Team Members :

Ahmed refaat 200017694

Yousef Atef 200016657

Ziad Abdelwahab 200013252

Mohamed Elshaer 200019118

Mostafa Arafa 200020784

Supervised By :

DR. Nehal A.Mohamed

Eng. Toka Ebrahim

Summary

A compiler is a software tool that translates human-readable source code written in a programming language into machine-readable code, typically in the form of binary instructions that a computer can execute. The compilation process involves several stages, starting with lexical analysis and ending with code generation and linking.

The lexical analyzer, often referred to as the laxer or scanner, is the initial phase of a compiler. Its primary task is to break down the source code into meaningful tokens. These tokens are the basic building blocks of the language and include keywords, identifiers, literals, and operators. The lexical analyzer ignores whitespace and comments, focusing solely on extracting tokens.

Introduction

Compilers are essential software tools that translate high-level programming languages into machine-readable code, enabling computers to execute programs. By undergoing a series of phases like lexical analysis, syntax parsing, semantic analysis, optimization, and code generation, compilers convert abstract code structures into efficient machine code. This succinct introduction aims to highlight the critical role and intricate processes involved in compilers, essential for software development across various computing platforms.

1.1. Phases of Compiler

1. **Lexical Analysis (Scanning):** This phase involves breaking the source code into tokens. Tokens are the smallest meaningful units in a programming language, such as keywords, identifiers, operators, etc. The output of this phase is a stream of tokens.
2. **Syntax Analysis (Parsing):** This phase involves analyzing the structure of the source code based on the rules of the programming language's grammar. It

creates a parse tree or syntax tree representing the syntactic structure of the program.

3. **Semantic Analysis:** This phase checks the source code for semantic correctness. It ensures that the program follows the rules of the language and detects any semantic errors. This includes type checking, scope resolution, and other checks.

4. **Intermediate Code Generation:** In this phase, the compiler translates the source code into an intermediate representation. This intermediate code is typically closer to the machine language but is still platform-independent. It simplifies subsequent optimization passes.

5. **Code Optimization:** This phase involves improving the intermediate code to make it more efficient. Optimization techniques aim to reduce execution time, memory usage, or both, without changing the program's functionality. Various optimizations include constant folding, loop optimization, and function in lining.

6. **Code Generation:** This phase translates the optimized intermediate code into the target machine language or bytecode. The output is typically in the form of assembly code or

object code, depending on whether the compiler generates code for a specific machine architecture or a virtual machine.

7. Lexical Analyzer

A lexical analyzer, is a program component that breaks down a source code file into a sequence of tokens. It is the first phase of a compiler or interpreter, and its main task is to read the input source code and produce a stream of tokens that can be processed by the next phase of the compiler.
The lexical analyzer scans the source code file character by character, grouping them into meaningful units called tokens. These tokens represent the basic building blocks of the programming language, such as keywords, identifiers, operators, and literals. The lexical also discards any comments or whitespace characters that are not relevant to the program's execution.
The output of the lexical analyzer is a stream of tokens that are passed on to the next phase of the compiler, which is usually the parser. The parser then uses these tokens to build a parse tree, which represents the syntactic structure of the program.

8. Software Tools

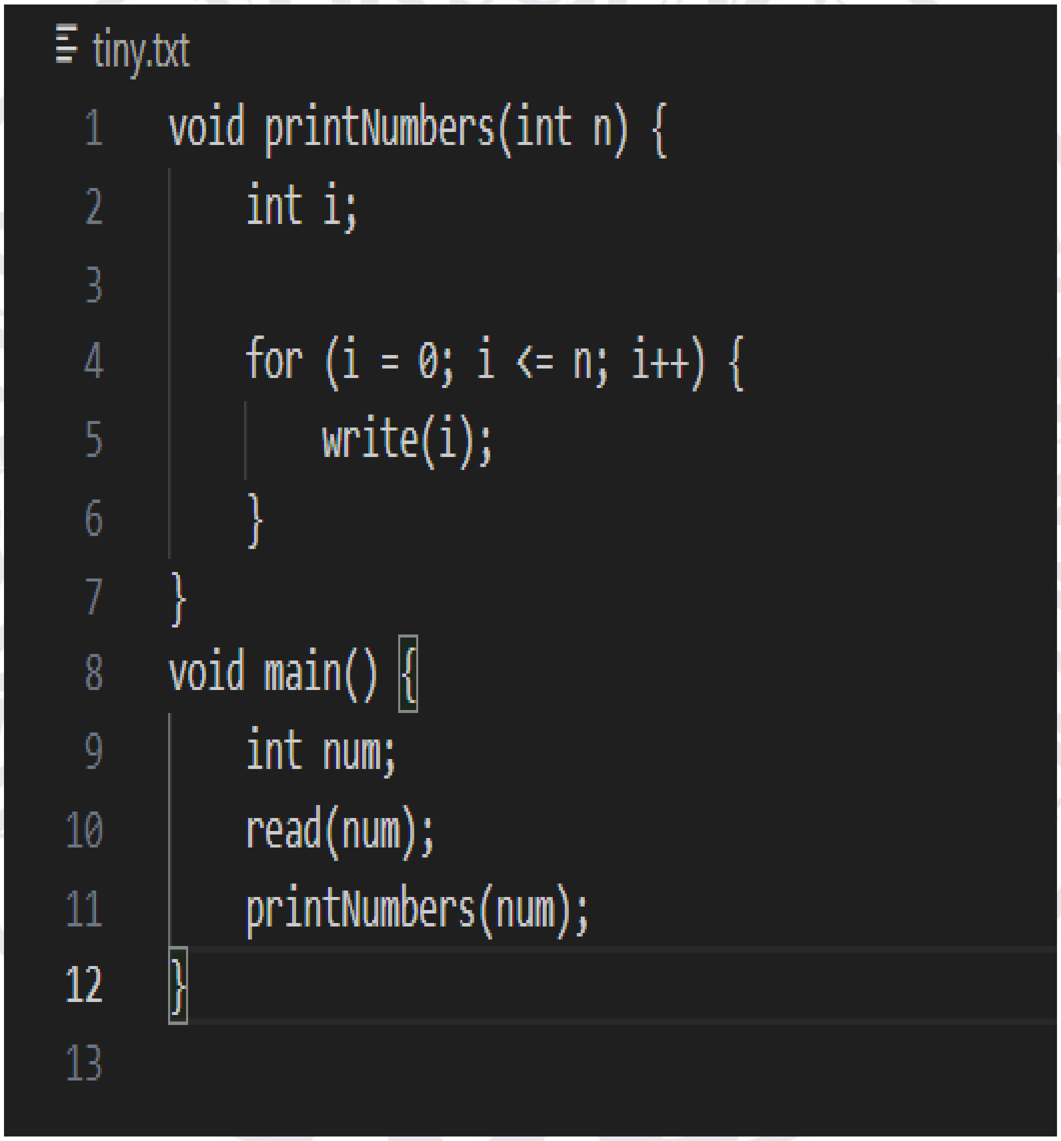
- **TINY compiler:** is a simple programming language designed for educational purposes. It is often used in compiler construction courses to demonstrate the implementation of basic compiler components.
  The TINY compiler serves as a learning tool for understanding the fundamentals of compiler design and implementation. It typically involves building a compiler that translates TINY source code into machine code or an intermediate representation.

- **Flex Software:** is a powerful tool for generating lexical analyzers (scanners) for programming languages. It is often used alongside Bison (a parser generator) in the development of compilers.
  Flex simplifies the process of creating lexical analyzers by allowing developers to specify patterns for token recognition using regular expressions. These patterns are then compiled into efficient C code for use within a compiler.

- **C-Minus Language:**
  Is a simple subset of the C programming language, designed for educational purposes It includes a minimal set of features found in C, making it suitable for teaching compiler construction and systems programming concepts. C-Minus serves as a practical vehicle for teaching compiler design and implementation. Its simplicity allows students to focus on core compiler concepts without being overwhelmed by the complexities of a full-fledged language like C.

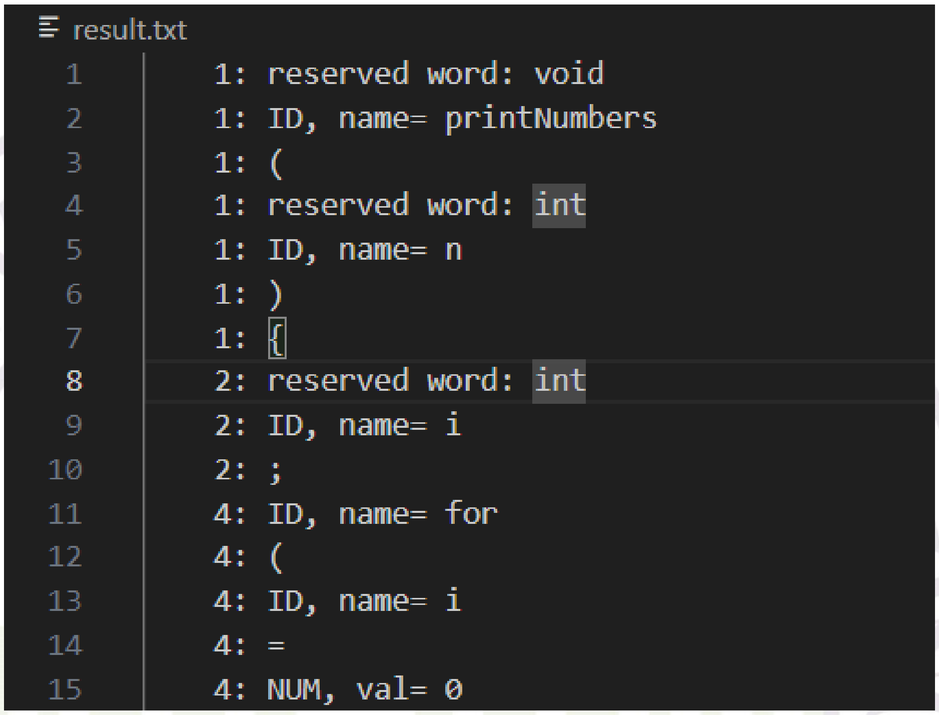
Screenshots of input and output:

Input :


tiny.txt

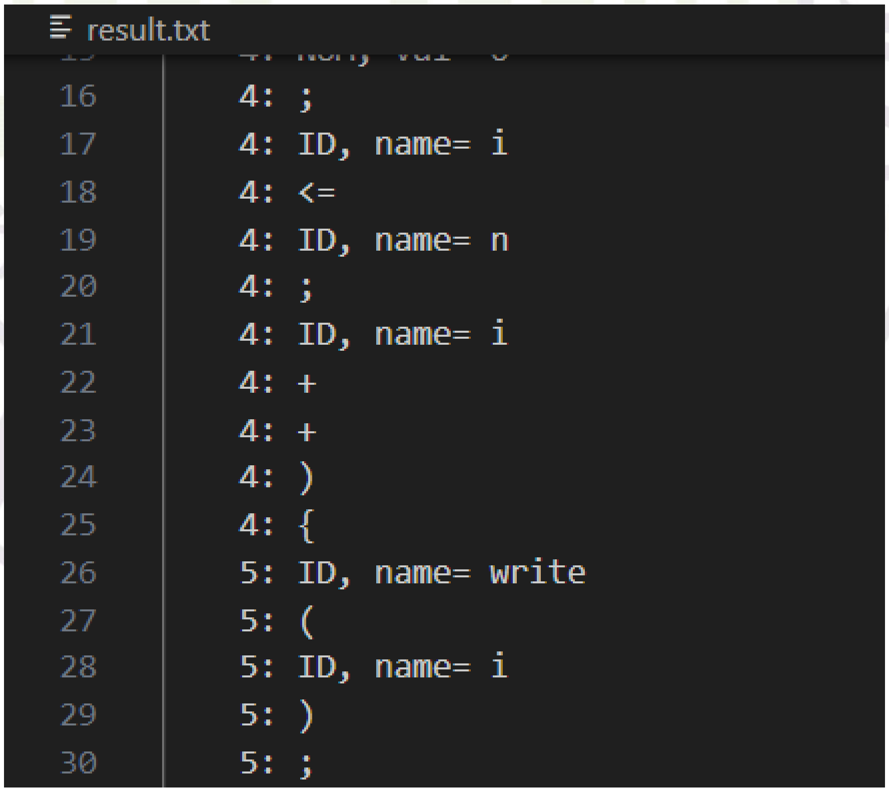
```
void printNumbers(int n) {
    int i;

    for (i = 0; i <= n; i++) {
        write(i);
    }
}
void main() {
    int num;
    read(num);
    printNumbers(num);
}

```

Output :



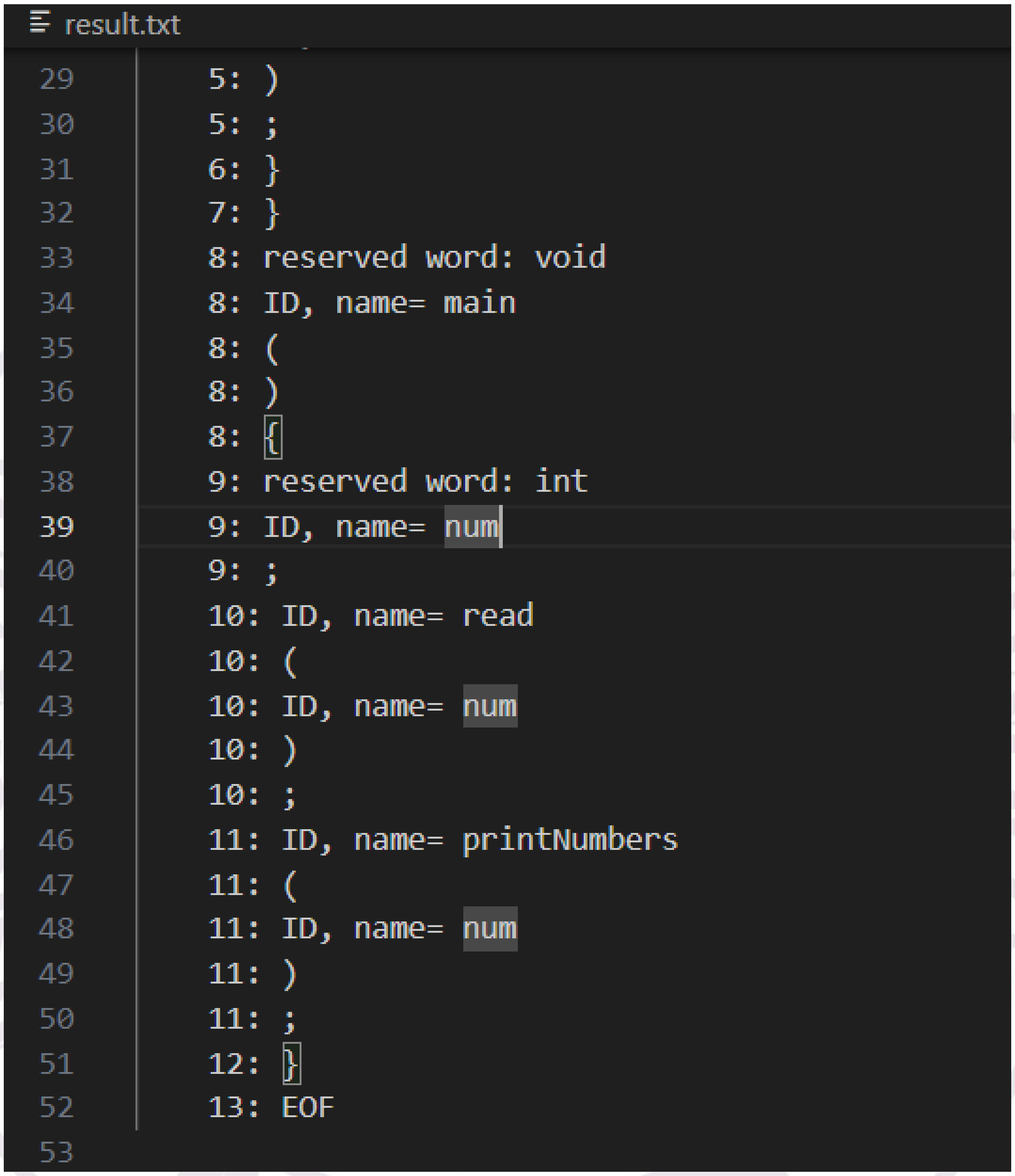
result.txt

```
1: reserved word: void
1: ID, name= printNumbers
1: (
1: reserved word: int
1: ID, name= n
1: )
1: {
2: reserved word: int
2: ID, name= i
2: ;
4: ID, name= for
4: (
4: ID, name= i
4: =
4: NUM, val= 0
```



result.txt

```
4: ;
4: ID, name= i
4: <=
4: ID, name= n
4: ;
4: ID, name= i
4: +
4: +
4: )
4: {
5: ID, name= write
5: (
5: ID, name= i
5: )
5: ;
```



result.txt

```
5: )
5: ;
6: }
7: }
8: reserved word: void
8: ID, name= main
8: (
8: )
8: {
9: reserved word: int
9: ID, name= num
9: ;
10: ID, name= read
10: (
10: ID, name= num
10: )
10: ;
11: ID, name= printNumbers
11: (
11: ID, name= num
11: )
11: ;
12: }
13: EOF
```

9. Conclusion

compilers and lexical analyzers are essential for software development, translating high-level code into machine-readable instructions. Compilers meticulously process code through various stages, ensuring efficient execution across platforms. Meanwhile, lexical analyzers break down source code into tokens, facilitating this process. Together, they enable programmers to express complex logic, driving innovation in the digital sphere.

10. References

- Stackoverflow
- geeksforgeeks
- ChatGPT

11. Appendices

- Compiler Constructions (Kenneth c. Louden)
- Compilers Principles & techniques and Tools.

HANK
YOU