SYNTAX ANALYSIS

This unit discusses about the role of the parser, its basic issues and how to construct a parse tree for a given string. As every compiler performs some type of syntax analysis, usually after lexical analysis, the input to a parser is typically a sequence of tokens and the output of the parser can be of many different forms. But in this chapter we assume that the output of a parser to be a representation of parse tree. Later on the ambiguity in grammars and how to eliminate left-recursion from the given grammar is also shown. The top down parsing and its various other types of parsing like Recursive decent parsing, Brute force parsing, Predictive parsing have also been discussed. The last topic that has been covered in this unit is LL(1) grammars.

Syntax analysis or Parsing is a major component of the front end of a compiler. Parsing is the process of determining if a string of tokens can be generated by a grammar. A parser scans an input token stream, from left to right, and groups the tokens in order to check the syntax of the input string by identifying a derivation by using the production rules of the grammar.

We can define a parser formally as a program p, for a context free grammar G, which takes a string w as input and outputs

1. a parse tree for w if w is a sentence of G i.e. $w \in L(G)$.
2. or an error message if w is not a sentence. It may also provide information about the type of error and the location of the error.

Funcions of a parser

The syntax analyzer (parser) plays an important role in the compiler design. It performs the following main tasks.

- It obtains a string of tokens from the lexical analyzer.
- It groups the tokens appearing in the input in order to identify larger structures in the program. This process is done to verify that the string can be generated by the grammar for the source language.
- It should report any syntax error in the program.
- It should also recover from the errors so that it can continue to process the rest of the input.

Basic issues in parsing

The basic issues in parsing are as follows:

- The specification of syntax of a programming language. How to describe the language syntax is the very basic issue. This specification should be precise and unambiguous. It must cover all the syntactic details of the language.

- The representation of the input after it has been successfully parsed. This information is important, as it is required by the subsequent passes of the compiler.
- The parsing algorithm. The algorithms fall into two categories, top-down and bottom-up. These terms refer to the order in which nodes in the parse tree are constructed.

Parse Trees

Parse tree is also called derivation tree or syntax tree. It is useful to display derivation as a tree i.e., Pictorial representation of a derivation is called parse tree.

If a non-terminal A has a production A-> XYZ, then a parse tree may have an interior node labeled A, with three children labeled X,Y and Z from left to right as

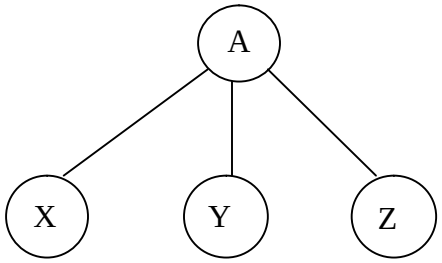


Fig 6.1: A Simple Parse Tree

Properties of parse trees

1. Root is labeled with start symbol.
2. Every vertex as label which is a variable or terminal or .
3. Label of interior vertex is variable or non-terminal .
4. The vertex n has label A and vertices n_1 , n_2 , ------n_k respectively then A -> x_1 x_2---------x_k must be a production in P.
5. If vertex n has a label then n is a leaf node and is the only son of its father.
6. The syntax tree is complete if no leaf node is labeled with non–terminal.
7. If a non-terminal has a production A -> XYZ then parse tree may have an interior node labeled with A and three children XYZ from left to right.

Construction of parse trees:

Example 1: Consider the following grammar

S -> ABC | ε
A -> Ba
B -> Ab
C -> c

Construct possible derivation tree for the given grammar.

Solution:

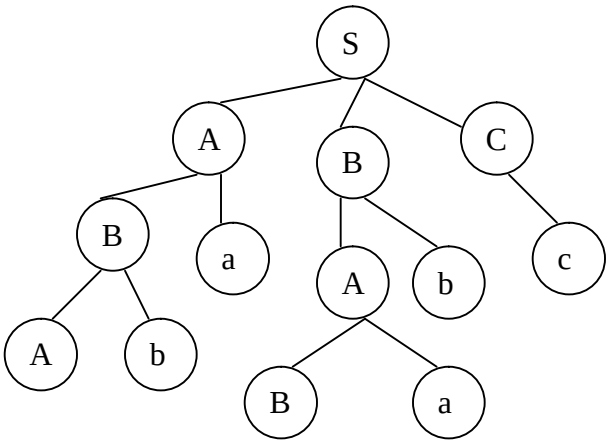


Fig 6.2: Parse Tree for the given grammar

Example 2: Construct a parse tree for the grammar
S -> S+S | S*S | (S) | a | b
String : a + a * b
Solution: Two parse trees can be constructed for the above grammar:

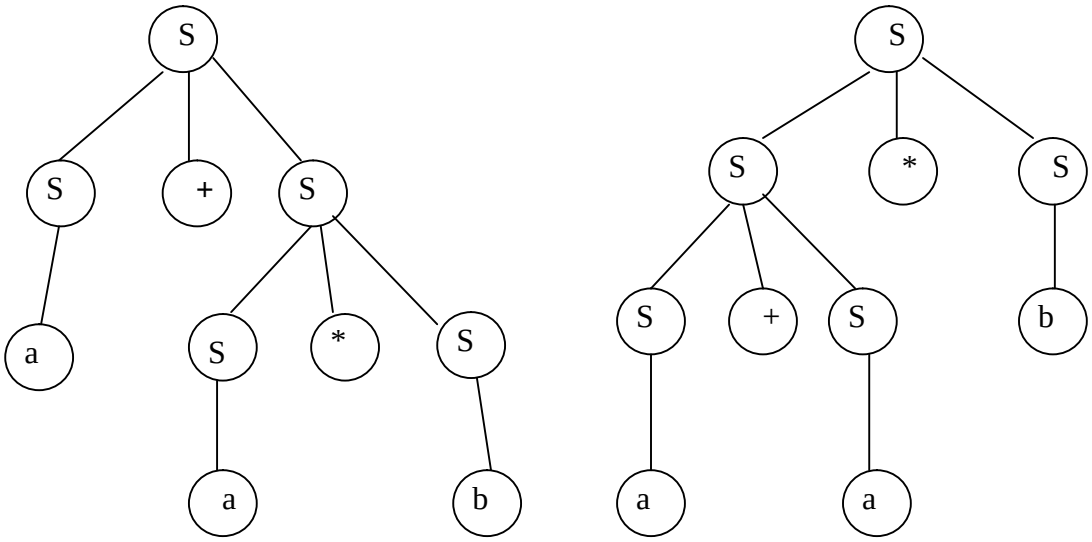


Fig 6.3: Parse Trees for the given grammar

Ambiguity in writing grammars:

A grammar that produces more than one parse tree or derivation tree for some sentence is said to be ambiguous. In other words, an ambiguous grammar is one that produces more than one left most or more than one rightmost derivation for some sentence.

Example 3: Consider the following "dangling-else" grammar:
stmt -> *if* **E** *then* **stmt**| *if* **E** *then* **stmt** *else* **stmt**

Draw the two parse trees for an ambiguous sentence.

1. If E1 then if E2 then S1 else S2
2. S:= if a>b then if a=b then a:= c+d else b:= c-d (replace E1 by a>b,E2 by a=b, S1 by a:=c+d, S2 by b:=c-d).s

Solution: Two parse trees for an ambiguous sentence:

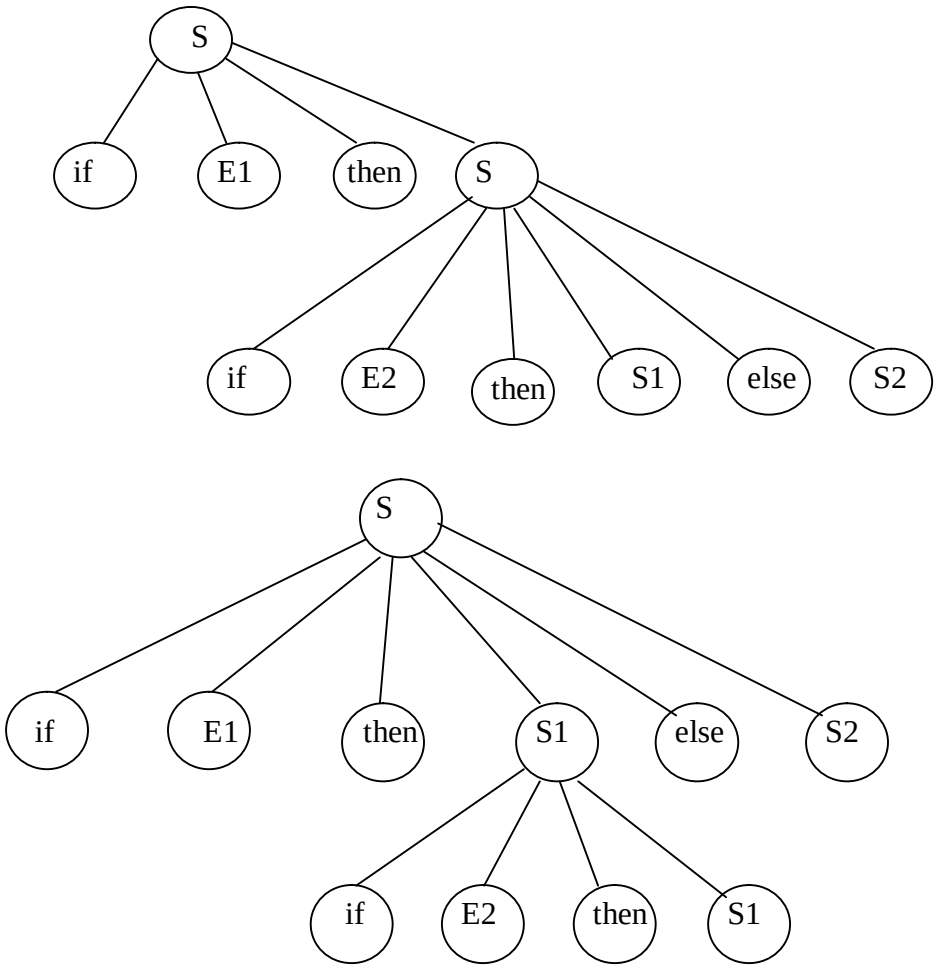


Fig 6.4: Parse Trees for an ambiguous sentence

Elimination of Left-Recursion:

A grammar is left-recursive if it has a non-terminal A such that there is a derivation A +=> Aα , for some string α . That is , a is left-recursive iff A can be expanded into a string that begins with A . A grammar which has one or more terminals that are left-recursive can be presented with strings to enter an infinite loop of production applications.

Consider the grammar : S -> aAc
A -> Ab

Non-terminating sequence of syntax tree generated from a left-recursive grammar

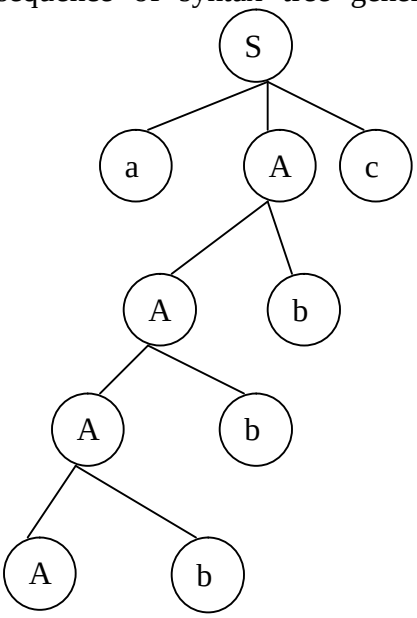


Fig Syntax Tree for the above grammar

Eliminate the left–recursion i.e., A -> bA

If we have the left recursive pair of productions :

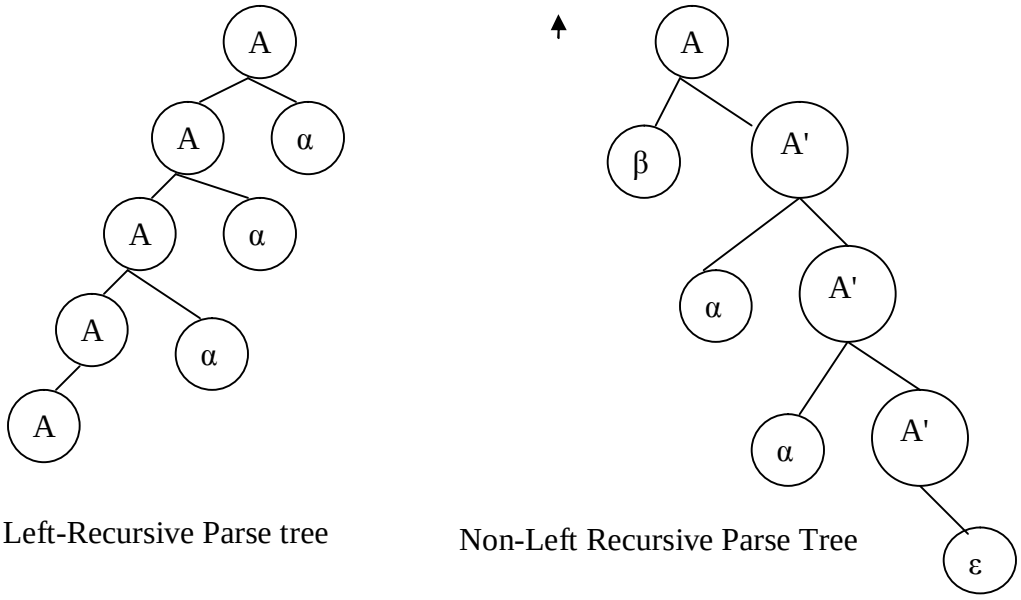
A -> Aα

A -> β

Where α and β are sequences of terminals and non-terminals that do not start with A, then we can eliminate the left-recursion by replacing this pair of productions with :

A -> βA'

A`-> αA' | ϵ



Left-Recursive Parse tree

Non-Left Recursive Parse Tree

Fig Left and Right Recursive Parse Trees

The general technique to eliminate immediate left-recursion among all A-Productions is :

First we group the A-Productions as

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$

where no βi begins with an A. Then, we replace the A-productions by

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots \mid \beta_n A'$

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

Example 4: Consider the grammar

E -> E+T | T
T -> T*F | F
F -> (E) | id

Construct an equivalent grammar with no left recursion.

Solution: Applying the immediate left-recursion for the E-production E -> E+T | T here α=+T β= T,hence resultant E-Productions as follows.

E -> TE'
E' -> +TE' | ε

Next we apply the immediate left-recursion for the T-production T -> T*F | F here α=*F β= F ,hence resultant T-Productions as follows.

T -> FT'
T' -> *FT' | ε

As there is no leftrecursion in F-production ,it remains as original production.

F -> (E) | id

Therefore the final grammar free from left –recursion is

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε F -> (E) | id

Algorithm to eliminate left-recursion from a grammar (With no cycles or ε–Productions)

1) Arrange the non-terminals in some order $A_1, A_2, \ldots A_n$.
2) For i := 1 to n do begin
   For j :=1 to i-1 do
   Begin
   Replace each production of the form $A_i \rightarrow A_{jr}$
   by the productions $A_i \rightarrow \delta_{1r} \mid \delta_{2r} \mid \ldots \mid \delta_{kr}$
   where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j productions;
   end;
3) Eliminate the immediate left-recursion among the A_i-Productions
   End;

Example 5: Eliminate the left-recursion for the following grammar

(a) A -> Ba | Aa | c
B -> Bb | Ab | d

Solution: From step1, removing the left-recursion of A

A -> BaA' | cA'
A' -> Aa' | ε
B -> Bb | Ab | d

From step2, replace A in B -> A

A -> BaA' | cA'
A' -> aA' | ε
B -> Bb | BaA'b | cA'b | d

From step3, remove the immediate left-recursion of B

A -> BaA' | cA'
A` -> aA' | ε
B ->cA'bB' | dB`
B' -> bB' | aA'bB' | ε

(b) E -> E+T | E-T | T
T -> T*F | T/F | F
F -> (E) | a

Put T' -> +T | - T
F' -> *F | /F

Solution: Non left-recursive grammar:

E -> TE'
E' -> T'E' | ε
T -> FT'
T' -> F'T' | ε
F -> (E) | a

By substituting T' and F', the above grammar can be written as follows:

E -> ET' | T
T -> TF' | F
F -> (E) | a

The grammar with out left –recursion is obtained as shown below:

E -> TE'
E' -> +TE' | -TE'| ε
T -> FT'
T' -> *FT' | /FT' | ε
F -> (E) | a

Left Factoring:

It is a process of factoring out the common prefixes of alternatives. That is left factoring is a grammar transformation (or) manipulation which is useful for producing a grammar suitable for recursive-decent parsing (or) predictive parsing.

1) For example, if we have the two productions: S -> iCtSeS | iCtS
On seeing the input token i, we cannot immediately tell which production to choose to expand S.

2) If A -> $\alpha\beta_1$ | $\alpha\beta_2$ are two productions, and the input begins with a non-empty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ (or) $\alpha\beta_2$. However, we

may defer the decision by expanding A to αA'. On left-factoring the original productions becomes

A -> $\alpha\beta_1 \mid \alpha\beta_2$ => A -> αA'
A' -> $\beta_1 \mid \beta_2$

Example 6: Consider the grammar

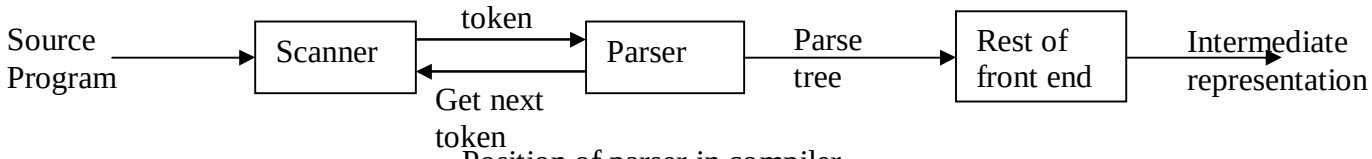
S -> iEtSeS | iEtS | a
E -> b

Solution : On left factoring we get

S -> iEtSS' | a
S' -> eS | ε
E -> b

Parsing:

In a natural language, parsing is the process of splitting a sentence into words. In compiler concepts, parsing is the process of determining if a string of tokens can be generated by a grammar.



Position of parser in compiler

A parser for a grammar is a program that takes as input a string and produces as output a parse tree. The input to a parser is typically a sequence of tokens. A parser can be constructed for any grammar. For any CFG, there is a parser that takes atmost O(n3) time to parse a string of n tokens. Programming language parsers almost always make a single left-to-right scan over the input, looking ahead one token at a time. The current token being scanned in the input is referred to as the lookahead symbol. Fig 6.7 shows position of parser in compilation process.

There are two types of parsing:

1. Top-down parsing
2. Bottom-up-parsing

In top down parsing, we attempt to construct the derivation of the input string or to build a parse tree starting from the top (root) to the bottom (leaves). On the other hand, in bottom up parsing, we build the parse tree, starting from the bottom (leaves) and work towards the top. In both cases input to the parser is scanned from left to right, one symbol at a time. Efficient parsers can be constructed more easily by hand using top-down methods. Parsers implemented by hand often work with LL grammars. Bottom-up methods can handle a larger class of grammars and translation schemes. Parsers for the larger class of LR grammars are usually constructed by automated tools.

Top-Down Parsing

The top-down construction of a parse tree is done by starting with the root, labeled with the starting non-terminal, and repeatedly performing the following two-steps

1. At node n, labeled with non-terminal A, select one of the productions for A and construct children at n for the symbols on the right side of the production.
2. Find the next node at which a sub tree is to be constructed.

For some grammars, the above steps can be implemented during a single left-to-right scan of the input string. The current token being scanned in the input is frequently referred to as the lookahead symbol. Initially, the lookahead symbol is the first, i.e., leftmost, token of the input string.

Top-down parsing can viewed as an attempt to find a leftmost derivation for an input string, i.e it can be viewed as an attempt to construct a parse tree in preorder. A general form of top-down parsing is called **recursive descent parsing**. A left recursive grammar can cause a top down parser (a recursive-descent parser) to go into an infinite loop. Therefore to use top-down parsing, we must eliminate all the left recursion from the grammar. The top-down parsing methods shown in the Fig

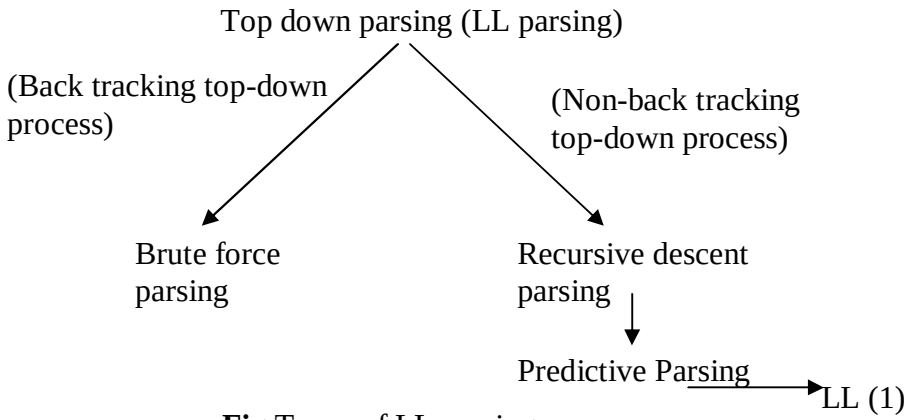


Fig Types of LL parsing

Brute-force parsing: This method operates as follows:

1. Given a particular non-terminal that is to be expanded, the first production for this non-terminal can be applied.
2. Then, within this newly expanded string, the left most non-terminal is selected for expansion and its first production is applied.
3. This process of applying productions is repeated for all subsequent non-terminals that are selected until the process cannot be continued.

Example 7: Consider the grammar and the input string :

w = accd

S -> aAd | aB

A -> b | c

B -> ccd | ddc

Trace the string w=accd of given grammar by Brute-Force Method

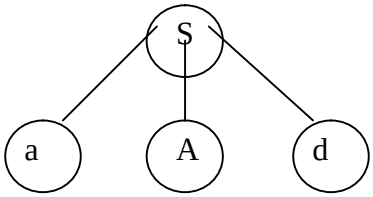
Solution : consider S first production, compare input string first syrmbol with generated symbol from the parse tree as shown in step 2 parse tree of Fig 6.9. The complete parsing steps are six as shown in the Fig 6.9.

Step1 :

Generated :

Input: ↑ accd

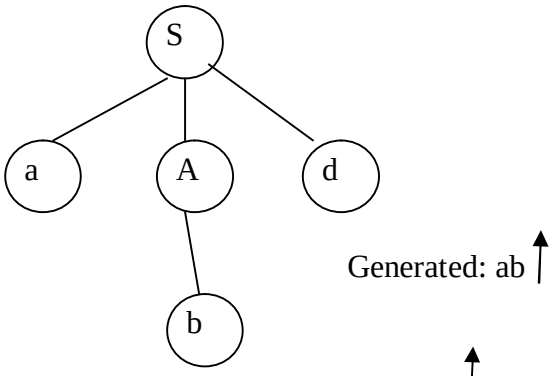
Step2:



Generated: a

Input: a↑ ccd

Sept 3:

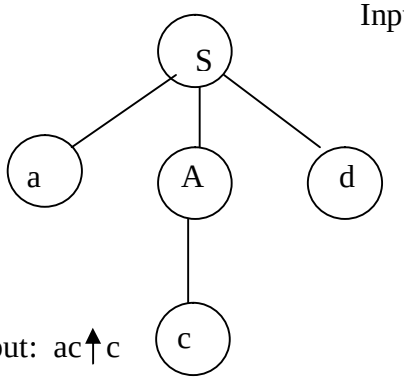


Generated: ab ↑

↑

Input: ac cd (mismatch b != c)

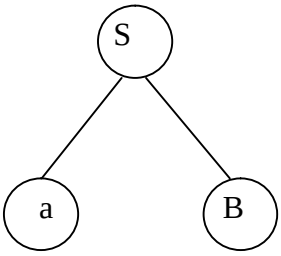
Step4:



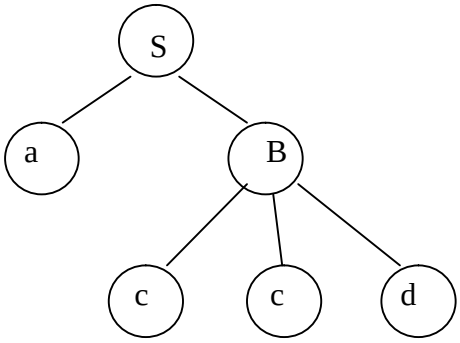
Generated: ac Input: ac↑c
Generated: acd
↑

Input: acc d (mismatch d != c)

Step5:



Generated: a
Input: a↑ccd



Generated: a ac | acc | accd |
Input: a|ccd ac | cd acc | d accd|

Fig: Parse Trees

Here the given string match with generated string so it is sucussfull parsing.

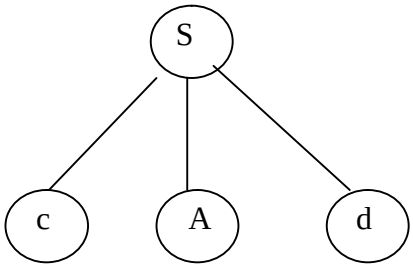
Example 8: Consider the grammar and the input string : cad

S -> cAd
A -> ab|a

Trace the string w=cad of given grammar by Brute-force method

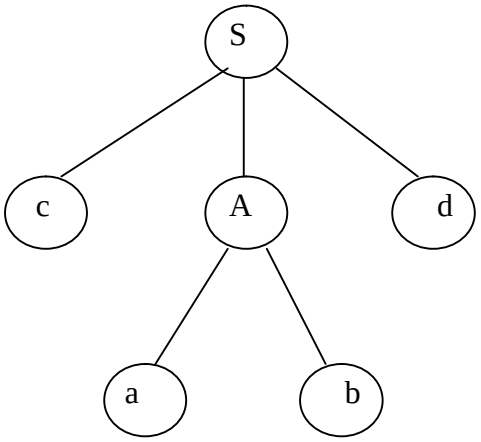
Solution : parsing steps of the string are shown in below Fig

Step1**:**



Generated: c
Input : c | ad

Step2:

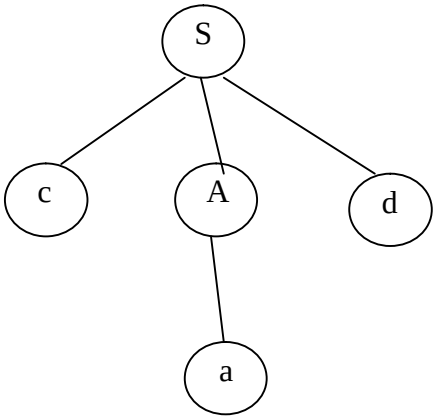


Generated: cab

Input: cad

(mismatch b!= d)

Step3:



Generated: cad
Input: cad
Successful completion of parsing

Recursive-Descent Parsing (RDP):

In order to find the left most derivation for an input string, we use the method top – down parsing. The general form of top-down method of syntax analysis is recursive descent parsing. A parser that uses a set of recursive procedures to recognize (to process) its input with no back tracking is called a **recursive-descent parser**. A procedure is associated with each non-terminal of a grammar. The sequence of procedures called in processing the input implicitly defines a parse tree for the input. Recursive descent parsing

eliminates the need for back tracking (i.e making repeated scans of the input) over the input; whereas Brute force method allows backtracking to process the input string.

Transition diagrams for recursive descent parsers:

We can create a transition diagram as a plan for a recursive descent parser. In case of the parser, there is one transition diagram for each non-terminal. The labels of edges are tokens and non-terminals. A transition on a non-terminal A is a call of the procedure for A. To construct the transition of a recursive descent parser from a grammar, first eliminate left recursion from the grammar and then left factor the grammar. Then for each non-terminal B do the following:

1. Create an initial n final state
2. For each production, B->X_1 X_2 ……X_n, create a path from the initial to the final state, with edges labeled X_1,X_2,X_3…X_n.

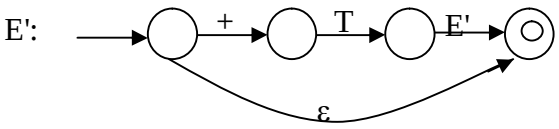
Consider the grammar G:
E -> E+T | T
T -> T * F | F
F -> (E) | id
Construct simplified transition diagram for G using RDP(or)
Explain RDP with suitable arithmetic expression.

Solution: we eliminate left recursion from given grammar G,on elimination of left recursion we get grammar G1 productions P1 as follows:

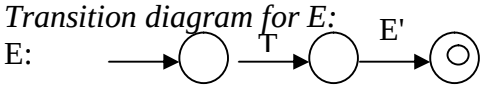
P1: E -> TE' T -> FT' F -> (E) |id
E' -> +TE' | ε T' -> *FT'| ε

Transition diagrams of each non terminal of grammar G shown in Fig

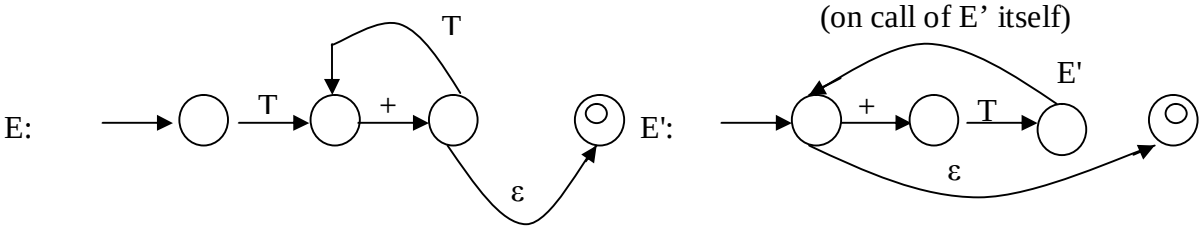
Transition diagram for E':



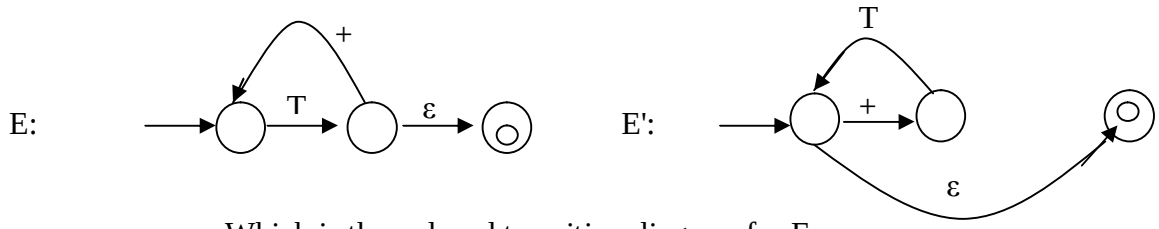
Transition diagram for E:



(on call of E' itself)

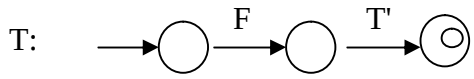


(on simplification)

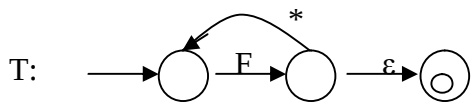


Which is the reduced transition diagram for E

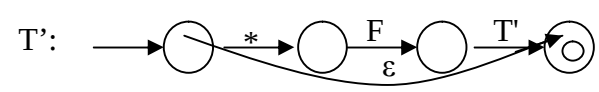
Transition diagram for T:

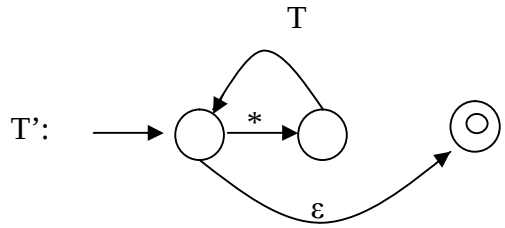


On simplification

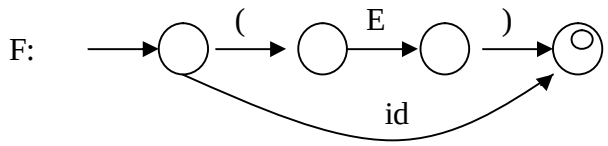


Transition diagram for T':





Transition diagram for F:



No simplification for F

Fig Transition Diagrams

Predictive parsing

Recursive-descent parsing is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. A procedure is associated with each non-terminal of a grammar. Predictive parsing is a special form of recursive-descent parsing in which the look ahead symbol unambiguously determines the procedure selected for each non-terminal. The sequence of procedures called in processing the input implicitly defines a parse tree for the input.

Designing a Predictive Parser

A predictive parser is a program consisting of a procedure for every non-terminal. Each procedure does two things.

1. It decides which production to use by looking at the look ahead symbol. The production with right side α is used if the look ahead symbol is in FIRST(α). If there is a conflict between two right sides for any look ahead symbol, then we

cannot use this parsing method on this grammar. A production with ε on the right side is used if the look ahead symbol is not in the FIRST set for any other right hand side.

2. The procedure uses a production by mimicking the right side. A non-terminal result in a call to the procedure for the non-terminal, and a token matching the look ahead symbol results in the next input token being read. If at some point the token in the production does not match the look ahead symbol, an error is detected.

Error Recovery in Predictive Parsing

An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when non terminal A is on top of the stack, a is the next input symbol, and the parsing table entry M[A, a] is empty.

Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.

Phase level recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert or delete symbols on the input and issue appropriate error messages. They may also pop from the stack.

Non-recursive Predictive Parsing

The key problem during predictive parsing is that of determining the production to be applied for a non-terminal. The non-recursive parser shown in the following Fig 6.12 looks up the production to be applied in a parsing table.

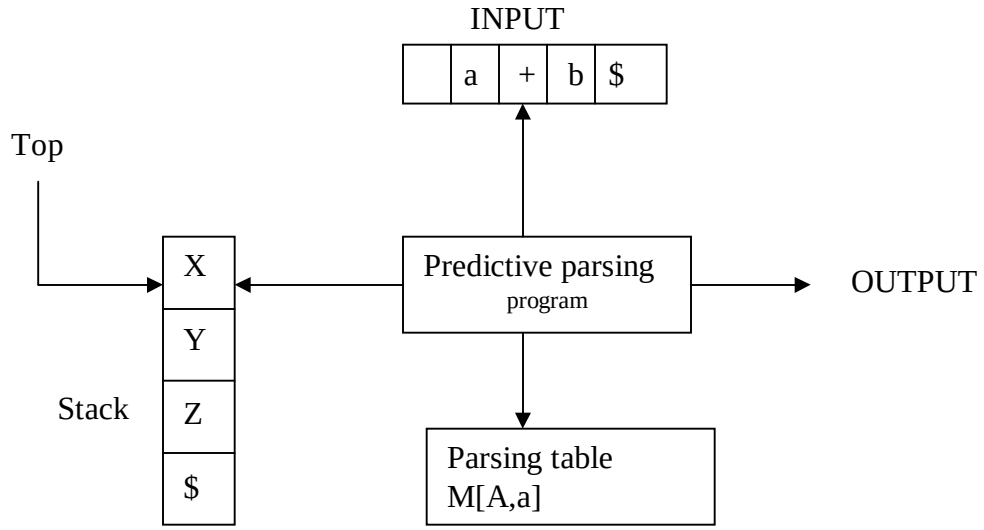


Fig Model of Non Recursive Predictive Parser

The non recursive predictive parser has

- An input buffer which contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string.
- A stack containing 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 $.
- Parsing table, a two-dimensional array M[A,a], where in A is a nonterminal, and a is a terminal or the symbol $.
- An output stream.

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

1. If X = a = $, the parser halts and announces successful completion of parsing.
2. If X = a ≠ $, the parser pops X off the stack and advances the 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. The entry will be either an X-production of the grammar or an error entry.

For example, if M[X,a]={X->ABCD}, the parser replaces X on the top of the stack by DCBA with A on top. If M[X,a]=error, the parser calls an error recovery routine.

FIRST and FOLLOW

The construction of a predictive parser is aided by two functions associated with a grammar G. These functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible. Sets of tokens yielded by the FOLLOW function can also be used as synchronizing tokens during panic-mode error recovery.

Rules for First Sets

1. If X is a terminal **then** First(X) is just X.
2. If there is a Production X → ε **then** add ε to first(X)
3. If there is a Production X → *a*B**then** add *a* to first(X) where *a* terminal and B string
4. If there is a Production X → Y1Y2..Yk **then** add first(Y1Y2..Yk) to first(X)
5. First(Y1Y2..Yk) is **either**
    1. First(Y1) (if First(Y1) doesn't contain ε)
    2. **OR** (if First(Y1) does contain ε) then First (Y1Y2..Yk) is everything in First(Y1) <except for ε > as well as everything in First(Y2..Yk)
    3. If First(Y1) First(Y2)..First(Yk) all contain ε **then** add ε to First(Y1Y2..Yk) as well.

Rules for Follow Sets

1. First put $ (the end of input marker) in Follow(S) (S is the start symbol)
2. If there is a production A → *a*B*b*, (where *a* and *b* can be a whole string) **then** everything in FIRST(*b*) except for ε is placed in FOLLOW(B).
3. If there is a production A → *a*B, **then** everything in FOLLOW(A) is in FOLLOW(B)
4. If there is a production A → *a*B*b*, where FIRST(*b*) contains ε, **then** everything in FOLLOW(A) is in FOLLOW(B)

Example : Compute FIRST and FOLLOW sets for the following grammar:

E->TE'
E'-> +TE'| ε
T -> FT'
T' -> *FT'| ε
F -> (E) | id

Solution:

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

FIRST (E') = {+, ε}

FIRST (T') = {*, ε}

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

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

FOLLOW (F) = FIRST (T') + FOLLOW (T') = {+, *,), $}

In the example, id and left parenthesis are added to FIRST (F) by rule (3) in the definition of FIRST with i =1 in each case, since FIRST (id) = {id} and FIRST ('(') = {(} by rule (1). Then by rule (3) with i=1, the production T->FT' implies that id and left parenthesis are in FIRST (T) as well in FIRST (E') by rule (2).

To compute FOLLOW sets, we put $ in FOLLOW (E) by rule (1) for FOLLOW. By rule (2) applied to production F-> (E), the right parenthesis is also in FOLLOW (E). By rule (3) applied to production E->TE', $ and right parenthesis are in FOLLOW (E'). Since E'=*>, they are also in FOLLOW (T).

Example : Find the FIRST and FOLLOW sets of each of the non- terminals for the Following grammar:

S -> aAB | bA | ε
A -> aAb | ε

B -> bB | c

Solution: For all the given production rule 3 is applicable therefore
FIRST (S) = {a, b, ε}
FIRST (A) = {a, ε}
FIRST (B) = {b, c}
As S is start symbol FOLLOW(S) = {$}, for the productions S -> aAB and A -> aAb, rule 2 and for the production S -> bA rule 3 are applicable therefore
FOLLOW (A) = FIRST (B) + FIRST{b}+ FOLLOW(S))
= {b,c} + {b} + {$}
= { b, c, $}
Consider the production S -> aAB apply rule 3 we get FOLLOW (B)=FOLLOW (S)={$}

Example : Compute FIRST and FOLLOW sets for the following

S-> iEtSS' | a
S' -> eS | ε
E -> b

Solution: FIRST(S) = {i , a}
FIRST(S') = {e, ε}
FIRST (E) = {b}
FOLLOW(S) = FIRST(S') + FOLLOW(S')
= {e , $ }
FOLLOW(S') = FOLLOW(S)
= {e , $}
FOLLOW (E) = {t}

Construction of Predictive parsing table

The following algorithm can be used to construct a predictive parsing table for a grammar G.
Algorithm: Construction of a predictive parsing table.

Step1:for each production A → α of the grammar, do steps 2 and 3.
Step2:For each terminal a in FIRST (α), add A → α to M [A, a].
Step3: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, $].
Step4:Make each undefined entry of M be error.

Example : Consider the following grammar
E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> (E) | id

(a) Compute FIRST and FOLLOW sets for each non terminal of the above grammar
Solution : Explanation for computing FIRST and FOLLOW set of this example given in Example 6.9. First set and Follow set s of all variables of grammar as follows.
First Sets:

FIRST (E) = FIRST (T) = FIRST (F) = {(,id}
FIRST (E') = {+, ε}
FIRST (T') = {*, ε}

Follow Sets:

FOLLOW (E) = {), $}
FOLLOW (E') = FOLLOW (E) = {) , $ }
FOLLOW (T) = FOLLOW (T') = FIRST (E') + FOLLOW (E') = {+,), $}
FOLLOW (T') = FOLLOW(T) = { +,), $}
FOLLOW(F) = FIRST(T') + FOLLOW(T) = { *, + ,), $}

(b) Construct predictive parsing table for the given grammar:

Solution: The predictive parsing table is two dimensional table, in which rows are corresponding to nonterminals and columns are corresponding to terminals of the grammar. We must remember that there should be extra column for $ symbol.
Consider production E -> TE' ,we determine FIRST(TE') is FIRST(T)= {(,id}.Therefore the entries of M[E, id]= E -> TE' and M[E, ( ]= E -> TE'. Next we consider E' production E' -> +TE',it is of the form $A \rightarrow \alpha$ here A= E' and α=+TE',determine FIRST(+TE').The FIRST(+TE')= {+} therefore the entry M[E',+] becomes E'->+TE'. Now consider the another E'-production E'-> ε, determine FOLLOW(E') because FIRST(E')={ ε}as per the rule 3. we know that FOLLOW(E')={ {), $}, hence M[E',)] and M[E', $]entries are filled with E'-> ε.In similar fashion we can find another entries of predictve parsing table as shown in Table

	id	+	*	(	)	$
E	E -> TE'			E -> TE'		
E'		E'->+TE'			E'-> ε	E'-> ε
T	T -> FT'		T -> FT'			
T'		T' -> ε	T'->*FT'		T' -> ε	T' -> ε
F	F ->id			F -> (E)		

The following example illustrates the top-down predictive parser using parser table .
String: id + id * id

MATCHED	STACK	INPUT	ACTION
	E$	id+id * id$	
	TE'$	id+id * id$	E->TE'

	FT'E'$	id+id * id$	T->FT'
	id T'E'$	id+id * id$	F->id
id	T'E'$	+id * id$	Match id
id	E'$	+id * id$	T'->Є
id	+TE'$	+id * id$	E'-> +TE'
id+	TE'$	id * id$	Match +
id+	FT'E'$	id * id$	T-> FT'
id+	idT'E'$	id * id$	F-> id
id+id	T'E'$	* id$	Match id
id+id	* FT'E'$	* id$	T'-> *FT'
id+id *	FT'E'$	id$	Match *
id+id *	idT'E'$	id$	F-> id
id+id * id	T'E'$	$	Match id
id+id * id	E'$	$	T'-> Є
id+id * id	$	$	E'-> Є

LL (1) Grammars:

The algorithm for predictive parsing table can be applied to any grammar G to produce a parsing table M. The parsing table M for some grammars may have one entry or multiple defined entries. If the grammar G is left recursive or ambiguous then M will have atleast one multiple defined entry, and such type of grammars are not considered to be LL(1) grammars.

A grammar whose parsing table has no multiply-defined entries is said to be LL (1).
The LL (1) grammars:

- are scanned from left-to-right
- are parsed by a leftmost derivation
- have one symbol look ahead.

Therefore, in the symbol LL(1) , the first 'L' stands for scanning the input from left to right, the second 'L' for producing a leftmost derivation, and the '1' for using one input symbol of lookahead at each step to make parsing action decisions. If we can show that a grammar is LL(1), we are guaranteed that it is backtrack free. We can construct a predictive parser that will not require any back tracking. The parsing table of LL(1) grammar should not contain multiply defined entry and this situation is obtained when the grammar is left recursive or ambiguous. So LL(1) grammar cannot be ambiguous or

left recursive. In order to obtain the parsing table for a given LL(1) grammar, we require the FIRST and FOLLOW sets.

Method 1: A grammar is LL (1) iff, whenever A->α and A-> β, are two distinct productions, then the following two conditions hold:
1. For nonterminal A, both α and β drive strings beginning with A.
2. Atmost one of α and β can derive the empty string.
3. If $\beta \Rightarrow^{*} \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A).
4. FIRST (α FOLLOW (A)) ∩ FIRST (β FOLLOW (A)) = Ø

Example : Verify whether the following grammar is LL (1) or not.

S -> iEtSS' | a

S' -> eS | ε

E -> b

Solution:
First Sets:
FIRST(S) = { i,a} as per the rule 2

FIRST(S') = {e, ε} as per the rule 2

FIRST(E) = {b} as per the rule 2

Follow Sets:

FOLLOW(S) = FIRST(S') + FOLLOW(S) = {e,$}

FOLLOW(S') = FOLLOW(S)

FOLLOW(E) = {t}

Therefore, S' entry: FIRST(S') FOLLOW(S') = {e, ε,$}

	a	b	E	i	t	$
S	S - > a			S'->iEtSS'		
S'			S'->ε S'->eS			S'->ε
E		E->b				

Since the above parsing table contains multiple entries, the grammar is not LL(1).

Method 2: To verify whether a grammar is LL(1) or not apply following rules:

1. A grammar G with out ε- rules is LL(1) if for each production of the form A -> $\alpha_1 | \alpha_2 | \ldots | \alpha_n$, FIRST($\alpha_1$), FIRST($\alpha_2$), …. FIRST($\alpha_n$) must be pair wise disjoint.

   $$FIRST(\alpha_1) \cap FIRST(\alpha_j) = \Phi \text{ for each } i \neq j$$

2. A grammar G with ε – rule is LL(1), if each production of the form A -> α | ε

   $$FIRST(\alpha) \cap FOLLOW(A) = \Phi$$

3. Left recursive G' cannot be LL(1).
4. Any ambiguous grammar is not LL(1).