Lexical Analysis and Flex

Introduction to Lexical Analysis

Primary tasks of a lex

- Converts source code into tokens
- Removes whitespace, comments
- Identifies keywords, identifiers, literals, operators, etc.
- Reporting lexical errors

Role of a Lexer

Lex acts as a bridge between raw code and the parser.

Three main roles of a lexer:

- Tokenization: Breaking input into meaningful symbols
- Error detection (e.g., illegal characters)
- Communicating with the parser

Example of Lexical Analysis

Input: int x = 10;

Output:

Example

```
#include <stdio.h>
int main() {
   int num = 5;
   printf("Number: %d", num);
   return 0;
}
```

<preprocessor, #include>

<library, stdio.h>

<keyword, int>

<identifier, main>

<left_paren, (>

<right_paren,)>

<left_brace, {>

<keyword, int>

<identifier, num>

<operator, =>

<integer, 5>

<semicolon, ;>

<identifier, printf>

<left_paren, (>

<string_literal, "Number: %d">

<comma, ,>

<identifier, num>

<right_paren,)>

<semicolon, ;>

<keyword, return>

<integer, 0>

<semicolon, ;>

<right_brace, }>

Introduction to Flex

A tool to generate lexical analyzers

Uses regular expressions to define patterns

Generates a lex.yy.c program that recognizes tokens

Uses: Compiler development, text processing, and custom scripting languages.

How Does Flex Work?

Define rules

Flex generates C code

flex lexer.l # Generates 'lex.yy.c'

Compile the generated lexer

gcc lex.yy.c -o -lfl # Compiles the lexer

Run the lexer

./a.out # Runs the lexer

Lex Program Structure

%{

  // Definitions (C code, headers, global variables)

%}

Declarations

%%

Rules Section (Pattern matching & Actions)

%%

// Code Section (Optional main function)

Problems

- Scans text character by character
- Look ahead character determines what kind of token to read and when the current token ends
- First character cannot determine what kind of token we are going to read
- Handling of blanks

Problems

intvariable = 10;

int @x = 10;

String x = "Hello;

a = b+++c;

Optimization in Lex

- Error handling
- Token Stream Preprocessing (if(..))
- Using Character Classes in Regex
  - [a-zA-Z_] [a-zA-Z0-9_]*
  - [_a-zA-Z][_a-zA-Z0-9]*

Symbol Table

Stores information for subsequent phases

Interface to the symbol table

- Insert(s,t): save lexeme s and token t and return pointer
- Lookup(s): return index of entry for lexeme s or 0 if s is not found

Types of tokens

- One token for each keyword
- Tokens for the operators, either individually or in classes
- One or more tokens representing constants
- Tokens for each punctuation symbol, such as left and right parenthesis, comma and semicolon

Applications of Lexical Analyzer

- Compiler Design
- Text Processing Tools
- Natural Language Processing
- Code Editors & IDEs
- Security & Malware Detection
- Plagiarism Detection

Makefile

A Makefile is a special file containing a set of rules to automate the compilation of programs.

Key Points:

- Specifies how source files depend on each other.
- Defines compilation steps.
- Automates dependency tracking.

Makefile Syntax

A Makefile consists of rules, dependencies, and commands.

```
target: dependencies
    command
```

target: The file to be created.

dependencies: Files required to build the target.

command: Command to create the target.

Code example for lex

```
%{
#include <stdio.h>
%}
INT int
letters [_A-Za-z]

%%

{INT}     { printf("Keyword: INT\n"); }
[0-9]+  { printf("Integer: %s\n", yytext); }
{letters}[a-zA-Z0-9_]* { printf("Identifier: %s\n", yytext); }
=        { printf("OPERATOR: =\n");}
; {printf("SPECIAL: ;\n");}

%%

int main() {
  yylex();
  return 0;
}
```

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

DIGIT      [0-9]+
ID         [a-zA-Z_][a-zA-Z0-9_]*
OPERATOR   [+\-*/=]

%%

"if"        { printf("<KEYWORD, if>\n"); }
"else"      { printf("<KEYWORD, else>\n"); }

{ID}        { printf("<IDENTIFIER, %s>\n", yytext); }
{DIGIT}     { printf("<NUMBER, %s>\n", yytext); }
{OPERATOR}  { printf("<OPERATOR, %s>\n", yytext); }

"("         { printf("<LEFT_PAREN, (>\n"); }
")"         { printf("<RIGHT_PAREN, )>\n"); }
"{"         { printf("<LEFT_BRACE, {>\n"); }
"}"         { printf("<RIGHT_BRACE, }>\n"); }
";"         { printf("<SEMICOLON, ;>\n"); }
[ \t\n]     ;
"//".*      ;
.           { printf("<UNKNOWN, %s>\n", yytext); }

%%

int main() {
    printf("Enter you test sample:\n");
    yylex();
    return 0;
}
```