CONTEXT FREE GRAMMAR:

Context free grammar Is a specification for the syntax of a programming language. Informally a Context Free Grammar (CFG) is defined as a finite set of rules used for deriving or generating strings or sentences in a language, which consists of:

- A set of replacement rules, each having a Left-Hand Side (LHS) and a Right-Hand Side (RHS).
- Two types of symbols called as variables and terminals.
- LHS of each rule is a single variable (non-terminal).
- RHS of each rule is a string of zero or more variables and/or terminals.

CFG is also defined as a set of variables (non-terminals) each of which represents a language. The languages represented by the variables are described recursively in terms of each other. The primitive symbols are called as terminals and the rules relating the variables are called as productions.

Context Free Grammar is formally defined as a 4 tuple (V, T, P, S) where:

- V and T are finite sets of variables and terminals respectively.
- P is the finite set of productions. Each production is of the form A→α where A is a variable and α is a string of symbols from (V U T)*
- S is a special variable called as start symbol.

Sentential form: A sentence that contains variables & terminals is called as a sentential form.

For example, If (α, β) ϵ P, then we write the production as α →β where β is the sentential form.

Context-free grammars are useful for describing arithmetic expressions and block structure. Example for context free grammar is given below:

Given a set of productions:

<exp> → <exp> + <exp>

<exp> → <exp> * <exp>

<exp> → (<exp>)

<exp> → id

Example E → E + E, E → E * E, E → (E), E → x, E → y

LANGUAGE OF CONTEXT FREE GRAMMAR:

A sequence of tokens is syntactically legal if it can be derived by applying the productions of the CFG. A context-free grammar defines a language which is a set of strings (sequences) of tokens (terminals), each string of tokens is derivable from the production rules of the CFG.

Consider the following simplified grammar for expressions

expr→ expr op expr | (expr) | id | num

op → + | – | * | /

- ✓ The string:*(id+num)*id* is syntactically legal and part of the language
- ✓ Similarly: *id*(num-id)* is also part of the language
- ✓ However: *(id+num* is NOT part of the language
- ✓ Similarly: *id*-num+* is NOT part of the language

DERIVATIONS:

Derivation is an ordered tree which is defined as sequence of replacements of a substring in a sentential form. To check whether a sequence of tokens is legal or not:

- ✓ We start with a nonterminal called the start symbol
- ✓ We apply productions, rewriting nonterminals, until only terminals remain
- ✓ derivation replaces a nonterminal on LHS of a production with RHS
- ✓ The → symbol denotes a derivation step

For example, a derivation for *(id + num)*id* is given below:

```
expr   →  expr op expr
       →  ( expr ) op expr
       → ( expr op expr ) op expr
       → ( expr + expr ) op expr
       → ( expr + expr ) * expr
       → ( id + expr ) * expr
       → ( id + num ) * expr
       → ( id + num ) * id
```

When deriving a sequence of tokens more than one nonterminal may be present and can be expanded. There are two types of derivations 1) left most derivation and 2) right most derivation.

A left most derivation chooses the leftmost nonterminal to expand and a right most derivation chooses the rightmost nonterminal to expand.

A leftmost derivation for `(id + num)*id`

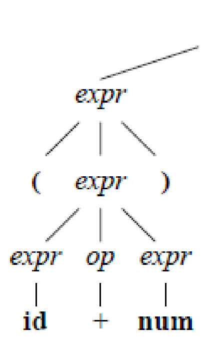
$$\begin{aligned} &\textit{expr} \Rightarrow_{lm} \textit{expr op expr} \Rightarrow_{lm} (\textit{ expr }) \textit{ op expr} \Rightarrow_{lm} (\textit{ expr op expr }) \textit{ op expr} \\ &\quad \Rightarrow_{lm} (\textbf{ id } \textit{op expr }) \textit{ op expr} \Rightarrow_{lm} (\textbf{ id } + \textit{ expr }) \textit{ op expr} \\ &\quad \Rightarrow_{lm} (\textbf{ id } + \textbf{ num }) \textit{ op expr} \Rightarrow_{lm} (\textbf{ id } + \textbf{ num }) * \textit{ expr} \Rightarrow_{lm} (\textbf{ id } + \textbf{ num }) * \textbf{ id} \end{aligned}$$

A rightmost derivation for `(id + num)*id`

$$\begin{aligned} &\textit{expr} \Rightarrow_{rm} \textit{expr op expr} \Rightarrow_{rm} \textit{expr op } \textbf{id} \Rightarrow_{rm} \textit{expr} * \textbf{ id} \Rightarrow_{rm} (\textit{ expr }) * \textbf{ id} \\ &\quad \Rightarrow_{rm} (\textit{ expr op expr }) * \textbf{ id} \Rightarrow_{rm} (\textit{ expr op } \textbf{num }) * \textbf{ id} \\ &\quad \Rightarrow_{rm} (\textit{ expr } + \textbf{ num }) * \textbf{ id} \Rightarrow_{rm} (\textbf{ id } + \textbf{ num }) * \textbf{ id} \end{aligned}$$

The graphical representation of a derivation is called parse tree, which filters out choice regarding replacement order and it is rooted by the start symbol S. Interior nodes represent nonterminals, Leaf nodes are terminals. The following is a parse tree for (id + num) * id

$$\begin{aligned} &\textit{expr} \rightarrow \textit{expr op expr} \\ &\text{expr} \rightarrow (\textit{ expr }) \\ &\text{expr} \rightarrow \textbf{id} \\ &\text{expr} \rightarrow \textbf{num} \\ &\textit{op} \rightarrow + \mid - \mid * \mid / \end{aligned}$$



AMBIGUITY:

A Context Free Grammar is ambiguous if some string w ϵ L(G) has two or more leftmost or rightmost derivations. Sometimes the language generated by an ambiguous grammar has an equivalent unambiguous grammar. Languages that can only be generated by ambiguous grammars are called inherently ambiguous. It can be solved in two ways 1) solved by precedence 2) solved by associativity

PARSING:

The term parsing is the process of analyzing a string of symbols, either in natural language or in computer languages, according to the rules of a formal grammar.

Programming language that follows some syntax to develop the code has to check for any syntax errors in the code, this will be done in parsing.

Parsing is the process of the generation of a parse tree (or its equivalents) which is a graphical representation of derivation that filters out the choice regarding replacement order; parse tree is the internal structure of compiler or interpreter; corresponding to a given input string w and grammar G. There are two types of parsing techniques. 1) Top Down Parsing and 2) Bottom Up parsing

TOP DOWN PARSING:

Top down Parsing begin with start symbol of grammar as root of tree and grow it towards leaves. Construction of parse tree in top down parsing is constructed from the top and from left to right. It is classified in to two types based on two forms (i.e. prediction and backtracking)

1) Recursive descent parsing

2) Predictive parsing without backtracking.

RECURSIVE DESCENT PARSING:

A recursive descent parser is a kind of top-down parser built from a set of mutually recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

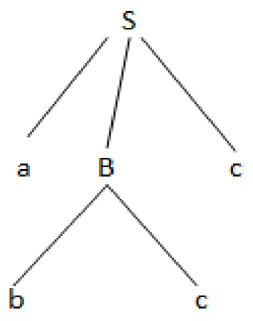
Recursive descent with backtracking is a technique that determines which production to use by trying each production in turn. Recursive descent with backtracking is not limited to LL(k) grammars, but is not guaranteed to terminate unless the grammar is LL(k). Even when they terminate, parsers that use recursive descent with backtracking may require exponential time. It consists of set of procedures one for each non terminal, which uses backtracking.

Now let us consider a grammar:

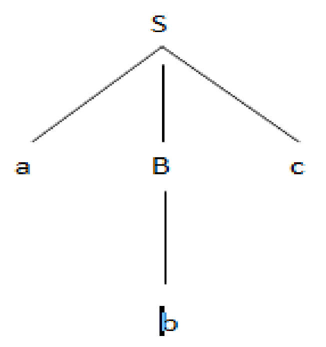
$S \rightarrow aBc$

$B \rightarrow bc/b$

Let the input string be abc then constructing a parse tree



Here it fails so it backtracks



It makes repeated scans of the input. A left recursive grammar can cause a recursive decent parser, even one with backtracking, to go into an infinity loop. A grammar is said to be left recursive if it has a non terminal A such that there is a derivation $A \rightarrow A\alpha$ for some string α.

Top down parsing methods cannot handle left recursive grammars, so a transformation that eliminates left recursion is needed. To get non-left recursive productions we rewrite the productions **$A \rightarrow A\alpha \mid \beta$** as

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

i.e. if $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \ldots \mid \beta_m$ then it can be written as follows:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \ldots \beta_m A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \ldots \mid \alpha_n A' \mid \varepsilon$

Let us consider an example:

E→E+T/T

T→T*F/F

F→(E)/id

The grammar can be written as

E→E+T
E→T
T→T*F
T→F
F→(E)
F→id

Now consider E→E+T | T this is in the form of A→Aα, so we can rewrite the grammar as

E→TE'

E'→+TE'/E

Similarly this can be T→FT'

T'→*FT'/E

F→(E)/id

PREDICTIVE PARSING:

A predictive parser is a recursive descent parser that does not require backtracking. Predictive parsing is possible only for the class of LL(k) grammars, which are the context-free grammars for which there exists some positive integer k that allows a recursive descent parser to decide which production to use by examining only the next k tokens of input.

The LL(k) grammars therefore exclude all ambiguous grammars, as well as all grammars that contain left recursion. Any context-free grammar can be transformed into an equivalent grammar that has no left recursion, but removal of left recursion does not always yield an LL(k) grammar. A predictive parser runs in linear time.

Predictive parsers can be depicted using transition diagrams for each non-terminal symbol where the edges between the initial and the final states are labeled by the symbols (terminals and non-terminals) of the right side of the production rule.

Left Factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. Now we have few steps to follow:

1) Collect all the productions with the same left side and begin with same symbols on right hand side.

   Example: E→T+E/T

   T→int/int*T/(E)

   T→int

   T→int*T

2) Combine the common strings into a single production and then append a new terminal symbol to the end of this new production.

   Example: T→int y

3) Create new production using this new non-terminal for each of the suffixes to the common production.

METHODS IN LL:

- FIRST(α)
- FOLLOW(A)

<u>FIRST (α):</u>

It is defined as the set of terminals that begin strings derived from α where α is any grammar symbol.

<u>FOLLOW (A):</u>

It is defined as the set of terminals that can appear immediately after A in some sentential form. We have some rules for both FIRST (α) and FOLLOW (A). They are:

<u>FIRST (α):</u>

- If α is a terminal then FIRST (α) is {α}.
- If α→ε is a production then add ε to FIRST (α).
- If α is a non-terminal and α→$Y_1,Y_2,Y_3,\ldots.Y_k$ is a production then place a in First(α) if for some I, a is in FIRST(Y_i) and ε is in all of FIRST(Y_1)….FIRST(Y_{i-1}) and if ε is in FIRST(Y_j) for all j=1,2,….k then add ε to FIRST(α).

<u>FOLLOW (A):</u>

- Place $ in FOLLOW (A) where A is a start symbol and $ is input right end marker.
- If there is a production A→αBβ then everything in FIRST(β) is in FOLLOW(β) except 'ε', irrespective of whether β may or may not contains ε.
- If there is a production A→αBβ (or) A→αB where FIRST(β) contains ε then everything in FOLLOW (A) is in FOLLOW (B).

Now let us consider an example grammar.

E→E+T/T

T→T*F/F

F→Id/(E)

The above example has lest recursion so we have determinate it.

First consider,

$E \rightarrow E+T$

$E \rightarrow T$

Here α=T, $\beta1$=+E β=E

So this can be written as

$E \rightarrow TE'$

$E' \rightarrow +E/E$

Now consider

$T \rightarrow T*F$ (or) $T \rightarrow F*T$

$T \rightarrow F$ $T \rightarrow F$

Here α=F, $\beta1$= *T, $\beta2$=E

$T \rightarrow FT'$

$T' \rightarrow *T/E$

So now after removing left recursion the productions are:

$E \rightarrow TE'$

$E' \rightarrow +E/E$

$T \rightarrow FT'$

$T' \rightarrow *T/E$

$F \rightarrow id/(E)$

Now let us calculate FIRST (α).

FIRST (E) = FIRST (T) = FIRST (F)

Here FIRST (F) = {id,c} so from rule (3)

FIRST (T) = {id,c}

FIRST (E') = {E, +} from rules (2&3)

FIRST (T’) = {E, *}

FIRST (+) = {+}

FIRST (*) = {*}

FIRST (id) = {id)

FIRST (( ) = {(}

FIRST ( )) = {)}

Now let us calculate the FOLLOW (A) for the given grammar:

FOLLOW (E) = {$,)}

FOLLOW (E’) = FOLLOW (E)

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

FOLLOW (T’) = FOLLOW (T)

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

Now let us construct a predictive parse table. It has some rules:

- For each production A→α of the grammar do the step 2 & 3
- For each terminal ‘a’ in FIRST (α) add A→α to M[A, a]
- If E is in FIRST (α) add A→ to M [A, b] for each terminal b in FOLLOW (A). If E is in FIRST (α) and $ is in FOLLOW (A) then add A→ to M [A, $].
- Make each undefined entry as an error, in the table M.

	+	*	id	(	)	$
E			E→TE’	E→TE’		
E’	E’→+E				E’→E	E’→E
T			T→FT’	T→FT’		
T’	T’→E	T’*T			T’→E	T’→E
F			F→id	F→(E)		