SLR PARSING

ABSTRACT :

In compiler design, SLR parsing is a widely used technique for parsing context-free grammars. The SLR parsing algorithm uses a bottom-up approach to parse the input string and construct a parse tree. This report provides an in-depth exploration of the SLR parsing technique, including the theory behind it and its implementation.

INTRODUCTION :

Compilers are essential tools in computer science that translate source code into executable programs. One of the key components of a compiler is the parser, which takes the source code and analyzes its structure to generate a parse tree. There are different types of parsers, including top-down and bottom-up parsers, each with their advantages and disadvantages.

SLR parsing is a bottom-up parsing technique that uses a deterministic finite automaton to parse the input string. The SLR parsing algorithm is efficient and can handle a wide range of context-free grammars. However, it has some limitations, such as its inability to handle left-recursive grammars.

This report provides a comprehensive overview of SLR parsing in compiler design. We will discuss the implementation of the SLR parsing algorithm

METHODOLOGY/TECHNIQUES :

The methodology for SLR parsing involves several techniques and algorithms that work together to parse the input string and construct a parse tree. Here are some of the key techniques used in SLR parsing:

1. Grammar Definition : The first step in SLR parsing is to define the context-free grammar that represents the programming language being parsed. The grammar should be written in a formal notation, such as BNF (Backus-Naur Form).
2. Item Sets : An item set is a set of production rules with a dot (.) placed at various positions within the rule. The item sets represent the state of the parsing process at a particular point.
3. LR(0) Automaton : The LR(0) automaton is a deterministic finite automaton that is constructed from the item sets. It represents the state transitions of the parser based on the input symbol.

METHODOLOGY/TECHNIQUES :

4. LR(0) Table Construction : The LR(0) table is constructed from the LR(0) automaton. The table contains the actions to be taken by the parser based on the current state and input symbol. The actions can be either a shift or a reduce operation.

5. Lookahead Sets : The lookahead sets represent the set of input symbols that can follow a particular item set.

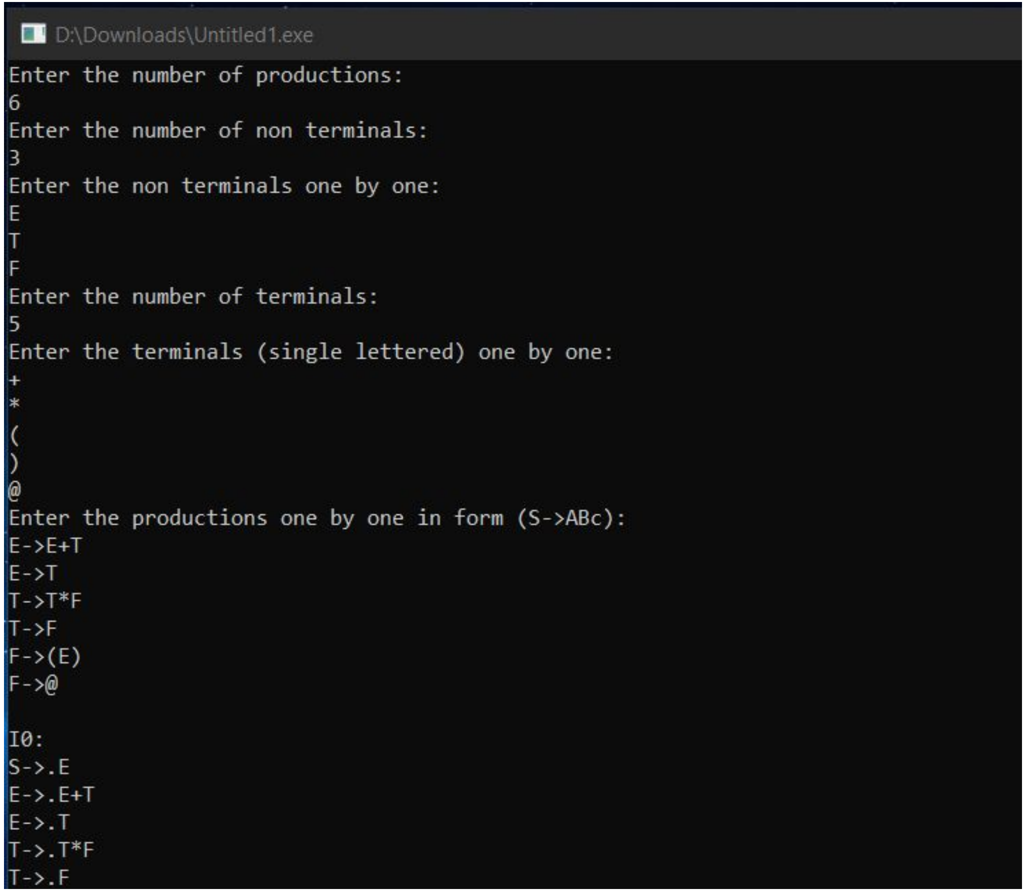
6. Conflict Resolution : Conflicts can arise in the parsing process when the parser encounters a state with multiple possible actions. The conflicts can be resolved using various techniques, such as precedence and associativity rules.

METHODOLOGY/TECHNIQUES :

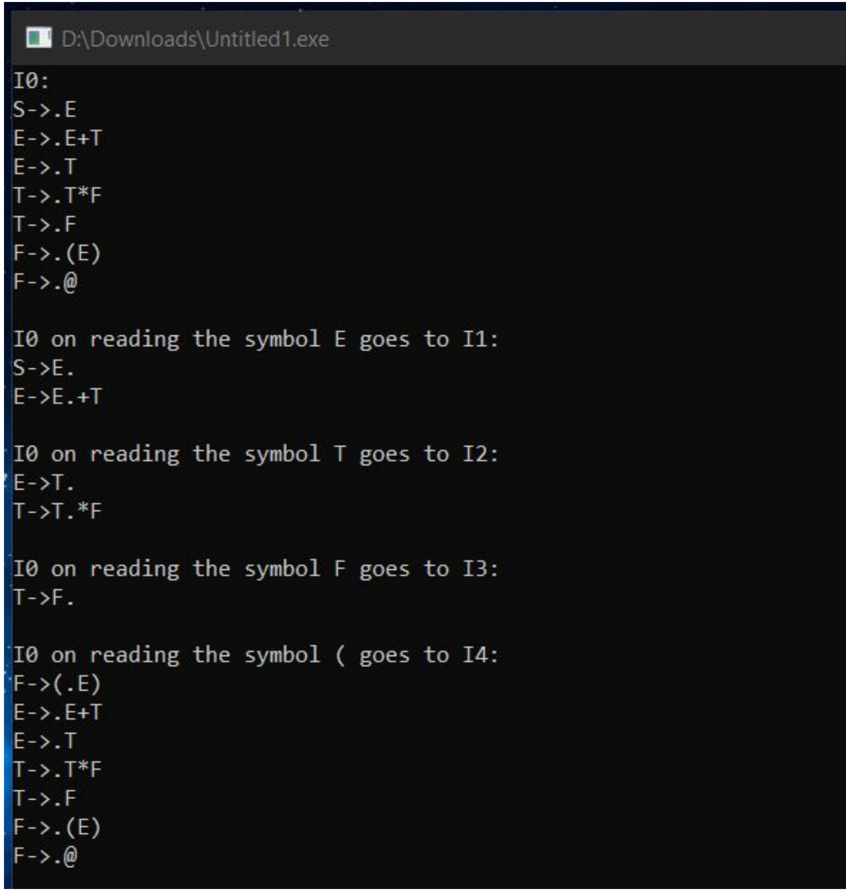
7. Parse Tree Construction : Once the parsing process is complete, the parse tree is constructed from the stack of symbols generated during the parsing process. The parse tree represents the syntactic structure of the input string.

SLR parsing is a powerful technique for parsing context-free grammars, and it is widely used in compiler design. However, it has some limitations, such as its inability to handle left-recursive grammars. To address this limitation, other parsing techniques, such as LR(1) and LALR(1), have been developed.

IMPLEMENTATION :

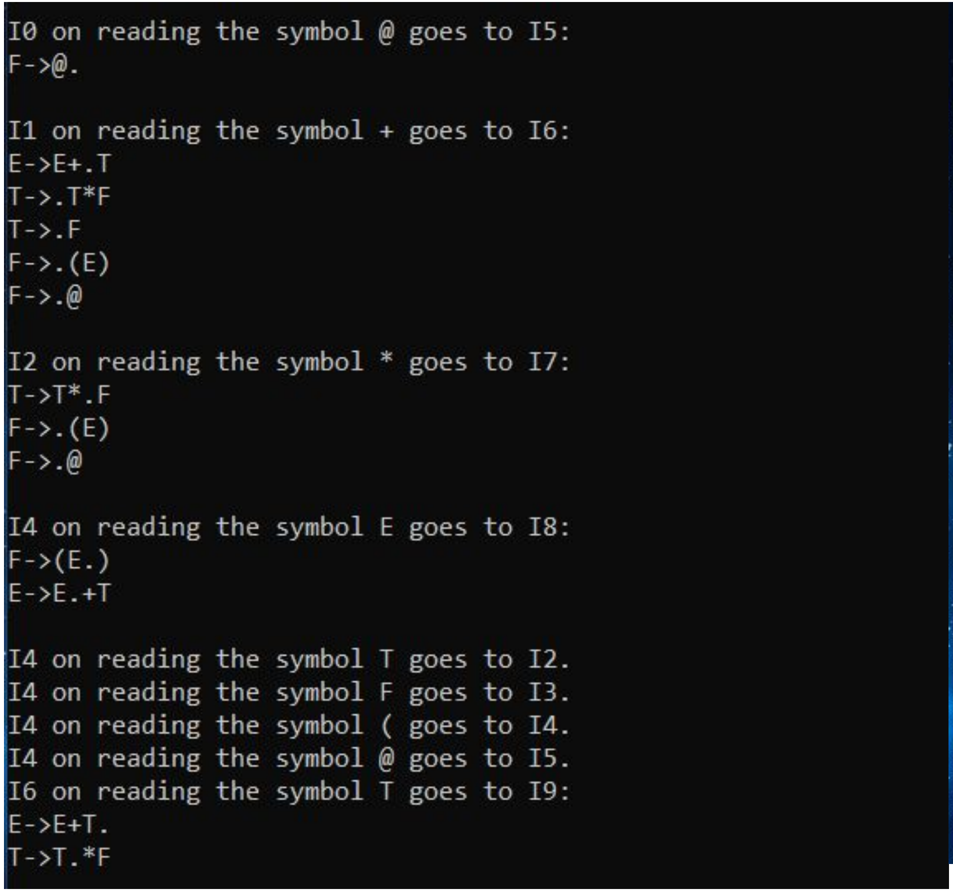


```
Enter the number of productions:
6
Enter the number of non terminals:
3
Enter the non terminals one by one:
E
T
F
Enter the number of terminals:
5
Enter the terminals (single lettered) one by one:
+
*
(
)
@
Enter the productions one by one in form (S->ABc):
E->E+T
E->T
T->T*F
T->F
F->(E)
F->@

I0:
S->.E
E->.E+T
E->.T
T->.T*F
T->.F
```

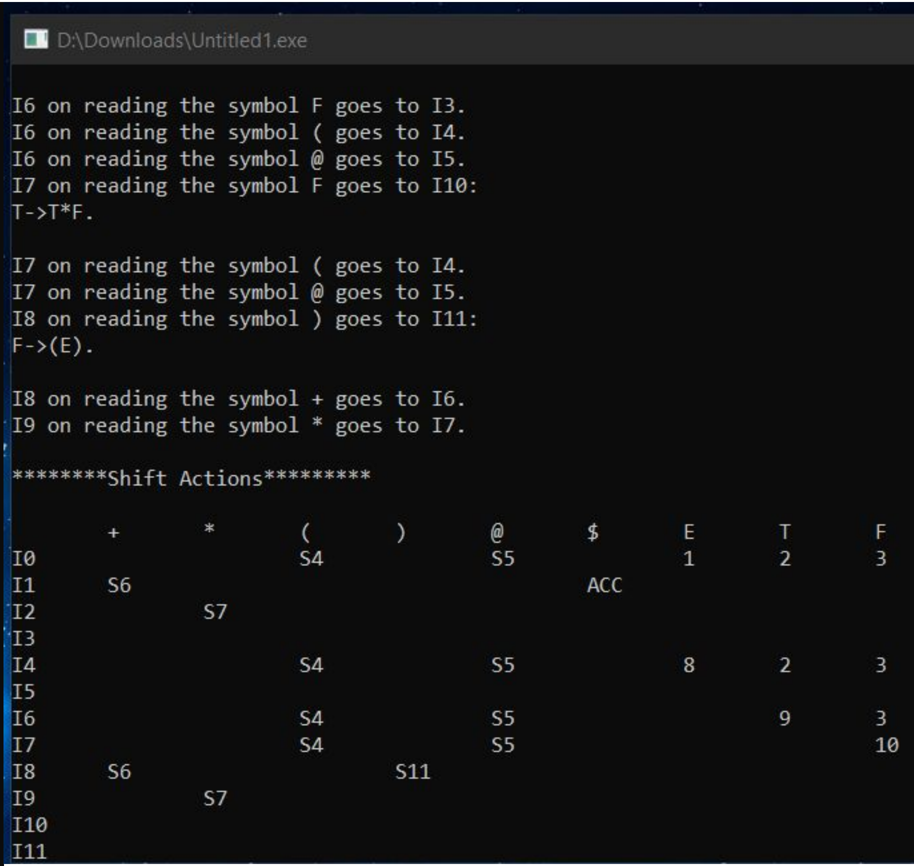


```
I0:
S->.E
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.@

I0 on reading the symbol E goes to I1:
S->E.
E->E.+T

I0 on reading the symbol T goes to I2:
E->T.
T->T.*F

I0 on reading the symbol F goes to I3:
T->F.

I0 on reading the symbol ( goes to I4:
F->(.E)
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.@
```

IMPLEMENTATION :

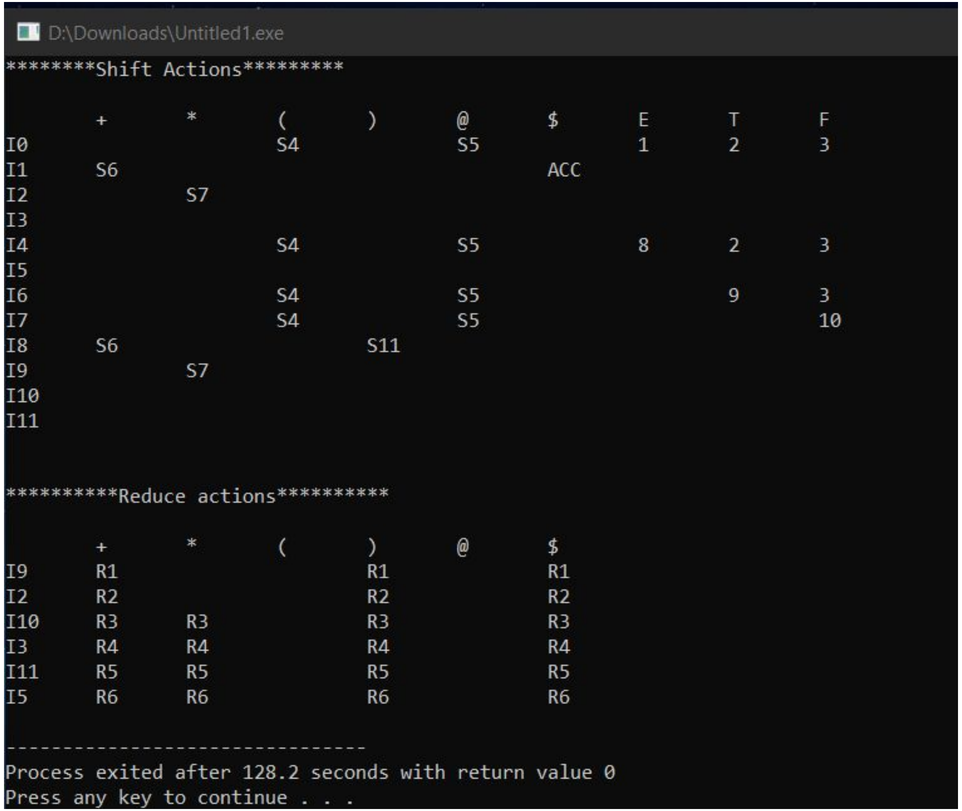


```
I0 on reading the symbol @ goes to I5:
F->@.

I1 on reading the symbol + goes to I6:
E->E+.T
T->.T*F
T->.F
F->.(E)
F->.@

I2 on reading the symbol * goes to I7:
T->T*.F
F->.(E)
F->.@

I4 on reading the symbol E goes to I8:
F->(E.)
E->E.+T

I4 on reading the symbol T goes to I2.
I4 on reading the symbol F goes to I3.
I4 on reading the symbol ( goes to I4.
I4 on reading the symbol @ goes to I5.
I6 on reading the symbol T goes to I9:
E->E+T.
T->T.*F
```



```
I6 on reading the symbol F goes to I3.
I6 on reading the symbol ( goes to I4.
I6 on reading the symbol @ goes to I5.
I7 on reading the symbol F goes to I10:
T->T*F.

I7 on reading the symbol ( goes to I4.
I7 on reading the symbol @ goes to I5.
I8 on reading the symbol ) goes to I11:
F->(E).

I8 on reading the symbol + goes to I6.
I9 on reading the symbol * goes to I7.

********Shift Actions*********

        +       *       (       )       @       $       E       T       F
I0                      S4              S5              1       2       3
I1      S6                                      ACC
I2              S7
I3
I4                      S4              S5              8       2       3
I5
I6                      S4              S5                      9       3
I7                      S4              S5                              10
I8      S6                      S11
I9              S7
I10
I11
```

IMPLEMENTATION :



```
********Shift Actions*********

        +       *       (       )       @       $       E       T       F
I0                      S4              S5              1       2       3
I1      S6                                      ACC
I2              S7
I3
I4                      S4              S5              8       2       3
I5
I6                      S4              S5                      9       3
I7                      S4              S5                              10
I8      S6                      S11
I9              S7
I10
I11


**********Reduce actions**********

        +       *       (       )       @       $
I9      R1                      R1              R1
I2      R2                      R2              R2
I10     R3      R3              R3              R3
I3      R4      R4              R4              R4
I11     R5      R5              R5              R5
I5      R6      R6              R6              R6

--------------------------------
Process exited after 128.2 seconds with return value 0
Press any key to continue . . .
```

RESULT :

Therefore, the SLR parsing is successfully implemented.

CONCLUSION :

In conclusion, SLR parsing is a bottom-up parsing technique that is widely used in compiler design. It is an efficient and effective technique for parsing a wide range of context-free grammars. However, it has some limitations, such as its inability to handle left-recursive grammars.

Despite its limitations, SLR parsing remains a valuable technique for compiler design. It is easy to implement and provides a good balance between efficiency and expressiveness. Furthermore, it serves as a foundation for more advanced parsing techniques, such as LR(1) and LALR(1) parsing.

REFERENCES :

- https://www.geeksforgeeks.org/slr-parser-with-examples/
- https://www.javatpoint.com/slr-1-parsing
- https://www.tutorialspoint.com/construct-the-slr-parsing-table-for-the-following-grammar-also-parse-the-input-string-a-b-plus-a