**Misr University for Science & Technology**

**College of Information Technology**

**CS 352 Compiler Project**

**Team Members**

**Adam Ashraf – 200019512**

**Ahmed Khaled – 200012564**

**Khaled Kamr – 200013171**

**Ali Mohamed – 200012168**

**Abdlrahman Ahmed - 200011829**

**Supervised by:**

**Dr/ Ehab Elsalamony**

**Ta/Toka Abdelhamid**

**Summary**

In this project a language called C minus is derived from the Tiny language. Then the lexical analyzer prints out the tokens of this derived language.

**Introduction to the Compiler**

A compiler is a program that translates source code written in a high-level programming language into a lower-level language, often machine code, which can be executed by a computer's processor.

**Phases of the Compiler**

**Lexical Analysis (Scanning):** This initial phase involves parsing the source code into a series of tokens, which are the smallest meaningful units such as keywords, identifiers, and symbols. A lexer or scanner is responsible for this task.

**Syntax Analysis (Parsing):** Following lexical analysis, the syntax analysis phase checks the validity of the token sequence against the language's grammar rules. It constructs a parse tree or abstract syntax tree (AST) representing the program's structure. Parsing is conducted by the parser.

**Semantic Analysis:** Once the syntax is verified, semantic analysis ensures the correctness of statements based on the language's semantics. This includes tasks like type checking, scope resolution, and creating symbol tables to manage identifiers and their properties.

**Intermediate Code Generation:** Here, the compiler produces an intermediate representation (IR) of the source code. This IR is platform-independent and serves as a common ground for subsequent optimizations and target platform translations.

**Optimization:** Optimization techniques are applied to enhance the efficiency of the intermediate code. These techniques range from simple transformations to more complex algorithms like loop optimization and data-flow analysis.

**Code Generation:** In this phase, the compiler translates the optimized intermediate code into machine code or assembly instructions tailored to the target hardware. It maps high-level language constructs to low-level instructions, considering factors like register allocation and instruction scheduling.

**Software Tools**

Compiler development often relies on a variety of software tools to assist in different stages of the compilation process. These tools aid in tasks such as parsing, optimization, code generation, debugging, and performance analysis. Here are some of the tools listed below.

Lex and Yacc (or Flex and Bison):

Lex (or Flex) and Yacc (or Bison) constitute classic tool combinations for lexical analysis (scanning) and syntax analysis (parsing), respectively. Lex generates lexical analyzers based on regular expressions, while Yacc generates parsers based on context-free grammars. These tools are commonly employed for generating the scanner and parser components of a compiler.

ANTLR (ANother Tool for Language Recognition):

ANTLR stands out as a potent parser generator that supports various programming languages and facilitates the creation of

parsers for intricate grammars. It finds extensive use in compiler construction and language development endeavors.

LLVM (Low-Level Virtual Machine):

LLVM serves as a compiler infrastructure offering a collection of modular and reusable compiler and toolchain components. It encompasses a compiler front-end for parsing and optimizing source code, a middle-end featuring diverse optimization passes, and a back-end for generating machine code across multiple target architectures. LLVM finds wide application in both academic research and industrial compiler development.

GCC (GNU Compiler Collection):

GCC emerges as a popular open-source compiler suite supporting multiple programming languages, such as C, C++, and Fortran. It encompasses front-ends for parsing source code written in these languages, alongside a comprehensive array of optimization and code generation tools.

Clang:

Clang, built upon LLVM, represents a C/C++ compiler delivering swift and precise compilation with expressive diagnostics. It frequently serves as a front-end in various toolchains and integrated development environments (IDEs).

Valgrind:

Valgrind stands as a potent debugging and profiling tool aiding in the detection of memory leaks, buffer overflows, and other memory-related errors in compiled programs. It furnishes a suite of tools, including Memcheck for memory debugging and Callgrind for profiling.

GDB (GNU Debugger):

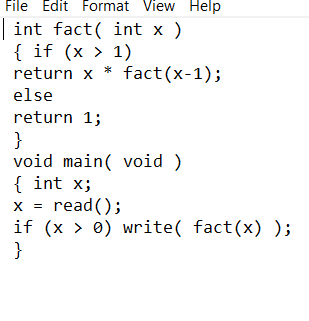
GDB stands as a widely utilized debugger for debugging compiled programs. It empowers developers to scrutinize and manipulate program execution, set breakpoints, inspect variables, and trace program flow.

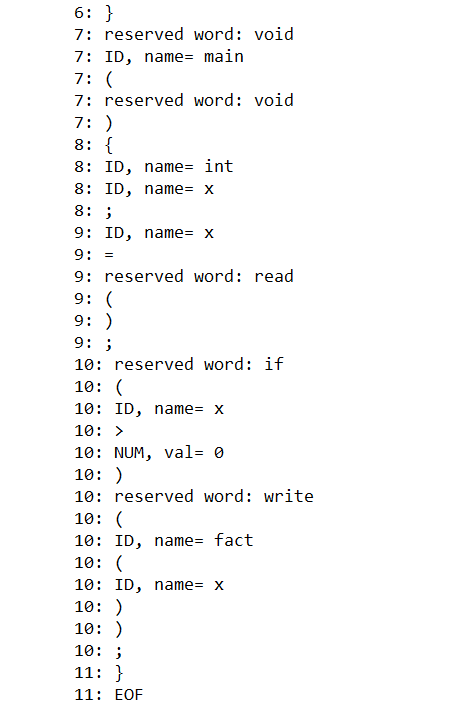
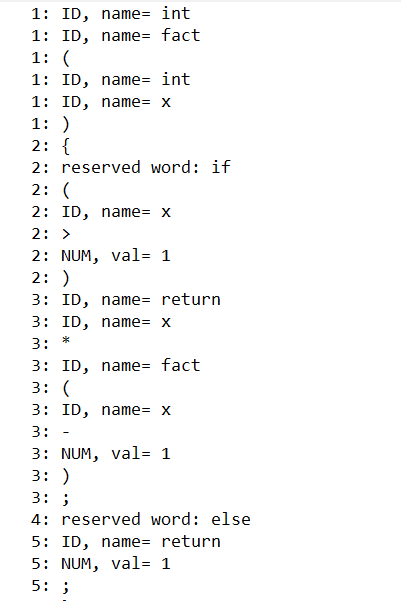
Perf:

Perf constitutes a performance analysis tool tailored for Linux systems, furnishing insights into program behavior, CPU usage, cache misses, and other performance metrics. It facilitates code optimization and identification of performance bottlenecks in compiled programs.

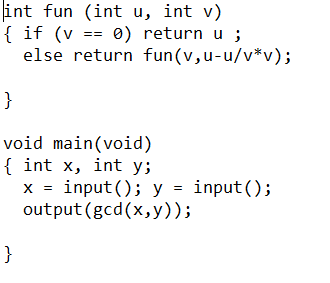
**Code**

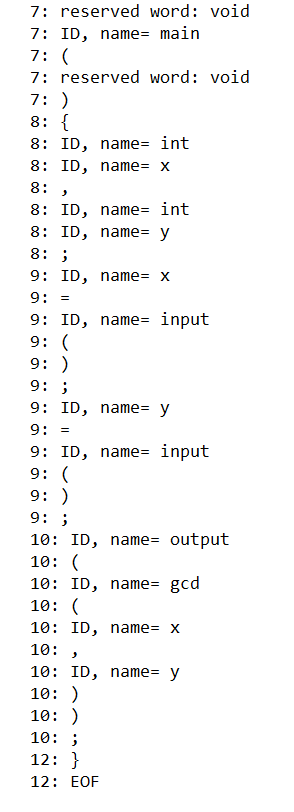
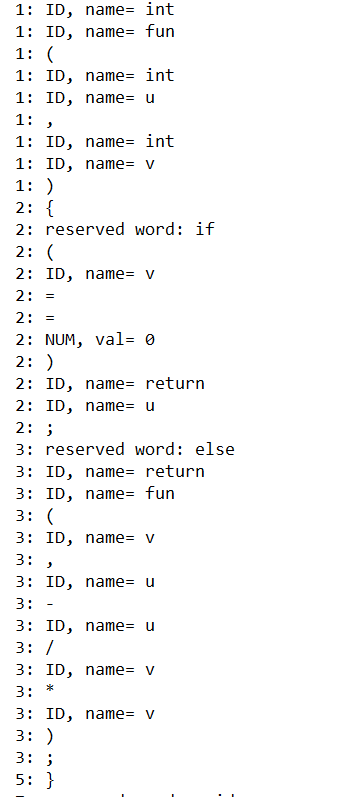
**Example 1:**





**Example 2:**





**Conclusion**

After finishing this project we now have a better understanding of how the compiler works and how the tokens are printed during the Lexical Analysis.

**References**

Compiler Construction: Principles and Practice

Kenneth C. Louden