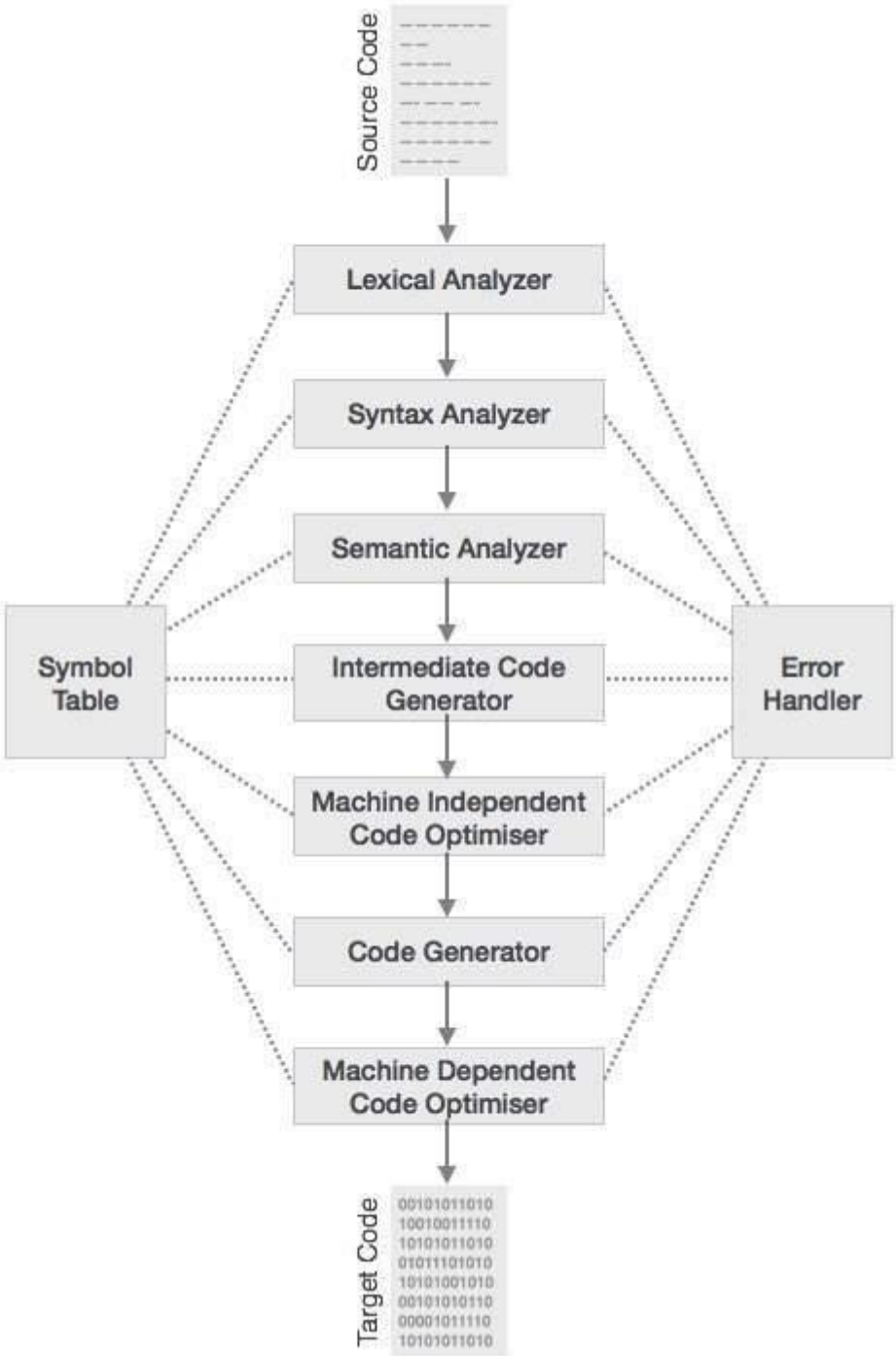
COMPILER DESIGN

Faculty Name: A.K.Rout Semester: 6th Module:1

Introduction: Phases of Compiler

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



Lexical Analysis

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

```
<token-name, attribute-value>
```

Syntax Analysis

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

Semantic Analysis

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

Intermediate Code Generation

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

Code Optimization

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

Code Generation

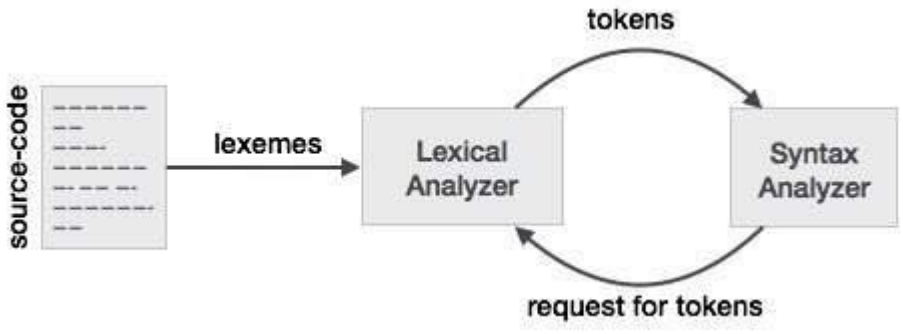
In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the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

Symbol Table

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

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Tokens

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

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

Deterministic Finite Automaton (DFA)

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton.

Formal Definition of a DFA

A DFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where −

- **Q is a finite set of states.**
- **∑ is a finite set of symbols called the alphabet.**
- **δ is the transition function where δ: Q × ∑ → Q**
- **q_0 is the initial state from where any input is processed (q_0 ∈ Q).**
- **F is a set of final state/states of Q (F ⊆ Q).**

Graphical Representation of a DFA

A DFA is represented by digraphs called state diagram.

- **The vertices represent the states.**
- **The arcs labeled with an input alphabet show the transitions.**
- **The initial state is denoted by an empty single incoming arc.**
- **The final state is indicated by double circles.**

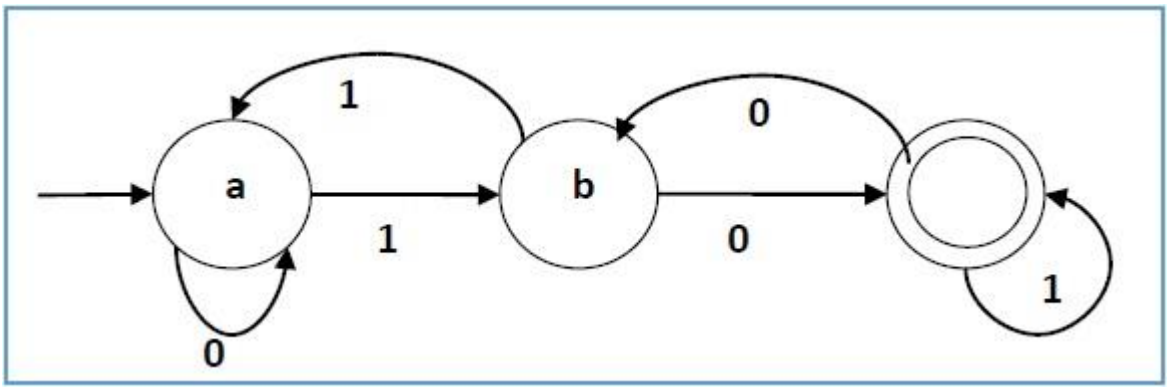
Example

Let a deterministic finite automaton be →

- **Q = {a, b, c},**
- **∑ = {0, 1},**
- **q_0 = {a},**
- **F = {c}, and**

Transition function δ as shown by the following table −

Present State	Next State for Input 0	Next State for Input 1
a	a	b
b	c	a
c	b	c



Non-deterministic Finite Automaton

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-deterministic Automaton. As it has finite number of states, the machine is called Non-deterministic Finite Machine or Non-deterministic Finite Automaton.

Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where −

- **Q is a finite set of states.**
- **∑ is a finite set of symbols called the alphabets.**
- **δ is the transition function where δ: Q × ∑ → 2^Q**

  (Here the power set of Q (2^Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)
- **q_0 is the initial state from where any input is processed (q_0 ∈ Q).**
- **F is a set of final state/states of Q (F ⊆ Q).**

Graphical Representation of an NDFA: (same as DFA)

An NDFA is represented by digraphs called state diagram.

- **The vertices represent the states.**
- **The arcs labeled with an input alphabet show the transitions.**
- **The initial state is denoted by an empty single incoming arc.**
- **The final state is indicated by double circles.**

Example

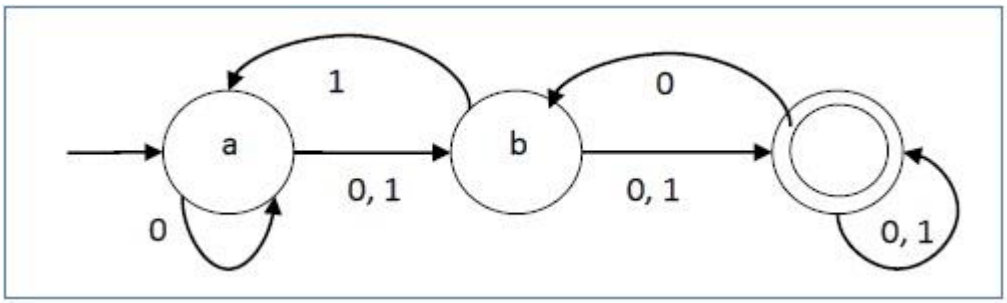
Let a non-deterministic finite automaton be →

- **Q = {a, b, c}**
- **∑ = {0, 1}**
- **q_0 = {a}**
- **F = {c}**

The transition function δ as shown below −

Present State	Next State for Input 0	Next State for Input 1
a	a, b	b
b	c	a, c
c	b, c	c

Its graphical representation would be as follows –



Regular Grammar:

A grammar is regular if it has rules of form A -> a or A -> aB or A -> ε where ε is a special symbol called NULL.

Type-3 grammars or regular grammar generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form X → a or X → aY

where X, Y ∈ N (Non terminal)

and a ∈ T (Terminal)

The rule S → ε is allowed if S does not appear on the right side of any rule.

Example

```
X → ε
X → a | aY
Y → b
```

Regular Expressions:
Regular Expressions are used to denote regular languages. An expression is regular if:

- **ϕ is a regular expression for regular language ϕ.**
- **ε is a regular expression for regular language {ε}.**
- **If a ∈ Σ (Σ represents the input alphabet), a is regular expression with language {a}.**
- **If a and b are regular expression, a + b is also a regular expression with language {a,b}.**
- **If a and b are regular expression, ab (concatenation of a and b) is also regular.**
- **If a is regular expression, a* (0 or more times a) is also regular.**

Regular Languages : A language is regular if it can be expressed in terms of regular expression.

Design of a LA Generator:

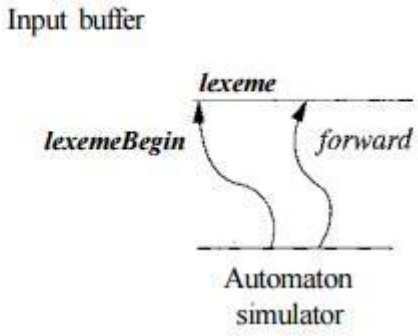
Two approaches:

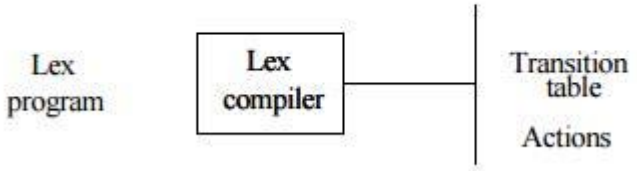
NFA-based

DFA-based

The Lex compiler is implemented using the second approach.

Generated LA





A Lex program is turned into a transition table and actions, which are used by a finite-automaton simulator

Syntax Analysis

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

CFG, on the other hand, is a superset of Regular Grammar, as depicted below:

It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

Context-Free Grammar

n this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.

A context-free grammar has four components:

- **A set of non-terminals (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.**
- **A set of tokens, known as terminal symbols (Σ). Terminals are the basic symbols from which strings are formed.**
- **A set of productions (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.**
- **One of the non-terminals is designated as the start symbol (S); from where the production begins.**

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

Example

We take the problem of palindrome language, which cannot be described by means of Regular Expression. That is, L = { w | w = w^R } is not a regular language. But it can be described by means of CFG, as illustrated below:

$G = (V, \Sigma, P, S)$

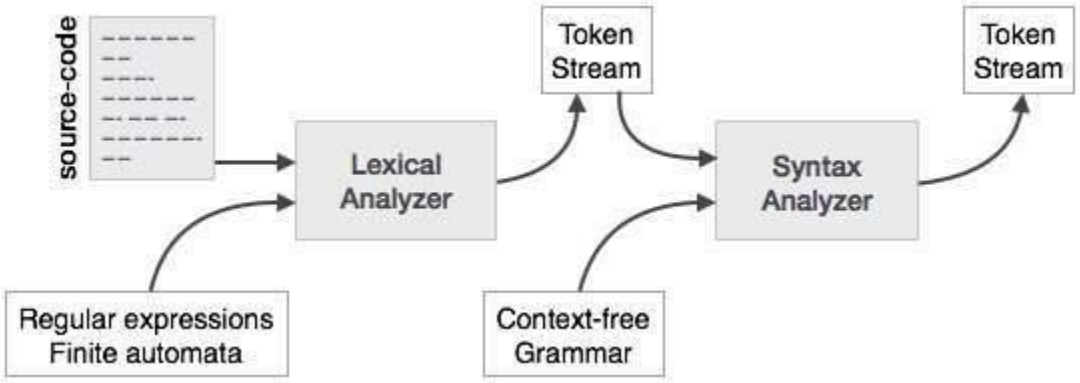
Where:

$V = \{ Q, Z, N \}$
$\Sigma = \{ 0, 1 \}$
$P = \{ Q \rightarrow Z \mid Q \rightarrow N \mid Q \rightarrow \varepsilon \mid Z \rightarrow 0Q0 \mid N \rightarrow 1Q1 \}$
$S = \{ Q \}$

This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc.

Syntax Analyzers

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.



This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- **Deciding the non-terminal which is to be replaced.**
- **Deciding the production rule, by which, the non-terminal will be replaced.**

To decide which non-terminal to be replaced with production rule, we can have two options.

Left-most Derivation

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

Right-most Derivation

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Example

Production rules:

```
E → E + E
E → E * E
E → id
```

Input string: id + id * id

The left-most derivation is:

```
E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id
```

Notice that the left-most side non-terminal is always processed first.

The right-most derivation is:

```
E → E + E
E → E + E * E
E → E + E * id
```

```
E → E + id * id
E → id + id * id
```

Parse Tree

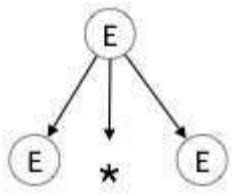
A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

We take the left-most derivation of a + b * c

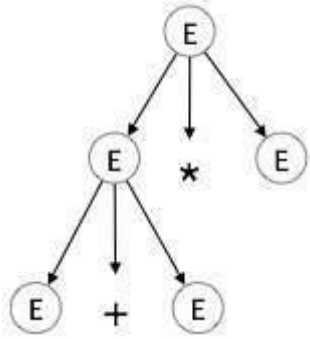
The left-most derivation is:

```
E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id
```

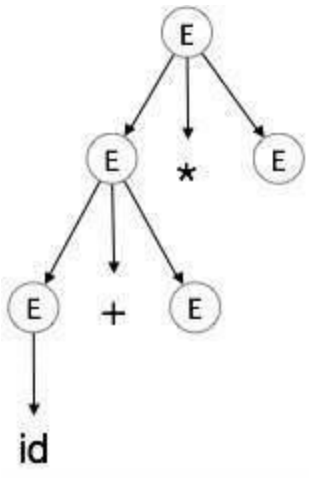
Step 1: E → E * E



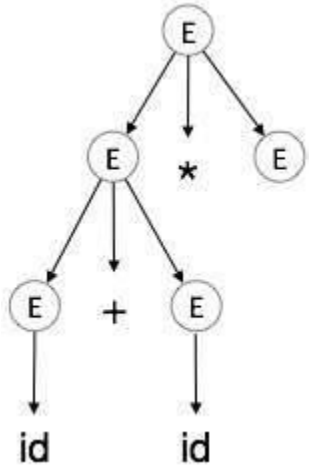
Step 2: E → E + E * E



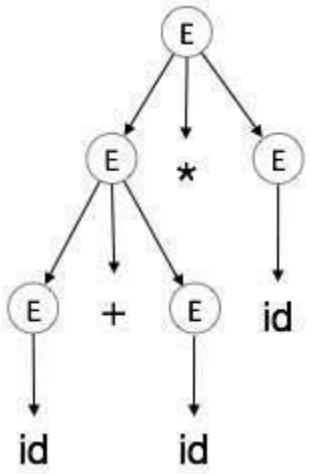
Step 3: E → id + E * E



Step 4: E → id + id * E



Step 5: E → id + id * id



In a parse tree:

- **All leaf nodes are terminals.**
- **All interior nodes are non-terminals.**
- **In-order traversal gives original input string.**

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

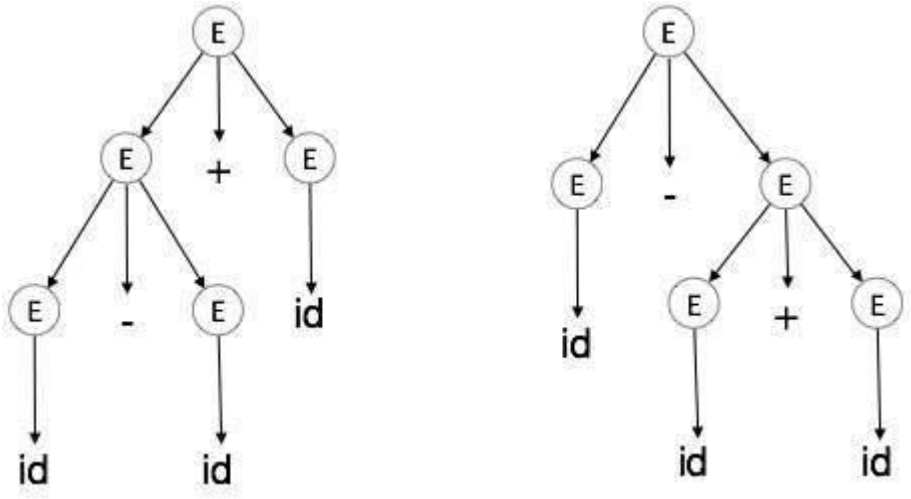
Ambiguity

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

Example

```
E → E + E
E → E – E
E → id
```

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



The language generated by an ambiguous grammar is said to be inherently ambiguous. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

Left Recursion

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

Example:

```
(1) A => Aα | β

(2) S => Aα | β
    A => Sd
```

(1) is an example of immediate left recursion, where A is any non-terminal symbol and α represents a string of non-terminals.

(2) is an example of indirect-left recursion.

Removal of Left Recursion

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

The production

```
A => Aα | β
```

is converted into following productions

```
A => βA'
A'=> αA' | ε
```

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

Second method is to use the following algorithm, which should eliminate all direct and indirect left recursions.

Left Factoring

If more than one grammar production rules has a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand.

Example

If a top-down parser encounters a production like

$A \Longrightarrow \alpha\beta \mid \alpha\gamma \mid \ldots$

Then it cannot determine which production to follow to parse the string as both productions are starting from the same terminal (or non-terminal). To remove this confusion, we use a technique called left factoring.

Left factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions.

Example

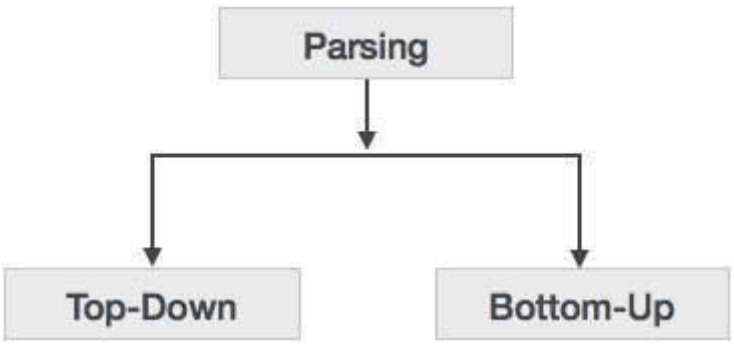
The above productions can be written as

```
A => αA'
A'=> β | γ | ...
```

Now the parser has only one production per prefix which makes it easier to take decisions.

Top Down Parsing:

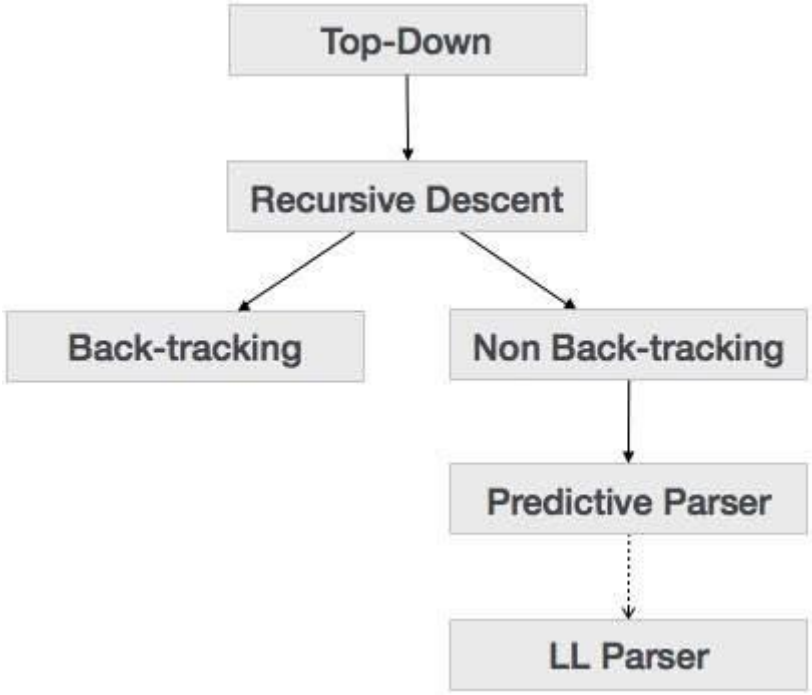
Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types : top-down parsing and bottom-up parsing.



Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

- **Recursive descent parsing : It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.**
- **Backtracking : It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.**



Recursive Descent Parsing

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

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

Back-tracking

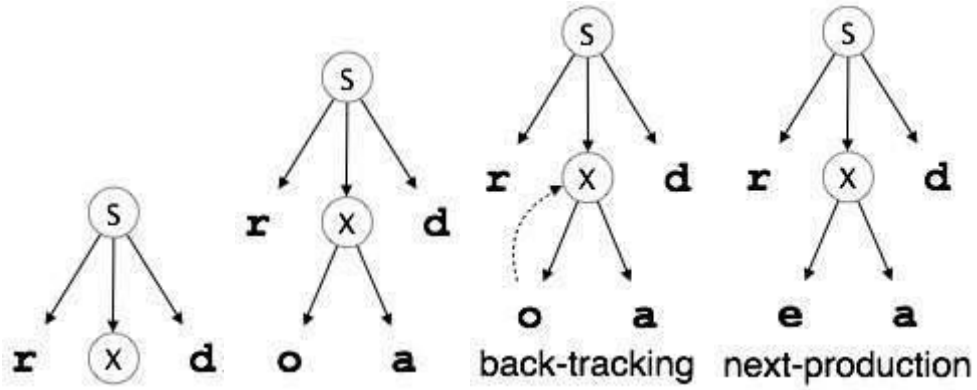
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

```
S → rXd | rZd
X → oa | ea
Z → ai
```

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

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

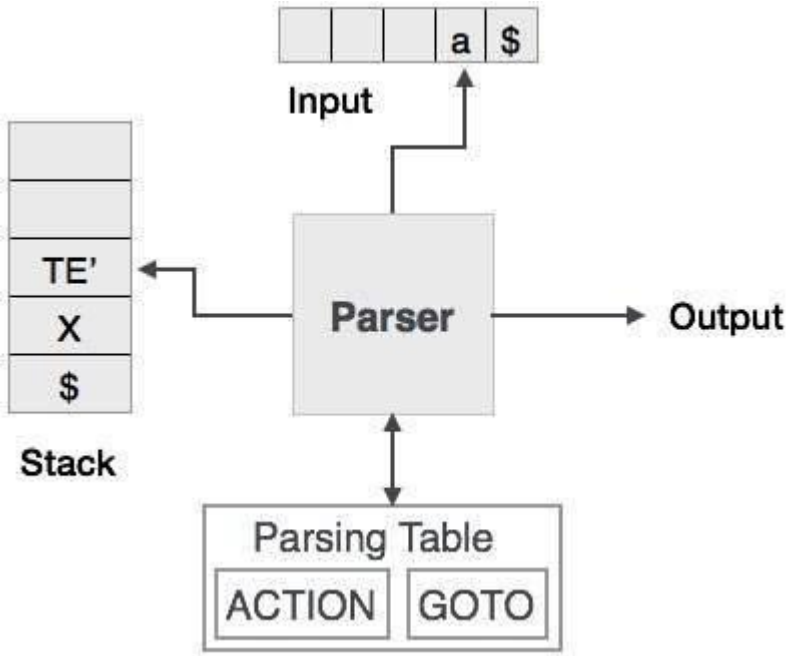
Now the parser matches all the input letters in an ordered manner. The string is accepted.



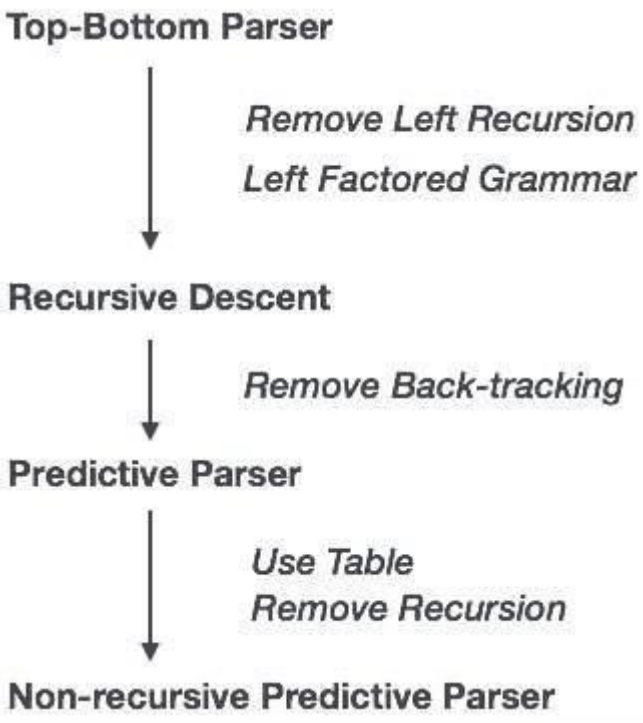
Predictive Parser

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

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



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

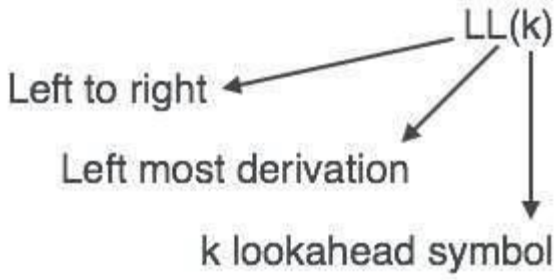


In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure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

LL Parser

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

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

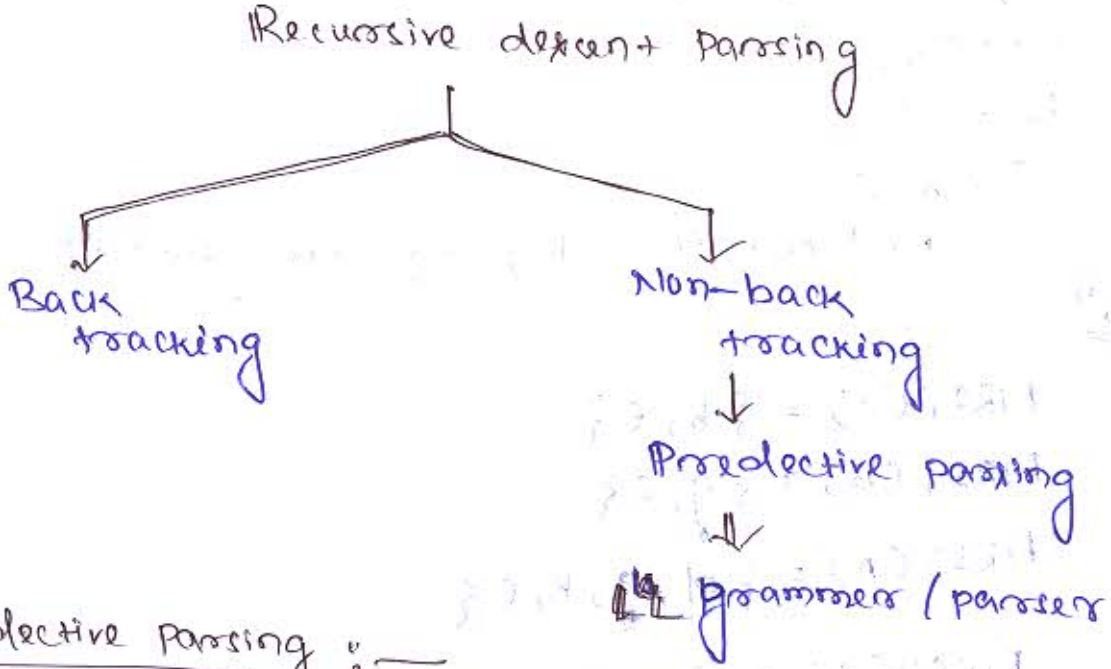


A grammar G is LL(1) if A → α | β are two distinct productions of G:

01/02/22

Top down parsing :—

Deriving the input / string from the starting symbol or root node of the derivation tree from a left most derivation. scanning from left to right.



Predective Parsing :—

For predective parsing 2 elements are used.

(i) FIRST
(ii) FOLLOW

FIRST :—

If there is a production rule $\alpha \rightarrow XYZ$ then,

$FIRST(\alpha) = \{ x \mid x \text{ is a terminal symbol} \}$

or

$FIRST(\alpha) = FIRST(XYZ) = FIRST(X)$, if X is a non-terminal and doesn't contain ϵ.

else (if X contain ϵ)

then, $FIRST(\alpha) = FIRST(X) - \{\epsilon\} \cup FIRST(YZ)$

N/B

calculate FIRST of any symbol from bottom to top of the grammer.

e.g:-

① $S \to ACB / CbB / Ba$

$A \to da / BC$

$B \to g / \epsilon$

$C \to h / \epsilon$

Find out the FIRST of these grammer.

Sol:-

$FIRST(C) = \{h, \epsilon\}$

$FIRST(B) = \{g, \epsilon\}$

$FIRST(A) = \{d, g, h, \epsilon\}$

$FIRST(S) = \{d, g, h, \epsilon\} \cup \{h, b, g, \epsilon\} \cup \{g, a\}$

$= \{a, b, d, g, h, \epsilon\}$

02/02/21

FOLLOW :-

Rule

① If S is a starting symbol of the grammer then FOLLOW (S) = $

② If there is a production rule of the form $A \to \alpha x \beta$

FOLLOW (x) = FIRST(β), if it does n't contain any ε.

else (If FIRST(β) contain ε)

FOLLOW (x) = FIRST(β) − {ε} ∪ FOLLOW(A)

Always calculate FOLLOW top to bottom -

e.g-

$E \rightarrow TA$

$T \rightarrow +TA / \epsilon$

$T \rightarrow FB$

$B \rightarrow *FB / \epsilon$

$F \rightarrow (E) / id$

Sol?

$FIRST(F) = FIRST\{(E)\} \cup FIRST\{id\}$

$= \{(\} \cup \{id\}$

$= \{(, id\}$

$FIRST(B) = \{*, \epsilon\}$

$FIRST(T) = FIRST(FB)$

$= FIRST(F)$

$= \{(, id\}$

$FIRST(T) = \{+, \epsilon\}$

$FIRST(E) = FIRST(T)$

$= \{+, \epsilon, id\}$

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

$FOLLOW(T) = \{(, id\}$

$FOLLOW(A) = \{\$,), (, id\}$

$FOLLOW(B) = \{\$,), (, id\}$

$FOLLOW(F) = \{*, \$,), (, id\}$

NB

FOLLOW doesn't contain any ϵ

Q:- $D \rightarrow type\ list\ ;$

$list \rightarrow id\ tlist\ ;$

$tlist \rightarrow , id\ tlist / \epsilon$

$type \rightarrow int / float$

Sol?

$FIRST(type) = \{int, float\}$ $FIRST(D) = \{int, float\}$

$FIRST(tlist) = \{,, \epsilon\}$

$FIRST(list) = \{id\}$

FOLLOW (D) = {$}
FOLLOW (list') = {;}
FOLLOW (+list') = {;}
FOLLOW (type) = {id}

Q:- $S_1 \rightarrow S\#$
$S \rightarrow qABC$
$A \rightarrow a / bbD$
$B \rightarrow a / \epsilon$
$C \rightarrow b / \epsilon$
$D \rightarrow c / \epsilon$

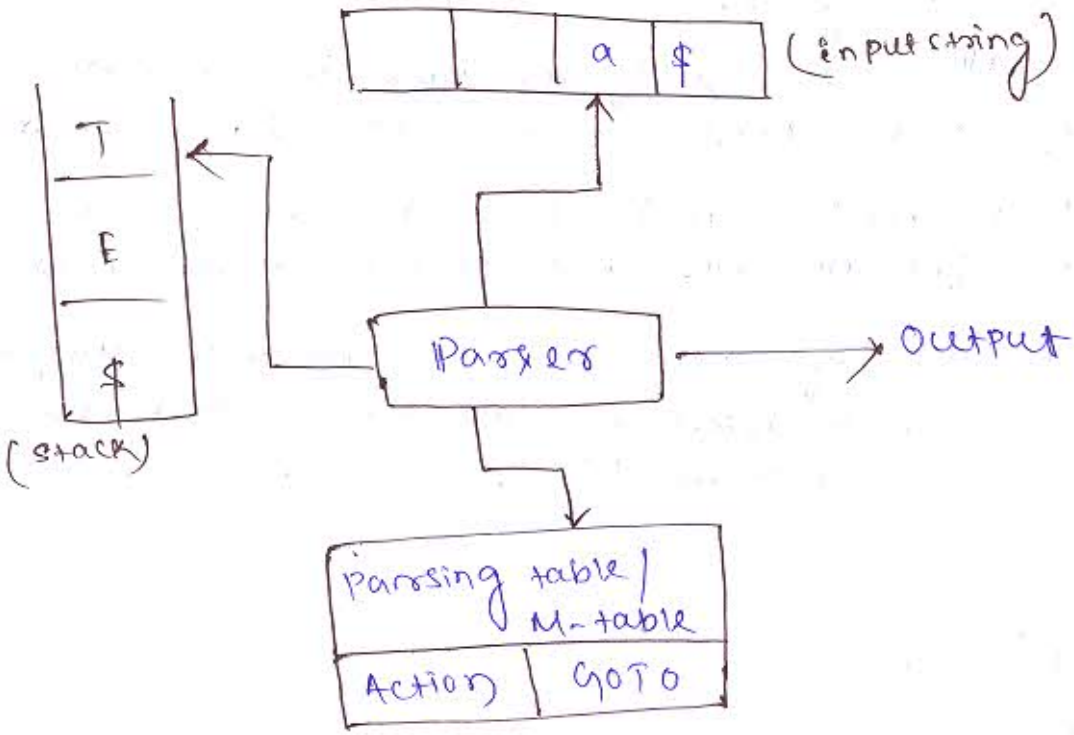
03/02/24

Predictive Parsing :—

(Non-recursive parsing without backtracking).

→ It predicts the production rule which can replace the input string.

→ The technique used for prediction is done through look ahead symbol by using a look ahead pointer of the parser.



LL parser :—

It performed on LL grammer and LL grammer is a subset of context free grammers with some restrictions.

LL(K) → no. of look ahead symbols
↓ → leftmost derivation
left to right scanner

→ A Grammer is said to be LL(K) where K=1.
→ If and only if
for a production $A \to \alpha / \beta$ there are 2 distinct productions of A.

To check a Grammer of LL(1) or not :-

Step

1. Find out the FIRST and FOLLOW of the grammer.
2. Create the parsing table or M-table.
   → The 1st column of the table containing all the non-terminal symbols.
   → The 1st raw of the table containing all the terminal symbols including $.
   → Fill other cells of the table by calculating the FIRST of there respective non-terminals without any
   → If ϵ present then Find the FOLLOW of non-terminal and fill the cells with respective productions
3. If all the cells of the table are filled with unique production rule without multiple entry then the grammer is said to be LL(1).

Q:- $E \to TE'$

$E' \to +TE' / \epsilon$

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to (E) / id$ check the grammer LL(1) or not.

Sol:

FIRST(F) = { (, id }

FIRST(T') = { *, ϵ }

FIRST(T) = { (, id }

FIRST(E') = { +, ϵ }

FIRST(E) = { (, id }

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

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

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

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

M-table / parsing table

	+	*	(	)	id	$
E			$E \to (E)$		$E \to id$	
E'	$E' \to +TE'$			$E' \to \epsilon$		$E' \to \epsilon$
T			$F \to (E)$		$F \to id$	
T'	$T' \to \epsilon$	$T \to *FT'$		$T' \to \epsilon$		$T' \to \epsilon$
F			$F \to (E)$		$F \to id$	

Hence, the parsing table doesn't entry multiple production rule in any ~~etc~~ cell then it is known as LL(1) grammer.

G// (1) $S \to A / a$
$A \to a$

(2) $S' \to AB / eDa$
$A \to ab / c$
$B \to dc$
$C \to eC / \epsilon$
$D \to \$D / \epsilon$ Check the grammar LL(1) or not.

Soln (1) $S \to A / a$
$A \to a$

~~FOLLOW(S) = { $ }~~
FOLLOW(A) = { $ }

$FIRST(A) = \{a\}$
$FIRST(S) = \{a\}$

	a	$
S	$S \to a$ $S \to A$	
A	$A \to a$	

So it is not a LL(1) grammer. Because it ~~does~~ cells contain multiple production rule.

04/02/22

Predictive Parsing Algorithm :—

→ Set the input pointer at the first symbol of input string w$.

→ Let 'x' be the stack top symbol and 'a' is the first symbol of input string.

→ If x = a = $, then parser halts and announces the successful completion of parsing.

→ If x = a ≠ $, then parser pop x from the top of the stack and advance the input pointer to the next symbol of input string.

→ If x is a non-terminal, then check the M-table for entry M[x, a], if there is a production $x \rightarrow Y_1 Y_2 Y_3 \cdots Y_k$ then pop x from the stack and push $Y_k, Y_{k-1}, \cdots Y_2$ to the stack.

e.g —

D → type list ;
list → id tlist ;
tlist → , id tlist / ∈
type → int / float
w = int id , id ;

parse the input string using predictive parsing algorithm.

FIRST(type) → {int, float}
FIRST(tlist) → {, , ∈}
FIRST(list) → {id}
FIRST(D) → {type int, float}

FOLLOW(D) → {$}
FOLLOW(list) → {;}
FOLLOW(tlist) → {;}
FOLLOW(type) → {id}

	id	;	,	int	float	$
D				D→type list ; type→int	D→type list ; type→float	
list	list→id tlist					
tlist		tlist→ϵ	tlist→, id tlist			
type				type→int	type→float	

Stack	Input	Action
$ D	int id , id ; $ ↑	D→type list ; ~~type→int~~
$ ~~int~~ ; list type ↑	int id , id ; $ ↑	~~pop int~~ type→int
$; list int	int id , id ; $ ↑	pop int
$; list	id , id ; $ ↑	list→id tlist
$; tlist id ④	id , id ; $ ↑	pop id
$; tlist	, id ; $ ↑	tlist→, id tlist
$; tlist id , ④	, id ; $ ↑	pop ,
$; tlist id ④	id ; $ ↑	pop id
$; tlist ④	; $ ↑	tlist→ϵ
$; ϵ $	; $ $	pop ; Accept

e.g.

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC/\epsilon$

$D \rightarrow EF$

$E \rightarrow g/\epsilon$

$F \rightarrow f/\epsilon$

$w : acbgfh$

$w : acbh$

Parse the input string using predictive parsing algorithm

05/02/26

Bottom-up Parsing :—

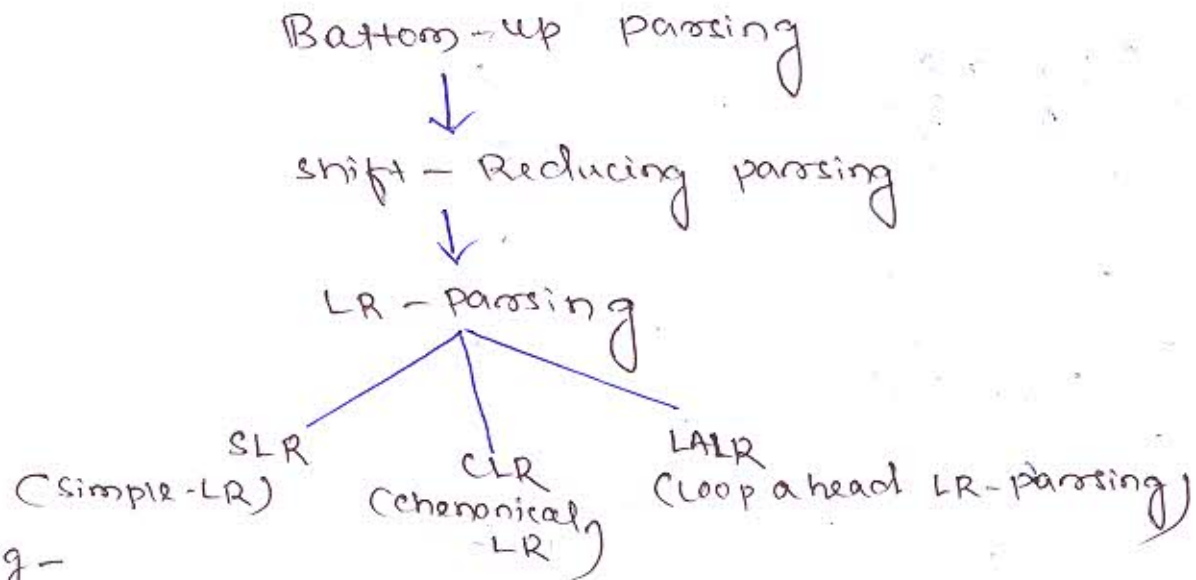
The process of reducing the given input string into its starting non-terminal symbol by using right most derivation in reverse order.

Handel —

In the deriving process when a sub-string matches with the right hand side production rule and any be replaced by a left hand side non-terminal then that sub-string is known as handel.

Reduction —

When a substring of the production rule is replaced by a left hand side of the production rule as a single non-terminal that procedure is known as reduction.



e.g-

Find the handles of the string (a,(a,a)) from the given grammer

$S \rightarrow (L) / a$

$L \rightarrow L,S / S$

Soln

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

Right sential form	Handel
(a,(a,a))	a
(S,(a,a))	S
(L,(a,a))	a
(L,(S,a))	S
(L,(L,a))	a
(L,(L,S))	L,S
(L,(L))	L
(L,S)	L,S
(L)	S

$E \rightarrow E+E / E*E / id$

$w = id+id*id$ handle the string from the given grammar

08-02-21

Shift reduce Parsing :-

(i) Shift
(ii) Reduce
(iii) Accept

Shift :-

The parser keeps moving all the symbols that is present in the input buffer to the stack one at a time.

Reduce :-

Once a handle is found on the top of the stack then it reduces it's left hand side non-terminal symbol.

Accept :-

The parser declares the successful completion of parsing when it encounters the starting symbol at the stack and $ at the input buffer.

e.g. —

Q// $E \to E+E \mid E*E \mid (E) \mid a \mid b \mid c$

$w = a * (b+c)$ Parse the string using shift reduce algorithm

Soln

$E \to \underline{E * E}$
$\to E * \underline{(E)}$
$\to E * (\underline{E+E})$
$\to E * (E + \underline{c})$
$\to E * (\underline{b} + c)$
$\to \underline{a} * (b+c)$

Right sential form	Handle
$a * (b+c)$	a
$E * (b+c)$	b
$E * (E+c)$	c
$E * (E+E)$	E+E
$E * (E)$	(E)
$E * E$	E*E
E	

Stack	input	Action
a	a*(b+c)$	shift a
E	*(b+c)$	Reduce E→a
E*	(b+c)$	shift *
E*(	b+c)$	shift (
E*(b	+c)$	Reduce E→b
E*(E	+c)$	shift +
E*(E+	c)$	shift (
E*(E+c	)$	Reduce E→(
E*(E+E	)$	Reduce E+E→E
E*(E	)$	shift)
E*(E)	$	Reduce (E)→E
E*E	$	Reduce E→E*E
E	$	Accepted

Q!- E → E+T / T
T → T*F / F
F → (E) / y

w = y+y+(y*y)

parse the string using shift reduce parsing algorithm.

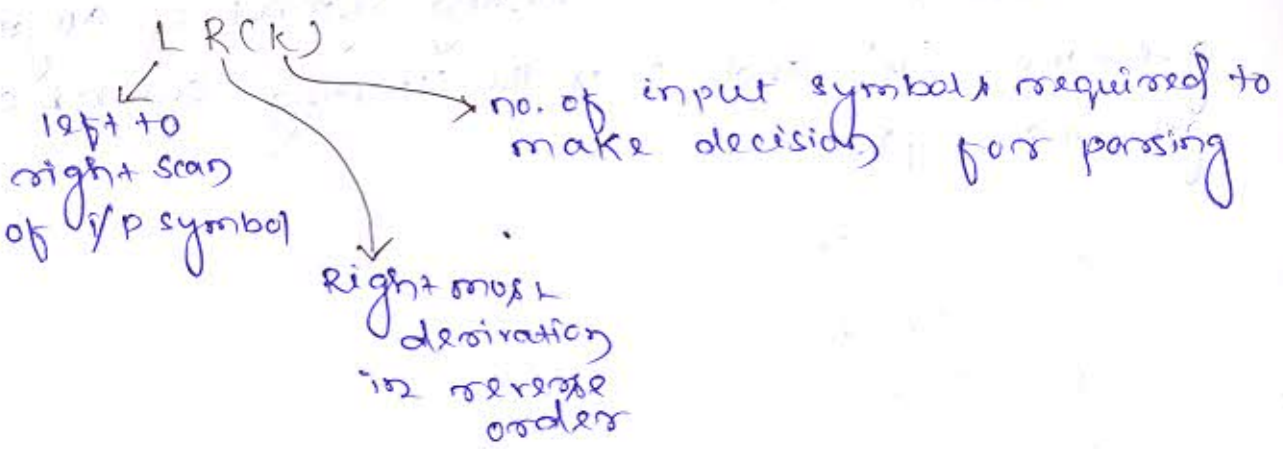
E → E+T
→ E+F
→ E+(E)
→ E+(T)
→ E+(T*F)
→ E+(T*y)
→ E+(F*y)
→ E+(y*y)
→ E+T+(y*y)
→ E+F+(y*y)
→ E+y+(y*y)
→ T+y+(y*y)
→ F+y+(y*y)
→ y+y+(y*y)

Drawback of LR :-

① Shift reduce conflict
② Reduce reduce conflict

09/02/22

LR(K) - Parsing :-



SLR(1) (Simple LR parsing)

LR(0) items :-

If a grammer G contains $A \to XYZ$ then LR(0) for this production is

$A \to \cdot XYZ$
$A \to X \cdot YZ$
$A \to XY \cdot Z$
$A \to XYZ \cdot$

e.g. $A \to \epsilon$
$A \to .$

10/02/22

SLR :-

1. LR(0) items
2. Augmented Grammar
3. Closure
4. Go to

Augmented Grammar

Augmented Grammar G' of a grammar G contains a production rule along with all other productions of the grammar G i.e $S' \rightarrow .S$

Where S' is the starting symbol of augmented grammar (G') and S is the starting symbol of grammar (G).

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

$G' = E' \rightarrow .E$

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

Closure

Closure of an item I i.e, closure (I) can be found out by the following rules

1. An item I is closure of itself.
2. $A \rightarrow \alpha . B\beta$ in closure (I) and there exist a production $B \rightarrow \gamma$, the add $B \rightarrow .\gamma$ to the closure (I)
3. Repeat step-2 until no more new items are found.

Closure $(E' \rightarrow .E)$

$E' \rightarrow .E$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E) / .id$

Go to (I, X)

Where I is an item set and X is any grammar symbol (may be terminal or non-terminal)

GOTO(I, X) = closure ($A \to \alpha X . B$)
if there is a production rule present as -
$A \to \alpha . XB$ in the item set I.

GOTO(I_0, C)

$F \to .(E)$

Closure ($F \to (.E)$)

$F \to (.E)$
$E \to .E+T / .T$
$T \to .T * F / .F$
$F \to .(E) / .id$

Q) $S \to AA$
$A \to aA / d$ Find out the canonical LR(0) item set for the above grammar.

Soln

Augmented
$S' \to .S$
$S \to AA$
$A \to aA / d$

closure ($S' \to .S$)

$S' \to .S$
$S \to .AA$
$A \to .aA / .d$] I_0

GOTO (I_0, S)

closure ($S' \to S.$)

$S' \to S. \; 3I_1$

GO TO (I_0, A)

closure (S → A.A)

S → A.A

A → .aA / .d } I_2

GOTO (I_0, a)

closure (A → a.A)

A → a.A

A → .aA / .d } I_3

GOTO (I_0, d)

closure (A → d.)

A → d. } I_4

GOTO (I_2, A)

closure (A → AA.)

A → AA. } I_5

GOTO (I_2, a)

closure (A → a.A)

A → a.A

A → .aA / .d } I_3

GOTO (I_2, d)

closure (A → d.)

A → d. } I_4

GOTO (I_3, A)

closure (A → aA.)

A → aA. } I_6

GOTO (I_3, a)

closure (A → a.A)

A → a.A

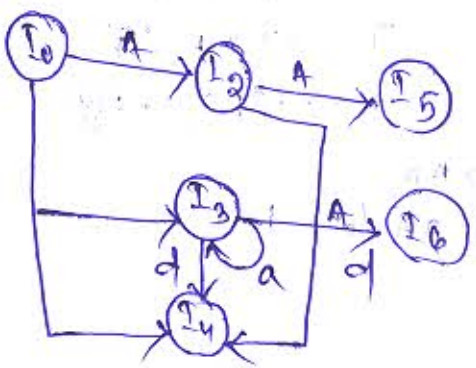
A → .aA / .d } I_3

GOTO (I_3, d)

closure (A → d.)

A → d. } I_4

C = { I_0, I_1, I_2, I_3, I_4, I_5, I_6



$S \to BB$
$B \to aB \mid b$

11/02/22

Steps to construct SLR - Parsing table :—

1. Construct cannonical item sets $C = \{I_0, I_1, I_2 \dots I_n\}$ for augmented grammar G'.

2. Construct i from state I_i
   → If there is a production rule -
   $A \to \alpha . a\beta$ in I_i, then
   Find GoTo $(I_i, a) = I_j$
   Set Action (i, a) = Shift j where a = terminal symbol
   if 'a' is a non-terminal. set Action $(i, a) = j$

3. If there is a product
   $A \to \alpha .$ in I_i, where α may be a terminal or non-terminal.

   Set Action (i, a) = reduce $A \to \alpha$, $\forall \alpha \in$ FOLLOW(A)
   Where A is not the starting symbol.

4. If A is the starting symbol of augmented grammar.
   Set Action $[i, a]$ = Accept

eg:-

$S \to BB$

$B \to aB/b$ Find out LR(0) item set and construct the SLR-parsing tab

Sol:-

Augmented grammer:-

$S' \to .S$

$S \to BB$

$B \to aB/b$

closure($S' \to .S$)

$S' \to .S$
$S \to .BB$
$B \to .aB/.b$] I_0

GoTo (I_0, S) closure ($S' \to S.$)

$S' \to S.$ { I_1

GoTo (I_0, B) ~~closure ($S \to B.B$)~~

~~$B \to$~~

$S \to B.B$
$B \to .aB/.b$ } I_2

GoTo (I_0, a)

$B \to a.B$
$B \to .aB/.b$ } I_3

GoTo (I_0, b)

$B \to b.$ { I_4

GoTo (I_2, B)

$S \to BB.$ } I_5

~~B~~

GOTO (I_2, a)

$B \to a \cdot B$
$B \to \cdot aB / \cdot b$ } I_3

GOTO (I_2, b)

$B \to b \cdot$ } I_4

GOTO (I_3, B)

$B \to aB \cdot$ } I_6

GOTO (I_3, a)

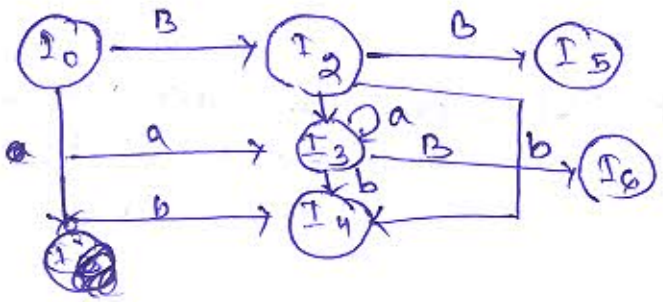
$B \to a \cdot B$
$B \to \cdot aB / \cdot b$ } I_3

GOTO (I_3, b)

$B \to b \cdot$ } I_4

GOTO

$C = \{I_0, I_1, I_2, I_3, I_4, I_5, I_6\}$



State	Action			GoTo	
	a	b	$	S	B
I_0	shift 3/ S3	shift 4/ S4		~~shift~~ 1	2
I_1			Accept	~~Accept~~	
I_2	shift 3/ S3	shift 4/ S4			5
I_3	shift 3/ S3	shift 4/ S4			6
I_4	Reduce 3/ r3	Reduce 3/ r3			
I_5			r1		
I_6	r2				

Q:- E → E+T / T

T → T*F / F

F → (E) / id Find out LR(0) item set and construct the SLR-parsing table.

12/02/21

C-L-R - Parsing :—

$$A \to \alpha \cdot B\beta, a$$

→ All the prosidure of CLR parsing are equivalent SLR parsing exceptes the following methods.

GoTo

If a production $[A \to \alpha \cdot B\beta, a]$ is present in item set I and their is a production $B \to \gamma$ present in grammar G.

then add $[B \to \cdot\gamma, b]$ to the item set I.

where $b = FIRST(\beta, a)$

Closure :

Find out the closure of the Starting symbol of Augmented Grammar G' as $[S' \to \cdot S, \$]$, and name the item set as I_0.

Find the chenonical LR(1) items for the given grammar.

$S \to CC$

$C \to cC / d$

Soln

Augmented grammer.

$S' \to .S$

$S \to CC$

$C \to cC / d$

closure $(S' \to \cdot S, \$)$

$S' \to \cdot S, \$$

$S \to \cdot CC, \$$

$C \to \cdot cC, c/d$

$C \to \cdot d, c/d$

I_0

GoTo (I_0, S)

closure $S' \to S.,\$$

$S' \to S.,\$$ } I_1

GoTo (I_0, C)

$S \to C.C,\$$
$C \to .cC,\$$
$C \to .d,\$$
} I_2

GoTo (I_0, c)

$C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d, c/d$
} I_3

GoTo (I_0, d)

$C \to d., c/d$ } I_4

GoTo (I_2, C)

$S \to CC., \$$ } I_5

GoTo (I_2, c)

$C \to c.C, \$$
$C \to .cC, \$$
$C \to .d, \$$
} I_6

GoTo (I_2, d)

$C \to d., \$$ } I_7

GoTo (I_3, C)

$C \to cC., c/d$ } I_8

GoTo (I_3, c)

$C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d, c/d$
} I_3

GoTo (I_3, d)

$C \to d., c/d$ } I_4

GoTo (I_6, C)

$C \to cC., \$$ } I_9

GoTo (I_6, c)

$C \to c.C, \$$
$C \to .cC, \$$
$C \to .d, \$$
} I_6

GoTo (I_6, d)

$C \to d., \$$ } I_7

State	Action			goto	
	c	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

18-02-22

LALR - Parsing :-

→ LALR parsing is same as C-L-R parsing while finding LR(1) item sets.

→ But while constructing the LALR parsing table check the item sets / state having same ~~code~~ core may differ in look ahead symbols will be treated as single state or item sets.

e.g.-

$S \to AA$

$A \to aA \mid b$ construct the LALR parsing table by finding the LR(1) items.

Sol^n

Augmented grammer

$S' \to .S$

$S \to AA$

$A \to aA / b$

closure $(S' \to .S)$

$S' \to .S, \$$
$S \to .AA, \$$
$A \to .aA, a/b$
$A \to .b, a/b$] I_0

GOTO (I_0, S)

$S' \to S., \$$] I_1

GOTO (I_0, A)

$S \to A.A, \$$
$A \to .aA, \$$ ~~a/b~~
$A \to .b, \$$] I_2

GOTO (I_0, a)

$A \to a.A, \$ / a/b$
$A \to .aA, a/b$
$A \to .b, a/b$] I_3

GOTO (I_0, b)

$A \to b., a/b$] I_4

GOTO (I_2, A)

$S \to AA., \$$] I_5

GOTO (I_2, a)

$A \to a.A, \$$
$A \to .aA, \$$
$A \to .b, \$$] I_6

Goto (I_2, b)

$A \to b.\ , \$]\ I_7$

Goto (I_3, A)

$A \to aA.\ , a/b]\ I_8$

Goto (I_2, a)

$A \to a.A, a/b$
$A \to .aA, a/b$
$A \to .b, a/b$] I_3

Goto (I_3, b)

$A \to b.\ , a/b]\ I_4$

Goto (I_6, A)

$A \to aA.\ , \$]\ I_9$

Goto (I_6, a)

$A \to a.A\ , \$$
$A \to .aA, \$$
$A \to .b, \$$] I_6

Goto (I_6, b)

$A \to b.\ , \$]\ I_7$

$C = \{I_0, I_1, \ldots, I_9\}$

state	Action			goto	
	a	b	$	S	A
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		

LALR Parsing Algorithm :—

Steps

1. Construct the chenonical LR(1) item set for the grammer.

2. If S is at the top of the stack symbol and a is a symbol pointed by the input pointer.

→ If Action [S, a] = shift j
then First push a, to the stack then push shift j to the stalk and move input pointer to the next sym

→ If Action [S, a] = reduce, such that A → r, pop 2x|r| from the top of the stack.

→ If S' is the top stack symbol and 'A is a non-terminal then push A and push S'.

→ If Action [S, a] = Accept, then return Parse successful.

Q. w = aabb check parse the string using LALR parsing alg

Stack	input	Action
S_0	a b b $ ↑	push a push S_{36}
S_0 a S_{36}	b b $ ↑	push b push S_{47}
S_0 a S_{36} b S_{47}	b $	A → b 2x|r| ②
S_0 a S_{36} A S_{89}		

19-02-21

Error Reporting and Recovery

Error

- Compile time errors
  - → Lexical phase error
  - → Syntax / Syntatic phase error
  - → Semantic phase error
- Run time error

① Error Detection
② Reporting error
③ Recovery

Lexical phase error :-

→ Exceeding input length
→ unmatched string
→ Any ~~~~ unnecessary ~~ illegal symbols.

e.g. -

```
printf ("SRINIX"); /*
printing the output */
```

Panic Mode recovery

→ Successive removal of inputs ~~~ one at a time until reaching to a specified / designated synchronized token

Advantage

Simple & easy to implement.

Disadvantage

It removes the input without forecasting the future errors may, come in next phase.

Syntatic phase error :-

→ structural error
→ missing operators
→ missing balanced parenthesis
→ Misspeced keywords

(i) panic mode recovery
(ii) Statement mode recovery
(iii) Error production
(iv) Global correction

Module-02

22/12/21

Intermediate Code Generation :-

→ Intermediate code is close to the target machine as compaired to source language.
→ It is machine independent and easily retarget the machine code.
→ Variaes optimization technique can implemented on intermediate code.

How to form intermediate code?

(1) Three address code (TAC)
(2) Abstract Syntax Tree (AST)
(3) postfix notation

Three Address Code (TAC) :—

It is a statement or expression where only One operator is allow at the right hand side (RHS)

Arithmatic Expression

① $x = y \text{ op } z$

② $x = \text{op } y$

③ $x = y$

Destination	Source-1	Source-2	Operators
x	y	z	op
x	-	y	op
x	y	-	-

e.g. -

$x = y + z$

$x = +z$

$x = y$

Jumping Statement :—

Conditional Jumping —

if (condition) Goto L, where L is a label.

unconditional

Goto L

23|02|22

①
```
i = 1
Sum = 0;
do
{
Sum = Sum + i * 5;
    i++;
} while (i<10);
```

②
```
if (x>y)
{
 while (x>d)
   a = a+b;
}
else
{
 do
 {
  p = p+q;
 }
 while (e<=f)
}
```

Soln

① if (x>y) goto 3
② goto 7
③ if (x>d) goto 5
④ goto 10
⑤ a = a+b
⑥ goto 3
⑦ p = p+q
⑧ if (e<=f) goto 7
⑨ goto 10
⑩ Exit

③
```
for (i=0; i<5; i++)
    x = y+2;
```

Ans:-

① i = 0
② if (i < 5) goto 4
③ goto 7
④ x = y+2
⑤ i = i+1
⑥ goto 2
⑦ Exit

④
```
switch (a+b)
{
case 1 : c = a+b*2;
         break;
case 2 : c = a+b;
         a = a-2;
         break;
case 3 : c = a;
         break;
default : c = a-2*b;
}
```

① t = a+b
② if (t = 1) goto 8
③ if (t = 2) goto 10
④ if (t = 3) goto 12
⑤ goto 13
⑥ t_1 = 2*b
⑦ c = a - t_1
⑧ t_2 = b*2
⑨ c = a + t_2
⑩ c = a+b
⑪ a = a-2
⑫ c = a
⑬ Exit

⑤
```
switch (a+b)
{
case 1 : x = x+1;
case 2 : y = y+2;
case 3 : z = z+3;
default : c = c-1;
}
while (i<10);
while (i<10);
```

}

⑥ switch (a+b)

{

case 2 : { x = y, break ; }

case 5 : { switch (x)

{

case 0 : { a = b+1 ; break ; }

case 1 : { a = b+3 ; break ; }

default : { a = 2 ; }

break ;

case 9 : { x = y-1 ; break ; }

default : { a = 2 ; }

Soln-1

① i = 1

② Sum = 0

③ t = i * 5

④ Sum = Sum + t

⑤ i = i + 1

⑥ if (i < 10) goto 3

⑦ goto 8

⑧ exit

Soln-2

① t = a+b
t = 2

② if (~~a+b~~) goto 8

③ if (~~t~~ = 5) goto 10

④ if (t = 9) goto 21

⑤ ~~if (t~~

⑤ goto 6

⑥ a = 2

⑦ goto 24

⑧ x = y

⑨ goto 4

⑩ $t_1 = x$

⑪ if (t_1 = 0) goto 15

⑫ if (t_1 = 1) goto 17

⑬ ~~goto~~

⑭ a = 2

⑮ goto 24

⑯ a = b+1

⑰ goto 4

⑱ a = b+3

⑲ goto 4

⑳ goto 13 goto 24

㉑ x = y-1

㉒ ~~goto~~ goto 24

㉓ goto 6

㉔ Exit

25/02/21

Three Address Code for Functions or Procedures / Translation of Expression

calling part —

① Arguments / parameters

```
param t
call P, n
```

t → the arguments passed to the func

P → name of the func / procedure

n → no. of arguments passed to the function.

callee part —

function return value x

```
retrive x
```

e.g. —

$$P(x_1, x_2, \ldots x_n)$$

So, TAC

① param x_1
② param x_2
⋮
⑤ param x_n
(n+1) call P, n

$x = f(0, y + z*t) - 5$

↳ y Returns p

Sol:

① param 0

② A = z * t

③ B = y + z * A

④ param B

⑤ call f, 2

⑥ retrive P

⑦ x = P - 5

Three-address code for Array reference:

Let a one-dimensional array named as 'a', whose subscript ranges from low(l) to high(h)

→ Let 'b' is the address of the starting element and 'w' is the size of each element present in the array, then the address of the array can be calculated as follows —

a[i]

address = b + (i * w) − (l * w)

= b − (l * w) + (i * w)

$= [b - (l \times w)] + (l \times w)$

t_2 t_1

↳ base address

e.g. — a[i] = 2 (∵ l = 1)

① $t_1 = i * w$

② $t_2 = \text{addr}(a) - w$

③ $t_3 = t_2[t_1]$

④ $t_3 = 2$

Q:- ① Main ()

```
{
  int a[10], b[10];
  sum = 0;
  for (i = 0; i < 10; i++)
    {
      sum = sum + a[i] + b[i];
    }
  cout << sum;
}
```

① sum = 0
② i = 0
③ if (i < 10) goto 5
④ goto 15
⑤ $t_1 = i * w$
⑥ $t_2 = addd(b) - w$
⑦ $t_3 = t_2[t_1]$
⑧ $t_4 = i * w$
⑨ $t_5 = addd$
⑩ $t_6 = t_5[t_4]$
⑪ $t_7 = t_6 + t_3$
⑫ sum = sum + t_7
⑬ i = i + 1
⑭ goto 3
⑮ Exit

② main ()

```
{
  int a = 5;
  int b[11];
  while (a <= 5)
    {
      b[a] = 3*a;
    }
}
```

① a = 5
② if (a <= 5) goto
③ goto 9
④ $t_1 = a * w$
⑤ $t_2 = addd(b) - w$
⑥ $t_3 = t_2[t_1]$
⑦ $t_3 = 3 * a$
⑧ goto 9
⑨ Exit

01/03/21

Three Address code for 2-D Array :—

A[i] [j]

① — Row major order
② — Column major order

In 2-D array the variable 'i' represents the row and the variable 'j' represents the column.

Row major order :—

base address of the 1st element of the 2-D array

$$\text{address} = b + (i - low_1)\,w + (j - low_2)\,w$$

$$= b + [(i - low_1)\,u + (j - low_2)]\,w$$

$$= b + iuw - low_1 uw + jw - low_2 w$$

$$= \underbrace{(iu + j)\,w}_{t_1} + \underbrace{b - w(low_1 u + low_2)}_{t_2}$$

where,

u = no. of elements of the column.

$t_1 = i * u$
$t_1 = t_1 + j$
$t_1 = t_1 * w$
$t_2 = Addr(A) - w$
$t_3 = t_2 [t_1]$

```
main()
{
int a[20][20], b[20][20], add=0, i=1, j=1;
  do
  {
  add = add + a[i][j] * b[j][i];
      i++;
      j++;
  } while ((i<=20) & (j<=20));
```

word size of each element is 4. (w = 4)

① add = 0
② i = 1
③ j = 1
④ $t_1 = i * 20$
⑤ $t_1 = t_1 + j$
⑥ $t_2 = t_1 * 4$
⑦ $t_2 = addr(a) - 84$
⑧ $t_3 = t_2[t_1]$
⑨ $t_4 = j * 20$
⑩ $t_4 = t_4 + i$
⑪ $t_4 = t_4 * 4$
⑫ $t_5 = addr(b) - 84$
⑬ $t_6 = t_5[t_4]$
⑭ $t_7 = t_3 * t_6$
⑮ add = add + t_7
⑯ i = i + 1
⑰ j = j + 1
⑱ ~~goto~~
⑲ if (i<=20) & (j<=20) goto ⑧
⑳ goto ④
㉑ Exit

02/03/21

Translation Boolean Expression to TAC :—

① Numerical method

② Jump method

— It is an expression which is constructed by boolean operation that is known as boolean expression.

e.g- E → E or E
E → E and E
E → not E
E → id relop id
E → True
E → False

→ It is evaluated by asigning numerical values to the boolean expression.

e.g- if a < b or (c > d & e < f)

Ans:-

① if (a < b) goto 7
② goto 3
③ if (c > d) goto 5
④ goto 8
⑤ if (e < f) goto 7
⑥ goto 8
⑦ statement
⑧ Next statement

Back patching :—

```
main( )
 {
 int a[i] , b[i] , i=1, b=1;
  Sum = 0;
  for (i=1 ; i<10; i++)
    Sum = Sum + a[i] + b[i];
  }
```

(1) i = 1
(2) b = 1
(3) sum = 0
(4) if(i < 10) goto 6
(5) goto 16
(6) $t_1 = i * w$
(7) $t_2 = addr(b) - w$
(8) $t_3 = t_2[t_1]$
(9) $t_4 = i * w$
(10) $t_5 = addr(a) - w$
(11) $t_6 = t_5[t_4]$
(12) $t_7 = t_3 + t_6$
(13) sum = sum + t_7
(14) i = i + 1
(15) goto 4
(16) Exit

03/03/21

Representation of Intermediate codes :—

1. Quadruples
2. Triple
3. Indirect triple
4. Postfix notation
5. Syntax tree
6. DAG (Directed Acycle Graph)

Quadruples :—

Quadruples has 4 fields which can represent the three address code in intermediate code generation.

Operator	Operand 1	Operand 2	Result/ destination

e.g.-

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

(1) $t_1 = a+b$

(2) $t_2 = -c$

(3) $t_3 = t_2/d$

~~(4) $t_3 = a+b$~~

~~(4) $x = t_3 * t_2$ $t_1 * t_2$~~

(4) $t_4 = t_1 * t_3$

(5) $x = t_4$

Operator	Operand-1	Operand-2	Result
+	a	b	t_1
−	c	−	t_2
/	t_2	d	t_3
*	t_1	t_3	t_4
* =	t_4		x

~~Code generation~~