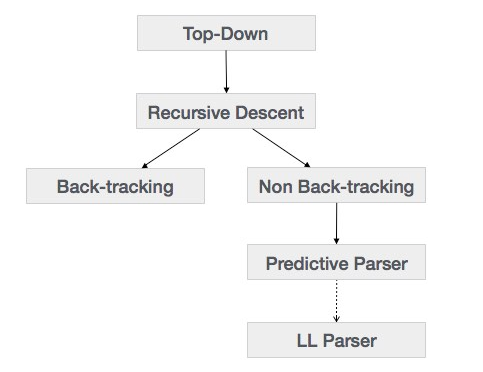
Top-Down Parsing

Top-down parsing can be viewed as attempt to find left most derivation for an input string. It can be viewed as attempting to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder .

Left-recursive grammar can cause a top-down parser to go into an infinite loop



**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.

***Example (backtracking)***

         Consider the grammar

S → cAd

A → ab | a                                                                               ***(Grammar 1)***

and the input string w = cad

·         To construct a parse tree for this string using top-down approach, initially create a tree consisting of a single node labeled S.

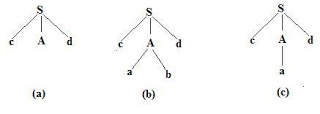


Fig 2

·         An input pointer points to c, the first symbol of w.

·         Then use the first production for S to expand the tree and obtain the tree (as in Fig 2(a)

·         The leftmost leaf, labeled c, matches the first symbol of w.

·         Next input pointer to a, the second symbol of w.

·         Consider the next leaf, labeled A.

·         Expand A using the first alternative for A to obtain the tree (as in Fig 2(b)).

·         Now have a match for the second input symbol. Then advance to the next input pointer d, the third input symbol and compare d against the next leaf, labeled b. Since b does not match d, report failure and go back to A to see whether there is another alternative. (Backtracking takes place).

·         If there is another alternative for A, substitute and compare the input symbol with leaf.

·         Repeat the step for all the alternatives of A to find a match using backtracking. If match found, then the string is accepted by the grammar. Else report failure.

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

·         For the grammar ***(Grammar 1)*** above write the recursive procedure for each nonterminal S and A.

***Procedure S***

procedure S() S → cAd

begin A → ab | a

      if input symbol = ‘c’ then

            begin

                  ADVANCE( );

                   if A( ) then

                       if input symbol = ‘d’ then

                                begin ADSVANCE( ); return true end

              end;

      return false

end

***Procedure A***

procedure A( )

begin

      isave := input-pointer;

      if input symbol = ‘a’ then

             begin

                   ADVANCE( );

                    if input symbol = ‘b’ then

                          begin ADVANCE( ); return true end

             end

       input-pointer := isave;

             /\* failure to find ab \*/

       if input symbol = ‘a’ then

               begin ADVANCE( ); return true end

       else return false

end

S → cAd

A → ab | a not left factoring

S → cAd

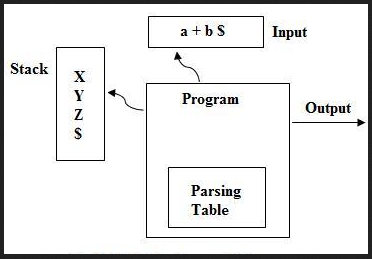
A → aC`

C`→ b | ε left factoring

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(1) grammar.

The first "L" in LL(1) 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.



predictive parser

Non recursive Predictive Parser

The predictive parser has an input, a stack, a parsing table, and an output.

-      The input contains the string to be parsed, followed by $, the right endmarker.

-      The stack contains a 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 nonterminal, and a is a terminal or the symbol $.

    -   The parser is controlled by a program that behaves as follows:

      -  The program determines *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. This entry will be either an X-production of the grammar or an error entry.

  If M[*X*,a] = {*X* → *UVW*}, the parser replaces *X* on top of the stack by *WVU* (with *U* on top).

  If M[*X*,a] = error, the parser calls an error recovery routine.

***Moves by Predictive parser using the input string***

***Predictive parsing program***

***repeat***

***begin***

            let *X* be the top stack symbol and a the next input symbol;

**if** *X* is a terminal or $ then

**if** *X* = *a* then

                          pop *X* from the stack and remove a from the input

**else**

                        ERROR( )

**else** /\* *X* is a nonterminal \*/

**if** M[*X*,a] = *X* *→* *Y1, Y2, … , Yk*then

***begin***

                             pop *X* from the stack;

                             push *Yk, Yk-1, … ,Y1* onto the stack, *Y1* on top

***end***

**else**

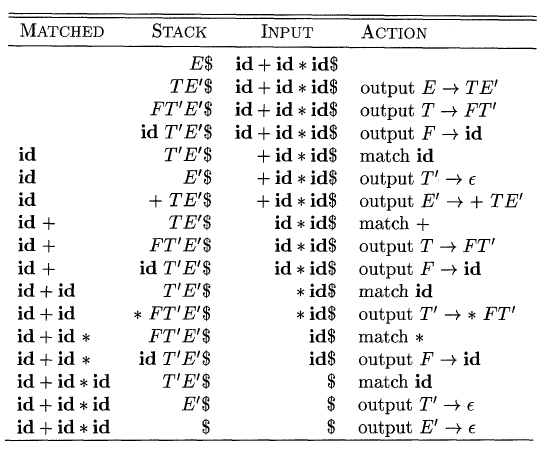
                        ERROR( )

***end***

***until*** *X* = $ /\* stack has emptied \*/

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 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) |  |  |

Predictive parser table



***Construction of Parsing Table:***

   Before constructing the parsing table, two functions are to be performed to fill the entries in the table.

FIRST( ) and FOLLOW( ) functions.

  These functions will indicate proper entries in the table for a grammar G.

To ***compute FIRST(X)*** for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set.

1. If *X* is terminal, then FIRST(X) is {X}.

2. If *X* is nonterminal and *X* → aα is a production, then add a to FIRST(X). If *X* → ε is a production, then add ε to FIRST(X).

3. If *X → Y1, Y2, … , Yk* is a production, then for all I such that all of *Y1*, … , Yi-1 are nonterminals and FIRST(*Yj*) contains ε for j = 1,2, … , i-1 (i.e., *Y1, Y2, … . Yi-1*  ε), add every non-ε symbol in FIRST(*Yi*) to FIRST(*X*). If ε is in FIRST(*Yj*) for all j = 1, 2, … , k, than add ε to FIRST(*X*).

  To ***compute FOLLOW(A)*** for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. $ is in FOLLOW(S), where S is the start symbol.

2. If there is a production A → αBβ, β ≠ ε, the everything in FIRST(β) but ε is   in FOLLOW(B).

3. If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε (i.e., β  ε), then everything in FOLLOW(A) is in FOLLOW(B).

***Example***

Consider the following grammar

E → E + T | T

T → T \* F | F

F → ( E ) | id                                                   ***(Grammar 2)***

  Compute the FIRST and FOLLOW function for the above grammar.

***Solution:***

        Here the ***(Grammar 2)*** is in left-recursion, so eliminate the left recursion for the ***(Grammar 2)*** we get;

FIRST FOLLOW

E → TE` {id , ( } {$ , ) }

E` → +TE` | ε { + , ε } {$ , ) }

T → FT` {id , ( } {+,$ , ) }

T` → \*FT` | ε { \* , ε } {+, $ , ) }

F → ( E ) | id              {id , ( }                  {+,\*, $ , ) }

***Example***

Consider the following grammar.   Compute the FIRST and FOLLOW function for the above grammar.

FIRST FOLLOW

S → ABCDE {a,b,c} {$}

A → a | ε {a, ε} {b,c}

B → b | ε {b, ε} {c}

C → c {c} {d,e,$}

D → d | ε {d, ε} {e,$}

E → e | ε {e, ε} {$}

***Example***

Consider the following grammar.   Compute the FIRST and FOLLOW function for the above grammar.

FIRST FOLLOW

S → Bb| Cd {a,b,c,d} {$}

B → aB | ε {a, ε} {b}

C → cC| ε {c, ε} {d}

***Exercise***

Consider the grammar

S → iEtSS` | a

S` → eS | ε

E → b                                                             ***(Grammar 4)***

·         Compute the FIRST and FOLLOW for the (Grammar 4)

***Construction of Predictive Parsing Table:***

The following algorithm can be used to construct a predictive parsing table for a grammar G

***Algorithm***  Constructing a predictive parsing table

***Input:*** Grammar G

***Output:*** Parsing table M

***Method:***

1. For each production A → α of the grammar, do step 2 and 3.

2. For each terminal a in FIRST(α), add A → α to M[A,a].

3. If ε is in FIRST(α), add A → α to M[A,b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A), add A → α to M[A,$].

4. Make each undefined entry of M error.

***Example***

  Construct the predictive parsing table for the ***(Grammar 3)*** using the ***Algorithm*** of constructing a predictive parsing table

FIRST FOLLOW

E → TE` {id , ( } {$ , ) }

E` → +TE` | ε {+, ε } {$ , ) }

T → FT` {id , ( } {+,$ , ) }

T` → \*FT` | ε { \* , ε } {+, $ , ) }

F → ( E ) | id              {id , ( }                  {+,\*, $ , ) }

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 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) |  |  |

***Example***

  Construct the predictive parsing table for the below grammar using the ***Algorithm*** of constructing a predictive parsing table

FIRST FOLLOW

S → aABC {a} {$}

A → a | bb {a, b} {a,b,$}

B → a | ε {a, ε} {b,$}

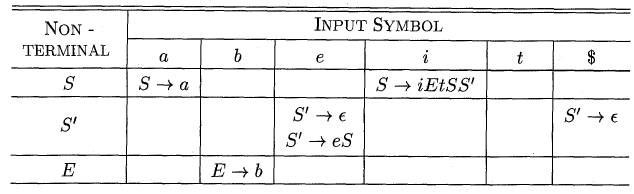
C → b | ε {b, ε} {$}

|  |  |  |  |
| --- | --- | --- | --- |
|  | A | b | $ |
| S | S → aABC |  |  |
| A | A → a | A → bb |  |
| B | B → a | B → ε | B → ε |
| C |  | C → b | C → ε |

predictive parsing table

***Example***

         Construct the predictive parsing table for the ***(Grammar 4)***



***LL(1) Grammars:***

o   When there is a situation that the parsing table consists of at least one multiply defined entries, then the easiest recourse is to transform the grammar by eliminating all left-recursion and then left-factoring whenever possible, to produce a grammar for which the parsing table has no multiply-defined entries.

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

o   A grammar is *LL(1)* if and only if whenever A → α | β are two distinct productions of G the following conditions hold:

1. For no terminal **a** do α and β derive strings beginning with **a**.

2. At most one of α and β can derive the empty string.

3. If β→\*  ε, then α does not derive any strings beginning with a terminal in FOLLOW(A).

Q1\ For the following regular expression a(b|c)\*

1. Construct the NFA using Thompson’s construction.
2. Convert the resulting NFA to the DFA
3. Minimize the states in the DFA’s

Q2\

a-Construct the predictive parsing table for the following grammar

S🡪 qABC

A🡪 a | bbD

B🡪 a | ϵ

C🡪 b | ϵ

D🡪 c | ϵ

b- parsing the following string qbbcab

Q1\ For the following regular expression a(b|c)\*

1. Construct the NFA using Thompson’s construction.
2. Convert the resulting NFA to the DFA
3. Minimize the states in the DFA’s

Q2\

a-Construct the predictive parsing table for the following grammar

S🡪 qABC

A🡪 a | bbD

B🡪 a | ϵ

C🡪 b | ϵ

D🡪 c | ϵ

b- parsing the following string qbbcab