LEXICAL ANALYZER

Create a Scanner of the C-Minus Compiler

Prepared By

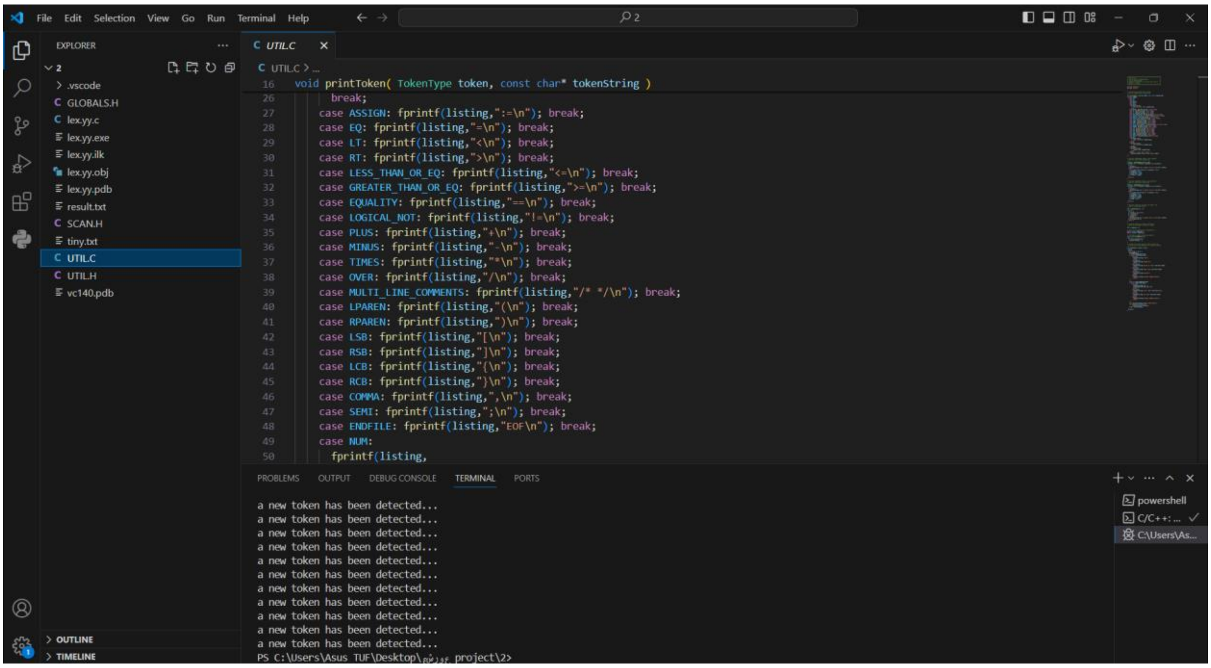
Name: Ahmed Eldesouky

id:200030833

Under Supervision

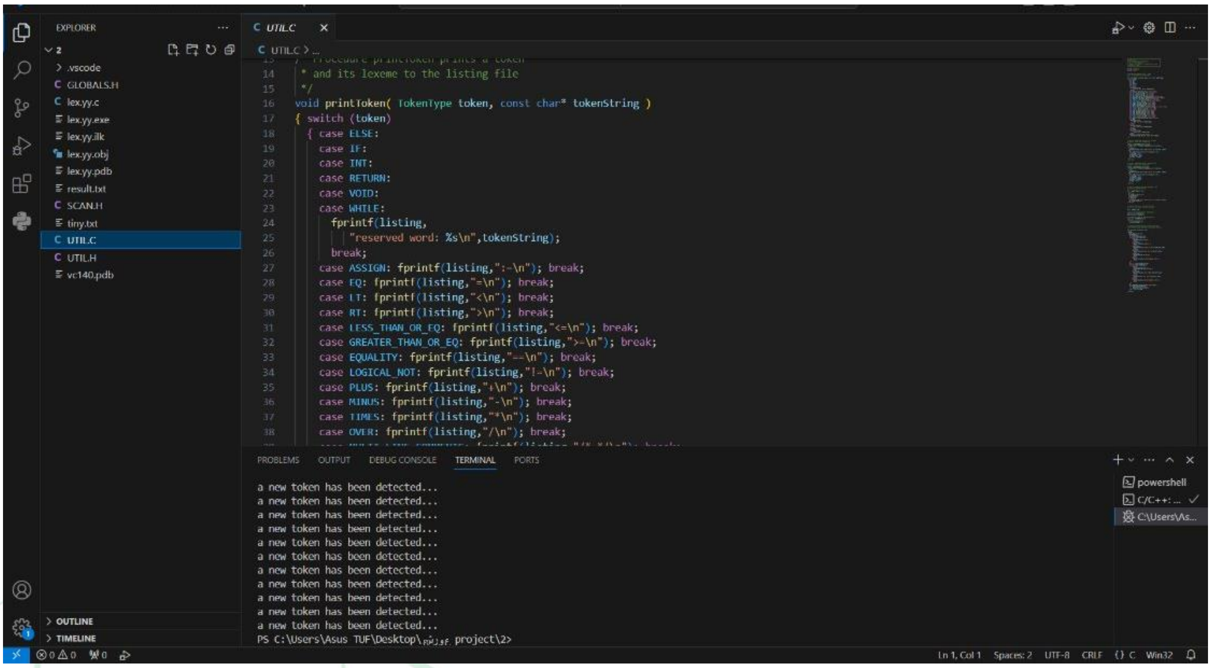
Name of Doctor:Nehal

Name of T. A. :faris



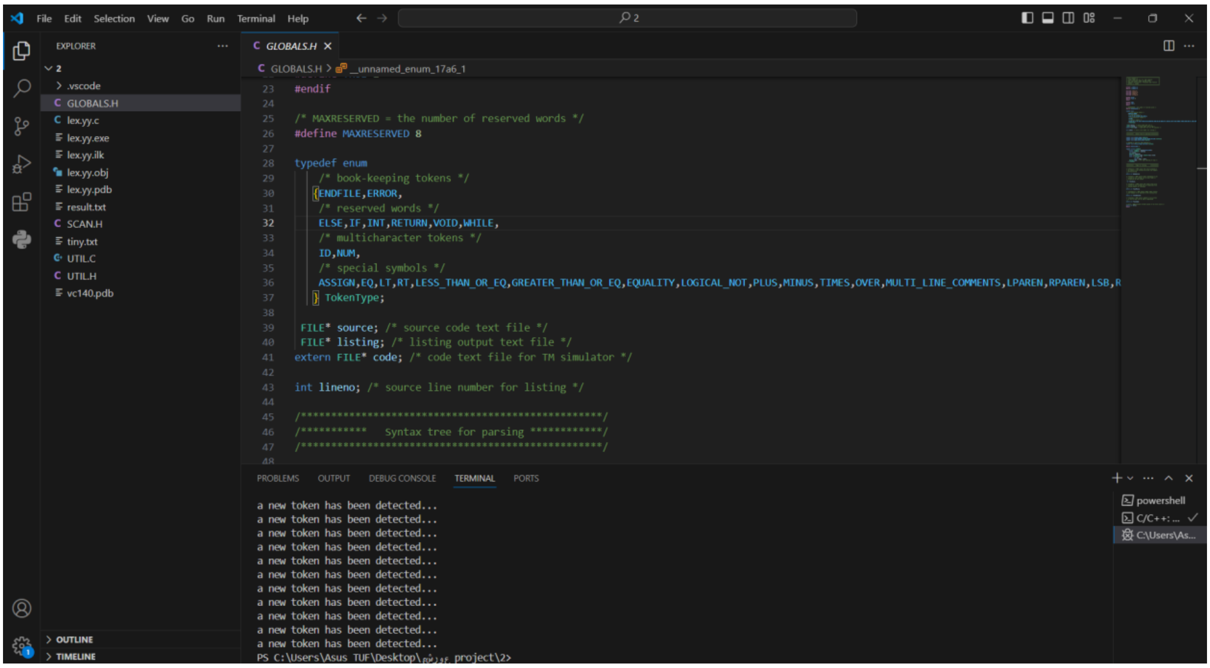
```c
void printToken( TokenType token, const char* tokenString )
      break;
    case ASSIGN: fprintf(listing,":=\n"); break;
    case EQ: fprintf(listing,"=\n"); break;
    case LT: fprintf(listing,"<\n"); break;
    case RT: fprintf(listing,">\n"); break;
    case LESS_THAN_OR_EQ: fprintf(listing,"<=\n"); break;
    case GREATER_THAN_OR_EQ: fprintf(listing,">=\n"); break;
    case EQUALITY: fprintf(listing,"==\n"); break;
    case LOGICAL_NOT: fprintf(listing,"!=\n"); break;
    case PLUS: fprintf(listing,"+\n"); break;
    case MINUS: fprintf(listing,"-\n"); break;
    case TIMES: fprintf(listing,"*\n"); break;
    case OVER: fprintf(listing,"/\n"); break;
    case MULTI_LINE_COMMENTS: fprintf(listing,"/* */\n"); break;
    case LPAREN: fprintf(listing,"(\n"); break;
    case RPAREN: fprintf(listing,")\n"); break;
    case LSB: fprintf(listing,"[\n"); break;
    case RSB: fprintf(listing,"]\n"); break;
    case LCB: fprintf(listing,"{\n"); break;
    case RCB: fprintf(listing,"}\n"); break;
    case COMMA: fprintf(listing,",\n"); break;
    case SEMI: fprintf(listing,";\n"); break;
    case ENDFILE: fprintf(listing,"EOF\n"); break;
    case NUM:
      fprintf(listing,
```

```
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
PS C:\Users\Asus TUF\Desktop\مشروع project\2>
```



```c
/* Procedure printToken prints a token
 * and its lexeme to the listing file
 */
void printToken( TokenType token, const char* tokenString )
{ switch (token)
  { case ELSE:
    case IF:
    case INT:
    case RETURN:
    case VOID:
    case WHILE:
      fprintf(listing,
          "reserved word: %s\n",tokenString);
      break;
    case ASSIGN: fprintf(listing,":=\n"); break;
    case EQ: fprintf(listing,"=\n"); break;
    case LT: fprintf(listing,"<\n"); break;
    case RT: fprintf(listing,">\n"); break;
    case LESS_THAN_OR_EQ: fprintf(listing,"<=\n"); break;
    case GREATER_THAN_OR_EQ: fprintf(listing,">=\n"); break;
    case EQUALITY: fprintf(listing,"==\n"); break;
    case LOGICAL_NOT: fprintf(listing,"!=\n"); break;
    case PLUS: fprintf(listing,"+\n"); break;
    case MINUS: fprintf(listing,"-\n"); break;
    case TIMES: fprintf(listing,"*\n"); break;
    case OVER: fprintf(listing,"/\n"); break;
```

```
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
PS C:\Users\Asus TUF\Desktop\مشروع project\2>
```



```
#endif

/* MAXRESERVED = the number of reserved words */
#define MAXRESERVED 8

typedef enum
    /* book-keeping tokens */
   {ENDFILE,ERROR,
    /* reserved words */
    ELSE,IF,INT,RETURN,VOID,WHILE,
    /* multicharacter tokens */
    ID,NUM,
    /* special symbols */
    ASSIGN,EQ,LT,RT,LESS_THAN_OR_EQ,GREATER_THAN_OR_EQ,EQUALITY,LOGICAL_NOT,PLUS,MINUS,TIMES,OVER,MULTI_LINE_COMMENTS,LPAREN,RPAREN,LSB,R
   } TokenType;

 FILE* source; /* source code text file */
 FILE* listing; /* listing output text file */
extern FILE* code; /* code text file for TM simulator */

int lineno; /* source line number for listing */

/**************************************************/
/***********   Syntax tree for parsing ************/
/**************************************************/
```

```
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
a new token has been detected...
PS C:\Users\Asus TUF\Desktop\مشروع project\2>
```

What is The Project ?

- The idea of the project is to create a scanner so that we can extract the tokens of the source code. This will be the first stage of the lexical parser compiler.

Definition About compiler Design

- Compiler design is an important field in computer science that focuses on developing and understanding how to build and design compilers, which are the tools used to convert source code from a programming language into a form that can be executed by a computer. Here's an introduction to compiler design:

  1. Main goal: The main goal of compiler design is to convert source code from a high-level programming language into a machine language (such as machine language or assembly language) that a computer can understand and execute.

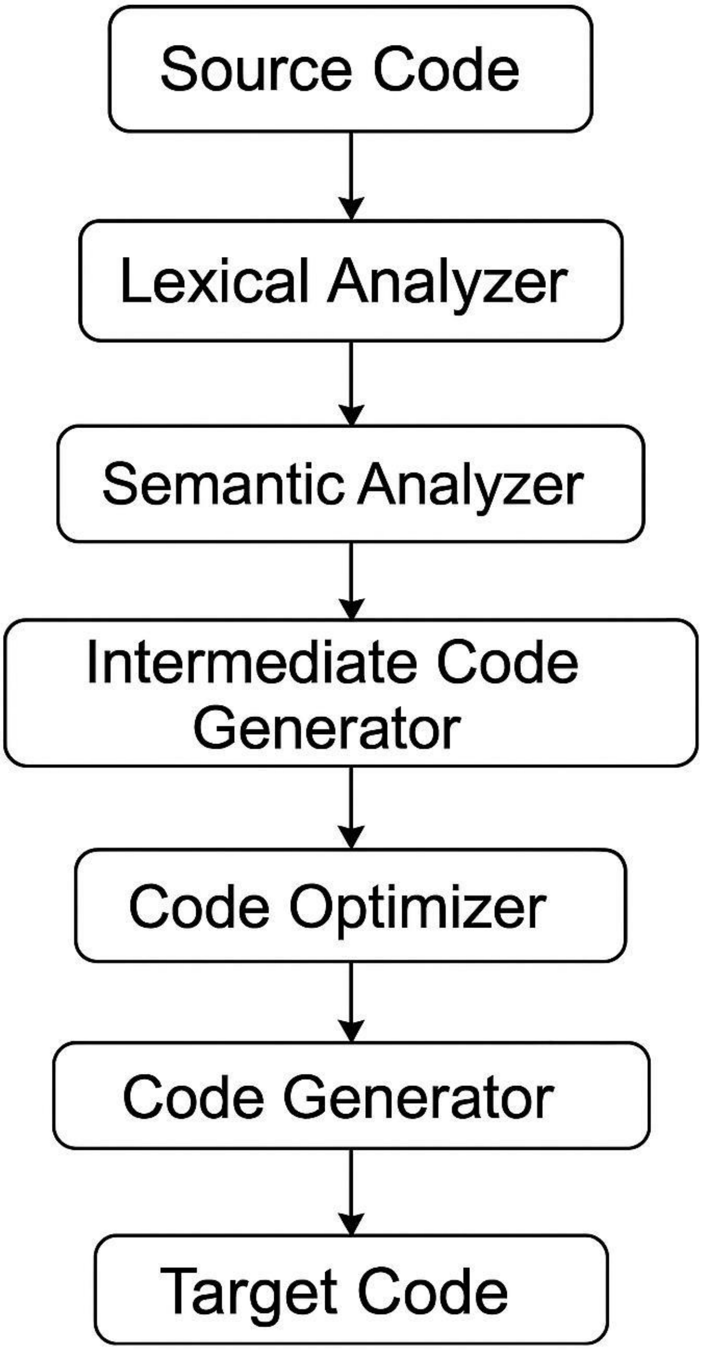
  2. Compiler components: A translator usually consists of three main components:

○ Analyzer: It divides the source code into small units and understands their linguistic structures.

○ Code Generator:It generates machine code or low-level programming code from source code.

○ Optimizer: It improves the generated source code to improve the performance of the resulting program.

Phases of Compiler



Source Code

Lexical Analyzer

Semantic Analyzer

Intermediate Code Generator

Code Optimizer

Code Generator

Target Code

Abstract

This technical report presents the design and implementation of a Lexical Analyzer (Scanner) as a frontend component of a C-Minus Compiler. The lexical analyzer is responsible for reading the source code, breaking it into tokens, and eliminating unnecessary characters such as whitespace and comments. The goal of this project is to develop a working compiler component that performs accurate lexical analysis using modern compiler construction tools and techniques.

Introduction

A compiler is a software tool that translates source code written in a high-level programming language into machine code that can be executed by a computer. The compilation process is typically divided into several phases, each responsible for a specific task. This project focuses on the frontend phases, including lexical, syntax, and semantic analysis.

The first phase, lexical analysis, reads the raw source code and converts it into a stream of tokens. The tokens represent basic elements such as identifiers, keywords, operators, and symbols. This token stream is then passed to the syntax analyzer to check the grammatical structure of the code.

In this report, we describe the development of a lexical analyzer for the C-Minus language, which is a simplified C-like programming language used for teaching compiler design. The report also discusses other frontend components of the compiler such as the syntax and semantic analyzers.

Description of Frontend of the Compiler

The frontend of the compiler includes the lexical analyzer, syntax analyzer, and semantic analyzer. These components work together to ensure that the source code adheres to the rules of the programming language before proceeding to code generation and optimization.

- **Lexical Analyzer: Converts source code into tokens.**
- **Syntax Analyzer: Checks if tokens form valid syntax structures.**
- **Semantic Analyzer: Ensures the logical correctness of the code (e.g., variable declarations).**

Lexical Analyzer

The lexical analyzer, also known as a scanner, processes the input source code and converts it into a series of tokens. Each token represents a meaningful element such as a keyword, identifier, operator, or delimiter. The scanner also removes comments and white spaces. The implementation involves writing regular expressions and a finite automaton that recognizes token patterns.

Syntax Analyzer

The syntax analyzer, or parser, receives the token stream from the lexical analyzer and verifies whether the tokens follow the grammar of the language. The parser constructs a syntax tree that represents the grammatical structure of the source code.

Semantic Analyzer

The semantic analyzer checks for semantic errors in the source code. This includes ensuring variables are declared before use, type compatibility, and proper use of control structures. It annotates the syntax tree with type and scope information.

Programming Language

The language used in this project is C-Minus, a small subset of the C programming language designed for compiler education. It includes basic programming constructs like variables, conditionals, loops, and functions.

List of Reserved Words

Reserved words in C-Minus include: if, else, int, return, void, while.

List of Keywords

Keywords are predefined, reserved identifiers that have special meanings in the language, such as: if, else, int, void, return, while.

Basic Grammar Rules

Basic grammar rules define the structure of valid statements and expressions in the C-Minus language. For example:

- Expression -> Expression + Term | Term
- Term -> Term * Factor | Factor
- Factor -> (Expression) | ID | NUM

Software Tool(s)

Tools used include:

- JFlex (for generating lexical analyzers)
- CUP (for generating parsers)
- Java (as the implementation language)

Discussion

The project demonstrates the process of building a compiler frontend. By constructing the lexical and syntax analyzers, we gain a deeper understanding of how compilers interpret source code and enforce programming language rules.

Input of Lexical Analyzer

The input is a text file containing C-Minus source code. It includes program elements such as variable declarations, expressions, and control structures.

Output of Lexical Analyzer

The output is a stream of tokens, each representing an element of the source code. For example, a statement like `int x = 5;` would produce tokens: [INT, ID(x), ASSIGN, NUM(5), SEMICOLON].

Conclusion

This report outlines the steps involved in constructing the lexical analyzer of a C-Minus compiler.

Through this project, we have learned how to apply formal language theory and tools like JFlex and CUP to build real-world compiler components.

References

1. **Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools. Pearson.**

2. **JFlex and CUP official documentation.**

Appendices

Appendix A: Sample C-Minus Program

Appendix B: Output Token Stream