Name: Digrijay Jondhale
Roll No: PC-32
Batch : C1
PRN: 10322017770

SSCD LCA - 06.

Aim: Implement Recursive Descent parser for given grammer:

$E \rightarrow E+T \mid T$

$T \rightarrow T^{*} F \mid F$

$F \rightarrow (E) \mid id$

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 syntatic 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 grammer for the language being parsed.
- Recursive Descent Parsing involves writing parsing functions or produres for non-terminals in the grammer. These functions recursively apply production rules to match and validate the input against the grammer.
- 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 grammer is LL(1) (left-to-right scan, leftmost derivation, one-token lookahead).

3. Elimation 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) grammers.

* To elimate left recursion, you can follow these steps:
(i) Identify left-recursive non-terminals in the grammer.
(ii) Replace left-recursive rules with equivalent right-recursive rules.
(iii) 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 grammer:
$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$
To eliminate the immediate left recursion, you can rewrite the grammer as follows:
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id$

In this modified grammer, 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 grammer. This ensures that the grammer can be parsed without issues related to left recursion.