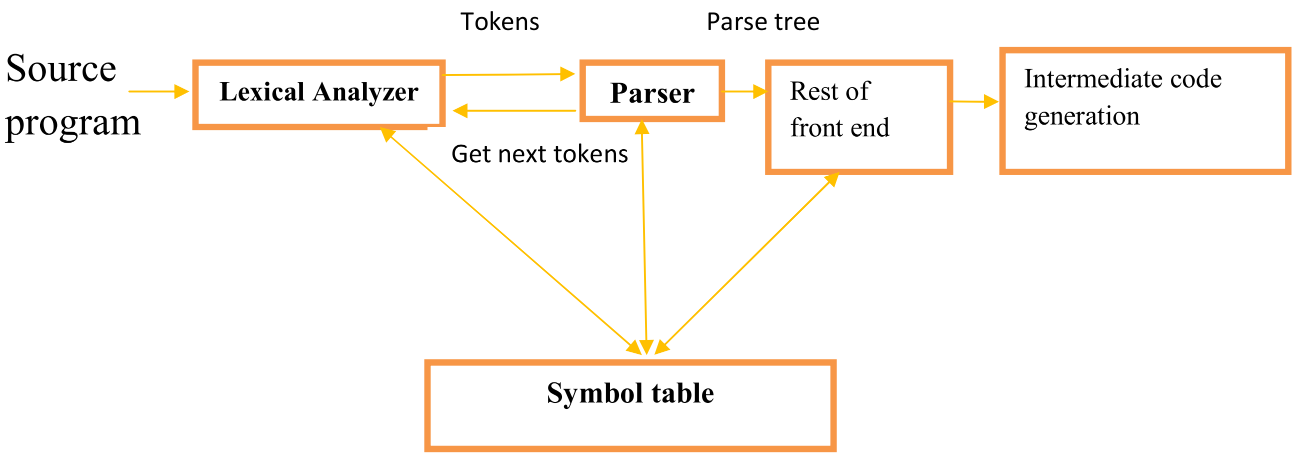
UNIT IV

Syntax Analysis



Position of parser in compiler model

- The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.
- It reports any syntax errors in the program and gets recovered from occurring errors and continues the process.
- Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.
- A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.
- **There are three types of parser for grammars:**
  a) Universal parser
  b) Top-down parser
  c) Bottom-up parser
- Universal parser can parse any grammar.
- Top-down methods build parse tree from top to bottom.
- Bottom-up methods build parse tree from bottom to top.
- In both the cases the input to parser is scanned from left to right.

- The output of the parser is some representation of parse tree which is comes from lexical analyzer.
- There are number of tasks that might be performed during parsing.
    a. It verifies the structure generated by tokens based on the grammar.
    b. It constructs the parse tree.
    c. It reports the errors.
    d. It performs error recovery.
- Parser can't detect errors such as:
    a. Variable re-declaration.
    b. Variable initialization before use.
    c. Data type mismatch for an operation.

Content free Grammar (CFG):

- A context free grammar (CFG) consisting of a finite set of grammar rules which is qudraples (N, T, P, S) Where,

  N→ set of non-terminal symbols.

  T→ set of terminals where N∩T=NULL

  P→ P is set of rules (Production rules)

  P: N→(N∪T)*

  The left hand side of the production rules P does have any right context or left context.

  S→ S is the start symbol
- CFG is the notation used for describing the syntax of programming languages.
- It is used to specify the syntax of program in particular language.
- It is also called as BNF(Backus Naur form).
- E.g:

  If S1 and S2 are statements and E is an expression then

  if

  E then S1

  else

  S2

- If S_1, S_2, S_3, ..., S_n are statements
  - Statement → Statement | Statements List.

- If E_1 and E_2 are two expression
  - Then $E \rightarrow E_1+E_2$

Production Rules:

$S \rightarrow aSa$
$S \rightarrow bSb$
$S \rightarrow c$

String abbcbba derived from given context free grammar.

$S \rightarrow aSa$
$S \rightarrow abSba \quad \therefore S \rightarrow bSb$
$S \rightarrow abbSbba \quad \therefore S \rightarrow bSb$
$S \rightarrow abbcbba \quad \therefore S \rightarrow c$

#Capabilities of context free grammar:

There are various capabilities of CFG:

- Context free grammar is useful to describe most of the programming languages.
- If the grammar is properly designed then an efficient parser can be constructed automatically.
- Using the features of associatively and precedence information, suitable grammars for expressions can be constructed.
- Context free grammar is capable of describing nested structures like balanced parenthesis, matching begin- end, corresponding if- then else and so on..
- Regular expressions are capable of describing the syntax of tokens.
- Any syntactic construct that can be described by a regular expression can also be described by a context free grammar.
- For example:
  - (a | b) (a | b | 0 | 1)* -regular expression and context free grammar.

$S \rightarrow aA \mid bA$

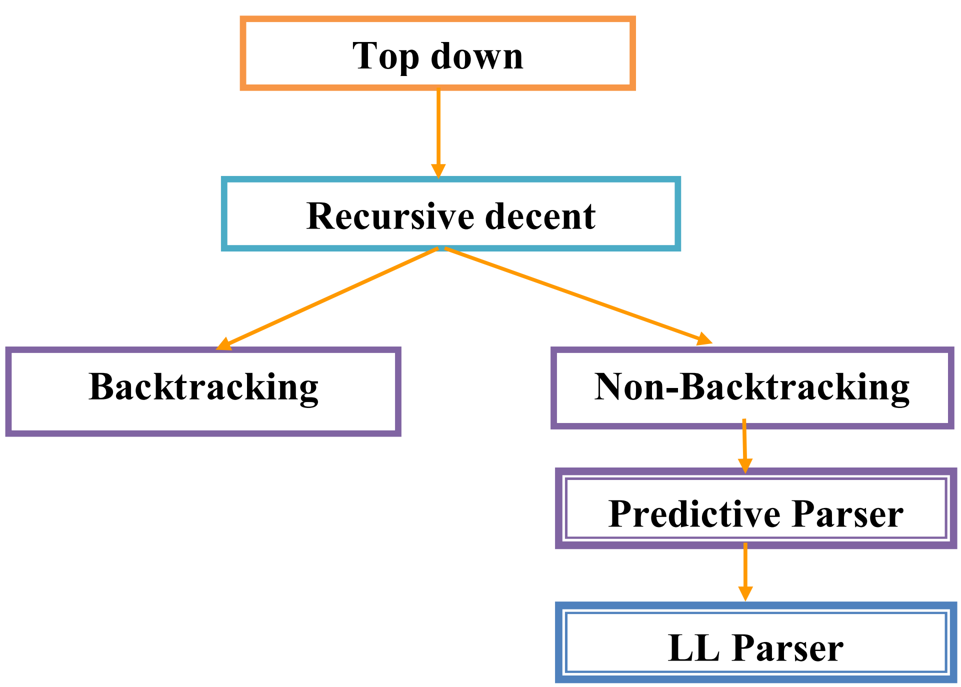
$A \rightarrow aA \mid bA \mid 0A \mid 1A \mid €$

Describe the same language.

- Regular expressions are most useful for describing the structure of lexical constructs such as identifiers, constants, keywords and so on.
- Context free grammar on the other hand are most useful in describing nested structures such as parenthesis matching, corresponding if-else's statements.
- These nested structures can't be described by regular expressions.

#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.
- Top-down parsing technique parses the input and starts constructing a parse tree form the root node gradually moving down to the leaf nodes.
- The types of top-down parsing are:



Eg.

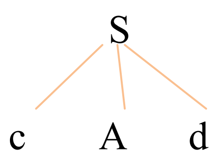
It starts to construct a parse tree for input starting from root.

Eg. $S \rightarrow cAd$

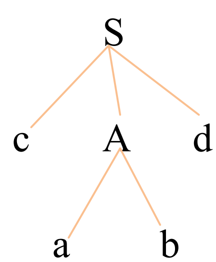
$A \rightarrow ab \mid a$

We initially create a tree consisting of single node S.

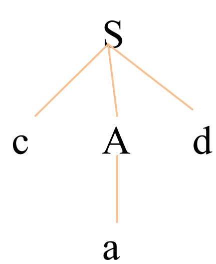
∴



The leftmost leaf ‘c’ matches the first symbol and it is a terminal then we expand A using the give production rule.



Now second alternate for A is



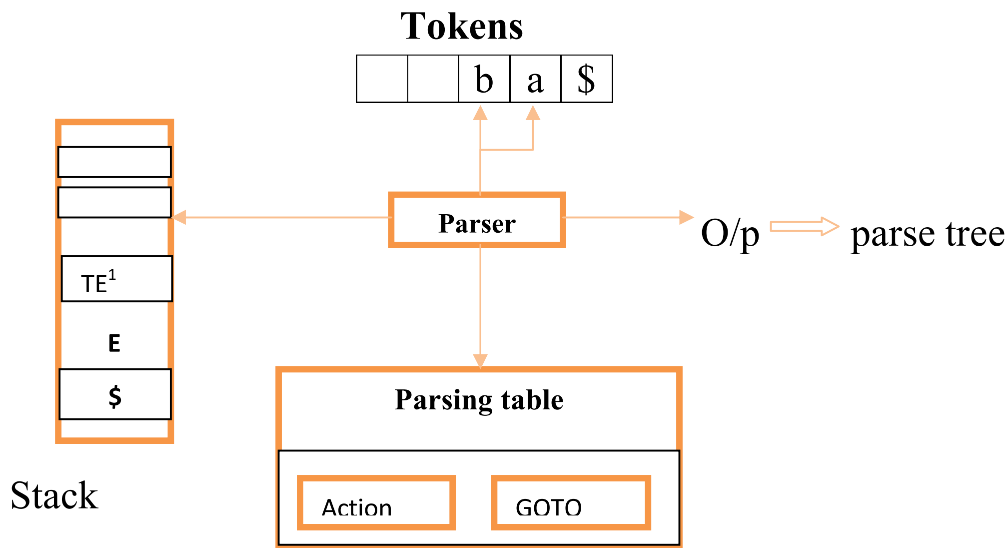
- An easy way to create a parser is a procedure for non-terminal.
- In simple grammar there is no need for recursion.
- But when grammar derive an infinite number of strings, recursion is necessary.
- **There are difficulties with top-down parsing**.
    a) The first problem is left recursion, it will go into infinite loop.
    b) A second problem is its backtracking.
    c) The third problem is the order in which alternates are used, it will affect the language Accepted.

Recursive decent parsing:

- Recursive decent 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 for procedures for every terminal and non-terminal entity.
- By carefully writing a grammar means eliminating left recursion and left factoring from it, the resulting grammar can be parsed by recursive decent parser.

Predictive parser:

- Predictive parser is used to construct a top-down parser that never backtracks.
- To do so we must perform a grammar in two ways:
    a) Eliminate left recursion
    b) Perform left factoring
- The predictive parser has an input, a stack, a parsing table and an output which is a parse tree.
- The input contains set of tokens to be parsed followed by $, the right end maker.
- The stack contains sequence of grammar symbols, preceded by $, the bottom of stack marker.
- Initially the stack contains the start symbol of the grammar preceded by $.
- The parsing table is a two-dimensional array M[A, a],
  Where, A- is a non-terminal
  a-is a terminal or $



Predictive parser divided into four steps:

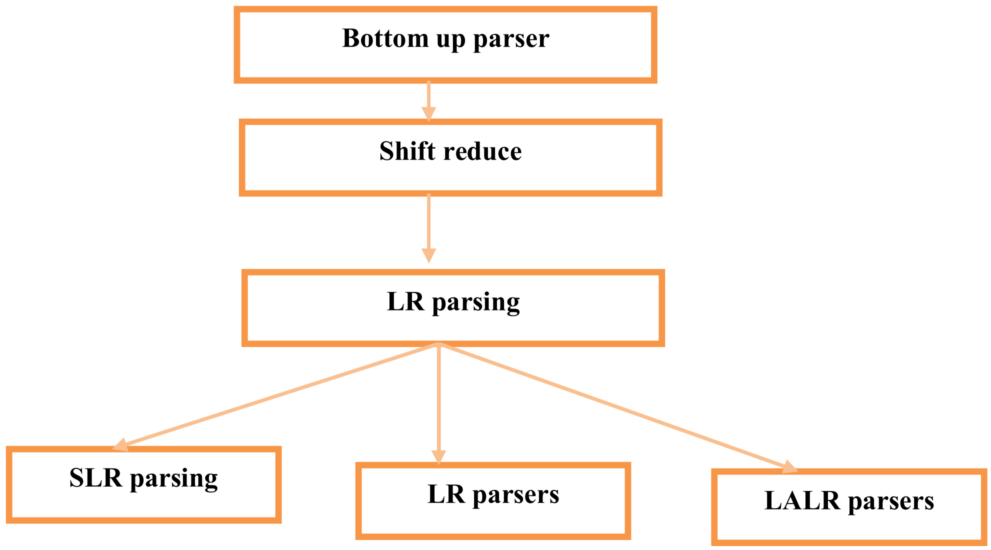
a) **First and follow method**
b) **Parsing table**
c) **Stack implementation**
d) **Parse tree/output**

- It is a recursive decent parser, which has the capability to predict which production is to be used to replace the input string.
- It does not suffer from backtracking.
- To accomplish its task, the predictive parser uses a lookahead pointer, which points to the next input symbols.
- To make the grammar backtracking free, the predictive parser puts some constrains on the grammar and accepts only a class of grammar known as LL(k) grammar.
- The parser refers to the parsing table to take any decision on the input and stack element combination.
- The behaviour of the parser we can describe in terms of stack.

Stack	input
$S	W$

- When S is a start symbol of the grammar and WW is a string to be parsed.

Bottom up parsing:



- Bottom up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node.
- In this parsing, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol.
- It is an attempt to reduce the input string W to the start symbol of grammar tracing out the right most derivations of W in reverse.
- Bottom-up parsing is also called as shift reduces parsing.
- Bottom-up parsing generally works by choosing next input and contents of stack whether to shift next input onto stack.
- We use $ to mark, the bottom of stack and show top of stack on right.
- During a left to right scan of input string, parser shifts zero or more symbol onto stack.

Following figure shows the steps of shift reduces parser.

Here input string is id_1*id_2

$F \rightarrow id_1$

$F \rightarrow T$

$T \rightarrow id_2$

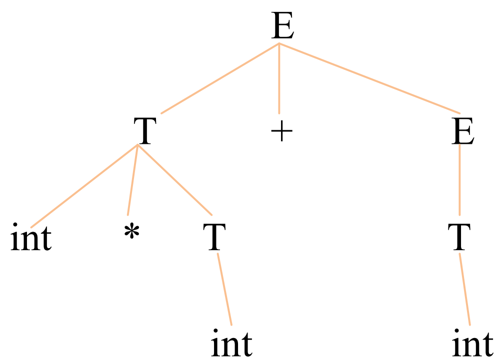
Stack	Input	Action
$	Id1*id2$	Shift
$id1	*id2$	Reduce F→id1
$F	*id2$	Reduce F→T
$T	*id2$	Shift
$T*	id2$	Shift
$T*id2	$	Reduce F→id2
$T*F	$	

- There are four possible actions of shift reduce parser:

1) Shift: shift next input symbol onto top of stack.
2) Reduce: The right end of string to be reduced and locate the left end of string
3) Accept: Announce successful completion of parsing.
4) Error: Discover a syntax error.

Eg.

int*int+int	T→int
int*T+int	T→int*T
T+int	T→int
T+T	E→int
T+E	E→T+E
E	E→T+E



w=int*int+int

- Bottom up parsing as the process of reducing a string W to the start symbol of the grammar.
- At each reduction step a substring that matches the body of a production is replaced by the non-terminal at the head of that production.

Operator precedence parsing:

- The following grammar for expressions.

  E→EAE | (E) | -E | id

  A→+ | - | * | /
- EAE has two repeated non-terminal
- If we substitute for A then we obtain following grammar.

  E→E+E | E-E | E*E | E/E | -E | id
- An operator precedence parser is a bottom up parser that interprets an operator grammar.
- This parser is only used for operator grammars.
- Ambiguous grammars are not allowed in any parser except operator precedence parser.
- This parser relies on the following three precedence relation.

  <, =>

  a) a<b → means "a yields precedence to b"

  b) a>b → means "a yields precedence over b"

  c) a=b → means "a has same precedence as b"
- As a general parsing technique operator precedence parser has number of disadvantages.

- For example- It is hard to handle tokens precedence (depending on whether it is binary or unary)
- It is a kind of shift reduce parsing method.
- It is applied to small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

a) No right hand side of any production has a €
b) No adjacent variables or non-terminals.

- Operator precedence grammar can only established between the terminals of the grammar. It ignores the non-terminals.

Input

Operator grammar → **Operator parser**

Parse tree

- The string id+id*id $ can be parsed by operator precedence parser using four steps:
  i. First check given grammar is operator precedence grammar or not.
  ii. Operator precedence relation table.
  iii. Parse the given string.
  iv. Generate the parse tree.

Eg. T→T+T | T*T | id

- Operator precedence relation table

	+	*	**id**	**$**
+	>	<	<	>
*	>	>	<	>
Id	>	>	**-**	>
$	<	<	<	**A**

String is accepted.

- Precedence level is decided in operator precedence parser
- id, a, b, c has highest priority where as $ has very low precedence
- Arithmatic operator has greater precedence than right hand side
- The purpose of precedence relations is to define handle of right side form with <•, •>, ∑ sign.

  Eg. W=id+id*id

  <•id•>+<•id•>*<•id•>
- The string between <• and •> is our handle

LR parser:

- LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.
- In the LR parsing 'L' stands for left to right scanning of the input.
- R stands for constructing right most derivation in reverse.
- LR parsing is divided into four parts.

  LR(0) parsing

  SLR parsing

  CLR parsing

  LALR parsing
- The LR algorithm requires stacks, input, output and parsing table.
- In all type of LR parsing, input, output and stack are same but parsing table is different.

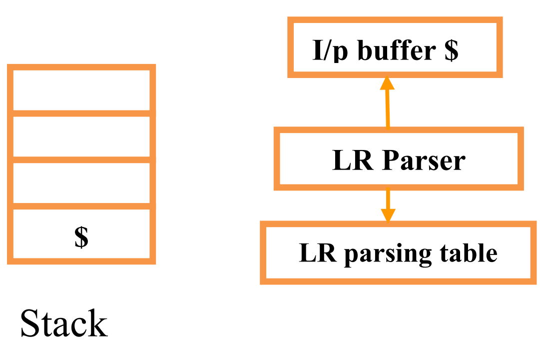


Fig: Block Diagram of LR parser

- Input (i/p) buffer is used to indicate end of input and it contains the string to be parsed followed by a $ symbol.
- A stack is used to contain a sequence of grammar symbols with a $ at the bottom of the stack.
- Parsing table is two dimensional arrays.

It contains two parts
- Action part
- GOTO part

Automatic construction of parser using YACC:

- A parser generator is a program that takes as input a specification of syntax and produces as output a procedure for recognizing that language.
- Historically they are also called compiler-compilers.
- YACC (yet another compiler-compiler) is an LALR (Look Ahead, left to right, Rightmost derivation procedure with 1 lookahead token) parser generator.
- YACC was originally designed for begin complemented by Lex.

Input file

YACC input file is divided in three parts

```
/* definitions*/
----------------
----------------
/*rules*/
---------
---------
%%
/*Auxilary routines*/
----------------
-----------------
```

Input file: Definition part

- The definition part includes information about the tokens used in the syntax definitions

% token number% token Id

- YACC automatically assigns numbers for tokens, therefore assigned tokens numbers should not overlap ASCII codes.
- The definition part include C code external to the definition of the parser and variable declarations within %{and %} in first column.
- It can also include the specification of the starting symbol in the grammar.

  % start non-terminal

Input file: rule part

- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in {} and can be embedded inside(translation schemes)

Input file: Auxiliary Routines part

- The Auxiliary routines part is only C code
- It includes function definition for every function needed in rules part.
- It can also contain main() function definition if the parser is going to be run as a program.

Output files:

- The output of YACC is a file named y.tab.c
- If it contains the main() definition, it must be compiled to be executable
- Otherwise, the code can be an external function definition for the function int yyparse()

Grammar.y→ **YACC Compiler** →y.tab.c

y.tab.c→ **C Compiler** →a.out (Syntax Analyzer File)

Tokens generated by lexical analyzer → **a.out** → Parser(Parse tree)