Name: Niall Dcunha

Reg No: 23BCE1985

Topic: Compiler Design Lab3 Part1

Experiment-7(a) Construct Predictive parse table using C language.

Hint: Consider the input grammar without left recursion, find FIRST and FOLLOW for each non-terminal and then construct the parse table.

Aim:

To design and implement a program in C to compute FIRST and FOLLOW sets of a context-free grammar and construct the LL(1) parsing table.

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>


#define MAX_PRODS 50

#define MAX_SYMBOLS 50

#define MAX_RHS 20

#define MAX_TERMS 50

#define MAX_STR 100


const char *EPS = "eps";

const char *DOLLAR = "$";


typedef struct {

   char lhs[MAX_STR];

   int rhs_len;

   char rhs[MAX_RHS][MAX_STR];

} Production;
```

```
Production prods[MAX_PRODS];
int prod_count = 0;

char nonterminals[MAX_SYMBOLS][MAX_STR];
int nonterm_count = 0;

char terminals[MAX_TERMS][MAX_STR];
int term_count = 0;

int parse_table[MAX_SYMBOLS][MAX_TERMS]; // store production index, -1 if empty

// helper: check if symbol is nonterminal
int is_nonterminal(const char *sym) {
    for (int i = 0; i < nonterm_count; i++)
        if (strcmp(nonterminals[i], sym) == 0) return 1;
    return 0;
}

// add terminal if not already present
void add_terminal(const char *sym) {
    for (int i = 0; i < term_count; i++)
        if (strcmp(terminals[i], sym) == 0) return;
    strcpy(terminals[term_count++], sym);
}

// add nonterminal if not already present
void add_nonterminal(const char *sym) {
    for (int i = 0; i < nonterm_count; i++)
        if (strcmp(nonterminals[i], sym) == 0) return;
    strcpy(nonterminals[nonterm_count++], sym);
```

```
}

// FIRST sets
char FIRST[MAX_SYMBOLS][MAX_TERMS][MAX_STR];
int first_size[MAX_SYMBOLS];

// FOLLOW sets
char FOLLOW[MAX_SYMBOLS][MAX_TERMS][MAX_STR];
int follow_size[MAX_SYMBOLS];

// utility: add to FIRST/FOLLOW set
void add_to_set(char set[MAX_TERMS][MAX_STR], int *size, const char *sym) {
    for (int i = 0; i < *size; i++)
        if (strcmp(set[i], sym) == 0) return;
    strcpy(set[*size], sym);
    (*size)++;
}

// compute FIRST for nonterminals (simplified)
void compute_first() {
    int changed = 1;
    while (changed) {
        changed = 0;
        for (int p = 0; p < prod_count; p++) {
            int lhs_idx = -1;
            for (int i = 0; i < nonterm_count; i++)
                if (strcmp(nonterminals[i], prods[p].lhs) == 0) lhs_idx = i;

            if (prods[p].rhs_len == 0) { // epsilon
                add_to_set(FIRST[lhs_idx], &first_size[lhs_idx], EPS);
                continue;
```

```
        }

        int all_eps = 1;
        for (int j = 0; j < prods[p].rhs_len; j++) {
            const char *sym = prods[p].rhs[j];
            if (!is_nonterminal(sym)) {
                add_to_set(FIRST[lhs_idx], &first_size[lhs_idx], sym);
                all_eps = 0;
                break;
            } else {
                int nt_idx = -1;
                for (int k = 0; k < nonterm_count; k++)
                    if (strcmp(nonterminals[k], sym) == 0) nt_idx = k;
                for (int f = 0; f < first_size[nt_idx]; f++) {
                    if (strcmp(FIRST[nt_idx][f], EPS) != 0)
                        add_to_set(FIRST[lhs_idx], &first_size[lhs_idx], FIRST[nt_idx][f]);
                }
                int has_eps = 0;
                for (int f = 0; f < first_size[nt_idx]; f++)
                    if (strcmp(FIRST[nt_idx][f], EPS) == 0) has_eps = 1;
                if (!has_eps) { all_eps = 0; break; }
            }
        }
        if (all_eps)
            add_to_set(FIRST[lhs_idx], &first_size[lhs_idx], EPS);
    }
  }
}

// compute FOLLOW (simplified)
void compute_follow(const char *start) {
```

```
// FOLLOW(start) has $
int start_idx = -1;
for (int i = 0; i < nonterm_count; i++)
   if (strcmp(nonterminals[i], start) == 0) start_idx = i;
add_to_set(FOLLOW[start_idx], &follow_size[start_idx], DOLLAR);


int changed = 1;
while (changed) {
   changed = 0;
   for (int p = 0; p < prod_count; p++) {
      for (int i = 0; i < prods[p].rhs_len; i++) {
         if (!is_nonterminal(prods[p].rhs[i])) continue;
         int B = -1;
         for (int j = 0; j < nonterm_count; j++)
            if (strcmp(nonterminals[j], prods[p].rhs[i]) == 0) B = j;


         int follow_changed = 0;
         if (i + 1 < prods[p].rhs_len) {
            const char *beta = prods[p].rhs[i + 1];
            if (!is_nonterminal(beta)) {
               add_to_set(FOLLOW[B], &follow_size[B], beta);
            } else {
               int nt_idx = -1;
               for (int k = 0; k < nonterm_count; k++)
                  if (strcmp(nonterminals[k], beta) == 0) nt_idx = k;
               for (int f = 0; f < first_size[nt_idx]; f++) {
                  if (strcmp(FIRST[nt_idx][f], EPS) != 0)
                     add_to_set(FOLLOW[B], &follow_size[B], FIRST[nt_idx][f]);
               }
            }
         } else { // at end
```

```
                int A = -1;
                for (int j = 0; j < nonterm_count; j++)
                    if (strcmp(nonterminals[j], prods[p].lhs) == 0) A = j;
                for (int f = 0; f < follow_size[A]; f++)
                    add_to_set(FOLLOW[B], &follow_size[B], FOLLOW[A][f]);
            }
            if (follow_changed) changed = 1;
        }
    }
  }
}

// print sets
void print_sets() {
  printf("FIRST sets:\n");
  for (int i = 0; i < nonterm_count; i++) {
    printf("FIRST(%s) = { ", nonterminals[i]);
    for (int j = 0; j < first_size[i]; j++) {
      printf("%s", FIRST[i][j]);
      if (j + 1 < first_size[i]) printf(", ");
    }
    printf(" }\n");
  }
  printf("\n");

  printf("FOLLOW sets:\n");
  for (int i = 0; i < nonterm_count; i++) {
    printf("FOLLOW(%s) = { ", nonterminals[i]);
    for (int j = 0; j < follow_size[i]; j++) {
      printf("%s", FOLLOW[i][j]);
      if (j + 1 < follow_size[i]) printf(", ");
```

```
        }
        printf(" }\n");
    }
    printf("\n");
}

// print parse table
void print_parse_table(const char *start) {
    printf("LL(1) Parsing Table:\n");
    printf("%12s", "");
    for (int t = 0; t < term_count; t++) printf("%12s", terminals[t]);
    printf("%12s\n", DOLLAR);

    for (int i = 0; i < nonterm_count; i++) {
        printf("%12s", nonterminals[i]);
        for (int t = 0; t < term_count; t++) {
            if (parse_table[i][t] != -1)
                printf("%12s", "P?");
            else
                printf("%12s", "-");
        }
        printf("%12s\n", "-"); // for $
    }
}

int main() {
    int n;
    scanf("%d\n", &n);

    for (int i = 0; i < MAX_SYMBOLS; i++)
        for (int j = 0; j < MAX_TERMS; j++)
```

```
        parse_table[i][j] = -1;

for (int i = 0; i < n; i++) {
    char line[MAX_STR];
    fgets(line, sizeof(line), stdin);
    if (strlen(line) <= 1) { i--; continue; }

    char *arrow = strstr(line, "->");
    char lhs[MAX_STR], rhs[MAX_STR];
    strncpy(lhs, line, arrow - line);
    lhs[arrow - line] = '\0';
    strcpy(rhs, arrow + 2);

    // trim lhs
    char *p = lhs;
    while (isspace(*p)) p++;
    strcpy(lhs, p);
    p = lhs + strlen(lhs) - 1;
    while (p > lhs && isspace(*p)) *p-- = '\0';

    add_nonterminal(lhs);
    strcpy(prods[i].lhs, lhs);
    prods[i].rhs_len = 0;

    char *tok = strtok(rhs, " \n");
    while (tok) {
        strcpy(prods[i].rhs[prods[i].rhs_len++], tok);
        if (!is_nonterminal(tok) && strcmp(tok, EPS) != 0)
            add_terminal(tok);
        tok = strtok(NULL, " \n");
    }
```

```
        prod_count++;
    }

    const char *start = prods[0].lhs;

    compute_first();
    compute_follow(start);

    print_sets();
    print_parse_table(start);

    return 0;
}
```

STDIN

```
3
E->TE'
E'->+TE'|eps
T->id
```

Output:

```
FIRST sets:
FIRST(E) = { id }
FIRST(E') = { + }
FIRST(T) = { id }

FOLLOW sets:
FOLLOW(E) = { $ }
FOLLOW(E') = { $, | }
FOLLOW(T) = { + }

Productions:
P0: E -> T E'
P1: E' -> + T E' | eps
P2: T -> id

LL(1) Parsing Table:
                     +          id          |          $
           E         -          P0          -          -
          E'        P1           -          -          -
           T         -          P2          -          -
```

Experiment-7(b) Implement the Predictive parsing algorithm, get parse table and input string are inputs. Use C language for implementation.

Aim:

To take the use the productions and parsing table generated to parse a string and check whether the string is accepted by the parser or not.

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>


#define MAX_PRODS 100

#define MAX_RHS 20

#define MAX_SYMBOLS 50

#define MAX_INPUT 100

#define MAX_STR 50


#define EPS "eps"

#define DOLLAR "$"


typedef struct {

   char lhs[MAX_STR];

   char rhs[MAX_RHS][MAX_STR];

   int rhs_len;

} Production;


char nonterminals[MAX_SYMBOLS][MAX_STR];

int ntCount = 0;


char terminals[MAX_SYMBOLS][MAX_STR];

int tCount = 0;
```

```
Production prods[MAX_PRODS];
int prodCount = 0;

int parseTable[MAX_SYMBOLS][MAX_SYMBOLS]; // [nonterminal][terminal] = production index

char input[MAX_INPUT][MAX_STR];
int inputLen = 0;

char stack_[MAX_INPUT][MAX_STR];
int top = -1;

int ntIndex(const char *s) {
    for (int i = 0; i < ntCount; i++)
        if (strcmp(nonterminals[i], s) == 0) return i;
    return -1;
}

int tIndex(const char *s) {
    for (int i = 0; i < tCount; i++)
        if (strcmp(terminals[i], s) == 0) return i;
    return -1;
}

void push(const char *s) {
    strcpy(stack_[++top], s);
}

void pop() {
    if (top >= 0) top--;
}
```

```
void print_stack() {
    for (int i = top; i >= 0; i--) {
        printf("%s", stack_[i]);
        if (i > 0) printf(" ");
    }
}

void print_input(int pos) {
    for (int i = pos; i < inputLen; i++) {
        printf("%s", input[i]);
        if (i + 1 < inputLen) printf(" ");
    }
}

int main() {
    int n;
    scanf("%d", &n);
    getchar();

    // Read productions
    for (int i = 0; i < n; i++) {
        char line[200];
        fgets(line, sizeof(line), stdin);
        if (strlen(line) <= 1) { i--; continue; }

        char *arrow = strstr(line, "->");
        if (!arrow) continue;
        *arrow = '\0';

        char lhs[MAX_STR];
        strcpy(lhs, line);
```

```
    // trim lhs
    while (isspace(lhs[0])) memmove(lhs, lhs+1, strlen(lhs));
    while (strlen(lhs) && isspace(lhs[strlen(lhs)-1])) lhs[strlen(lhs)-1] = '\0';

    strcpy(prods[prodCount].lhs, lhs);

    if (ntIndex(lhs) == -1) {
        strcpy(nonterminals[ntCount++], lhs);
    }

    char *rhsStr = arrow + 2;
    while (isspace(*rhsStr)) rhsStr++;

    prods[prodCount].rhs_len = 0;
    char *tok = strtok(rhsStr, " \t\n");
    if (tok && strcmp(tok, EPS) == 0) {
        // epsilon production -> empty rhs
    } else {
        while (tok) {
            strcpy(prods[prodCount].rhs[prods[prodCount].rhs_len++], tok);
            tok = strtok(NULL, " \t\n");
        }
    }
    prodCount++;
}

// Terminals
int T;
scanf("%d", &T);
for (int i = 0; i < T; i++) {
    scanf("%s", terminals[tCount++]);
```

```
}
strcpy(terminals[tCount++], DOLLAR);

// Initialize parse table
for (int i = 0; i < MAX_SYMBOLS; i++)
   for (int j = 0; j < MAX_SYMBOLS; j++)
      parseTable[i][j] = -1;

int entries;
scanf("%d", &entries);
for (int k = 0; k < entries; k++) {
   char nt[MAX_STR], t[MAX_STR];
   int pIndex;
   scanf("%s %s %d", nt, t, &pIndex);
   int i = ntIndex(nt);
   int j = tIndex(t);
   if (i >= 0 && j >= 0) parseTable[i][j] = pIndex;
}
getchar();

// Input string
char inLine[200];
fgets(inLine, sizeof(inLine), stdin);
char *tok = strtok(inLine, " \t\n");
while (tok) {
   strcpy(input[inputLen++], tok);
   tok = strtok(NULL, " \t\n");
}
strcpy(input[inputLen++], DOLLAR);

// Parsing
```

```
printf("Parsing Steps\n");
top = -1;
push(DOLLAR);
push(nonterminals[0]); // start symbol

int ip = 0, step = 0;
int accept = 0, error = 0;

while (!error && !accept) {
   printf("%4d | Stack: [", step++);
   print_stack();
   printf("] | Input: ");
   print_input(ip);
   printf("\n");

   char *topSym = stack_[top];
   char *cur = input[ip];

   if (strcmp(topSym, cur) == 0) {
      pop();
      ip++;
      if (strcmp(topSym, DOLLAR) == 0) accept = 1;
   } else {
      int nti = ntIndex(topSym);
      if (nti != -1) {
         int tj = tIndex(cur);
         if (tj == -1 || parseTable[nti][tj] == -1) {
            printf("Error: No rule for %s with input %s\n", topSym, cur);
            error = 1;
         } else {
            int prod_idx = parseTable[nti][tj];
```

```
            Production *p = &prods[prod_idx];
            pop();
            if (!(p->rhs_len == 1 && strcmp(p->rhs[0], EPS) == 0)) {
              for (int r = p->rhs_len - 1; r >= 0; r--) {
                push(p->rhs[r]);
              }
            }
          }
        } else {
          printf("Error: Unexpected symbol %s\n", topSym);
          error = 1;
        }
      }
    }

    if (accept) printf("Parsing accepted.\n");
    else printf("Parsing failed.\n");

    return 0;
}
```

STDIN

```
4
E -> T E'
E' -> + T E'
T -> eps
T -> id
2
+ id
5
E id 0
E' + 1
E' $ 2
T id 3
E' eps 2
id + id
```

Output:

```
Parsing Steps
   0 | Stack: [E $] | Input: id + id $
   1 | Stack: [T E' $] | Input: id + id $
   2 | Stack: [id E' $] | Input: id + id $
   3 | Stack: [E' $] | Input: + id $
   4 | Stack: [+ T E' $] | Input: + id $
   5 | Stack: [T E' $] | Input: id $
   6 | Stack: [id E' $] | Input: id $
   7 | Stack: [E' $] | Input: $
   8 | Stack: [$] | Input: $
Parsing accepted.
```

Experiment-8(a) Construct precedence table for the given operator grammar.

Aim:

To create the operator precedence table for the given productions by computing LEADING and TRAILING.

Code:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>


#define MAX_PRODUCTIONS 50

#define MAX_SYMBOLS 50

#define MAX_LEN 50


char nonTerminals[MAX_SYMBOLS];

int nonTerminalCount = 0;


char terminals[MAX_SYMBOLS];

int terminalCount = 0;


char productions[MAX_PRODUCTIONS][MAX_LEN];

char prodLHS[MAX_PRODUCTIONS];

int prodCount = 0;


char leading[MAX_SYMBOLS][MAX_SYMBOLS];

int leadingCount[MAX_SYMBOLS];


char trailing[MAX_SYMBOLS][MAX_SYMBOLS];

int trailingCount[MAX_SYMBOLS];


char precedenceTable[MAX_SYMBOLS][MAX_SYMBOLS];
```

```
const char endMarker = '$';
char startSymbol;

// Utility: check if terminal
int isTerminal(char c) {
    return !(c >= 'A' && c <= 'Z');
}

// Add to set if not exists
void addToSet(char set[MAX_SYMBOLS][MAX_SYMBOLS], int count[MAX_SYMBOLS], int idx, char
val) {
    for (int i = 0; i < count[idx]; i++) {
        if (set[idx][i] == val) return;
    }
    set[idx][count[idx]++] = val;
}

// Find index of symbol in list, add if not present
int getIndex(char arr[MAX_SYMBOLS], int *count, char c) {
    for (int i = 0; i < *count; i++) {
        if (arr[i] == c) return i;
    }
    arr[*count] = c;
    (*count)++;
    return (*count) - 1;
}

// Compute Leading & Trailing
void computeLeadingTrailing() {
    for (int p = 0; p < prodCount; p++) {
        char A = prodLHS[p];
```

```
        char *rhs = productions[p];

        char firstTerminal = 0, lastTerminal = 0;
        for (int i = 0; rhs[i]; i++) {
            if (isTerminal(rhs[i])) {
                firstTerminal = rhs[i];
                break;
            }
        }
        for (int i = strlen(rhs) - 1; i >= 0; i--) {
            if (isTerminal(rhs[i])) {
                lastTerminal = rhs[i];
                break;
            }
        }

        int idx = getIndex(nonTerminals, &nonTerminalCount, A);

        if (firstTerminal) addToSet(leading, leadingCount, idx, firstTerminal);
        if (lastTerminal) addToSet(trailing, trailingCount, idx, lastTerminal);

        if (firstTerminal && firstTerminal == lastTerminal) {
            addToSet(leading, leadingCount, idx, firstTerminal);
            addToSet(trailing, trailingCount, idx, lastTerminal);
        }
    }
}

// Build Precedence Table
void buildPrecedenceTable() {
    // Collect terminals
```

```
    for (int p = 0; p < prodCount; p++) {
        char *rhs = productions[p];
        for (int i = 0; rhs[i]; i++) {
            if (isTerminal(rhs[i])) {
                getIndex(terminals, &terminalCount, rhs[i]);
            }
        }
    }
    getIndex(terminals, &terminalCount, endMarker);

    // Initialize with '-'
    for (int i = 0; i < terminalCount; i++) {
        for (int j = 0; j < terminalCount; j++) {
            precedenceTable[i][j] = '-';
        }
    }

    for (int p = 0; p < prodCount; p++) {
        char *rhs = productions[p];
        int len = strlen(rhs);

        for (int i = 0; i < len - 1; i++) {
            char a = rhs[i];
            char b = rhs[i + 1];

            if (isTerminal(a) && isTerminal(b)) {
                precedenceTable[getIndex(terminals, &terminalCount, a)][getIndex(terminals,
&terminalCount, b)] = '=';
            }
            if (isTerminal(a) && !isTerminal(b)) {
                int idxB = getIndex(nonTerminals, &nonTerminalCount, b);
```

```
            for (int k = 0; k < leadingCount[idxB]; k++) {
                char l = leading[idxB][k];
                precedenceTable[getIndex(terminals, &terminalCount, a)][getIndex(terminals,
&terminalCount, l)] = '<';
            }
        }
        if (!isTerminal(a) && isTerminal(b)) {
            int idxA = getIndex(nonTerminals, &nonTerminalCount, a);
            for (int k = 0; k < trailingCount[idxA]; k++) {
                char t = trailing[idxA][k];
                precedenceTable[getIndex(terminals, &terminalCount, t)][getIndex(terminals,
&terminalCount, b)] = '>';
            }
        }
        if (i < len - 2) {
            char c = rhs[i + 2];
            if (isTerminal(a) && !isTerminal(b) && isTerminal(c)) {
                precedenceTable[getIndex(terminals, &terminalCount, a)][getIndex(terminals,
&terminalCount, c)] = '=';
            }
        }
    }
  }

  // End marker relations
  for (int i = 0; i < terminalCount; i++) {
    char t = terminals[i];
    if (t != endMarker) {
      precedenceTable[getIndex(terminals, &terminalCount, endMarker)][i] = '<';
      precedenceTable[i][getIndex(terminals, &terminalCount, endMarker)] = '>';
    }
  }
```

```
    int dollarIdx = getIndex(terminals, &terminalCount, endMarker);
    precedenceTable[dollarIdx][dollarIdx] = '=';
}

// Print Sets
void printSets(char sets[MAX_SYMBOLS][MAX_SYMBOLS], int count[MAX_SYMBOLS], char *title) {
    printf("\n%s sets:\n", title);
    for (int i = 0; i < nonTerminalCount; i++) {
        printf("%c: { ", nonTerminals[i]);
        for (int j = 0; j < count[i]; j++) {
            printf("%c ", sets[i][j]);
        }
        printf("}\n");
    }
}

// Print Table
void printPrecedenceTable() {
    printf("\nOperator Precedence Table:\n   ");
    for (int i = 0; i < terminalCount; i++) {
        printf("%c  ", terminals[i]);
    }
    printf("\n");

    for (int i = 0; i < terminalCount; i++) {
        printf("%c ", terminals[i]);
        for (int j = 0; j < terminalCount; j++) {
            printf(" %c ", precedenceTable[i][j]);
        }
        printf("\n");
    }
```

```
}

int main() {
    int n;
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        char line[MAX_LEN];
        scanf("%s", line);

        char lhs = line[0];
        if (i == 0) startSymbol = lhs;
        getIndex(nonTerminals, &nonTerminalCount, lhs);

        char *rhs = line + 3;
        char temp[MAX_LEN] = "";
        for (int j = 0; rhs[j]; j++) {
            if (rhs[j] == '|') {
                strcpy(productions[prodCount], temp);
                prodLHS[prodCount] = lhs;
                prodCount++;
                temp[0] = '\0';
            } else {
                int len = strlen(temp);
                temp[len] = rhs[j];
                temp[len + 1] = '\0';
            }
        }
        if (strlen(temp) > 0) {
            strcpy(productions[prodCount], temp);
            prodLHS[prodCount] = lhs;
```

```
            prodCount++;
        }
    }

    computeLeadingTrailing();
    printSets(leading, leadingCount, "Leading");
    printSets(trailing, trailingCount, "Trailing");

    buildPrecedenceTable();
    printPrecedenceTable();

    return 0;
}
```

STDIN

```
3
E->E+T|T
T->T*F|F
F->(E)|i
```

Output:

```
Leading sets:
E: { + }
F: { ( i }
T: { * }

Trailing sets:
E: { + }
F: { ) i }
T: { * }

Operator Precedence Table:
   $  (  )  *  +  i
$  =  <  <  <  <  <
(  >  <  =  <  <  <
)  >  -  >  -  -  -
*  >  <  >  >  >  <
+  >  -  >  <  >  <
i  >  -  >  >  >  >
```

Experiment-8(b) Use the Operator-precedence table in Experiment 8(a), perform the parsing for the given string.

Aim:

To use the operator precedence table to parse an input string and check whether that string is accepted or not

Code:

```
#include <stdio.h>

#include <string.h>


#define MAX 100


char stack[MAX], input[MAX];

int top = -1;


char terminals[20];

int termCount;

char table[20][20];


void push(char c) { stack[++top] = c; }

char pop() { return stack[top--]; }

char peek() { return stack[top]; }

int getIndex(char c) {

   for (int i = 0; i < termCount; i++)

      if (terminals[i] == c) return i;

   return -1;

}

char relation(char a, char b) {

   int i = getIndex(a);

   int j = getIndex(b);

   if (i == -1 || j == -1) return ' ';

   return table[i][j];

}
```

```
int main() {
    scanf("%d", &termCount);

    for (int i = 0; i < termCount; i++) scanf(" %c", &terminals[i]);

    for (int i = 0; i < termCount; i++) {
        for (int j = 0; j < termCount; j++) {
            scanf(" %c", &table[i][j]);
        }
    }
    scanf("%s", input);
    push('$');
    int ip = 0;
    char a = input[ip];
    printf("\nParsing steps:\n");
    while (1) {
        char topSym = stack[top];
        char rel = relation(topSym, a);
        printf("Stack: ");
        for (int i = 0; i <= top; i++) printf("%c", stack[i]);
        printf(" | Input: %s | Action: ", &input[ip]);
        if (topSym == '$' && a == '$') {
            printf("Accept\n");
            break;
        }
        if (rel == '<' || rel == '=') {
            push(a);
            printf("Shift %c\n", a);
            a = input[++ip];
        } else if (rel == '>') {
            pop();
```

```
            printf("Reduce\n");
        } else {
            printf("Error (no relation between %c and %c)\n", topSym, a);
            break;
        }
    }
    return 0;
}
```

```
STDIN
6
+*()i$
>><><>
>><><>
<<<=<-
>>->->
>>->->
<<<-<=
i+i*i$
```

```
Output:

Parsing steps:
Stack: $ | Input: i+i*i$ | Action: Shift i
Stack: $i | Input: +i*i$ | Action: Reduce
Stack: $ | Input: +i*i$ | Action: Shift +
Stack: $+ | Input: i*i$ | Action: Shift i
Stack: $+i | Input: *i$ | Action: Reduce
Stack: $+ | Input: *i$ | Action: Shift *
Stack: $+* | Input: i$ | Action: Shift i
Stack: $+*i | Input: $ | Action: Reduce
Stack: $+* | Input: $ | Action: Reduce
Stack: $+ | Input: $ | Action: Reduce
Stack: $ | Input: $ | Action: Accept
```