**NOTRE DAME UNIVERSITY BANGLADESH**



Compiler Design

Report

**Submitted to**:

Humayara Binte Rashid (Lecturer NDUB)

**Submitted by**:

Shazidul Alam 09

Anisur Rahman Naeem 15

Faisal Morshed Akash 07

Batch: CSE-19

**1. Introduction**

Compiler Overview:

The definitions provided are concise summaries found in compiler design literature and textbooks such as "Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, also known as the "Dragon Book."

a) Lexical Analyzer: A lexical analyzer reads source code character by character, identifies tokens like keywords and identifiers, and produces a stream of tokens as output.

b) Syntax Analyzer: A syntax analyzer, or parser, checks the structure of source code using grammar rules, ensuring correctness and detecting syntax errors.

c) Semantic Analyzer: A semantic analyzer checks the meaning of source code beyond its structure, verifying type compatibility, variable declarations, and function calls.

d) Intermediate Code Generator: An intermediate code generator creates an intermediary representation of source code, which is independent of the target machine architecture, facilitating optimization and portability.

e) Machine-Independent Code Optimizer: A machine-independent code optimizer improves the efficiency and performance of intermediary code, regardless of the target machine, through various optimization techniques.

f) Code Generator: A code generator translates optimized intermediary code into target-specific instructions, such as machine code or assembly language, for execution on the target hardware.

g) Machine-Dependent Code Optimizer: A machine-dependent code optimizer further enhances the efficiency and performance of generated code by tailoring optimizations to the specific characteristics of the target hardware architecture.

**2. Problem Statemen**t:

Our project aims to develop a lexer for C code using Flex. The lexer will tokenize input C code, identifying keywords, identifiers, numbers, operators, and header files. The tokenized output will assist in syntax highlighting, code analysis, and understanding of the structure of C programs. This lexer will be helpful for developers, compilers, and code editors in processing C code efficiently and accurately.

**3. Tools**

1) Flex Installation: Download flex from any browser then install it.

2) Path Selection: Copy the bin file location from flex and paste it on system environment variables.

3) Text File: Create a text file. Rename it and add .l as extension. i.e: demo.l .

4) File: Create a completely separate file and put the .l file in it.

5) Command Promt: Firstly, run the .l file as flex demo.l

Secondly run the lex file gcc lex.yy.c

Thirdly run the .exe file as a.exe

**4. Token Table**:

|  |  |  |
| --- | --- | --- |
| **Token** | **Description** | **Example** |
| if | Conditionally execute a block of code | if (x > 5){  printf("x is greater than 5\n"); |
| else | The "else" keyword specifies a block of code that executes when the "if" condition is false. | if (x > 5) {  printf("x is greater than 5\n"); } else { printf("x is 5 or less\n"); |
| while | The "while" keyword in programming languages is used to create a loop. | while (count < 5) { printf("Count: %d\n", count); |
| do | The "do" keyword is used in programming languages to create a loop that executes a block of code at least once. | do { printf("Count: %d\n", count);  count++; |
| switch | The "switch" keyword is used in programming languages to create a selection statement. | switch (day) {  case 1: printf("Monday\n"); break;  case 2: printf("Tuesday\n"); break; |
| case | The "case" keyword is used within a switch statement in programming languages to define different branches of code | switch (num) {  case 1: printf("one\n"); break;  case 2: printf("two\n"); break; |
| for | The "for" keyword is used in programming languages to create a loop that repeats a block of code. | for (int i = 0; i < 5; i++) {  printf("Value of i: %d\n", i); |

|  |  |  |
| --- | --- | --- |
| **Token** | **Description** | **Example** |
| char | The char keyword is used to declare a variable of character. | char letter = 'A'; |
| const | The const keyword in programming languages such as C is used to declare a variable whose value cannot be changed once it is initialized. | const int num = 42; |
| continue | The continue keyword is used to skip the remaining code in the current iteration of a loop and proceed to the next iteration. | for (int i = 0; i < 5; i++) {  if (i == 2) { continue; |
| default | The default keyword is used in a switch statement to specify the code block that should execute if none of the specified case options match the switch expression. | case 2: printf("Value is 2\n");  break;  default: printf  ("Value is not 1 or 2\n"); break; |
| double | The double keyword used to declare a variable that can store double-precision floating-point numbers. | double pi = 3.141592653589793; |
| enum | The enum keyword is used to define a new data type that consists of a set of named integer constants. | enum Color{RED, GREEN,  BLUE };  int main(){ enum Color favoriteColor = GREEN; |

|  |  |  |
| --- | --- | --- |
| **Token** | **Description** | **Example** |
| extern | The extern keyword is used to declare a variable or function that is defined in another source file. | int externVar = 10;  void externFunction() {  printf("This is an externl function.\n"); |
| float | The float keyword is used to declare a variable that can store single-precision floating-point numbers. | float pi = 3.14f; |
| goto | The goto keyword is used to transfer control unconditionally to a specified label in the code. | if (num > 3) { goto label;  label: printf("This is the label. \n"); |
| int | The int keyword is used to declare a variable that can store integer values. | int number = 10; |
| long | The long keyword is used to declare a variable that can store long integer values. | long largeNumber = 1000000000L; |
| register | The register keyword is a storage class specifier that suggests to the compiler that a variable should be stored for faster access. | register int counter; |
| return | The return keyword is used such as C to exit a function and return control to the caller. | return 0; |

|  |  |  |
| --- | --- | --- |
| **Token** | **Description** | **Example** |
| short | The short keyword is a data type specifier used to declare a variable that can store integer values but with a smaller range. | short smallNumber = 32767; |
| signed | The signed keyword is a data type specifier used to declare a variable that can store both positive and negative integer values. | signed int number = -10;  printf("The signed integer value is: %d\n", number); |
| sizeof | The sizeof operator is used to determine the size of a data type or a variable in bytes. | int intSize = sizeof(int); |
| static | The static keyword in C is used to control the visibility and lifetime of variables and functions. | static int count = 0; |
| struct | struct (short for "structure") is a composite data type that allows us to group different data types together into a single unit. | struct Person { char name[50];  int age; |
| typedef | The typedef keyword is used to create a new name or alias for an existing data type. | typedef unsigned long ulong;  int main() ulong largeNumber = 1000000UL; { |

|  |  |  |
| --- | --- | --- |
| **Token** | **Description** | **Example** |
| break | The break keyword in programming languages is used to exit a loop.  (for, while, or do-while) | case 1: printf("One\n");  break; |
| union | The union keyword is used to define a user-defined data type that allows multiple members to share the same memory location. | union Data { int i;  float f;  char str[20]; |
| unsigned | The unsigned keyword is a data type specifier that declares a variable or integer type as only capable of holding non-negative values. | int main() {  unsigned int positiveNumber = 4294967295; |
| void | The void keyword is a special data type that represents the absence of any type. | void printMessage() {  printf("This function returns nothing.\n"); |
| volatile | The volatile keyword is used as a qualifier for variables to indicate that their values may change unexpectedly and frequently, outside of the normal program flow. | volatile int interruptFlag = 0; |

**5. Code**

 digit [0-9]

letter [A-Za-z]

identifier {letter}({letter}|{digit})\*

number ({digit})+((\.({digit})+)?("e^"[+-]?({digit})+)?)?

%{

#include <stdio.h>

int keyword\_count = 0;

int operator\_count = 0;

int function\_count = 0;

int header\_count = 0;

int additional\_token\_count = 0;

int idcount=0;

%}

%%

"#include <stdio.h>"|"#include <string.h>"|"#include <stdlib.h>"|"#include <math.h>" {printf("Header File:  %s\n",yytext);}

"#include <time.h>"|"#include <ctype.h>"|"#include <stdbool.h>" {printf("Header File:  %s\n",yytext);}

"#include <stdint.h>"|"#include <stddef.h>"|"#include <stdarg.h>" {printf("Header File:  %s\n",yytext);}

"#include <assert.h>"|"#include <errno.h>"|"#include <fcntl.h>" {printf("Header File:  %s\n",yytext);}

"#include <limits.h>"|"#include <locale.h>"|"#include <signal.h>" {printf("Header File:  %s\n",yytext);}

"#include <setjmp.h>"|"#include <unistd.h>"|"#include <wchar.h>" {printf("Header File:  %s\n",yytext);}

"#include <wctype.h>"|"#include <arpa/inet.h>"|"#include <dirent.h>" {printf("Header File:  %s\n",yytext);}

"#include <dlfcn.h>"|"#include <errno.h>"|"#include <fcntl.h>" {printf("Header File:  %s\n",yytext);}

"#include <grp.h>"|"#include <ifaddrs.h>"|"#include <netdb.h>" {printf("Header File:  %s\n",yytext);}

"#include <netinet/in.h>"|"#include <netinet/tcp.h>"|"#include <pthread.h>" {printf("Header File:  %s\n",yytext);}

"#include <pwd.h>"|"#include <sched.h>"|"#include <semaphore.h>" {printf("Header File:  %s\n",yytext);}

"#include <sys/ioctl.h>"|"#include <sys/mman.h>"|"#include <sys/msg.h>" {printf("Header File:  %s\n",yytext);}

"#include <sys/resource.h>"|"#include <sys/select.h>"|"#include <sys/socket.h>" {printf("Header File:  %s\n",yytext);}

"#include <sys/stat.h>"|"#include <sys/time.h>"|"#include <sys/epoll.h>" {printf("Header File:  %s\n",yytext);}

"#include <sys/wait.h>"|"#include <termios.h>"|"#include <unistd.h>" {printf("Header File:  %s\n",yytext);}

"#include <sys/types.h>"|"#include <sys/uio.h>"|"#include <sys/un.h>" {printf("Header File:  %s\n",yytext);}

"if"|"else"|"while"|"do"|"switch"|"case"|"for" {printf("Keyword:  %s\n",yytext);}

"char"|"const"|"continue"|"default"|"double"|"enum" {printf("Keyword:  %s\n",yytext);}

"extern"|"float"|"goto"|"int"|"long"|"register"|"return" {printf("Keyword:  %s\n",yytext);}

"short"|"signed"|"sizeof"|"static"|"struct"|"typedef" {printf("Keyword:  %s\n",yytext);}

"break"|"union"|"unsigned"|"void"|"volatile" {printf("Keyword:  %s\n",yytext);}

{identifier}  {idcount++; printf("Identifier:  %s, length of the identifier: %d\n", yytext, yyleng); }

{number} {printf("Number:  %s\n",yytext);}

"+"|"-"|"\*"|"/"|"%"|"="|"=="|"!="|"<"|">" {printf("Arithmetic Operator: %s\n",yytext);}

"<="|">="|"++"|"--"|"&&"|"||"|"!"|"&"|"|" {printf("Arithmetic Operator: %s\n",yytext);}

"(" { printf("Left Parenthesis\n"); }

")" { printf("Right Parenthesis\n"); }

"{" { printf("Left Curly Brace\n"); }

"}" { printf("Right Curly Brace\n"); }

"[" { printf("Left Square Bracket\n"); }

"]" { printf("Right Square Bracket\n"); }

";" { printf("Semicolon\n"); }

":" { printf("Colon\n"); }

"," { printf("Comma\n"); }

"." { printf("Dot\n"); }

"->" { printf("Arrow Operator\n"); }

"..." { printf("Ellipsis\n"); }

"+=" { printf("Addition Assignment Operator\n"); }

"-=" { printf("Subtraction Assignment Operator\n"); }

"\*=" { printf("Multiplication Assignment Operator\n"); }

"/=" { printf("Division Assignment Operator\n"); }

"%=" { printf("Modulus Assignment Operator\n"); }

"&=" { printf("Bitwise AND Assignment Operator\n"); }

"|=" { printf("Bitwise OR Assignment Operator\n"); }

"^=" { printf("Bitwise XOR Assignment Operator\n"); }

"<<=" { printf("Left Shift Assignment Operator\n"); }

">>=" { printf("Right Shift Assignment Operator\n"); }

"?" { printf("Question Mark\n"); }

%%

int yywrap()

{

return 1;

}

int main()

{

yyin= fopen("myfile.txt","r");

printf("Here in file:\n");

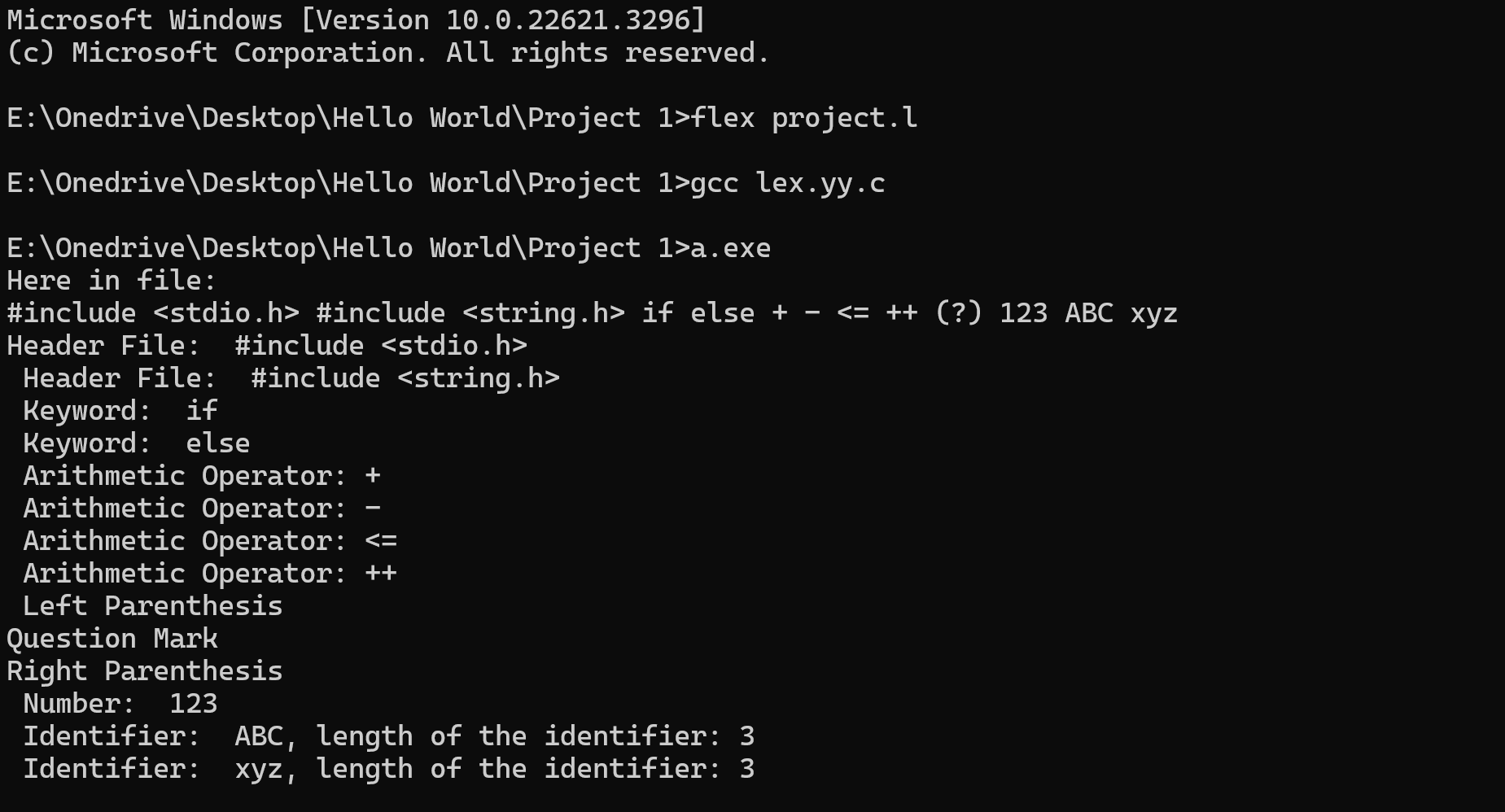
yylex();

printf("Number of id: %d",idcount);

return 0;

}

**6. Screenshot of result**



**7. Special Features**:

a) Header File Detection: Our lexer is capable of detecting and categorizing various standard header files included in C code, providing insights into the dependencies and libraries utilized in the program.

b) Comprehensive Tokenization: The lexer identifies and categorizes a wide range of tokens, including keywords, identifiers, numbers, operators, and header files, providing a detailed analysis of the lexical structure of the input C code.

c) Operator Classification: In addition to basic arithmetic and assignment operators, our lexer classifies logical, relational, and bitwise operators, enhancing the understanding of code logic and behavior.

d) Length of Identifiers: Our lexer calculates and displays the length of each identifier encountered in the C code, facilitating analysis of naming conventions and variable usage patterns.

e) Modular and Extensible: The lexer code is modular and extensible, allowing for easy integration with other tools and systems. Users can customize and extend the lexer functionality to suit specific requirements.

f) Integration with File I/O: Our lexer seamlessly integrates with file input/output operations, enabling users to tokenize C code stored in external files and process large codebases efficiently.

g) Educational Tool: With its detailed tokenization and categorization capabilities, our lexer serves as an educational tool for learning about the lexical analysis phase of compilation and understanding the internal workings of C code parsing.

**8. Limitations**

a) Lack of error recovery may lead to inaccurate results.

b) Limited support for complex preprocessor directives.

c) Challenges with comment handling, especially nested comments.

d) Basic operator classification, may not handle custom scenarios.

e) Limited error reporting may hinder debugging.

f) Performance concerns with large or nested code.

g) Platform dependence affects functionality.

**9. Conclusion**

Our lexer successfully tokenizes C code, providing insights into its lexical structure. It identifies keywords, identifiers, numbers, operators, and header files, facilitating code analysis. However, limitations such as lack of semantic analysis and error recovery pose challenges. Future directions could focus on enhancing error handling, improving preprocessor support, and incorporating semantic analysis for more robust code analysis. Additionally, optimizing performance and ensuring platform independence would enhance the usability of the lexer across diverse environments.