8. Write a C program to design predictive parser for the given language

Procedure:
A predictive parser is a top-down parsing technique that uses a predictive parsing table to predict the next production rule to be used in parsing a given input string. The parsing table is built based on the grammar of the language being parsed and the first and follow sets of its non-terminal symbols.

1. To implement a predictive parser in C, you would typically follow these steps:
2. Define the grammar of the language to be parsed using a set of production rules.
3. Compute the first and follow sets for each non-terminal symbol in the grammar.
4. Construct a predictive parsing table based on the grammar and the first and follow sets.
5. Write a parsing function in C that takes as input a string to be parsed, uses the predictive parsing table to predict the next production rule, and constructs a parse tree.
6. Test the parsing function by providing it with sample input strings and verifying that it produces the correct parse tree.

Example:

```
E -> E + T
  | E - T
  | T

T -> T * F
  | T / F
  | F

F -> ( E )
  | num
```

Assuming you have computed the first and follow sets for each non-terminal symbol, you can then construct a predictive parsing table that looks something like this:

	+	-	*	/	(	)	num	$
E	1	2			3		3	
T			4		5		5	
F					6	7	7	

In this table, each cell contains the production rule number to be used when the parser encounters the corresponding non-terminal symbol and input token. For example, when parsing the input string "2 + 3 * (4 - 1)", the parser would use the production rules in the following order: T -> F -> num, E -> T -> T * F -> F -> num, E -> E + T -> T -> F -> (E) -> E -> T -> F -> num, E -> T -> T * F -> T -> T / F -> F -> (E) -> E -> T -> F -> num, E -> T -> T / F -> T -> F -> (E) -> E -> T -> F -> num, E -> E - T -> T -> F -> (E) -> E -> T -> F -> num.

Program(Source Code):
```
#include <stdio.h>
#include <string.h>
 char prol[7][10] = { "S", "A", "A", "B", "B", "C", "C" };
char pror[7][10] = { "A", "Bb", "Cd", "aB", "@", "Cc", "@" };
char prod[7][10] = { "S->A", "A->Bb", "A->Cd", "B->aB", "B->@", "C->Cc", "C->@" };
char first[7][10] = { "abcd", "ab", "cd", "a@", "@", "c@", "@" };
char follow[7][10] = { "$", "$", "$", "a$", "b$", "c$", "d$" };
char table[5][6][10];
 int numr(char c)
{
   switch (c)
   {
      case 'S':
         return 0;

      case 'A':
         return 1;

      case 'B':
         return 2;

      case 'C':
         return 3;

      case 'a':
         return 0;

      case 'b':
         return 1;

      case 'c':
         return 2;

      case 'd':
         return 3;

      case '$':
         return 4;
   }

   return (2);
}

int main()
{
   int i, j, k;

   for (i = 0; i < 5; i++)
      for (j = 0; j < 6; j++)
         strcpy(table[i][j], " ");

   printf("The following grammar is used for Parsing Table:\n");
```

```
for (i = 0; i < 7; i++)
  printf("%s\n", prod[i]);

printf("\nPredictive parsing table:\n");

fflush(stdin);

for (i = 0; i < 7; i++)
{
  k = strlen(first[i]);
  for (j = 0; j < 10; j++)
    if (first[i][j] != '@')
      strcpy(table[numr(prol[i][0]) + 1][numr(first[i][j]) + 1], prod[i]);
}

for (i = 0; i < 7; i++)
{
  if (strlen(pror[i]) == 1)
  {
    if (pror[i][0] == '@')
    {
      k = strlen(follow[i]);
      for (j = 0; j < k; j++)
        strcpy(table[numr(prol[i][0]) + 1][numr(follow[i][j]) + 1], prod[i]);
    }
  }
}

strcpy(table[0][0], " ");

strcpy(table[0][1], "a");

strcpy(table[0][2], "b");

strcpy(table[0][3], "c");

strcpy(table[0][4], "d");

strcpy(table[0][5], "$");

strcpy(table[1][0], "S");

strcpy(table[2][0], "A");

strcpy(table[3][0], "B");

strcpy(table[4][0], "C");

printf("\n-------------------------------------------------------- \n");

for (i = 0; i < 5; i++)
  for (j = 0; j < 6; j++)
  {
    printf("%-10s", table[i][j]);
```

```
          if (j == 5)
            printf("\n-------------------------------------------------------\n");
        }
  }
```

Output :

```
The following grammar is used for Parsing Table:
S−>A
A−>Bb
A−>Cd
B−>aB
B−>@
C−>Cc
C−>@

Predictive parsing table:

---------------------------------------------------------------------------
        a        b        c        d        $
---------------------------------------------------------------------------
S       S−>A     S−>A     S−>A     S−>A
---------------------------------------------------------------------------
A       A−>Bb    A−>Bb    A−>Cd    A−>Cd
---------------------------------------------------------------------------
B       B−>aB    B−>@     B−>@              B−>@
---------------------------------------------------------------------------
C                         C−>@     C−>@     C−>@
---------------------------------------------------------------------------
```

9. write a lex program to implement a lexical analyser using lex tool

Source code :

```
%{
#include <stdio.h>
%}

// Regular expressions for tokens
DIGIT       [0-9]
LETTER      [a-zA-Z]
ID          {LETTER}({LETTER}|{DIGIT})*
INT         {DIGIT}+
WS          [ \t\n]+

%%

{ID}        printf("Identifier: %s\n", yytext);
{INT}       printf("Integer: %s\n", yytext);
if          printf("Keyword: if\n");
else        printf("Keyword: else\n");
while       printf("Keyword: while\n");
{WS}        // Ignore whitespace

.           printf("Unrecognized: %s\n", yytext);

%%

int main() {
   yylex();
```

```
    return 0;
}
```

End of the source program.

To compile and use this Lex program, you'll need to:

Save the above code into a file named lexer.l.
Install the Lex tool if you haven't already. On many systems, you might use flex as a modern replacement for Lex.
Open a terminal and navigate to the directory containing lexer.l.
Run the following commands to generate the lexer source code and compile it:

```
flex lexer.l
gcc lex.yy.c -o lexer -ll
```

10. Write a C program for implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar.

Algorithm:

1. Initialize the stack with the start symbol of the grammar and the end-of-input marker symbol ($).
2. Read the next input symbol from the input string.
3. If the top of the stack is a terminal symbol, pop it off the stack and compare it with the input symbol. If they match, read the next input symbol from the input string and repeat from step 2. If they do not match, report an error and halt.
4. If the top of the stack is a non-terminal symbol, look up the appropriate entry in the parsing table based on the current input symbol and the non-terminal symbol at the top of the stack. If there is no entry in the table, report an error and halt.
5. If the table entry is a shift action, push the input symbol onto the stack and push the state specified in the table entry onto the stack.
6. If the table entry is a reduce action, pop the right-hand side of the production rule off the stack and replace it with the left-hand side of the rule. Look up the new top of stack in the parsing table, and push the new state specified in the table entry onto the stack.
7. If the table entry is an accept action, report success and halt.
8. If none of the above conditions hold, report an error and halt.

Program(Source Code)

```
#include  <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 100

char input[MAX];
char stack[MAX];
int top = -1;

void push(char ch) {
    if (top == MAX - 1) {
        printf("Stack Overflow!\n");
        exit(1);
    }
    stack[++top] = ch;
}
```

```
char pop() {
    if (top == -1) {
        printf("Stack Underflow!\n");
        exit(1);
    }
    return stack[top--];
}

int shift() {
    int len = strlen(input);
    if (len == 0) {
        return 0;
    }
    push(input[0]);
    memmove(input, input + 1, len);
    input[len - 1] = '\0';
    return 1;
}
int reduce() {
    int i, len = strlen(stack);
    char temp[MAX];
    memset(temp, '\0', MAX);
    for (i = 0; i < len; i++) {
        temp[i] = stack[i];
        if (temp[i] == 'E' && temp[i - 1] == '(' && temp[i + 1] == ')') {
            temp[i - 1] = 'E';
            memmove(temp + i, temp + i + 2, len - i - 1);
            len -= 2;
            i = -1;
        }
        if (temp[i] == 'T' && temp[i - 1] == '(' && temp[i + 1] == ')') {
            temp[i - 1] = 'T';
            memmove(temp + i, temp + i + 2, len - i - 1);
            len -= 2;
            i = -1;
        }
    }
    if (strcmp(temp, stack) == 0) {
        return 0;
    }
    memset(stack, '\0', MAX);
    top = -1;
    len = strlen(temp);
    for (i = len - 1; i >= 0; i--) {
        push(temp[i]);
    }
    return 1;
}


int main()
 {
    strcpy(input, "(id+id)*id$");
    push('$');
    push('E');
    while (1) {
        printf("\nStack: %s", stack);
        printf("\nInput: %s\n", input);
        if (stack[top] == '$' && input[0] == '$') {
            printf("\nString Accepted!\n");
            break;
        }
        if (stack[top] == input[0]) {
            pop();
            shift();
        } else {
            if (!reduce()) {
                printf("\nString Not Accepted!\n");
                break;
            }
        }
    }
    return 0;
}
```

This program implements a Shift Reduce Parser using a stack data structure to accept an input string of a given grammar.

E -> E + T | T
T -> T * F | F
F -> (E) | id

The program starts by pushing the start symbol E and the end-of-stack symbol $ onto the stack. It then reads in the input string and begins parsing it by repeatedly performing the shift and reduce operations. The shift operation takes the first character of the input string and pushes it onto the stack, while the reduce operation checks if the top of the stack matches the right-hand side of any production rule and replaces it with the left-hand side of that rule.

11. Write a C program to implement the Brute force technique of Top Down Parsing.

Procedure:
Consider if we define a grammar with two production rules:
S -> aSb
S -> ε (empty string)

The **parseS** function is a recursive function that tries to match these production rules. Here's how the parsing works:
The **parseS** function starts at the current index in the input string.
If the current character is 'a', it attempts to apply the 'aSb' production rule. It recursively calls **parseS** to match the 'S' non-terminal and then checks if the next character is 'b'.
If the 'aSb' production is successful, it returns the new index where the parsing stopped.
If the 'aSb' production fails, it falls back to the 'ε' production rule, which matches an empty string and returns the current index.
The main function reads an input string from the user and calls the **parseS** function to check if the input string is valid according to the grammar. If the parsing process successfully consumes the entire input

string, it's considered valid. Otherwise, it's invalid.
Keep in mind that this is a simplified example of a top-down parser, and real-world parsers for more complex grammars involve additional features and optimizations.

```
#include <stdio.h>
#include <string.h>

// Define a simple grammar with production rules
// S -> aSb | ε (empty string)
// This grammar generates strings of the form 'a^n b^n' for n >= 0

// Function to parse the non-terminal 'S'
int parseS(char input[], int index);

int main() {
    char input[100];

    // Read the input string from the user
    printf("Enter a string: ");
    scanf("%s", input);

    // Start parsing from the first character of the input
    if (parseS(input, 0) == strlen(input)) {
        // If the parser successfully consumes the entire input string,
        // it means the input string is valid according to the grammar.
        printf("Valid string\n");
    } else {
        printf("Invalid string\n");
    }

    return 0;
}

int parseS(char input[], int index) {
    // Base case: If we have reached the end of the input string, return the current index.
    if (input[index] == '\0') {
        return index;
    }

    // Try to apply the production S -> aSb
    if (input[index] == 'a') {
        int newIndex = parseS(input, index + 1);
        if (input[newIndex] == 'b') {
            return parseS(input, newIndex + 1);
        }
    }

    // If the 'aSb' production didn't match, try the production S -> ε (empty string)
    return index;
}
```

Output:

```
Enter a string: 0
Invalid string

Enter a string: aaabbb
Valid string

Enter a string: abcabc
Invalid string

Enter a string: (just press enter)
valid string
```

12. Write a C Program to implement a Recursive descent Parser

ALGORITHM:

```
Step 1: start.
Step 2: Declare the prototype functions for any given grammar
Step 3: Read the string to be parsed.
Step 4: Check the productions
Step 5: Compare the terminals and Non-terminals
Step 6: Read the parse string.
Step 7: stop the production
```

Source Code:

```
For grammar
E  -> T E'
E' -> + T E' | - T E' | ε
T  -> F T'
T' -> * F T' | / F T' | ε
F  -> ( E ) | number

#include <stdio.h>
#include <stdbool.h>

#define MAX_EXPRESSION_LENGTH 100

// Function declarations for non-terminals
```

```
int E(char* input);
int E_prime(char* input);
int T(char* input);
int T_prime(char* input);
int F(char* input);

// Helper function to skip whitespace
void skipWhitespace(char* input, int* index) {
    while (input[*index] && isspace(input[*index])) {
        (*index)++;
    }
}

// Function to parse and evaluate an expression
int E(char* input) {
    int index = 0;
    int result = T(input);
    return E_prime(input, &index, result);
}

// Function to handle E' -> + T E' and E' -> - T E'
int E_prime(char* input, int* index, int left) {
    skipWhitespace(input, index);

    char op = input[*index];
    if (op == '+' || op == '-') {
        (*index)++;
        int right = T(input);
        if (op == '+') {
            left += right;
        } else {
            left -= right;
        }
        return E_prime(input, index, left);
    }
    return left;
}

// Function to parse and evaluate a term
int T(char* input) {
    int index = 0;
    int result = F(input);
    return T_prime(input, &index, result);
}

// Function to handle T' -> * F T' and T' -> / F T'
int T_prime(char* input, int* index, int left) {
    skipWhitespace(input, index);

    char op = input[*index];
    if (op == '*' || op == '/') {
        (*index)++;
        int right = F(input);
        if (op == '*') {
            left *= right;
        } else {
            if (right == 0) {
                printf("Error: Division by zero\n");
                return 0;
            }
            left /= right;
        }
        return T_prime(input, index, left);
    }
    return left;
}

// Function to parse and evaluate a factor
int F(char* input) {
    int index = 0;
    skipWhitespace(input, &index);

    if (input[index] == '(') {
        // Handle ( E )
        index++; // Consume '('
        int result = E(input);
        skipWhitespace(input, &index);

        if (input[index] == ')') {
            index++; // Consume ')'
        } else {
            printf("Error: Unmatched '(' or ')'\n");
            return 0;
        }

        return result;
    } else if (isdigit(input[index])) {
        // Handle numeric literals
        int value = 0;

        while (isdigit(input[index])) {
            value = value * 10 + (input[index] - '0');
            index++;
        }

        return value;
    } else {
        printf("Error: Invalid character '%c'\n", input[index]);
        return 0;
    }
}

int main() {
    char input[MAX_EXPRESSION_LENGTH];

    printf("Enter an arithmetic expression: ");
    fgets(input, MAX_EXPRESSION_LENGTH, stdin);

    int result = E(input);

    if (input[0] != '\0' && input[0] != '\n') {
        printf("Error: Invalid expression\n");
        return 1;
    }
    printf("Result: %d\n", result);

    return 0;
}
```

Output :

```
Enter an arithmetic expression: (5 + 3) * 2
Result: 16

Enter an arithmetic expression: (1 + 2) * 3
Result: 09s
```