Universidad Nacional Autónoma de México
Faculty of Engineering

Computer Engineering.

Compilers

LEXER

STUDENT:
320254536
315293728
423108596
320246881
320250648

Group:
5
Semester:
2026-I

México, CDMX. September 2025

Contents

1 Introduction

Throughout this project, we will develop a lexical analyzer for the Go programming language using Python. The development will be done by applying the concepts learned theoretically such as context-free grammar, lexemes, tokens using tools like regular expressions, and Python libraries optimized for compiler creation.

The main objective of this project is to take string inputs from a source program in Go, and be able to categorize each one of them into a token successfully, as well as output a valid token stream for the next compiler phases with the correct count of tokens identified in different example files. Building a correct and maintainable lexer is the key for any subsequent compiler phase. In practice, many parsing errors or ambiguous mistakes are made due to imprecise token boundaries, mishandled whitespace or comments, or numeric forms. When designing a new language, or designing a compiler for an existing language, the first task is to define precisely what characters are permitted in each category of token [1]. The tokens produced by the lexical analyzer essentially are the building blocks of our program. In the next steps of compilation, we are going to validate the correct program structure using these building blocks, so the effectiveness of our compiler will depend directly on the correct work of the lexical analyzer.
Recalling the main objective of the compiler course, which is to apply techniques and tools for the development of interpreters, compilers, and translators in general, this work seeks to carry out the first stage of a compiler.

The implementation targets the usual token categories generally presented in compiler texts: keywords, identifiers, literals, operators, and punctuation—while adopting Go-specific nuances that affect tokenization. The lexer runs once over the input and, whenever it finds something it recognizes, it creates a token with its type and the exact text. To check that everything works, we will test the lexer with short Go examples that cover common cases. The rules are organized so they're easy to extend later.

This project also serves a second objective, which is to become familiar with the Go programming language syntax, structure, and what sets it apart from other programming languages.

2 Theoretical Framework

The implementation of the project is founded on the principles of lexical analysis, specifically the use of formal grammars, regular expressions, and the finite state machine model. These concepts are fundamental to translating a high-level language into a stream of recognizable tokens, which is the initial step of the compilation process.

2.1 Lexical Analysis and Tokenization

As we said, the lexical analysis is the first phase of the compiler [2]. The main task of the lexical analyzer is to read the input characters (character stream) of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program [3].

A **token** is a pair consisting of a token name and an optional attribute value, and it represents a class of lexemes. The token names are the input symbols that the parser processes [3]. Most languages will have tokens in these categories [1]:

1. **Keywords** are words of the language structure, like *if*, *return* or *while*. They have to be chosen wisely to reflect the natural structure of the language, because they can't be use in names of variables or othe identifiers.

2. **Identifiers** are the names of variable, functions, classes or other code elements that the programmer can chose. Typically, identifiers are arbitrary sequences of letters and in some cases, numbers.

3. **Numbers or Constants** are numbers formatted as integers, floating point values, fractions, binary, octal or hexadecimal. Is important that echa format is clearly distingued, to avoid confusions for the programmer.
4. **Strings or Literals** are literal character sequences that must be clearly distinguished from keywords or identifiers. These are normally quoted with single or double quotes.
5. **Comments and whitespace** albeit not a token, the lexer should identify them in order to ignore them.

Another important function of the lexical analyzer is to remove comments and whitespaces from the source code, as these elements are not relevant to the syntantic or semantic structure of the program.

2.2 The Finite State Machine Model

Lexical analyzers are typically implemented using regular expressions to define the patterns for tokens. A **regular expression** is a concise notation for specifying a set of strings, such as the pattern for an identifier or a number. These regular expressions can be systematically converted into finite automata, which is an abstract machine that recognize regular languages. There are two principal models used:

- **Nondeterministic Finite Automata (NFA):** has no restrictions on the labels of their edges. A symbol may label several transitions from the same state, and ε (the empty string) is also allowed as a label [3].
- **Deterministic Finite Automata (DFA):** for each state and input symbol, there is at most one possible transition, making them suitable for efficient implementation in lexical analyzers [3].

Both finite automatas are capable of recognizing the same languages. In fact these languages are exactly the same, called the regular languages, that regular expressions can describe[3].

2.3 Context-Free Grammars (CFG) and Parsing Concepts

Once the lexical analyzer has produced a stream of tokens, the next step of compilation phase is syntax analysis, which checks whether the sequence of tokens forms a valid sentence in the programming language. The structure of programming language is formally described using **Context-Free Grammar**.

A Context-Free Grammar consists of terminals, nonterminals, a start symbol, and productions:

- **Terminals**: are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name. We also assume that the terminals are the first components of the tokens output by the lexical analyzer [3].
- **Nonterminals**: are syntactic variables that denote sets of strings. Represent syntactic categories such as expressions, statements, or declarations [3].
- **Start symbol**: represents the root of all valid derivations
- **Productions**: specify the manner in which the terminals and nonterminals can be combined to form strings [3]. Each production consists of:
  - A nonterminal called the head or left side of the production; this production defines some of the strings denoted by the head [3].
  - The symbol $\rightarrow$ [3].

- A body or right side consisting of zero or more terminals and non-terminals. The components of the body describe one way in which strings of the non-terminal at the head can be constructed [3].

CFGs are powerful enough to capture nested structures, for example, balanced parentheses, nested conditional statements, and recursive function calls. which cannot be described by regular expressions alone. In order for a grammar to be suitable for certain types of parsers, like top-down predictive parser, it needs to avoid **ambiguity** and **nondeterminism**. Two important transformations are used to achive this: eliminating left recursion and applying left factoring [4].

2.4 Eliminating Left Recursion

A grammar is left recursive if it has a nonterminal A such that there is a derivation A $\Longrightarrow$ Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion [3]. A simple example is:

$$A \to A\alpha \mid \beta$$

The solution is to transform the grammar to remove left recursion while preserving the language it defines. The general method replaces productions of the form

$$A \to A\alpha_1 \mid A\alpha_2 \mid \beta$$

(where β does not begin with A) with an equivalent grammar:

$$A \to \beta A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \epsilon$$

This new form is right recursive, which is suitable for top-down parsers. It allows the parser to consume input tokens before making recursive calls. Thus, eliminating left recursion is an essential preprocessing step in grammar design for compilers.

2.5 Left Factoring

This is a grammar transformation technique used to eliminate non-determinism. As we now non-determinism occurs when a non-terminal has multiple rules that share a common prefix. By applying Left Factoring, we can remove ambiguity in rule selection, creating a unique and deterministic path for the analyzer [3]. In general, if $A \to \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$. However, we may defer the decision by expanding A to $\alpha A'$ Then, after seeing the input derived from α, we expand A' to β_1 or to β_2 citedave2012compilers. That is, left-factored, the original productions become

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2$$

2.6 Go Language

Go, often referred to as Golang, is a general-purpose language designed for programming systems, its philosophy is based on minimal syntax, fast compilation, and concurrency over parallelism. It is an open source project to make programmers more productive [5]. This language has rapidly ascended since its inception by Google in 2009 because of its simplicity and efficiency [6].

The Go notation uses a variant of EBNF: a Production is defined as *name = [Expression] "."*, where an expression is one or more terms separated by |; a term is one or more factors; a factor can be a production name, a token, an option *[...]* meaning 0 or 1 occurrence, or a repetition ... meaning 0 to many occurrences [5].

The tokens from the Go language are four classes: identifiers, keywords, operators, punctuation, and literals. White space, horizontal tabs, carriage returns, and newlines are ignored except if they combine into a single token. Also, Go uses semicolons ";" as terminators in a number of productions.

3 Development

3.1 Project structure and general functionality

The project is written in python and has the following structure.

```
Lexer
 └─ src
     └─ Lexer.py
└─ tests
 ├─ fib.go
 ├─ fail.go
 ├─ helloworld.go
 └─ point.go
├─ main.py
└─ README.MD
```

The lexing is done via RPLY, a python library that is a pure python lexer and parser generator.[7].

The Lexer class implements the LexerGenerator class from rply, on initialization it creates a dictionary to store tokens, starts a token count and adds the rules for token recognition to the lexer. To define the tokens we use the method lexer.add(name, rule), name is a string that defines the type of token that is being detected by the rule, rule is a regular expression used to match the input string to a token type. rply matches the inputs greedily so the rules must be defined in the correct order to assign the input strings to the correct token type, the tokens for our implementation are defined in this order:

- Keywords
- Data types
- Literals
- Operators (multi character followed by single character)
- Punctuation / Delimiters
- Identifiers

To finalize the rule definition process we add the ignore rules to discard comments(single line and block) and whitespace.

main.py receives the source program's file name as input (either as a command line argument or specified during the execution of the program) and creates an instance of the Lexer class, it initiates the lexing process and handles any errors raised during its execution.

3.2 Tokens

There are six classes of tokens in the program: identifiers, types, keywords, operators, punctuation, and literals; the latter has two sub-classifications: a numerical literal and a string literal.

3.2.1 Identifiers

For identifiers, the first character must be a letter, which can be uppercase or lowercase. After that, you can use letters, digits, or underscores.

3.2.2 Types

The following types are supported:

```
int  float32  float64  bool string
```

3.2.3 Keywords

The following words are reserved by the language and cannot be used as identifiers:

```
break     default      func      interface   select
case      defer        go        map         struct
chan      else         goto      package     switch
const     fallthrough  if      range       type
continue  for          import    return      var
```

3.2.4 Operators

The operators are defined as:

```
+=  ++  -=  --  <-  *=  /=  %=  &=  &&  ^=
&^=   &^   |=   ||   ^=   <<=   >>=   ==
!=    <=   >=   :=   ...
+     -    *    /    %    &     |     ^
<<    >>   =    <    >    !     ~     .
```

3.2.5 Punctuation

The punctuation symbols are:

```
(   )   [   ]   {   }   ,   ;   :
```

3.2.6 Literals

```
    LIT_FLOAT   LIT_INT   LIT_STR   LIT_BOOL
```

3.3 Output stream

For the valid token stream we chose two options that we will use depending on which one will integrate better with the next phase of the compiler: in the first option we load it to memory, inside a `defaultdict` structure and on the second option it is loaded to a `.txt` file.

To classify and store all the tokens we use a hash table implemented as Python dictionary, as an attribute of the Lexer class. It creates a dictionary where each key is a token category, and each value is a list containing all tokens found for each category. As the lexer processes the source code, each token is added to the appropriate list in this dictionary using the `categorize_token` method.

After all the tokens are processed, we add the method `save_tokens_to_file`, which writes the contents of the dictionary into a `.txt` file. For each category, it outputs all tokens found, including repetitions, and the total number of tokens. This allow us to keep a complete record of all tokens and their classifications for later analysis. This file is rewritten with every execution of the main python file.

3.4 Error Handling

From the library implementation, when one error is found the token processing stops, and it has attributes which work for extracting the line and column it was found in, which we use to print its location and specify which token was the invalid one so the user can make the corrections. We didn't implement a more robust error handling method like panic-recovery mode.

3.5 Context Free Grammar

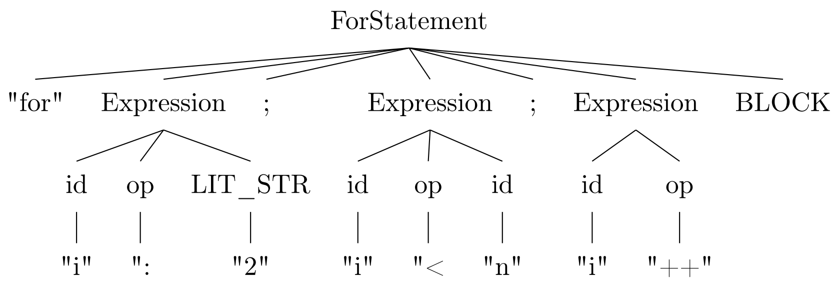
Context free Grammar Fibonnaci;
grammar Fibonnaci;
primary_expression

```
        : Keyword
        | Types
        | Literals
        | Operators
        | Punctuation
        | Identifier
        ;
Keyword
        : Package
        | import
        | func
        | if
        | return
        | var
        | for
        ;
Types
        : int
        ;
Literals
        : LIT_STR
        | LIT_INT
        ;
LIT_STR
        : "fmt"
        ; LIT_INT
        : 1
        | 2
        | 9
        ;
Operators
        : <=
        | =
        | :=
        | ++
        | +
        ;
Punctuation
        : {
        | }
        | (
        | )
        | .
        ;
Identifier
        : main
        | fibonnaciIterative
        | n
        |n1
        |n2
        ;
Block
        :"{"[Statement|,]* "}"
Statement
```

```
        :IfStatement
        |ForStatement
        |ReturnStatement
        |FunctionDecl
        |Expression
        |Call
        ;
IfStatement
        :"if" Expression Block
ForStatement
        :"for" Expression ";" Expression ";" Expression Block
ReturnStatement
         "return" Identifier.
FunctionDecl
        "func" Identifier Parameters Type [ Block ] .
Expression
        : [Identifier|LIT_INT] operator [Identifier|LIT_INT]
        : [Identifier|LIT_INT] operator
;
Package
        : package identifier
        ;
Call
        :Identifier "(" Parameters ")"
        ;
Parameters
        : identifier
        | LIT_STR
        | LIT_INT
        | CALL
        ;
```

3.6 CFG

Parse tree of the for statement in fib.go



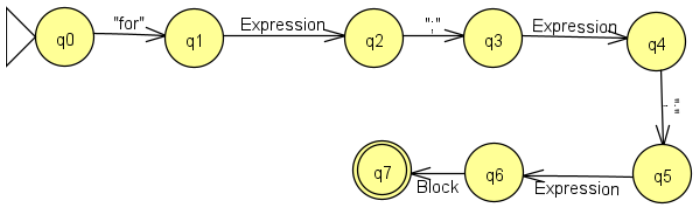
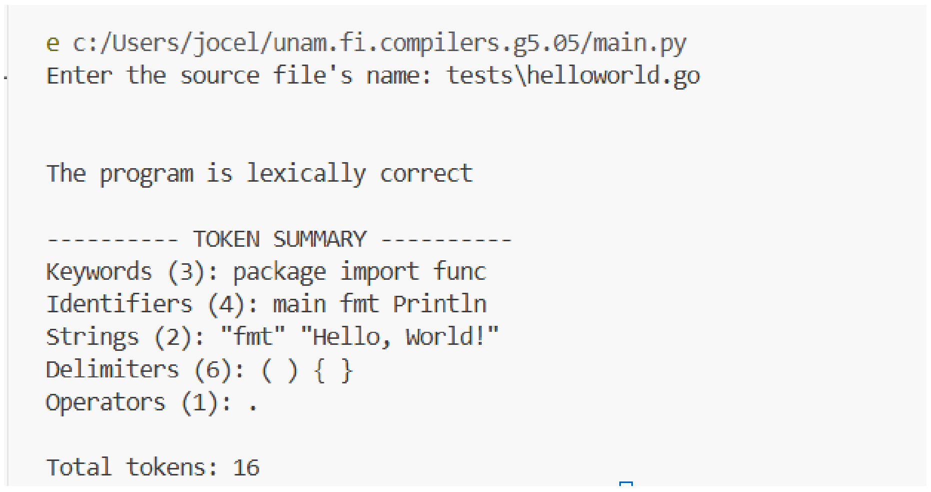


Figure 1: DFA for statement in fib.go

4 Results

4.1 Test Cases

OP01	**Code test for Hello world program** – ***Valid***
Program that prints "Hello, world!" in Go	
Steps to follow to perform the test	**Expected Result**
• Run the lexer on `hello.go`: • Inspect token stream order and categorize them. • Save the token stream to file.	• Valid token stream. • Comments/whitespace ignored; line/column preserved. • File `helloworld.txt` generated with the token stream.



```
e c:/Users/jocel/unam.fi.compilers.g5.05/main.py
Enter the source file's name: tests\helloworld.go


The program is lexically correct

---------- TOKEN SUMMARY ----------
Keywords (3): package import func
Identifiers (4): main fmt Println
Strings (2): "fmt" "Hello, World!"
Delimiters (6): ( ) { }
Operators (1): .

Total tokens: 16
```

Figure 2: Result of the execution of helloworld.go

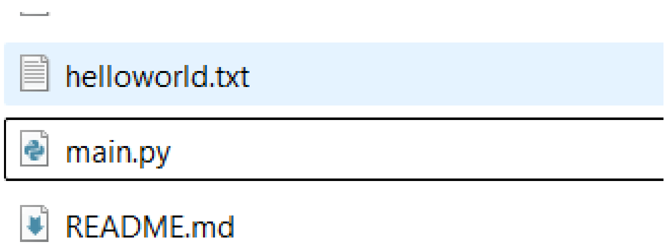
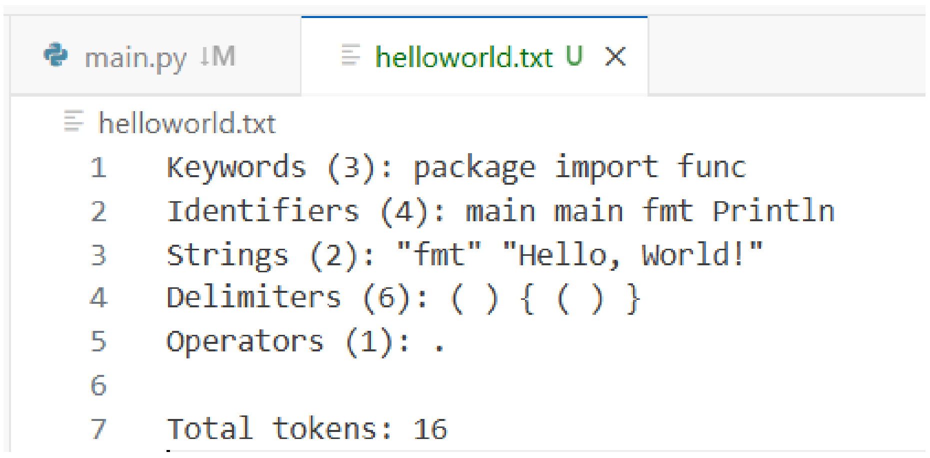


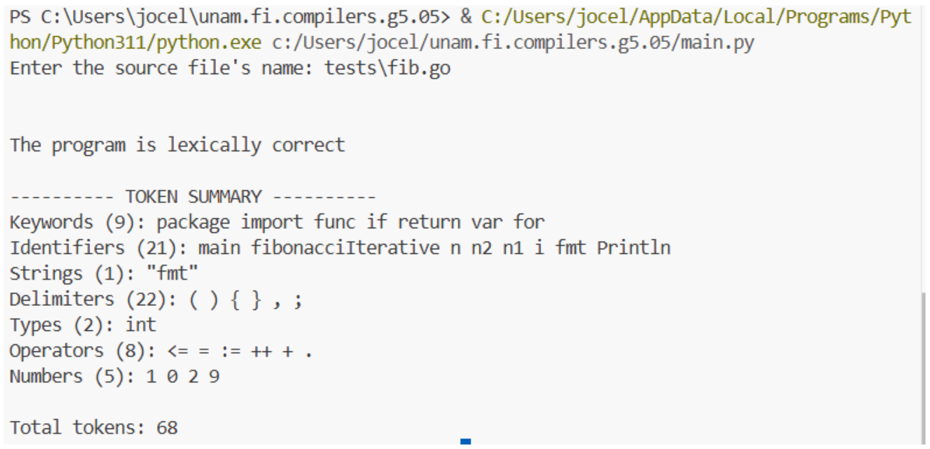
Figure 3: File generated with all the tokens including its repetitions of helloworld.go



```
Keywords (3): package import func
Identifiers (4): main main fmt Println
Strings (2): "fmt" "Hello, World!"
Delimiters (6): ( ) { ( ) }
Operators (1): .

Total tokens: 16
```

Figure 4: Content of the file with all the tokens: helloworld.txt

OP02	**Code test for Fibonacci sequence program** – ***Valid***
Program that computes the n-th Fibonacci number iteratively, including a leading block comment and line comments in Go.	
Steps to follow to perform the test	**Expected Result**
• Run the lexer on `fib.go`: • Inspect token stream order and categorize them. • Save the token stream to file.	• Valid token stream. • Comments and whitespace ignored: the entire leading `/* ... */` block. • File `fib.txt` generated with the token stream.



```
PS C:\Users\jocel\unam.fi.compilers.g5.05> & C:/Users/jocel/AppData/Local/Programs/Pyt
hon/Python311/python.exe c:/Users/jocel/unam.fi.compilers.g5.05/main.py
Enter the source file's name: tests\fib.go


The program is lexically correct

---------- TOKEN SUMMARY ----------
Keywords (9): package import func if return var for
Identifiers (21): main fibonacciIterative n n2 n1 i fmt Println
Strings (1): "fmt"
Delimiters (22): ( ) { } , ;
Types (2): int
Operators (8): <= = := ++ + .
Numbers (5): 1 0 2 9

Total tokens: 68
```

Figure 5: Result from the execution of the fib.go

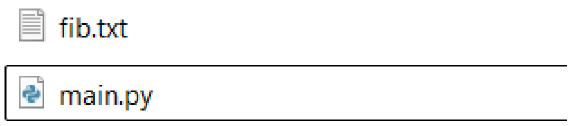
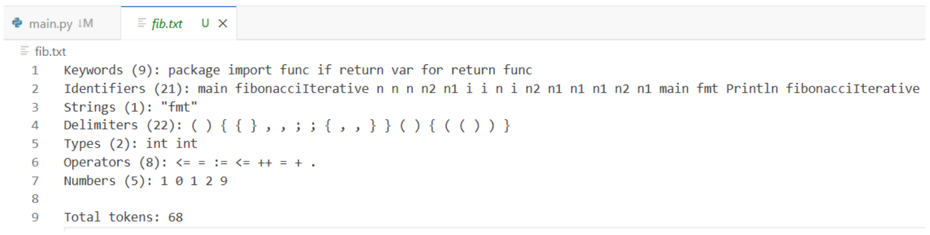


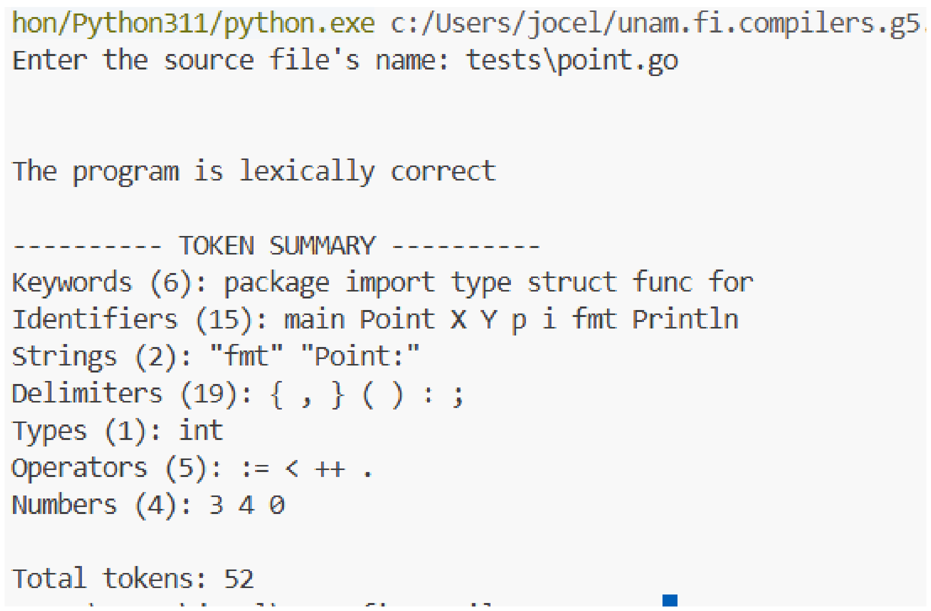
Figure 6: File generated with all the tokens including its repetitions of fib.go



```
Keywords (9): package import func if return var for return func
Identifiers (21): main fibonacciIterative n n n n2 n1 i i n i n2 n1 n1 n1 n2 n1 main fmt Println fibonacciIterative
Strings (1): "fmt"
Delimiters (22): ( ) { { } , , ; ; { , , } } ( ) { ( ( ) ) }
Types (2): int int
Operators (8): <= = := <= ++ = + .
Numbers (5): 1 0 1 2 9

Total tokens: 68
```

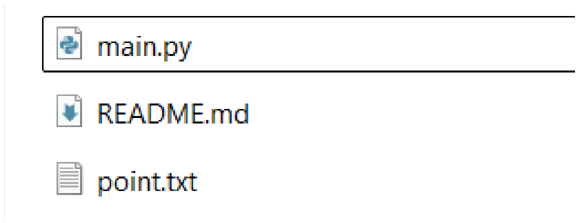
Figure 7: Content of the file with all the tokens: fib.txt

OP03	Code test for struct composite literal with for loop – *Valid*
Program that declares a `struct` type and prints it inside a for loop in go	
Steps to follow to perform the test	**Expected Result**
• Run the lexer on `point.go`: • Inspect token stream order and categorize them. • Save the token stream to file.	• Valid token stream. • Comments and whitespace ignored: the entire leading `/* ... */` block. • File `point.txt` generated with the token stream.



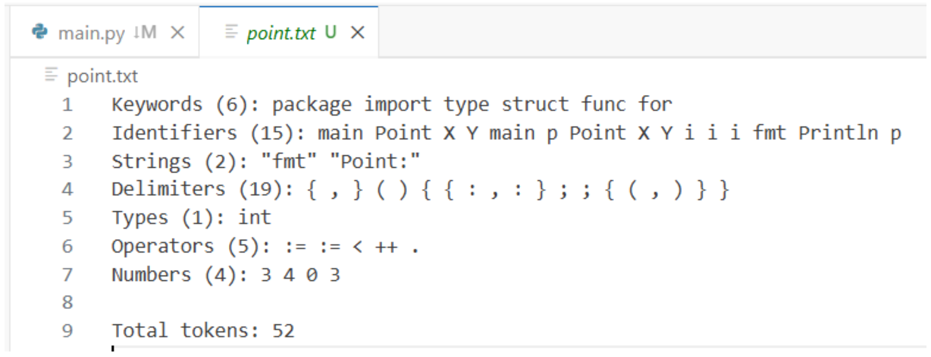
```
hon/Python311/python.exe c:/Users/jocel/unam.fi.compilers.g5
Enter the source file's name: tests\point.go


The program is lexically correct

---------- TOKEN SUMMARY ----------
Keywords (6): package import type struct func for
Identifiers (15): main Point X Y p i fmt Println
Strings (2): "fmt" "Point:"
Delimiters (19): { , } ( ) : ;
Types (1): int
Operators (5): := < ++ .
Numbers (4): 3 4 0

Total tokens: 52
```

Figure 8: Result of the execution of point.go



```
main.py
README.md
point.txt
```

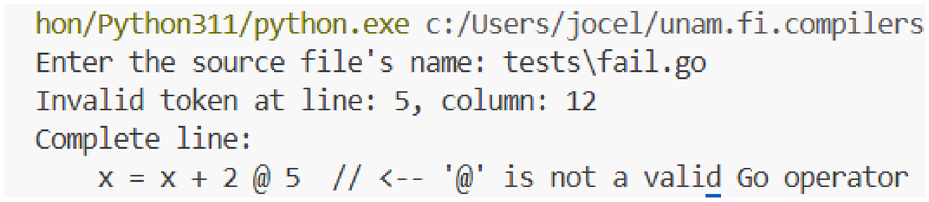
Figure 9: File generated with all the tokens including its repetitions of point.go



```
main.py M    point.txt U
point.txt
Keywords (6): package import type struct func for
Identifiers (15): main Point X Y main p Point X Y i i i fmt Println p
Strings (2): "fmt" "Point:"
Delimiters (19): { , } ( ) { { : , : } ; ; { ( , ) } }
Types (1): int
Operators (5): := := < ++ .
Numbers (4): 3 4 0 3

Total tokens: 52
```

Figure 10: Content of the file with all the tokens: point.txt

OP04	**Code test with operations – –*Invalid***
Program with an invalid operator character `@` in an expression in Go.	
Steps to follow to perform the test	**Expected Result**
• Run the lexer on `fail.go`: • Inspect token stream. • Stop the scanning after the first error.	• Recognizes the token until the error • Comments and whitespace ignored: the entire leading `/* ... */` block until the error. • Stop de execution when the lexer founds the first error, the invalid token @.



```
hon/Python311/python.exe c:/Users/jocel/unam.fi.compilers
Enter the source file's name: tests\fail.go
Invalid token at line: 5, column: 12
Complete line:
    x = x + 2 @ 5  // <-- '@' is not a valid Go operator
```

Figure 11: File with the tokens of fail.go

5 Conclusions

The implementation of the lexical analyzer proved to be a successful and illustrative application of the theoretical concepts from the compilers course. As we saw, the primary objective of transforming high-level language source code into a stream of categorized tokens was effectively achieved. The result, as presented in `tokens.txt` file, demonstrates that the Python script successfully recognized and classified a variety of syntactic elements including **keywords**, **identifiers**, **operators**, **constants**, and **punctuation**.

The results demonstrated the expected behavior across the programs: helloworld (OP01) for a minimal program, Fibonacci (OP02) to visualize the proper discard of the initial block comment and line comments, struct-for (OP03) to use a program with structures, and an invalid operator test (OP04) for an invalid symbol. In the valid cases, the tokens were categorized correctly in the right order, and with the total count ignoring the comments and whitespace. On the other hand, the invalid case, the lexer stopped at the first error with the location and the out limit defined symbol (@). This is useful for managing the limit of the tokens of our grammar for later parser implementations, and also for users.

All of this confirms the essential role of lexical analysis as the foundation for subsequent stages. By correctly delimiting token boundaries, the analyzer provides a reliable input for parsing and semantic analysis, which helps avoid ambiguities and prevents errors in later stages of the compilation process.

References

[1] D. Thain, *Introduction to Compilers and Language Design*, 2nd ed. Self-published, 2023, Revision Date: August 24, 2023, ISBN: 979-8-655-18026-0. [Online]. Available: `http://compilerbook.org`.

[2] S. Chattopadhyay, *Compiler Design*. PHI Learning Pvt. Ltd., 2022, ISBN: 9789391818760.

[3] P. H. Dave and H. B. Dave, *Compilers: Principles and Practice*, English. Pearson India, 2012, p. 504, Intermediate to Advanced.

[4] Salgado Cruz Emiliano Roman, *Notas del curso de compiladores*, UNAM, Facultad de Ingeniería, Notas personales con base en las clases., 2025.

[5] The Go Authors. "The go programming language specification." version go1.25, Accessed: Sep. 25, 2025. [Online]. Available: `https://go.dev/ref/spec`.

[6] W. Z. Feng Yaping, "Towards understanding bugs in go programming language," *IEEE*, 2024.

[7] PyPI. Python Package Index, *Rply 0.7.8*, `https://pypi.org/project/rply/`, Accessed: September 24, 2025.