Recursive Descent Parser

3.3.4 Elimination of Left Recursion

A grammar is said to be "left recursive" if it has a NT A such that there is a derivation $A \Rightarrow A\alpha$ for some string α (i.e., leftmost NT is same as left hand side NT). Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed.

Left recursion can be eliminated as follows:

- If there is a production $A \rightarrow A\alpha|\beta$, that can be replaced with sequence of two productions

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' | \varepsilon$$

$E \rightarrow TE'$
$E' \rightarrow +TE' / \varepsilon$

without changing the set of strings derivable from A.

(eg.) Consider the following grammar for arithmetic expressions.

$$E \rightarrow E+T|T \quad \Rightarrow A \rightarrow A\alpha / \beta$$
$$T \rightarrow T * F|F \qquad A = E \quad \alpha = +T$$
$$F \rightarrow (E)|id \qquad \beta = T$$

First eliminate the left recursion for E (of the form $A \rightarrow A\alpha|\beta$) as

$$E \rightarrow TE'$$
$$E' \rightarrow +TE'|\varepsilon$$

$A \rightarrow A\alpha / \beta$
$T \rightarrow T*F / F$

(here $\alpha = +T$, $\beta = T$)

Then eliminate for T as

$A = T \quad \alpha = *F$
$\beta = F$

$$T \rightarrow FT'$$
$$T' \rightarrow *FT'|\varepsilon$$

Thus the obtained grammar after eliminating reft recursion is

$T \rightarrow FT$
$T' \rightarrow *FT' /$

$$E \rightarrow TE'$$
$$E' \rightarrow +TE'|\varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT'|\varepsilon$$
$$F \rightarrow (E)|id$$

- In general, we can eliminate immediate left recursion if there are production of the form

$$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_m | \beta_1 | \beta_2 | \ldots | \beta_n$$

Transform to

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

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

where

A – left recursive NT

α – sequence of NT_3 and T_3 that is not ε

β – sequence of NT_3 and T_3 that does not start with A.

- There is a situation that left recursion may occur by two or more grammars.

(eg.) $S \rightarrow Aa|b$

$A \rightarrow Ac|Sd|\varepsilon$

There is a left recursion because

$$S \Rightarrow Aa \Rightarrow Sda$$

But it is not immediately left recursive, that cannot be eliminated using the above stated algorithm. Here, left recursion is removed. Systematically, if the grammar has no cycles ($A \Rightarrow A$) or ε-productions ($A \rightarrow \varepsilon$). In this type of situation, left-recursion is eliminated using the following algorithm.

Algorithm: Eliminating left recursion.

Input. Grammar G with no cycles or ε-productions.

Output. An equivalent grammar with no left recursion.

Method. Apply the algorithm for dangling-else grammer. Note that the resulting non-left-recursive grammar may have 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 $A_i \rightarrow A_j \gamma$

        by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$.

        where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current A_j-productions;

    end

    eliminate the immediate left recursion among the A_i-productions

end

Algorithm to eliminate left recursion from a grammar.

Using the above algorithm, the grammar can be re-written as follows:

After 1st step (substitute S by its right hand side in the rules)

$S \rightarrow Aa|b$

$A \rightarrow Ac|Aad|bd|\varepsilon$

After 2nd step (removal of left recursion)

$S \rightarrow Aa|b$

$A \rightarrow bd\ A'\ |\ A'$

$A' \rightarrow C\,A'\ |ad\ A'\ |\ \varepsilon$

3.3.5 Left Factoring

Left factoring is a grammar transformation that is usedful for producing a grammar suitable for predictive parsing (Top-down parsing). In top-down parsing when it is not clear which of two alternative productions to use to expand a NT A, defer the decision till we have seen enough input.

If there are productions of the form

$$A \rightarrow \alpha\beta_1|\ \alpha\beta_2,$$

then left factoring is done by re-writing the grammar with following two productions

$A \rightarrow \alpha\, A'$

$A' \rightarrow \beta_1|\beta_2$

(eg.) Consider the abstract of "dangling-else" grammar.

$S \rightarrow iEtS\ |\ iEtSeS\ |\ a$

$E \rightarrow b$

(i–if, t–then, e–else, E – expression, S – statement, α = iEtS, $\beta_1 - \varepsilon$, $\beta_2 - eS$).

If top-down parser want to derive a string (say ibta) that starts with symbol i, it is not clear to choose either the first or second production of S. Left factoring solves this problems.

Grammar after left factoring is as follows:

$S \rightarrow iEtS\,S'\ |a$

$S' \rightarrow eS|\varepsilon$

$E \rightarrow b$

Now for "*ibta*", the derivation becomes

$S \rightarrow iEtS\,S'$

$\rightarrow ibtS\,S'$

$\rightarrow ibta\,S'$

$\rightarrow ibta$

4.1.1 Difficulties with Top-Down Parsing

An easy way to implement a top-down parser is to create a procedure for each non-terminal. When grammars derive an infinite number of strings, recursive procedures are essential. The above grammar generates only two sentences (cabd or cad). Hence there is no need for recursive procedures.

There are several difficulties with top-down down parser when writing the procedures.

1. Left-Recursive Grammars

Left-recursive grammar can cause a top-down parser to go into an infinite loop when writing procdure. To use top-down parsing, we must eliminate all left-recursion from the grammar.

(eg.) Consider the grammar.

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Top-down parser writes procedures for E, T and F.

```
Procedure E( )
begin
   E( )
   .
   .
   .
end
```

```
Procedure T( )
 begin
 T( )
 .
 .
 end
```

```
Procedure F( )
begin
.
.
.
end
```

Here procedure E( ) infinitely calls E( ) that leads to infinite loop.

Details about the left-recursive grammar and its elimination are explained in the previous chapter.

2. Backtracking

If we make a sequence of erroneous expansions and subsequently discover a mismatch, we may have to undo the semantic effects of making these erroneous expansions. (eg.) entries made in the symbol table might have to be removed. Since undoing semantic actions requires a substantial overhead, it is reasonable to consider top-down parsers that do no backtrackings. There are top-down parsers with no backtracking such as **recursive-descent parser** and **predictive parser.**

2M

3. Rejection of Valid Strings

Due to backtracking, we may reject some valid sentences. For example, in the above explained grammar, we used **a** and then **ab** as the order of the alternates for A. There we could fail to accept cabd.

4. Error Reporting

When failure is reported in top-down parsing, we have very little idea where the error actually occured.

4.1.2 Recursive-Descent Parsing

In many practical cases a top-down parser needs no backtrack. A parser that uses a set of recursive procedures to recognize its input with no backtracking is called as **recursive-descent** parser. The recursive procedures can be quite easy to write and fairly efficient if written in a language that implements procedure calls efficiently.

Need for Recursive-Descent Parsing

Before writing procedures (recursive) for recursive-descent parser it is necessary to do the following actions for the given grammar.

1. Elimination of left-recursion
2. Left factoring) 2M.

Example for Recursive-Descent Parsing

Consider the grammar that recognizes arithmetic expressions

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

This grammar contains left recursion (leftmost symbol of right side of production in same as left side of production). If we tried to write procedures for this grammar that lead to infinite loop. Hence it is necessary to eliminate left-recursion. After eliminating the left-recursion, the grammar becomes

$$E \rightarrow TE^{1}$$

$$E^{1} \rightarrow + TE^{1} \mid \varepsilon$$

$$T \rightarrow FT^{1}$$

$$T^{1} \rightarrow * FT^{1} \mid \varepsilon$$

$$F \rightarrow (E) \mid id$$

Now we can write procedures for the grammar as follows:

Recursive Procedures

```
procedure E( );
begin
    T( );
    EPRIME ( )
end;

procedure EPRIME ( );
If input-symbol = '+' then
    being
        ADVANCE ( );
        T.( );
    EPRIME ( )
    end;
```

```
procedure T( );
begin
        F( );
        TPRIME ( )
end;

procedure TPRIME ( );
if input-symbol = '*' then
        begin
            ADVANCE ( );
            F( );
            TPRIME( )
        end;


procedure F( );
If input-symbol = 'id' then
        ADVANCE ( )
else if input-symbol = '(' then
        begin
            ADVANCE ( );
            E( );
            if input-symbol = ')' then
               ADVANCE( )
            else ERROR( )
         end
 else ERROR( )
```

Fig. : Mutually recursive procedures to recognize arithmetic expressions

The sequence of procedures called using the above parser to recognize a string **id + id * id** are

Procedures	Input String	Comment
E( )	**id** + id * id	
T( )	**id** + id * id	
F( )	**id** + id * id	
ADVANCE( )	id + id * id	input symbol is 'id'
TPRIME( )	id + id * id	input symbol is not '*' So return.
EPRIME( )	id + id * id	input symbol is '+'
ADVANCE( )	id + **id** * id	
T( )	id + **id** * id	
F( )	id + **id** * id	
ADVANCE( )	id + id * id	input symbol is 'id'
TPRIME( )	id + id * id	
ADVANCE( )	id + id * **id**	input symbol is '*'
F( )	id + id * **id**	
ADVANCE( )	id + id * **id**	input symbol is 'id'
TPRIME( )	id + id * **id**	input symbol is not '*' So return. This eventually return to E( ). Then halt. Thus deriving a valid string without back tracking.