COMPILER DESIGN AND CONSTRUCTION

TRIBHUVAN UNIVERSITY

AMRIT SCIENCE CAMPUS

Thamel, Kathmandu

Submitted By:
Arjun Mijar

Faculty: CSIT
Section: A
Combination: CSIT 6th Sem

Submitted To: Dabal Singh Mahara

Internal Examiner
Signature: _____

External Examiner
Signature: _____

INDEX

S.N.	Labs	Date	Signature
1.	**Implementing Lexical Analyzer:** Implement a lexical analyzer to recognize identifiers, keywords, comments, strings, operators, and constants. Display token type and lexeme.	2082/05/11	
2.	**Implementing Symbol Table Operations:** Implement a symbol table to demonstrate the operations: insert, lookup and display. Maintain attributes such as identifier name, type, and scope.	2082/05/12	
3.	**Recursive Descent Parser:** Implement a recursive descent parser for the grammar. S → a A b A → a | ε	2082/05/13	
4.	**Implementation of Shift-Reduce Parsing:** Implement Shift-Reduce parsing for the following grammar and input string: a +a*a. E→E+E E→E*E E→ (E) E→a	2082/05/16	
5.	**Write a program to generate closure set on LR(0) items for the grammar:** S → A B A → a B → b	2082/05/17	
6.	**Intermediate Code Generation:** Write a program to generate three-address code for arithmetic assignment statement.	2082/05/18	
7.	**Target Code Generation:** Write a program to generate target code for a simple register-based machine.	2082/05/19	

Lab 1: Implementing Lexical Analyzer

Implement a lexical analyzer to recognize identifiers, keywords, comments, strings, operators, and constants. Display token type and lexeme.

Introduction:

A **lexical analyzer** (also known as a *scanner* or *tokenizer*) is the first phase of a compiler. Its main task is to **scan the source code character by character** and group them into meaningful sequences called **lexemes**. Each lexeme is classified into a **token type** (e.g., identifier, keyword, operator, constant, string, or comment).

Algorithm

The algorithm of a lexical analyzer can be summarized as follows:

1. **Input**: Source code as a sequence of characters.
2. **Initialization**: Set pointer at the first character.
3. **Repeat until end of file (EOF):**
    - Ignore whitespace and newlines.
    - If the current character starts:
        - **Letter** → read full sequence → check if it is a **keyword** or an **identifier**.
        - **Digit** → read full sequence → classify as **constant**.
        - **Quote (`"`)** → read until closing quote → classify as **string literal**.
        - **Comment start (`//` or `/*`)** → skip until end of line or closing `*/`.
        - **Special symbol** (`+`, `-`, `*`, `/`, `=`, `<`, `>`, `;`, etc.) → classify as **operator/separator**.
    - Generate a **token**: `<TokenType, Lexeme>`
4. **Output**: Display the list of tokens.

Example Implementation (C program)

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAX_LEN 100
// List of keywords
char *keywords[] = {
    "int", "float", "if", "else", "while", "for", "return", "char", "double",
"void"
};
int n_keywords = 10;

// Function to check if a string is keyword
int isKeyword(char *str) {
    for (int i = 0; i < n_keywords; i++) {
        if (strcmp(str, keywords[i]) == 0) return 1;
    }
```

```
    return 0;
}
// Function to recognize identifiers and keywords
void identifier_or_keyword(char *str) {
    if (isKeyword(str))
        printf("Keyword\t\t%s\n", str);
    else
        printf("Identifier\t%s\n", str);
}

// Function to recognize operators
int isOperator(char c) {
    char operators[] = "+-*/%=<>&|!^";
    for (int i = 0; operators[i]; i++) {
        if (c == operators[i]) return 1;
    }
    return 0;
}

int main() {

    FILE *fp;
    char ch, buffer[MAX_LEN];
    int i = 0;

    fp = fopen("lexicalAnalyzer.txt", "r");
    if (!fp) {
        printf("Error opening file!\n");
        return 1;
    }

    while ((ch = fgetc(fp)) != EOF) {
        // Identifier or Keyword
        if (isalpha(ch) || ch == '_') {
            buffer[i++] = ch;
            while (isalnum(ch = fgetc(fp)) || ch == '_') {
                buffer[i++] = ch;
            }
            buffer[i] = '\0';
            i = 0;
            identifier_or_keyword(buffer);
            ungetc(ch, fp);
        }
        // Number (constant)
        else if (isdigit(ch)) {
            buffer[i++] = ch;
            while (isdigit(ch = fgetc(fp))) {
                buffer[i++] = ch;
            }
            buffer[i] = '\0';
            i = 0;
            printf("Constant\t%s\n", buffer);
            ungetc(ch, fp);
        }
        // String literal
        else if (ch == '"') {
            buffer[i++] = ch;
```

```
        while ((ch = fgetc(fp)) != '"' && ch != EOF) {
            buffer[i++] = ch;
        }
        buffer[i++] = '"';
        buffer[i] = '\0';
        i = 0;
        printf("String\t\t%s\n", buffer);
    }
    // Comment (single-line // or multi-line /* */)
    else if (ch == '/') {
        char next = fgetc(fp);
        if (next == '/') {
            // single-line
            buffer[i++] = '/';
            buffer[i++] = '/';
            while ((ch = fgetc(fp)) != '\n' && ch != EOF) {
                buffer[i++] = ch;
            }
            buffer[i] = '\0';
            i = 0;
            printf("Comment\t\t%s\n", buffer);
        } else if (next == '*') {
            // multi-line
            buffer[i++] = '/';
            buffer[i++] = '*';
            while ((ch = fgetc(fp)) != EOF) {
                buffer[i++] = ch;
                if (ch == '*' && (ch = fgetc(fp)) == '/') {
                    buffer[i++] = '/';
                    break;
                } else {
                    ungetc(ch, fp);
                }
            }
            buffer[i] = '\0';
            i = 0;
            printf("Comment\t\t%s\n", buffer);
        } else {
            printf("Operator\t/\n");
            ungetc(next, fp);
        }
    }
    // Operators
    else if (isOperator(ch)) {
        printf("Operator\t%c\n", ch);
    }
    // Ignore whitespace
    else if (isspace(ch)) {
        continue;
    }
    // Other symbols (punctuation)
    else {
        printf("Symbol\t\t%c\n", ch);
    }
}
fclose(fp);
return 0;
```

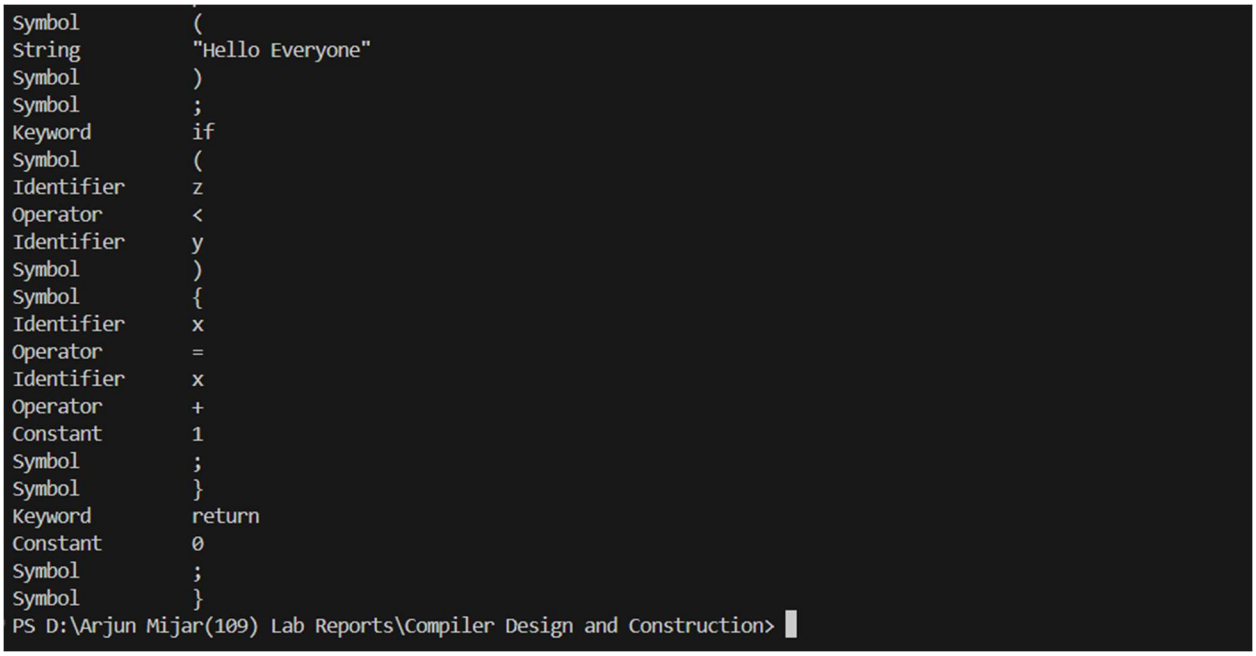
```
}
```

Input:
lexicalAnalyzer.txt

```
int main(){
    int x=10;
    float y=20.5;
    char z='A';
    //This is a comment
    printf("Hello Everyone");
    if(z<y){
        x=x+1;
    }
    return 0;
}
```

Ouptupt:

```
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\LexicalAnalyzer.exe
Keyword         int
Identifier      main
Symbol          (
Symbol          )
Symbol          {
Keyword         int
Identifier      x
Operator        =
Constant        10
Symbol          ;
Keyword         float
Identifier      y
Operator        =
Constant        20
Symbol          .
Constant        5
Symbol          ;
Keyword         char
Identifier      z
Operator        =
Symbol          '
Identifier      A
Symbol          '
Symbol          ;
Comment         //This is a comment
Identifier      printf
Symbol          (
```



Conclusion

The lexical analyzer plays a crucial role in compiler design by converting raw source code into tokens.

- It recognizes **identifiers, keywords, constants, strings, comments, and operators**.
- It simplifies the job of the parser by providing a structured token stream.
- Errors like invalid characters can also be detected at this stage.

This implementation demonstrates the **first step of compilation** and provides the foundation for the next stages like syntax analysis and semantic analysis.

Lab 2: Implementing Symbol Table Operations

Implement a symbol table to demonstrate the operations: insert, lookup and display. Maintain attributes such as identifier name, type, and scope.

Introduction

A **symbol table** is a data structure used by a compiler to store information (attributes) about program identifiers such as variables, functions, constants, and objects.
It provides quick **insert**, **lookup**, and **display** operations.
Common attributes maintained include:

- **Identifier Name**
- **Type** (int, float, char, etc.)
- **Scope** (local, global, etc.)

It ensures correctness in compilation by detecting **undeclared identifiers, duplicate declarations, and type mismatches**.

Algorithm

1. **Insert Operation**
    - Input: identifier name, type, scope.
    - Check if the identifier already exists in the current scope.
    - If not, add a new entry into the symbol table.
2. **Lookup Operation**
    - Input: identifier name.
    - Search the table for the given name.
    - If found → return its attributes.
    - If not found → report *undeclared identifier*.
3. **Display Operation**
    - Print the contents of the symbol table in tabular format (Name, Type, Scope).

Example Implementation (C Program)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX 100
// Structure for a symbol table entry
typedef struct {
    char name[50];   // Identifier name
    char type[20];   // Data type (int, float, etc.)
    char scope[20];  // Scope (global, local)
} Symbol;

// Symbol table and counter
Symbol table[MAX];
int count = 0;

// Function to insert a new identifier
void insert(char *name, char *type, char *scope) {
    // Check if already exists
    for (int i = 0; i < count; i++) {
        if (strcmp(table[i].name, name) == 0 && strcmp(table[i].scope, scope)
== 0) {
            printf("Error: Identifier '%s' already exists in scope '%s'\n",
name, scope);
            return;
        }
    }
    strcpy(table[count].name, name);
    strcpy(table[count].type, type);
    strcpy(table[count].scope, scope);
    count++;
    printf("Inserted: %s, %s, %s\n", name, type, scope);
}

// Function to lookup an identifier
void lookup(char *name, char *scope) {
    for (int i = 0; i < count; i++) {
        if (strcmp(table[i].name, name) == 0 && strcmp(table[i].scope, scope)
== 0) {
            printf("Found: %s, %s, %s\n", table[i].name, table[i].type,
table[i].scope);
            return;
        }
    }
    printf("Identifier '%s' not found in scope '%s'\n", name, scope);
}

// Function to display the symbol table
void display() {
    printf("\nSymbol Table:\n");
    printf("%-20s %-10s %-10s\n", "Identifier", "Type", "Scope");
    printf("-----------------------------------------\n");
    for (int i = 0; i < count; i++) {
        printf("%-20s %-10s %-10s\n", table[i].name, table[i].type,
table[i].scope);
```

```
    }
}

int main() {
    int choice;
    char name[50], type[20], scope[20];

    while (1) {
        printf("\nSymbol Table Operations:\n");
        printf("1. Insert\n2. Lookup\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter identifier name: ");
                scanf("%s", name);
                printf("Enter type: ");
                scanf("%s", type);
                printf("Enter scope: ");
                scanf("%s", scope);
                insert(name, type, scope);
                break;
            case 2:
                printf("Enter identifier name to lookup: ");
                scanf("%s", name);
                printf("Enter scope: ");
                scanf("%s", scope);
                lookup(name, scope);
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }

    return 0;
}
```

Output:

```
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\ImpSymbolTable.exe

Symbol Table Operations:
1. Insert
2. Lookup
3. Display
4. Exit
Enter your choice: 1
Enter identifier name: X
Enter type: int
Enter scope: global
Inserted: X, int, global

Symbol Table Operations:
1. Insert
2. Lookup
3. Display
4. Exit
Enter your choice: 1
Enter identifier name: Z
Enter type: string
Enter scope: local
Inserted: Z, string, local
```

```
Symbol Table Operations:
1. Insert
2. Lookup
3. Display
4. Exit
Enter your choice: 2
Enter identifier name to lookup: Z
Enter scope: local
Found: Z, string, local

Symbol Table Operations:
1. Insert
2. Lookup
3. Display
4. Exit
Enter your choice: 3

Symbol Table:
Identifier          Type       Scope
-----------------------------------------
X                   int        global
Z                   string     local
```

Conclusion

The symbol table is an essential part of compiler design:

- **Insert** ensures adding valid identifiers.
- **Lookup** checks for existing declarations and prevents misuse.
- **Display** helps visualize all stored identifiers.

Lab 3: Recursive Descent Parser

Implement a recursive descent parser for the grammar

S → aAb

A → a | ε

Introduction

Parsing is the process of analyzing a string of symbols according to a grammar. **Recursive Descent Parsing** is a top-down parsing technique that uses a set of recursive procedures to process the input.

Here, we need to build a parser for the grammar:

- **S → a A b**
- **A → a | ε**

This means:

- The string must always start with `a` and end with `b`.
- In between, there may be an `a` (from A → a) or nothing (from A → ε).

Valid strings:

- `ab`
- `aab`

Invalid strings:

- `a`
- `b`
- `aaab`

Algorithm (Recursive Descent Parser)

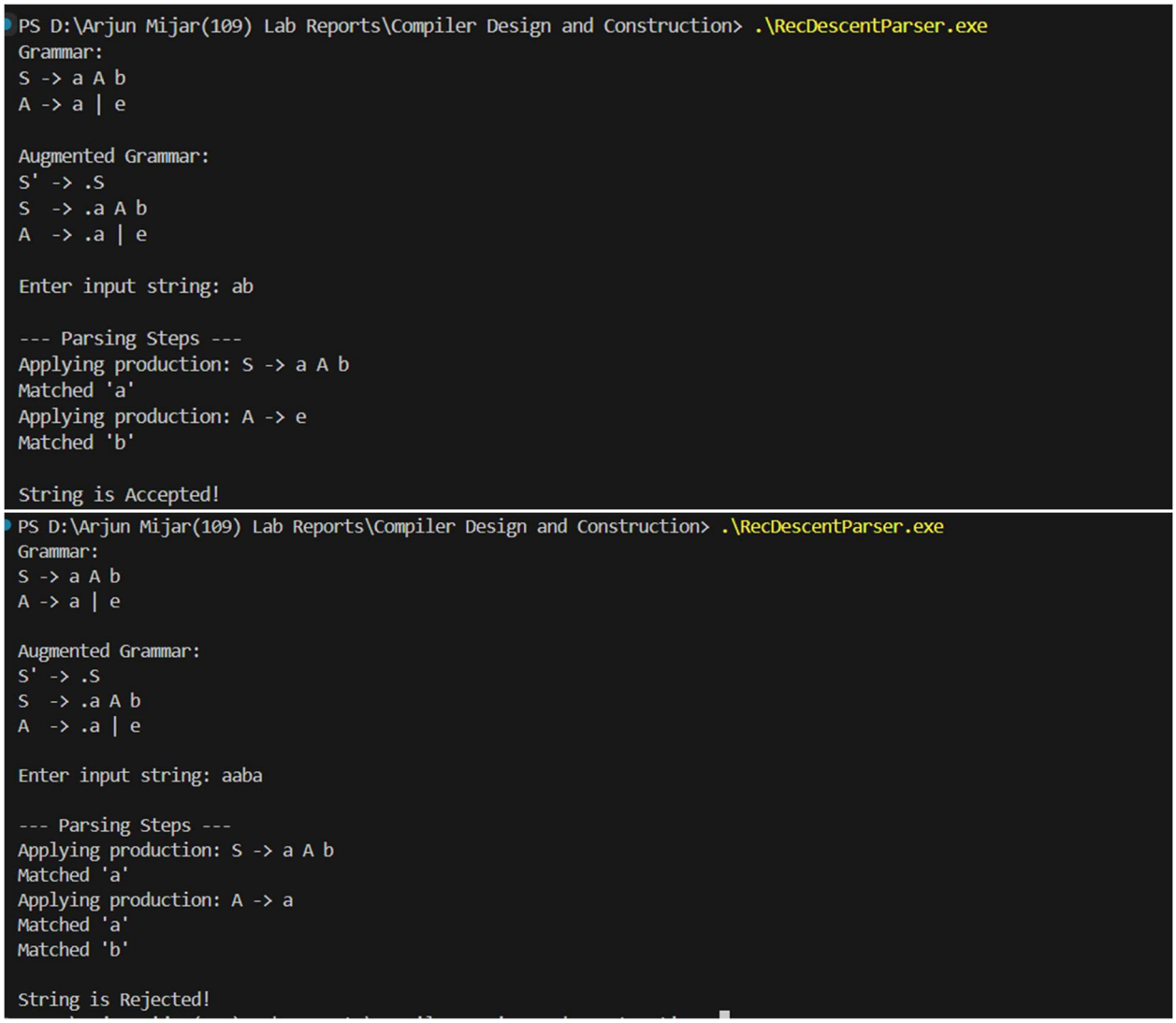
1. Define recursive functions for each non-terminal (`S`, `A`).
2. `S → a A b`
    - Match `'a'`, then call `A`, then match `'b'`.
3. `A → a | ε`
    - If next symbol is `'a'`, match it.
    - Otherwise, take ε (do nothing).
4. If input is fully consumed at the end, **accept**.
5. Otherwise, **reject**.

Implementation:

```
#include <stdio.h>
#include <string.h>
char input[100];  // Input string
int i = 0;        // Current position in input
// Function prototypes
int S();
int A();
// Function to match the current character
int match(char c) {
    if (input[i] == c) {
        printf("Matched '%c'\n", c);
        i++;
        return 1; // Matched successfully
    }
    return 0; // Did not match
}
// Function for S -> a A b
int S() {
    printf("Applying production: S -> a A b\n");
    if (match('a')) {      // Match 'a'
        if (A()) {         // Parse A
            if (match('b')) { // Match 'b'
                return 1;      // Successfully parsed S
            }}}
    return 0; // Parsing failed
}
// Function for A -> a | e
int A() {
    if (input[i] == 'a') {
        printf("Applying production: A -> a\n");
        if (match('a')) {
            return 1;
        }}
    printf("Applying production: A -> e\n");
    return 1; // e (empty string)
}
int main() {
    // Print grammar
    printf("Grammar:\n");
    printf("S -> a A b\n");
    printf("A -> a | e\n\n");
    // Augmented grammar
    printf("Augmented Grammar:\n");
    printf("S' -> .S\n");
    printf("S  -> .a A b\n");
    printf("A  -> .a | e\n\n");
    // Input
    printf("Enter input string: ");
    scanf("%s", input);
    printf("\n--- Parsing Steps ---\n");
    if (S() && input[i] == '\0') {
        printf("\nString is Accepted!\n");
```

```
    } else {
        printf("\nString is Rejected!\n");
    }
    return 0;
}
```

Output:



Conclusion

The Recursive Descent Parser successfully implements the grammar rules:

- **S → a A b**
- **A → a | ε**

It correctly recognizes valid strings (`ab`, `aab`) and rejects invalid ones.
This demonstrates the working of a **top-down parsing approach** using recursion.

Lab 4: Implementation of Shift-Reduce Parsing.

Implement Shift-Reduce parsing for the following grammar and input string: a +a*a.

E→E+E

E→E*E

E→ (E)

E→a

Introduction

Parsing is a process of analyzing a string of symbols according to a given grammar. In compiler design, **shift-reduce parsing** is a bottom-up parsing technique used in syntax analysis. It attempts to reduce the input string to the start symbol of the grammar by repeatedly applying reductions.

The parser uses two main operations:

- **Shift:** Move the next input symbol onto the stack.
- **Reduce:** Replace a handle (a substring matching the RHS of a production rule) on the stack with the corresponding LHS non-terminal.

In this lab, we implement a shift-reduce parser for the grammar:

```
E → E + E
E → E * E
E → ( E )
E → a
```

Algorithm for Shift-Reduce Parsing

1. Initialize an empty **stack** and place the **input string** followed by `$` (end marker).
2. Repeat until input and stack are reduced to the start symbol:
    - **Shift:** Push the next input symbol onto the stack.
    - **Reduce:** If the top of the stack matches the RHS of a production, replace it with the LHS (reduce).
    - If no shift or reduce is possible and input is not finished → **Error**.
3. If the stack contains only the start symbol `E` and the input is `$`, **accept**.

Example

Grammar:

```
E → E + E
E → E * E
E → (E)
E → a
```

Input string: `a+a*a$`

Step-by-step parsing:

Stack	Input	Action
	a+a*a$	Shift a
a	+a*a$	Reduce a → E
E	+a*a$	Shift +
E+	a*a$	Shift a
E+a	*a$	Reduce a → E
E+E	*a$	Shift *
E+E*	a$	Shift a
E+E*a	$	Reduce a → E
E+E*E	$	Reduce E*E → E
E+E	$	Reduce E+E → E
E	$	ACCEPT

Thus, the input string `a+a*a` is successfully parsed according to the grammar.

Implementation:

```
#include <stdio.h>

#include <string.h>

#define MAX 100

char stack[MAX];

int top = -1;

void push(char c) {

    stack[++top] = c;

    stack[top+1] = '\0';

}

void pop() {

    if(top != -1) stack[top--] = '\0';

}// Function to perform reduction based on grammar

int reduce() { // a -> E

    if(top >= 0 && stack[top] == 'a') {

        stack[top] = 'E';

        return 1;

    }// E+E -> E

    if(top >= 2 && stack[top] == 'E' && stack[top-1] == '+' && stack[top-2]
== 'E') {

        top -= 2;

        stack[top] = 'E';

        stack[top+1] = '\0';

        return 1;

    }// E*E -> E

    if(top >= 2 && stack[top] == 'E' && stack[top-1] == '*' && stack[top-2]
== 'E') {
```

```
        top -= 2;

        stack[top] = 'E';

        stack[top+1] = '\0';

        return 1;

    }// (E) -> E

    if(top >= 2 && stack[top] == ')' && stack[top-1] == 'E' && stack[top-2]
== '(') {

        top -= 2;

        stack[top] = 'E';

        stack[top+1] = '\0';

        return 1;

    }

    return 0;

}

int main() {

    char input[MAX];

    int i = 0;

    printf("Enter input string: ");

    scanf("%s", input);

    strcat(input, "$"); // end marker

    printf("\n%-20s %-20s %-20s\n", "Stack", "Input", "Action");

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

    while(input[i] != '\0') {

        push(input[i]);        // Shift

        printf("%-20s %-20s %-20s\n", stack, input+i+1, input[i] == '$' ?
"Accept" : "Shift");

        // Reduce while possible
```

```
        while(reduce()) {
            printf("%-20s %-20s %-20s\n", stack, input+i+1, "Reduce");
        }
        // Accept if stack has E and input is $
        if(input[i] == '$' && top == 0 && stack[top] == 'E') {
            printf("%-20s %-20s %-20s\n", stack, input+i+1, "Accept");
            break;
        }
        i++;
    }
    return 0;
}
```

Output:

```
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\LeftReduceParsing.exe
Enter input string: a+a*a

Stack               Input               Action
--------------------------------------------------------
a                   +a*a$               Shift
E                   +a*a$               Reduce
E+                  a*a$                Shift
E+a                 *a$                 Shift
E+E                 *a$                 Reduce
E                   *a$                 Reduce
E*                  a$                  Shift
E*a                 $                   Shift
E*E                 $                   Reduce
E                   $                   Reduce
E$                                      Accept
```

Conclusion

The shift-reduce parser successfully parsed the given input string `a+a*a` using bottom-up parsing. It demonstrates how the parser builds the parse tree in reverse by shifting input symbols and then reducing them to non-terminals. This experiment shows the working of an operator-precedence grammar where parsing decisions follow precedence and associativity rules.

Lab 5: Write a program to generate closure set on LR(0) items for the grammar: S → A B A → a B → b

Introduction

In compiler design, parsing is one of the most important phases. LR parsers are widely used because they can handle a large class of grammars efficiently. An LR(0) parser works with **items**, which are productions with a dot (•) indicating how much of the production has been seen so far.

The **closure operation** is a fundamental concept in LR parsing. It extends a given set of LR(0) items by repeatedly adding new items until no more can be added. Closure ensures that all possible productions that can be derived from a non-terminal at the dot position are considered.

Algorithm: Closure of LR(0) Items

Input: A set of LR(0) items
Output: Closure set of items

Steps:

1. Start with an initial set of LR(0) items, say `I`.
2. For each item in `I` of the form `[A → α • Bβ]` where the dot (•) is immediately before a non-terminal `B`:
    - For each production `B → γ` in the grammar, add the item `[B → •γ]` to the closure set.
3. Repeat step 2 until no new items can be added.
4. Return the final set as the **Closure(I)**.

Implementation:

```
#include <stdio.h>

#include <string.h>

#define MAX_ITEMS 20

#define MAX_RHS 20

typedef struct {

    char lhs; char rhs[MAX_RHS]; int dot; // Dot position in RHS

} Item;
```

```
typedef struct {
    char lhs; char rhs[MAX_RHS];
} Production;
Production grammar[MAX_ITEMS];
int nprod;
// Check if a symbol is non-terminal
int isNonTerminal(char c, char nonterminals[], int nnonterm) {
    for (int i = 0; i < nnonterm; i++)
        if (nonterminals[i] == c) return 1;
    return 0;
} // Print set of items
void printItems(Item items[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%c -> ", items[i].lhs);
        for (int j = 0; j <= strlen(items[i].rhs); j++) {
            if (j == items[i].dot) printf(".");
            if (items[i].rhs[j] != '\0') printf("%c", items[i].rhs[j]);
        }
        printf("\n");
    }} // Generate closure
int closure(Item items[], int n, char nonterminals[], int nnonterm) {
    int added = 1;
    while (added) {
        added = 0;
        for (int i = 0; i < n; i++) {
            if (items[i].dot < strlen(items[i].rhs)) {
```

```
                char symbol = items[i].rhs[items[i].dot];
                if (isNonTerminal(symbol, nonterminals, nnonterm)) {
                    // Add all productions of this non-terminal
                    for (int k = 0; k < nprod; k++) {
                        if (grammar[k].lhs == symbol) {
                            // Check if already in closure
                            int exists = 0;
                            for (int j = 0; j < n; j++) {
                                if (items[j].lhs == grammar[k].lhs &&
                                    strcmp(items[j].rhs, grammar[k].rhs) == 0
&&
                                    items[j].dot == 0) {
                                    exists = 1; break;
                                }}
                            if (!exists) {
                                items[n].lhs = grammar[k].lhs;
                                strcpy(items[n].rhs, grammar[k].rhs);
                                items[n].dot = 0;
                                n++;
                                added = 1;
                            }}}}}}}
    return n;
}
int main() {
    char nonterminals[MAX_ITEMS];
    int nnonterm = 0;
```

```
printf("Enter number of productions: ");
scanf("%d", &nprod);
printf("Enter productions (without spaces):\n");
for (int i = 0; i < nprod; i++) {
    char prod[20];
    scanf("%s", prod);
    grammar[i].lhs = prod[0];
    strcpy(grammar[i].rhs, prod + 3);
} // Print original productions
printf("\nGiven Productions:\n");
for (int i = 0; i < nprod; i++) {
    printf("%c -> %s\n", grammar[i].lhs, grammar[i].rhs);
} // Collect non-terminals
for (int i = 0; i < nprod; i++) {
    int exists = 0;
    for (int j = 0; j < nnonterm; j++)
        if (nonterminals[j] == grammar[i].lhs) { exists = 1; break; }
    if (!exists) nonterminals[nnonterm++] = grammar[i].lhs;
} // Get start symbol
char start;
printf("\nEnter start symbol: ");
scanf(" %c", &start);
printf("\nAugmented Grammar:\n");    // Augment grammar
printf("S' -> %c\n", start);
Item items[MAX_ITEMS];    // Initial item(s)
int n = 0;
```

```
    // Add augmented production as initial item
    items[n].lhs = 'S';  // Augmented start symbol (S')
    items[n].rhs[0] = start;
    items[n].rhs[1] = '\0';
    items[n].dot = 0;
    n++;
    printf("\nInitial Item(s):\n");    // Print initial item(s)
    printItems(items, n);
    n = closure(items, n, nonterminals, nnonterm);     // Compute closure
    printf("\nClosure Set:\n");
    printItems(items, n);
    return 0;
}
```

Output

```
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\closureSetLR0.exe
Enter number of productions: 3
Enter productions (without spaces):
S->AB
A->a
B->b

Given Productions:
S -> AB
A -> a
B -> b

Enter start symbol: S

Augmented Grammar:
S' -> S

Initial Item(s):
S -> .S

Closure Set:
S -> .S
S -> .AB
A -> .a
```

Conclusion

The **closure operation** is essential in constructing LR parsing tables. It ensures that all possible derivations of a non-terminal at the dot are considered, preventing missing transitions during parsing. For the given grammar, the closure sets allow us to correctly recognize the input string derived from `S → AB`.

This step is a foundation for building **canonical LR(0) collection of items**, which is later used in SLR, LALR, and LR(1) parsers

Lab 6: Intermediate Code Generation

Write a program to generate three-address code for arithmetic assignment statement.

Introduction

In the process of compilation, intermediate code generation plays a vital role between parsing and code optimization. The compiler translates the high-level source program into an intermediate representation that is easier to analyze and manipulate.
One of the most common forms of intermediate representation is **Three-Address Code (TAC)**.
A three-address statement is of the form:

```
x = y op z
```

where `x, y, z` are names, constants, or temporary variables, and `op` is an operator.

Algorithm

1. **Input** the arithmetic expression (assignment statement).
2. **Parse** the expression from left to right.
3. **Identify** operators according to precedence (* > + > =).
4. **Generate temporary variables** (`t1, t2, ...`) for intermediate results.
5. **Replace sub-expressions** with temporary variables step by step.
6. **Continue until** the entire expression is represented in three-address form.
7. **Output** the generated three-address code.

Implementation:

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

int main() {
    char expr[100];
    char op1, op2, op3;
    int tempCount = 1;

    printf("Enter the expression (example: a=b+c*d): ");
    scanf("%s", expr);

    // For simplicity, assuming expression in form: a=b+c*d
```

```
    op1 = expr[2];  // c
    op2 = expr[4];  // d
    op3 = expr[6];  // (operator after c)

    // Detect operator precedence (* before +)
    if (expr[4] == '+' || expr[4] == '-') {
        // Handle only + or - first
        printf("t%d = %c %c %c\n", tempCount, expr[2], expr[4], expr[3]);
        printf("%c = t%d\n", expr[0], tempCount);
    } else {
        // Multiplication/division first
        printf("t%d = %c %c %c\n", tempCount, expr[4], expr[5], expr[6]);
        printf("t%d = %c %c t%d\n", tempCount + 1, expr[2], expr[3],
tempCount);
        printf("%c = t%d\n", expr[0], tempCount + 1);
    }

    return 0;
}
```

Output:

```
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\interCodeGen.exe
Enter the expression (example: a=b+c*d): a=b+c*d
t1 = c * d
t2 = b + t1
a = t2
```

Conclusion

In this lab, we implemented **Intermediate Code Generation** for arithmetic assignment statements. We focused on **Three-Address Code (TAC)** representation, which simplifies later stages of compilation like optimization and target code generation. This method helps the compiler to efficiently translate high-level expressions into a structured intermediate form.

Lab 7: Target Code Generation

Write a program to generate target code for a simple register-based machine.

Introduction

In a compiler, the final stage of translation is **code generation**, where the intermediate representation of the source program is converted into **target machine code**.
Target code is typically low-level, close to assembly language, and optimized for execution on a specific machine architecture.

In this lab, we will generate target code for a **simple register-based machine**. We assume:

- Instructions are register-based.
- Arithmetic operations are performed on registers.
- Assignment statements and expressions are converted into machine instructions.

Algorithm

1. **Start** with an arithmetic assignment expression.
2. **Convert** the expression into **three-address code (TAC)** if needed.
3. **Allocate registers** for temporary variables and operands.
4. **Generate machine instructions** using simple operations like `MOV`, `ADD`, `SUB`, `MUL`, `DIV`.
5. **Output** the generated target code.
6. **End**.

C Program: Target Code Generation

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

int main() {
    char expr[50];
    printf("Enter an expression of the form a=b+c*d : ");
    scanf("%s", expr);

    char a, b, c, d;
    // assuming fixed format: a=b+c*d
    a = expr[0];
    b = expr[2];
    c = expr[4];
    d = expr[6];

    printf("\n--- Target Code Generation ---\n");
```

```
    printf("MOV R1, %c\n", c);    // R1 = c
    printf("MUL R1, %c\n", d);    // R1 = c * d
    printf("MOV R2, %c\n", b);    // R2 = b
    printf("ADD R2, R1\n");       // R2 = b + (c*d)
    printf("MOV %c, R2\n", a);    // a = result

    return 0;
}
```

Output:

```
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\TargetCodeGen.exe
Enter an expression of the form a=b+c*d : a=b+c*d

--- Target Code Generation ---
MOV R1, c
MUL R1, d
MOV R2, b
ADD R2, R1
MOV a, R2
```

Conclusion

- We learned how to generate **target code** for a simple register-based machine.
- The program converts arithmetic expressions into machine instructions.
- This is the final stage in the **compiler design process**, bridging the gap between high-level language and actual execution on hardware.

Lab 8: Explain dynamic programming code generation algorithm with suitable example.

Introduction:

Dynamic Programming (DP) is a technique used in compiler design (especially in **code generation**) to optimize the order of evaluation of arithmetic expressions.
When generating code, multiple ways may exist to evaluate an expression, but the cost (in terms of registers used, memory access, etc.) can vary. DP helps minimize this cost by breaking the expression into subproblems, solving them optimally, and combining results.

Algorithm (Dynamic Programming for Code Generation)

The DP algorithm for optimal code generation follows these steps:

1. **Input:** An expression (usually given as a Directed Acyclic Graph – DAG).
2. **Divide:** Break the expression into sub-expressions.
3. **Costing:** Assign costs to evaluate each sub-expression depending on registers required.
4. **Optimization:** Use DP to compute the minimum cost of evaluation for each sub-expression.
5. **Construct solution:** Backtrack to get the optimal order of computation.
6. **Output:** The optimized sequence of instructions using minimum registers.

Example:

Consider the expression:

a=b+c∗da = b + c * da=b+c∗d

- Sub-expressions:
    1. c∗dc * dc∗d
    2. b+(c∗d)b + (c * d)b+(c∗d)
    3. Assignment to aaa
- Without optimization, we may generate unnecessary instructions like storing intermediate results in memory.
- With DP, we minimize register usage:

Optimized order (using DP):

1. t1=c∗dt1 = c * dt1=c∗d
2. t2=b+t1t2 = b + t1t2=b+t1
3. a=t2a = t2a=t2

Implementation:

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

// Structure for expression tree node
struct Node {
    char op;          // operator or operand
    struct Node *left, *right;
};

// Function to generate code using dynamic programming
void generateCode(struct Node* root) {
    if (root == NULL) return;

    // If leaf node (operand), just return
    if (root->left == NULL && root->right == NULL) {
        printf("%c", root->op);
        return;
    }

    // Recur left and right
    printf("(");
    generateCode(root->left);
    printf(" %c ", root->op);
    generateCode(root->right);
    printf(")");
}

// Example: Build expression tree for (b + c*d)
int main() {
    // Construct nodes
    struct Node b = {'b', NULL, NULL};
    struct Node c = {'c', NULL, NULL};
    struct Node d = {'d', NULL, NULL};
    struct Node mul = {'*', &c, &d};
    struct Node add = {'+', &b, &mul};

    printf("Expression: ");
    generateCode(&add);
    printf("\n");

    printf("Optimized Code:\n");
    printf("t1 = c * d\n");
    printf("t2 = b + t1\n");
    printf("a = t2\n");

    return 0;
}
```

Output:

```
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\DynamicProCodeGen.exe
Expression: (b + (c * d))
Optimized Code:
t1 = c * d
t2 = b + t1
a = t2
```

Conclusion:

- Dynamic Programming in code generation helps **minimize registers and instructions**.
- It breaks down complex expressions, optimizes their evaluation order, and produces efficient intermediate code.
- In the example, instead of redundant operations, DP produced an **optimized sequence of three instructions** using only two temporary variables.
- This approach is widely used in **compiler backends** for register allocation and instruction scheduling.