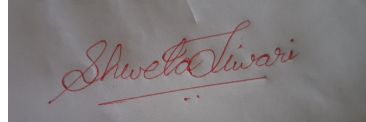


CD: COMPILER DESIGN
CD: UNIT-2 09/2022

SEPTEMBER 2022 / IT-3rd year, Vth semester
FALL SEMESTER, YEAR (Vth, 3rd)
FALL SESSION (2022-23)
(CD)

MS. SHWETA TIWARI
Published: SEPTEMBER, 2022

PREPARED FOR
Engineering Students
All Engineering College



CD: COMPILER DESIGN

TOPIC On : UNIT-2 UNIT-2 TOP-DOWN PARSER-RECURSIVE DESCENT PARSER

By SHWETA TIWARI

Under On: Basic Parsing Techniques

TOPIC On : UNIT-2 TOP-DOWN PARSER-RECURSIVE DESCENT PARSER

II-TOP-DOWN-RECURSIVE DESCENT

Top-Down Parsing- Recursive Descent Parsing:

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

A general form top-down parsing called recursive descent parsing, involves backtracking, that is making repeated scans of the input. A special case of recursive descent parsing called predictive parsing, where no backtracking is required.

Consider the grammar

$$S \rightarrow cAd$$

$$A \rightarrow ab \mid a$$

and the input string $w=cad$. Construction of the parse is shown in fig 1.

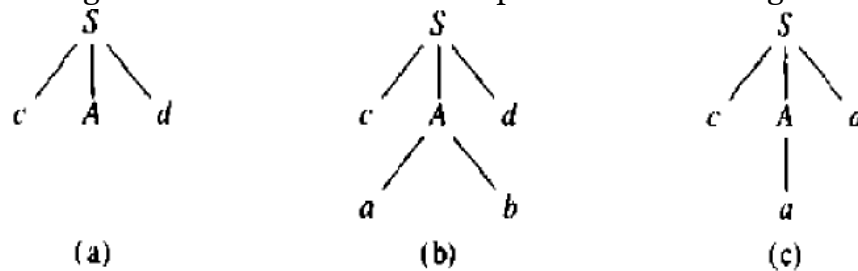


Fig 1 Steps in Top-down Parse

The leftmost leaf, labeled c , matches the first symbol of w , hence advancing the input pointer to a , the second symbol of w . Fig 1(b) and (c) shows the backtracking required to match the input string.

Recursive Descent parser

- It is a top down parser.
- It starts from the top that is the root and applies several productions and gets a bottom that has leaf nodes.
- The name suggests a specific meaning. It contains a recursive procedure.

- So grammar contains three non-terminal that's why this parser contains **three recursive procedures.**
- Recursive means "A function which is called by itself".
- Means the calling function and called function are the same.
- So in this we have to write a recursive procedure of each and every non terminals that is available in Grammar.

Steps of construction for recursive Descent parser

1. If input is a non terminal/variable then call the corresponding procedure of the non terminal. (*meaning call corresponding function*).
2. If input is a terminal symbol then compare terminal with input string (*corresponding input symbol*). If they are the same, then we have to increment the input pointer. (*meaning we have to compare the next input with corresponding terminals.*)
3. If a non terminal procedures 3 productions then we have to write those three productions in the corresponding non terminals. (*meaning produce more than 1 production then all production codes should be written in corresponding function or procedure.*)
4. There is no need to define any main function or no need to define any variable. (*If we define a main function then in the main function. We have to call the start symbol/production/ function.*)

EXAMPLE

$E \rightarrow iE'$

$E' \rightarrow +iE' \mid \epsilon$

Here are two non-terminals.

So, define 2 functions or procedures.

**** Do follow C Language Rule: It is basic Programming Language.**

$E()$

{

if(input=="i")

input++ ;

Epro();

}

Epro()

{

if(input=="+")

{

input++ ;

if(input=="i")

input++ ;

Epro();

```

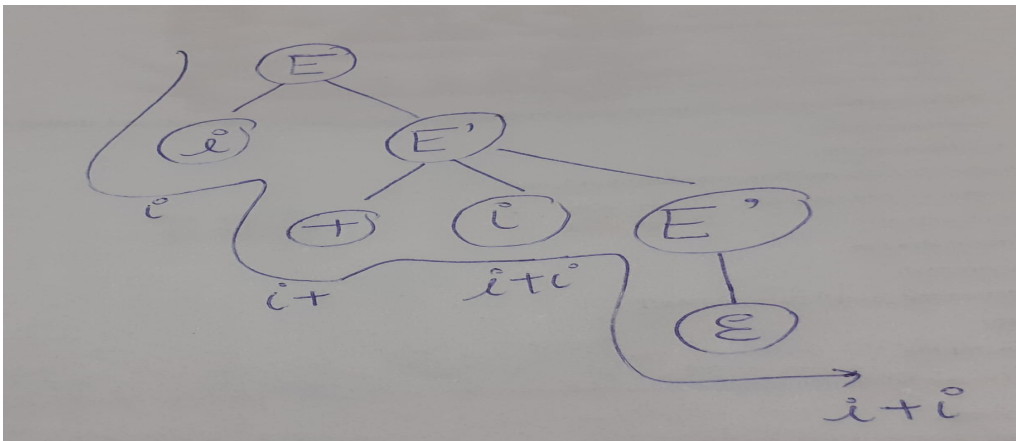
}
else
return ;
}

```

Parse this string $w=i+i$

Add the \$ symbol at the end of the string which shows that there is no more terminal for parse.
And looks like, $w=i+i\$$

After accepting string($w=i+i$) by the above program of given grammars, the output is generated in the form of a syntax tree.



Question: Write a recursive function for grammar :

And is this string parse $w=id+id$?

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid$

$\epsilon T \rightarrow FT'$

$T' \rightarrow *FT' \mid$

$\epsilon F \rightarrow (E)$

$\mid id$

Define function for each Non-Terminal. In this there are 5 Non-Terminal. So, define 5 functions.

**** Do follow C Language Rule: It is basic Programming Language.**

E()

{ T();

Epro();

}

Epro()

{ if (input == "+"

{ input++;

T();

Epro();

```

}
else
return( );
}

```

$$\begin{aligned} & \mathbf{T}() \\ & \{ F(); \\ & \quad T_{pro}(); \\ & \} \end{aligned}$$

```
Tpro()
{ if(input== “*”)
{ input++;
F();
Tpro();
}
else
return;
}
```

```

F()
{ if ( input==“(” )
{
input++;
E();
if (input==”)”)
input++;
}
else
if (input==”id”)
input++;
}

```

Add the \$ symbol at the end of the string which shows that there is no more terminal for parse. And looks like, $w=id+id\$$

After accepting string($w=id+id$) by the above program of given grammars, the output is generated in the form of a syntax tree.

