MINI ASSIGNMENT – 3

Kagita Meenakshi(MA23BTECH11013)

Meka Bhavya Kumari(MA23BTECH11016)

Reddy Roshni(MA23BTECH11021)

1. **Explain how lexical errors are handled in a compiler. Give an example where a lexical analyzer would flag an error.**

**ANS :**

**How Lexical Errors Are Handled in a Compiler**

Introduction

Lexical analysis is the first phase of a compiler where the source code is read character by character and grouped into meaningful sequences called *tokens*. Occasionally, during this process, the lexical analyzer may encounter character sequences that do not conform to the language's lexical rules. These are referred to as lexical errors.

Types of Lexical Errors

Some common examples of lexical errors include:

* Unrecognized symbols: Characters not part of the language's alphabet (e.g., @ in C).
* Invalid identifiers: Names that do not follow the rules (e.g., 9abc in C).
* Incorrect numeric formats: Numbers with invalid syntax (e.g., 12..34).
* Unterminated string literals: Strings without a closing quote (e.g., "Hello).

Error Detection

Lexical errors are typically detected when the input fails to match any of the regular expressions specified in the lexical specification (like the rules defined in Lex/Flex). The scanner then invokes an error-handling routine to manage the situation gracefully.

**Error Recovery Techniques**

1. **Panic Mode Recovery:**
   * The lexer skips characters until it finds a recognizable delimiter (e.g., space, semicolon, newline).
   * This is a simple and safe technique, but may skip too much input.
2. **Local Correction:**
   * The lexer tries to fix the error by making minimal changes:
     + Inserting missing characters
     + Deleting illegal characters
     + Replacing incorrect symbols
3. **Using Default Tokens:**
   * Some compilers define a special token like INVALID for unrecognized input, which is reported and then ignored.
4. **Error Reporting and Logging:**
   * The lexer reports the type of error, its location (line and column), and possibly a helpful message.
   * This helps developers trace and fix the problem quickly.

**Example**

Consider the following C code snippet:

int 9value = 10;

Here, 9value is not a valid identifier because identifiers in C cannot begin with a digit. The lexical analyzer will detect this as a lexical error. It may:

* Flag and skip the invalid token
* Replace it with a placeholder
* Report: "Lexical error at line 1: invalid identifier '9value'"

1. **Describe the role of a symbol table. How does lexical analysis contribute to its construction?**

**ANS :**

Role of the Symbol Table and the Contribution of Lexical Analysis:

What is a Symbol Table?

A symbol table is a crucial data structure used in compilers to record information about identifiers in a program. Each entry in the symbol table corresponds to a unique name and stores attributes relevant to that identifier.

Contents of the Symbol Table

Each entry typically contains:

* Name: Identifier or symbol name
* Type: Data type (int, float, char, etc.)
* Scope: Where the symbol is valid (global, local, function-level)
* Memory address/offset: Used during code generation
* Additional attributes: Parameter types, array dimensions, etc.

Why is the Symbol Table Important?

The symbol table plays a central role in:

* Semantic analysis: Ensuring variables are declared before use
* Type checking: Verifying type compatibility in expressions
* Code generation: Determining memory locations and type-specific instructions
* Optimization: Reusing memory or detecting constant expressions

Role of Lexical Analysis in Building the Symbol Table

The lexical analyzer directly contributes to symbol table construction by:

1. Recognizing Identifiers:
   * Whenever it encounters a valid identifier (e.g., variable, function name), it generates a token and triggers a symbol table lookup or insertion.
2. Avoiding Duplicates:
   * It ensures that identifiers are not redundantly inserted.
3. Providing Metadata:
   * If the context allows (e.g., after reading a datatype keyword), it can help annotate the type of the identifier.
4. Efficiency:
   * Since lexical analysis is the first phase, an efficient symbol table helps subsequent phases operate faster.

Example

Given the input:

float rate = 3.5;

* The lexer recognizes float as a datatype keyword.
* rate is identified as an identifier and added to the symbol table with type float.
* The number 3.5 is tagged as a floating-point constant.

The parser and semantic analyzer then use this information for type checking and memory allocation.

1. **Write a lex program (lex3.l) to count and print the number of words (space separated), letters and vowels in a given sentence. Also, print the set of vowels observed in the given sentence.**

**ANS :**

Approach

* A word is any sequence of characters separated by spaces.
* Letters are matched using the regular expression [A-Za-z].
* Vowels are tracked using a lookup on the character.
* The program reads file names from the command line (./lex3.out input.txt output.txt) and redirects stdin and stdout.

Code for lex3.l

%{

#include <stdio.h>

#include <ctype.h>

#include <string.h>

int word\_count = 0;

int letter\_count = 0;

int vowel\_count = 0;

int in\_word = 0;

int vowels\_seen[256] = {0}; // ASCII-based tracking

void add\_vowel(char c) {

if (!vowels\_seen[(int)c])

vowels\_seen[(int)c] = 1;

}

%}

%%

[ \t\n] { in\_word = 0; } // Whitespace ends a word

[A-Za-z] {

letter\_count++;

if (!in\_word) {

word\_count++;

in\_word = 1;

}

char lower = tolower(yytext[0]);

if (lower == 'a' || lower == 'e' || lower == 'i' || lower == 'o' || lower == 'u') {

vowel\_count++;

add\_vowel(yytext[0]);

}

}

. { /\* Ignore non-letter characters \*/ }

%%

int main(int argc, char \*\*argv) {

if (argc != 3) {

fprintf(stderr, "Usage: %s <input-file> <output-file>\n", argv[0]);

return 1;

}

FILE \*infile = fopen(argv[1], "r");

FILE \*outfile = fopen(argv[2], "w");

if (!infile || !outfile) {

fprintf(stderr, "Error opening input or output file.\n");

return 1;

}

yyin = infile;

yyout = outfile;

yylex();

fprintf(outfile, "Words = %d\n", word\_count);

fprintf(outfile, "Letters = %d\n", letter\_count);

fprintf(outfile, "Vowels = %d\n", vowel\_count);

fprintf(outfile, "Set = ");

for (int i = 0; i < 256; i++) {

if (vowels\_seen[i])

fprintf(outfile, "%c ", i);

}

fprintf(outfile, "\n");

fclose(infile);

fclose(outfile);

return 0;

}

1. **Write a lex program (lex4.l) to recognize and log the occurrences of the following patterns from a subset of C.**

**ANS:**

Notes

* Valid identifiers: must begin with a letter or underscore, followed by letters/digits/underscores.
* Constants: integer (123), float (3.14), and boolean (true, false).
* Lex rules are prioritized in top-down order—more specific patterns must appear earlier.

**Code for lex4.l**

%{

#include <stdio.h>

%}

IDENT [a-zA-Z\_][a-zA-Z0-9\_]\*

INTCONST [0-9]+

FLOATCONST [0-9]+\.[0-9]+

%%

"int" | "float" | "bool" | "char" | "void" { printf("datatype : %s\n", yytext); }

"for" | "while" | "do" | "break" | "continue" { printf("keyword : %s\n", yytext); }

"if" | "else" | "const"| "unsigned" | "return" { printf("keyword : %s\n", yytext); }

"+" | "-" | "/" | "\*" | "=" { printf("operator : %s\n", yytext); }

"<" | ">" | "==" | "&&" | "||" { printf("operator : %s\n", yytext); }

"(" | ")" | "[" | "]" | "{" | "}" | ";" | "'" { printf("punctuation : %s\n", yytext); }

"true" | "false" { printf("constant : %s\n", yytext); }

{FLOATCONST} { printf("constant : %s\n", yytext); }

{INTCONST} { printf("constant : %s\n", yytext); }

{IDENT} { printf("identifier : %s\n", yytext); }

[ \t\n] { /\* Skip whitespace \*/ }

. { /\* Ignore unknown characters \*/ }

%%

int main() {

yylex();

return 0;

}