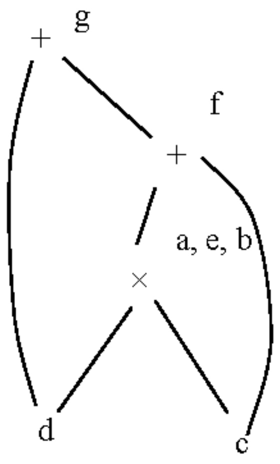
e := d * c

b := e

f := b + c

g := f + d

Diagram:



The optimized code can be generated by traversing the DAG. The local optimization on the above block can be done as:

1) A common sub expression e=d*c which is actually b*c (since d:=b) is eliminated
2) A dead code for b=e is eliminated

The optimized code and basic block is

a := b * c

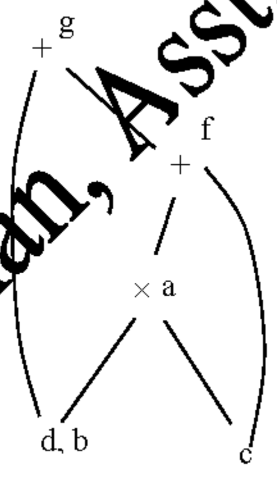
d := b

f := a + c

g := f + d

optimized code

Diagram:



@ GATE Previous questions

Q) Some code optimizations are carried out on the intermediate code because [2008-1M]

a) They enhance the portability of the compiler to other target processors.

b) Program analysis is more accurate on intermediate code than on machine code

c) The information form data flow analysis cannot otherwise be used for optimization

d) The information from the front end cannot otherwise be used for optimization

Q) Which of the following statements are TRUE? [2009-1M]

a) There exist parsing algorithms for some programming languages whose complexities are less than $\theta(n^3)$

b) A programming language which allows recursion can be implemented with static storage allocation.

c) No L-attribute definition can be evaluated in the framework of bottom-up parsing

d) Code improving transformations can be performed at both source language and intermediate code level

a) I and II b) I and IV c) III and IV d) I,III and IV