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 */.
        - **Operator or 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>

char keywords[8][10] =
{"int","float","if","else","while","for","return","char"};

int isKeyword(char *str) {
    for(int i=0;i<8;i++) {
        if(strcmp(str, keywords[i])==0)
            return 1;
    }
    return 0;
}

int main() {
```

```
    char src[200];
    printf("Enter source code:\n");
    fgets(src,200,stdin);

    int i=0;
    while(src[i]!='\0') {
        if(isalpha(src[i])) { // identifier or keyword
            char buf[50]; int j=0;
            while(isalnum(src[i])) {
                buf[j++]=src[i++];
            }
            buf[j]='\0';
            if(isKeyword(buf))
                printf("<Keyword, %s>\n", buf);
            else
                printf("<Identifier, %s>\n", buf);
        }
        else if(isdigit(src[i])) { // number
            char buf[50]; int j=0;
            while(isdigit(src[i])) {
                buf[j++]=src[i++];
            }
            buf[j]='\0';
            printf("<Constant, %s>\n", buf);
        }
        else if(src[i]=='"') { // string literal
            char buf[100]; int j=0;
            buf[j++]='"'; i++;
            while(src[i]!='"' && src[i]!='\0') {
                buf[j++]=src[i++];
            }
            if(src[i]=='"') buf[j++]='"';
            buf[j]='\0'; i++;
            printf("<String, %s>\n", buf);
        }
        else if(src[i]=='/' && src[i+1]=='/') { // single-line comment
            while(src[i]!='\n' && src[i]!='\0') i++;
        }
        else if(strchr("+-*/=;(){}",src[i])) { // operators/symbols
            printf("<Operator, %c>\n", src[i]);
            i++;
        }
        else i++; // skip spaces, tabs, etc.
    }
    return 0;
}
```

Ouptupt:

```
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\LexicalAnalyzer.exe
Enter source code:
int a=10; float b=20.5; if(a<b){print("hello world");}
<Keyword, int>
<Identifier, a>
<Operator, =>
<Constant, 10>
<Operator, ;>
<Keyword, float>
<Identifier, b>
<Operator, =>
<Constant, 20>
<Constant, 5>
<Operator, ;>
<Keyword, if>
<Operator, (>
<Identifier, a>
<Identifier, b>
<Operator, )>
<Operator, {>
<Identifier, print>
<Operator, ;>
<Operator, }>
```

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>

#define SIZE 50   // max number of symbols

// Structure of a symbol table entry
struct Symbol {
    char name[30];
    char type[10];
    char scope[10];
};

struct Symbol table[SIZE];
int count = 0;

// Insert into symbol table
void insert(char name[], char type[], char scope[]) {
    // check duplicate
    for(int i=0; i<count; i++) {
        if(strcmp(table[i].name, name)==0 && strcmp(table[i].scope,
scope)==0) {
            printf("Error: Duplicate entry for %s 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);
}

// Lookup in symbol table
int 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)
            return i;
    }
    return -1;
}

// Display symbol table
void display() {
    printf("\n--- Symbol Table ---\n");
    printf("%-15s %-10s %-10s\n", "Identifier", "Type", "Scope");
    printf("-----------------------------------\n");
    for(int i=0; i<count; i++) {
        printf("%-15s %-10s %-10s\n", table[i].name, table[i].type,
table[i].scope);
    }
}
```

```
int main() {
    insert("x","int","global");
    insert("y","float","global");
    insert("x","int","local");
    insert("z","char","local");

    display();

    char id[10]="x", sc[10]="local";
    int pos = lookup(id, sc);
    if(pos!=-1)
        printf("\nLookup: %s found at position %d with type %s\n", id, pos,
table[pos].type);
    else
        printf("\nLookup: %s not found in scope %s\n", id, sc);

    return 0;
}
```

Output:

```
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\ImpSymbolTable.exe
Inserted: x, int, global
Inserted: y, float, global
Inserted: x, int, local
Inserted: z, char, local

--- Symbol Table ---
Identifier      Type        Scope
-----------------------------------
x               int         global
y               float       global
x               int         local
z               char        local

Lookup: x found at position 2 with type int
```

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

$S \rightarrow aAb$

$A \rightarrow a \mid \varepsilon$

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)

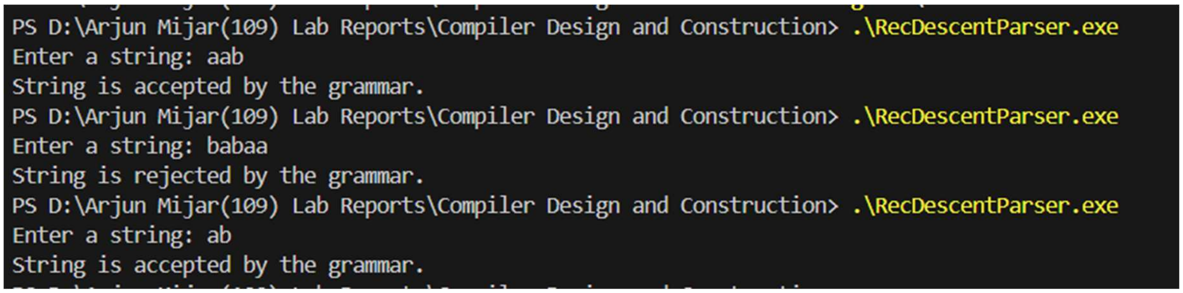
1. Define recursive functions for each non-terminal (`S`, `A`).
2. `S → a A b`
   - Match `'a'`, then call `A`, then match `'b'`.
3. `A → a | ε`
   - If next symbol is `'a'`, match it.
   - Otherwise, take ε (do nothing).
4. If input is fully consumed at the end, **accept**.
5. Otherwise, **reject**.

Implementation:

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

char input[100];   // input string
int pos = 0;       // current position in input
int length;        // length of input

// Function declarations
int S();
int A();

// Function to match a character
int match(char expected) {
    if (pos < length && input[pos] == expected) {
        pos++;
        return 1; // success
    }
    return 0; // failure
}

// Grammar rule: S → a A b
int S() {
    int start = pos;
    if (match('a')) {
        if (A()) {
            if (match('b')) {
                return 1; // success
            }
        }
    }
    pos = start; // backtrack
    return 0;
}

// Grammar rule: A → a | ε
int A() {
    int start = pos;
    if (match('a')) {
        return 1; // matched 'a'
    }
    // ε (empty string) is always valid
    return 1;
}

int main() {
    printf("Enter a string: ");
    scanf("%s", input);

    length = strlen(input);

    if (S() && pos == length) {
        printf("String is accepted by the grammar.\n");
```

```
    } else {
        printf("String is rejected by the grammar.\n");
    }

    return 0;
}
```

Output:



```
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\RecDescentParser.exe
Enter a string: aab
String is accepted by the grammar.
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\RecDescentParser.exe
Enter a string: babaa
String is rejected by the grammar.
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\RecDescentParser.exe
Enter a string: ab
String is accepted by the grammar.
```

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>

char input[50];         // input string

char stack[50];         // parsing stack

int top = -1;           // stack pointer

int i = 0;              // input pointer

// Push function

void push(char c) {

    stack[++top] = c;

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

}

// Pop function

void pop() {

    stack[top] = '\0';

    top--;

}

// Try reductions

void check() {

    // E -> a

    if (stack[top] == 'a') {

        pop();

        push('E');

        printf("\tReduce E->a\n");

    }

    // E -> (E)
```

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

        pop(); pop(); pop(); // remove ( E )

        push('E');

        printf("\tReduce E->(E)\n");

    }



    // E -> E+E

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

        pop(); pop(); pop();

        push('E');

        printf("\tReduce E->E+E\n");

    }



    // E -> E*E

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

        pop(); pop(); pop();

        push('E');

        printf("\tReduce E->E*E\n");

    }

}



int main() {

    printf("Enter input string: ");

    scanf("%s", input);
```

```
printf("\nSHIFT-REDUCE PARSING STEPS\n");

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

printf("Stack\tInput\tAction\n");

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


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

    // Shift

    push(input[i]);

    printf("%s\t%s\tShift\n", stack, input+i+1);

    i++;


    // Try reductions

    check();

}


// Final check

while (top > 0) {

    check();

}


if (strcmp(stack, "E") == 0) {

    printf("\nString accepted!\n");

} else {

    printf("\nString rejected!\n");

}
```

```
    return 0;
}
```

Output:

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

SHIFT-REDUCE PARSING STEPS
--------------------------------------------
Stack           Input           Action
--------------------------------------------
a               +a*a            Shift
                                Reduce E->a
E+              a*a             Shift
E+a             *a              Shift
                                Reduce E->a
                                Reduce E->E+E
E*              a               Shift
E*a                             Shift
                                Reduce E->a
                                Reduce E->E*E

String accepted!
```

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 10



// Structure for an item

struct Item {
```

```
    char lhs;

    char rhs[MAX];

    int dotPos;  // position of the dot

};


// Function to print an item

void printItem(struct Item item) {

    int i;

    printf("%c -> ", item.lhs);

    for (i = 0; i < strlen(item.rhs); i++) {

        if (i == item.dotPos) {

            printf(".");

        }

        printf("%c", item.rhs[i]);

    }

    if (item.dotPos == strlen(item.rhs)) {

        printf(".");

    }

    printf("\n");

}


// Closure function (basic for LR(0))

void closure(struct Item items[], int n) {

    printf("\nClosure Set:\n");

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

        printItem(items[i]);
```

```
        // If dot is before a non-terminal, add its productions
        if (items[i].dotPos < strlen(items[i].rhs)) {
            char nextSymbol = items[i].rhs[items[i].dotPos];

            if (nextSymbol == 'A') {
                struct Item newItem = {'A', "a", 0};
                printItem(newItem);
            }
            else if (nextSymbol == 'B') {
                struct Item newItem = {'B', "b", 0};
                printItem(newItem);
            }
        }
    }
}

int main() {
    // Initial Item: S -> .AB
    struct Item items[MAX];
    int n = 1;

    items[0].lhs = 'S';
    strcpy(items[0].rhs, "AB");
    items[0].dotPos = 0;
```

```
    printf("Grammar:\n");
    printf("S -> AB\n");
    printf("A -> a\n");
    printf("B -> b\n");

    closure(items, n);

    return 0;
}
```

Output

```
PS D:\Arjun Mijar(109) Lab Reports\Compiler Design and Construction> .\closureSetLR0.exe
Grammar:
S -> AB
A -> a
B -> b

Closure Set:
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.