

Name: Digvijay Jondhale

Roll No: PC-32

Batch: C1

PRN: 1032201770

SSCP LCA - 06.

Aim: Implement Recursive Descent parser for given

grammar: $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Theory:

- CFG, Non-Terminals, Terminals, Productions, Derivation Sequence:
- > Context-Free Grammar (CFG): A CFG is a formal grammar used to describe the syntax of programming languages and other formal languages. It consists of a set of non-terminals, a set of terminals, a set of production rules, and a start symbol.
- > Non-Terminals: Non-terminals are symbols representing syntactic categories or variables in a CFG. They are placeholders for language constructs and can be replaced by other symbols through production rules.
- > Terminals: Terminals are symbols that appear in the input language and cannot be further expanded. They represent actual tokens or characters.
- > Productions: Productions define the rules for generating valid sentences or expressions in the language. They specify how non-terminals can be replaced by a sequence of non-term & term.

> Derivation Sequence:

A derivation sequence is a sequence of production rule applications that starts from the start symbol and transforms it into a valid sentence or expression in the language.

2. Introduction to Recursive Descent Parser:

- A Recursive Descent Parser is a type of top-down parser used in compiler construction to analyze the syntax of programming languages. It corresponds closely to the structure of the context-free grammar for the language being parsed.
- > Recursive Descent Parsing involves writing parsing functions or procedures for non-terminals in the grammar. These functions recursively apply production rules to match and validate the input against the grammar.
- > Each non-terminal in the CFG corresponds to a parsing function, and each production rule corresponds to a set of choices or alternatives in the parsing code.
- > Recursive Descent Parsers are easy to understand and implement, especially when the grammar is LL(1) (left-to-right scan, leftmost derivation, one-token lookahead).

3. Elimination of Left Recursion:

- > Left recursion occurs in a CFG when a non-terminal can produce itself directly or indirectly through a sequence of other non-terminals and terminals.
- > Left recursion can lead to infinite loops in recursive descent parsers and is generally not allowed in LL(1) grammars.

- * To eliminate left recursion, you can follow these steps:
 - 1) Identify left-recursive non-terminals in the grammar.
 - 2) Replace left-recursive rules with equivalent right-recursive rules.
 - 3) Introduce new non-terminals for the right-recursive rules to maintain the original language's structure.

4. Example: Elimination of Left Recursion:

Given left-recursive grammar:

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

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

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

To eliminate the immediate left recursion, you can rewrite the grammar as follows:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

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

In this modified grammar, E and T are no longer directly left-recursive, and the non-terminals E' and T' are introduced to handle the recursive parts of the grammar. This ensures that the grammar can be parsed without issues related to left recursion.

CODE:

```
import java.util.*;
import java.io.*;

class As6
{
    /*The following grammar has been implemented:
    * E -> TE'
    * E' -> +TE'|sigma
    * T -> FT'
    * T' -> *FT'|sigma
    * F -> (E)|id
    */
    static int index=0;
    static String input;

    public static void E()
    {
        T();
        Eprime();
    }

    public static void Eprime()
    {
        if(input.charAt(index)=='+')
        {
            index++;
            T();
            Eprime();
        }
    }

    public static void T()
    {
        F();
        Tprime();
    }

    public static void Tprime()
    {
        if(input.charAt(index)=='*')
        {
            index++;
            E();
        }
    }

    public static void F()
    {
        if(input.charAt(index)=='i' && input.charAt(index+1)=='d')
        {
            index+=2;
        }
        else if(input.charAt(index)=='(')
        {
            index++;
            E();
            if(input.charAt(index)=='')
            index++;
        }
    }
}
```

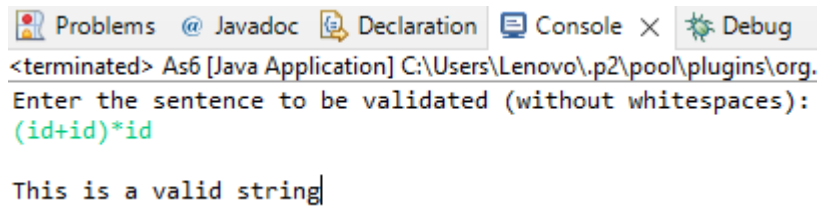
```

    }

    public static void main(String[] args) throws IOException
    {
        Scanner s = new Scanner(System.in);
        System.out.println("Enter the sentence to be validated (without whitespaces):");
        input = s.nextLine();
        input = input.concat("$"); //Terminal character
        E();
        if(input.charAt(index) == '$')
            System.out.println("\nThis is a valid string");
        else
            System.out.println("\nThis is an invalid string");
        s.close();
    }
}

```

OUTPUT:



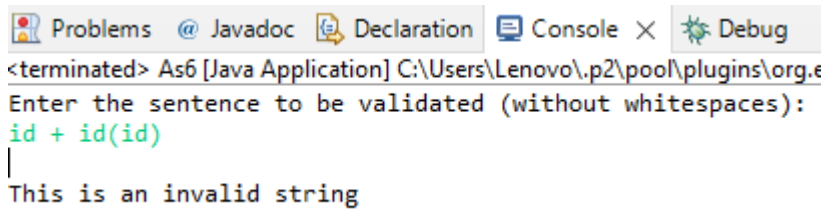
The screenshot shows an IDE window with tabs for Problems, Javadoc, Declaration, Console, and Debug. The Console tab is active, displaying the following text:

<terminated> As6 [Java Application] C:\Users\Lenovo\.p2\pool\plugins\org.

Enter the sentence to be validated (without whitespaces):

(id+id)*id

This is a valid string|



The screenshot shows the same IDE window with the Console tab active, displaying the following text:

<terminated> As6 [Java Application] C:\Users\Lenovo\.p2\pool\plugins\org.e

Enter the sentence to be validated (without whitespaces):

id + id(id)

|

This is an invalid string