Experiment No.6
Implementation of Shift Reduce Parser (SRP)
Date of Performance: 10-03-2025
Date of Submission: 17-03-2025

Aim: Implementation of Shift Reduce Parser (SRP).

Objective: To develop an ability to design and develops the Analysis phase of Compiler.

Theory:

Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom-up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of the shift-reduce parser is the LR parser. This parser requires some data structures i.e.

- An input buffer for storing the input string.
- A stack for storing and accessing the production rules.

Basic Operations –

- Shift: This involves moving symbols from the input buffer onto the stack.
- Reduce: If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of a production rule is popped out of a stack and LHS of a production rule is pushed onto the stack.
- Accept: If only the start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accepted action is obtained, it is means successful parsing is done.
- Error: This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Example 1 – Consider the grammar

S –> S + S

S –> S * S

S –> id

Perform Shift Reduce parsing for input string "id + id + id".

Stack	Input Buffer	Parsing Action
$	id+id+id$	Shift
$id	+id+id$	Reduce S->id
$S	+id+id$	Shift
$S+	id+id$	Shift
$S+id	+id$	Reduce S->id
$S+S	+id$	Reduce S->S+S
$S	+id$	Shift
$S+	id$	Shift
$S+id	$	Reduce S->id
$S+S	$	Reduce S->S+S
$S	$	Accept

There are two main categories of shift reduce parsing as follows:
1. Operator-Precedence Parsing
2. LR-Parser

Operator-Precedence Parsing:
Operator precedence grammar is a kind of shift reduce parsing method. It is applied to a small class of operator grammars. A grammar is said to be operator precedence grammar if it has two properties:
- No R.H.S. of any production has a∈.
- No two non-terminals are adjacent.

Operator precedence can only be established between the terminals of the grammar. It ignores the non-terminal. There are the three operator precedence relations:
a ⋗ b means that terminal "a" has the higher precedence than terminal "b".

a ⋖ b means that terminal "a" has the lower precedence than terminal "b".

a ≐ b means that the terminal "a" and "b" both have same precedence.

LR Parser:
LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.
In the LR parsing, "L" stands for left-to-right scanning of the input. "R" stands for constructing a right most derivation in reverse. "K" is the number of input symbols of the look ahead used to make number of parsing decisions.

LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing

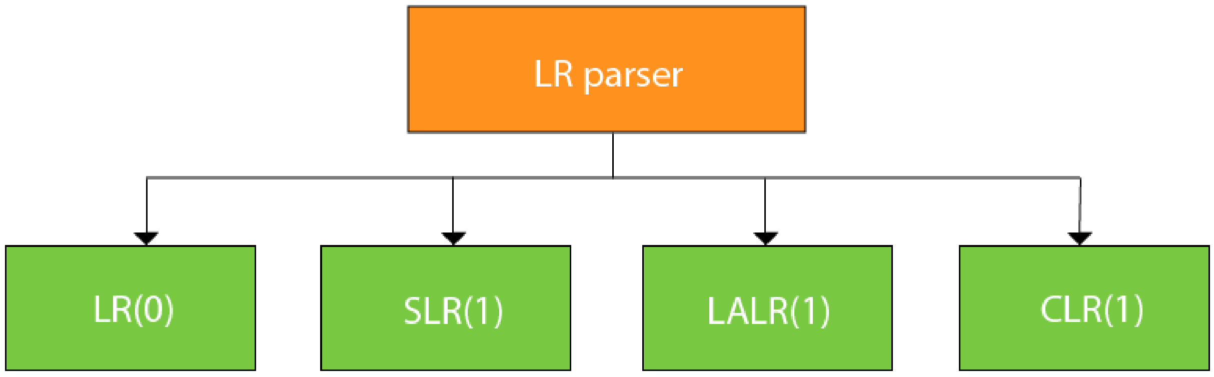


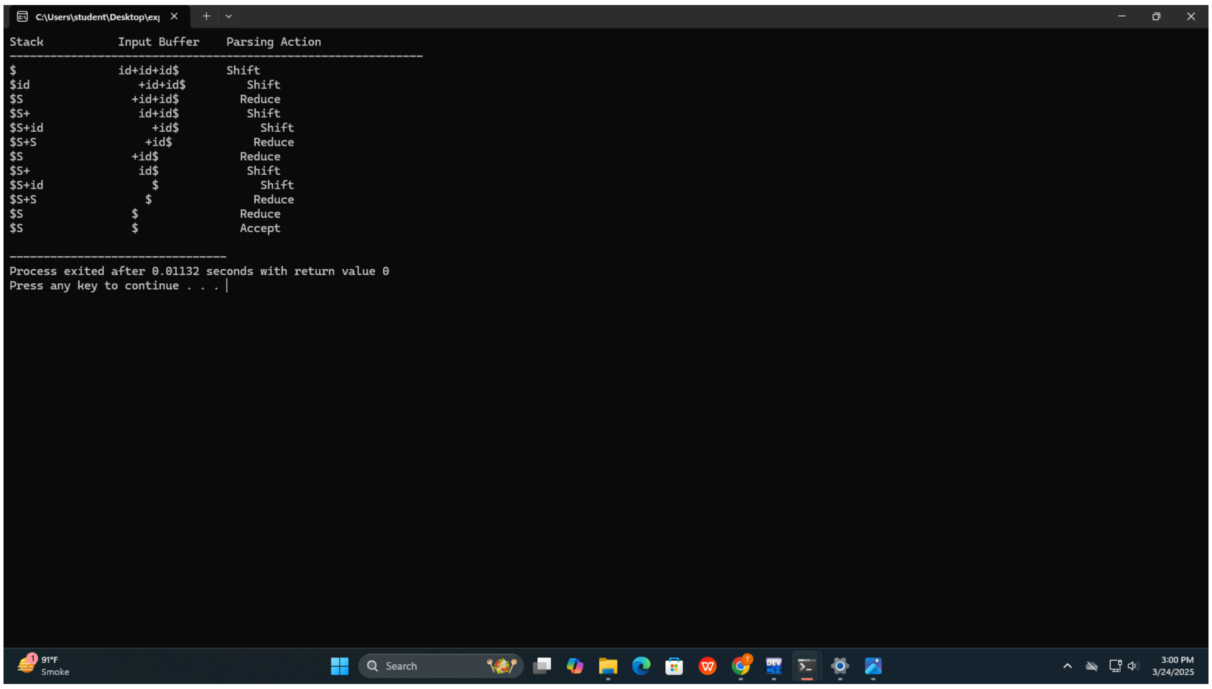
Fig: Types of LR parser

Code:

```
#include <stdio.h>

#include <string.h>



#define MAX 100



char stack[MAX][3];

int top = -1;



void push(char *c) {

    strcpy(stack[++top], c);

}
```

```
void pop() {

   top--;

}



void displayStack() {

   for (int i = 0; i <= top; i++) {

      printf("%s", stack[i]);

   }

}



int checkReduction() {

    if (top >= 2 && strcmp(stack[top], "S") == 0 && strcmp(stack[top - 1], "+") == 0 &&
strcmp(stack[top - 2], "S") == 0) {

      pop(); pop(); // Remove + and last S

      strcpy(stack[top], "S");

      return 1; // Reduce S -> S + S

   }

   if (top >= 0 && strcmp(stack[top], "id") == 0) {

      strcpy(stack[top], "S"); // Reduce S -> id

      return 1;
   }
   return 0;
}


void shiftReduceParser(char input[]) {
   printf("%-15s %-15s %-15s\n", "Stack", "Input Buffer", "Parsing Action");
   printf("-----------------------------------------------------------\n");


   push("$");
   printf("%-15s %-15s %-15s\n", "$", input, "Shift");


   int i = 0;
   while (input[i] != '$') {
      if (input[i] == 'i' && input[i + 1] == 'd') {
          push("id");
          i += 2;
      } else {
          char temp[2] = {input[i++], '\0'};
          push(temp);
      }
      displayStack();
      printf("%-15s %-15s %-15s\n", "", &input[i], "Shift");


      while (checkReduction()) {
          displayStack();
          printf("%-15s %-15s %-15s\n", "", &input[i], "Reduce");
      }
   }


   while (checkReduction()) {
      displayStack();
      printf("%-15s %-15s %-15s\n", "", "$", "Reduce");
   }


   if (top == 1 && strcmp(stack[1], "S") == 0) {
      displayStack();
      printf("%-15s %-15s %-15s\n", "", "$", "Accept");
   } else {
      displayStack();
      printf("%-15s %-15s %-15s\n", "", "$", "Reject");
    }
}


int main() {
    char input[] = "id+id+id$"; // Input string to be parsed
    shiftReduceParser(input);
    return 0;
}
```

Output:



```
Stack           Input Buffer    Parsing Action
-------------------------------------------------------------
$               id+id+id$       Shift
$id                +id+id$         Shift
$S                +id+id$         Reduce
$S+                id+id$         Shift
$S+id                +id$           Shift
$S+S                +id$           Reduce
$S                +id$          Reduce
$S+                id$          Shift
$S+id                $              Shift
$S+S                $              Reduce
$S                $             Reduce
$S                $             Accept

--------------------------------
Process exited after 0.01132 seconds with return value 0
Press any key to continue . . .
```

Conclusion:

→1. The implementation of the **Shift Reduce Parser (SRP)** successfully demonstrates the fundamental mechanism of **bottom-up parsing**, which forms an integral part of the **syntax analysis phase** in a compiler. The parser operates by repeatedly applying **shift** and **reduce operations**, making decisions based on the current input symbol and the top of the parsing stack until the string is either **reduced to the start symbol** or an **error is encountered**.

The **grammar assumed** for this implementation was carefully chosen to ensure clarity and simplicity in demonstrating SRP operations. It was designed to be **unambiguous and suitable for bottom-up parsing**, and it served effectively in illustrating the key concepts such as **handle identification, shift, and reduction steps**.

→2. The **output of the parser** clearly indicates whether a given string is **accepted or rejected** based on the grammar rules. The **accepted strings** are those that are successfully reduced to the start symbol using valid shift and reduce transitions, whereas **rejected strings** result from invalid operations or unresolvable conflicts, signifying syntactic errors in the input.

Overall, the Shift Reduce Parser program provides valuable insight into how a compiler performs syntax analysis and validates the structure of source code during compilation.