Universidad Nacional Autónoma de México
Faculty of Engineering

Computer Engineering.

Compilers

PARSER & SDT

STUDENT:
320254536
315293728
423108596
320246881
320250648

Group:
5
Semester:
2026-I

México, CDMX. November 2025

Contents

1 Introduction

Syntactic and semantic analysis are fundamental stages in the compilation process, since they allow to verify the structure of a program according to the rules of its grammar. The previous stage was the lexical analysis, which generated the tokens from the source code. This project phase takes those tokens as input to perform **syntactic analysis (parsing)** and **semantic analysis (SDT)**. The parser's primary role is to verify that the token stream conforms to the structure of a predefined Context-Free Grammar [1], ensuring syntactic analysis. The analyzer design is based on the theoretical principles covered in class, but its practical implementation takes advantage of the RPLY library. This tool allowed us to define the grammar productions and automatically generate an efficient LALR(1) parser. Once parsing succeeds, we perform semantic analysis using a Syntax-Directed Translation (SDT) approach implemented with a visitor over the parse tree, semantic actions (SDT rules) are embedded directly into these grammar productions to perform semantic validation concurrently with parsing. The objective of the project are as follows:

- Design and implement a parser for a subset of the Go language using rply, which is an implementation of ply, based on a context-free grammar and the RPLY library, which generates an LALR(1) parser.
- Design and verify the CFG identifying the fundamental elements to establish the rules of production.
- Integrate a **Syntactic-Directed Translation (SDT) rules** into the parser productions to perform semantic analysis, focusing on key aspects such as variable declaration and type-checking.
- Validate the implementation against test cases that correctly differentiate between syntactic failures (Parsing error...) and semantic failures (Parsing Success! STD error...), as specified in the project requirements.

In general, the toolchain is modular: the parser focuses on recognizing structure and building the tree, then the semantic analyzer focuses on meaning and constraints, and also *saveprint.py* converts internal nodes into an NLTK tree to make lexical and parse outputs easy to visualize and to append to the project's report.

2 Theoretical Framework

This project phase is built upon the formal theories of Context-Free Grammars, which are practically applied through a parsing algorithm.

2.1 Context-Free Grammars (CFG)

A context-free grammar (CFG) provides a formal specification for the hierarchical, syntactic structure of a programming language [2]. Formally, a CFG consists of four components: terminals, nonterminals, a set of productions, and a start symbol[2]. The term "context-free" is used because the replacement of a nonterminal on the left side of a production by its body (the right side) can be done regardless of the context in which the nonterminal appears[2].

2.1.1 Terminals

Terminals are the basic symbols from which strings in the language are formed[2]. In the context of a compiler, the "token name" is a synonym for the term "terminal"[2]. These terminals are the input symbols that the parser processes[2].

2.1.2 Non-Terminals

Nonterminals are syntactic variables that denote sets of strings. The nonterminals define the hierarchical structure of the language and help to group sets of strings into syntactic categories, such as ‘expression‘ or ‘statement‘[2].

2.1.3 Productions

The product ions of a grammar specify the manner in which terminals and nonterminals can be combined to form strings[2]. Each production consists of:

- (a) A nonterminal, which is called the **head** or left side of the production[2].
- (b) The symbol $\rightarrow$ (read as "goes to" or "can be")[2].
- (c) A **body** (or right side) consisting of zero or more terminals and nonterminals[2]. The body describes one way in which strings of the head nonterminal can be constructed[2].

2.1.4 Start Symbol

In a grammar, one nonterminal is specially designated as the **start symbol**. This symbol represents the entire set of strings defined by the grammar; it denotes the language itself[2]. By convention, the start symbol is often the head of the grammar's first production[2].

2.2 Syntactic Analysis (Parsing)

Syntactic analysis is the process by which it is determined how a string of terminals can be generated by a grammar [2]. To parse a computer program, the form of valid sentences in a language is needed, this is made by a CFG as we previously described, a context free grammar is more powerful than regular expressions because they allow for recursion [1]. In general, a parser builds a parse tree in which the root is labeled as the start symbol, each nonleaf corresponds to a production, and each lead is labeled with a terminal or the empty string [2], so the parse tree shows the string of terminals at the leaves, read from left to right.

2.2.1 The Role of the Parser

In the compiler process, the syntactic analysis obtains the string of tokens from the lexical analyzer, and also verifies that the string of tokens can be generated by the grammar of the established language [2]. In addition, for well-formed programs, the parsers constructs a parse tree and passes it to the rest of the compiler process [2], so the output of the parser is a parse tree that captures the grammatical structures of the program. The parse tree also remembers where in the source file each construct appeared, so it is able to generate targeted error messages, if needed [1].

2.2.2 Parse Trees

To establish what is a parse tree, first we will develop the basic elements of the tree terminology [2]. A tree is made up of nodes, which can have labels (often grammar symbols). The root is the topmost node and has no parent. Each node, except the root, has one parent and can have one or more children. Children of the same node are called siblings and are ordered from left to right. A node with no children is called a leaf, while nodes with children are interior nodes. Finally, a descendant of a node includes its children, grandchildren, and so on, while an ancestor is any node above it in the hierarchy. A parse tree is a graphical representation of a derivation of the grammar. Each internal node of the tree represents the application of a production. For example, if we have $A \rightarrow BC$, then the node labeled as A, will have two children B and C. The leaves of the tree are the terminals or non-terminals at the end of the derivation. Read from left to right, they form the so-called “sentential form” or “yield” of the tree: this is the string generated by the grammar. [2]. The parse tree will follow the next properties [2]:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a terminal or by ϵ.
3. Each interior node is labeled by a nonterminal.

2.2.3 Syntactic Error Detection

A fundamental aspect of compiler design involves not only translating correct programs but also detecting and managing errors that arise during the compilation process. Although most programming languages lack explicit specifications for how compilers should respond to errors, planning an appropriate error-handling strategy from the start can simplify the compiler's architecture and enhance its reliability. Proper error management also assists programmers in identifying and correcting mistakes efficiently. The Syntactic errors occur when the program violates the grammatical rules of the language. Examples include misplaced semicolons, extra or missing braces, or the appearance of a `case` statement outside an enclosing `switch` block in C or Java.

To maintain the robustness of parsing, compilers employ strategies to recover from syntactic errors and continue processing the remaining code. Two general approaches are commonly discussed:

- **Panic-mode recovery:** This strategy discards input symbols until a synchronizing token (such as a semicolon or closing brace) is found, allowing the parser to resume normal operation without getting stuck.
- **Phrase-level recovery:** In this approach, small local corrections are applied to the input, such as inserting or deleting a token, to allow parsing to continue smoothly.

2.3 Parsing Methodologies

There are three general types of parsers for grammars [2].

- Universal methods, such as the Cocke-Younger-Kasami algorithm and Earley's algorithm that can parse any grammar, but in practices are really inefficient for a real use, so we will not get into them in depth.

Also, the methods usually implemented in compilers can be classified as top-down or bottom-up as we will develop in the next subsections.

2.3.1 Top-Down Parsing

Top-down parsing is a technique for constructing the parse tree from the root to the leaves. It starts with the initial symbol of the grammar and applies productions until the leaves match the input string.

The construction follows two repetitive steps:

1. **Select a production:**
   At a node containing a nonterminal A, choose one of its productions (for example, $A \to X_1 X_2 \dots X_n$) and create children for each symbol X_i.
2. **Move to the next unexpanded node:**
   Choose the *leftmost* nonterminal that has not yet been expanded, and repeat the previous step.

2.3.2 Bottom-Up (Shift-Reduce) Parsing

For a more general purpose, programming languages uses LR(1) grammar and associated bottom-up parsing techniques [1].

LR(1) grammars must be parsed using the **shift-reduce** parsing technique. This strategy begins with the tokens and looks for rules that can be applied to reduce sentential forms into non-terminals. [1]. So, if there is a sequence of reductions that lead to the start symbol, then the parse is successful. This is how it is established [1]:

- **Shift**. Consumes one token from the input stream and pushes it onto the stack.
- **Reduce**. Applies one rule of the form $A \rightarrow \alpha$ from the grammar, replacing the sentential form α on the stacj withe the non-terminal A.

2.3.3 LALR(1) Parsing (The RPLY Approach)

Lookahead LR(LALR) parsing is the practical answer to disadvantages presented in LR(0) and LR(1). To construct an LALR parser, we first create the LR(1) automaton, and then merge states that have the same core [1]. With LALR parsers, the objective is to achieve the benefits provided by the LR(1) method, but with the cost of the SLR method and therefore using the smallest number of states that the latter requires. [3]. So, remembering, PLY is a Python-based implementation of the classic compiler tools lex and yacc. It supports LALR(1) parsing and offers thorough input checking, detailed error messages, and diagnostic tools. Therefore, developers familiar with yacc in other programming languages should find PLY easy to learn and apply [4].

2.4 Semantic Analysis and Syntax-Directed Translation (SDT)

This section of the compiler is responsible for checking the program for semantic errors and for gathering type information for the subsequent code-generation phase[2]. This is achieved by associating information with language constructs through attributes attached to grammar symbols. A **syntax-directed definition (SDD)** is a generalization of a context-free grammar in which each grammar production is associated with a set of semantic rules[2].

2.4.1 The Need for Semantic Analysis

Many language constructs cannot be specified using context-free grammars alone. For example, a CFG cannot check that a variable has been declared before it is used, or that an operation is being applied to an operand of the correct type[2]. These "context-sensitive" aspects require semantic analysis, which is implemented by adding attributes and semantic rules to the grammar's productions.

2.4.2 Syntax-Directed Translation (SDT) Rules

A **syntax-directed translation scheme (SDT)** is a context-free grammar in which semantic rules are embedded within the production bodies[2]. These semantic rules are fragments of code that are executed as the parser recognizes the corresponding productions. By convention, semantic rules are enclosed in curly braces[2]. These actions can be used to compute attribute values, build a syntax tree, or perform semantic checks.

2.4.3 The Symbol Table

A symbol table is a critical data structure used by a compiler to keep track of information about identifiers[2]. This information is entered into the symbol table during lexical and syntactic analysis, and is used during semantic analysis and code generation[2]. The symbol table stores information about the scope and binding of names, as well as data-type information for each identifier.

2.4.4 Type Checking

A **type checker** is a component of the compiler that verifies that the type of a construct matches that expected by its context[2]. A compiler needs to check that the type of an operand is valid for a given operator[2]. This process ensures that operations are performed on compatible types, preventing type-related errors during program execution.

2.4.5 Semantic Error Detection

If the type checker finds that the type of an operand is not valid for a given operator, the compiler must report an error[2]. This is the core of semantic error detection. Other examples of semantic errors include using an undeclared variable, multiple declarations of the same identifier within a scope, or calling a function with an incorrect number or type of arguments.

3 Development

3.1 Project structure and general functionality

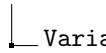
The project is written in python and has the following structure.

```
unam
README.MD
  fi
    compilers
      g5
        05
          doc
            Team05_Compilers_Lexer.pdf
            05-Compilers-Parser.pdf
          src
            main.py
            modules
              lexer.py
              parser.py
              savePrint.py
              semantic.py
          test
            fib.go
            fail.go
            helloworld.go
            point.go
            operation.go
            type_mismatch.go
            undeclared.go
          README.MD
```

3.2 Grammar Specification

The grammar implemented was done by following Go language's own grammar specification [5], but not entirely. The general structure it represents is the following:

```
SourceFile
  Package Declaration
  Import Declarations
  Top-level Declarations
    Type Declarations
    Function Declarations
```



3.2.1 Token Types

Keywords

KW_BREAK, KW_DEFAULT, KW_FUNC, KW_CASE, KW_STRUCT, KW_ELSE, KW_PACKAGE, KW_SWITCH, KW_IF, KW_TYPE, K

Type Tokens

TYPE_INT, TYPE_FLOAT32, TYPE_FLOAT64, TYPE_BOOL, TYPE_STR

Literal Tokens

LIT_FLOAT, LIT_INT, LIT_STR, LIT_BOOL

Operator Tokens

OP_PLUSEQ, OP_PLUSPLUS, OP_MINUSEQ, OP_MINUSMINUS, OP_MULEQ,
OP_DIVEQ, OP_MODEQ, OP_ANDEQ, OP_ANDAND, OP_ANDNOTEQ, OP_OREQ,
OP_OROR, OP_XOREQ, OP_SHLEQ, OP_SHREQ, OP_EQEQ, OP_NEQ, OP_LTE,
OP_GTE, OP_COLONEQ, OP_PLUS, OP_MINUS, OP_MUL, OP_DIV, OP_MOD,
OP_EQ, OP_LT, OP_GT, OP_NOT, OP_DOT

Punctuation Tokens

PUNC_LPAREN, PUNC_RPAREN, PUNC_LBRACK, PUNC_RBRACK, PUNC_LBRACE,
PUNC_RBRACE, PUNC_COMMA, PUNC_SEMI, PUNC_COLON, IDENT

3.2.2 Source File Structure

SourceFile	= PackageClause { ImportDecl } { TopLevelDecl }
PackageClause	= **package** IDENT
ImportDecls	= ϵ | ImportDecls ImportDecl
ImportDecl	= **import** ImportSpec
ImportSpec	= LIT_STR | "." LIT_STR | IDENT LIT_STR
TopLevelDecls	= ϵ | TopLevelDecls TopLevelDecl
TopLevelDecl	= TypeDecl | FunctionDecl | VarDecl | **error**

The program source file consists of a package declaration, zero or more import declarations, or zero or more top-level declarations. More importantly, TopLevelDecls represents the main content of a source file, all the declarations that exist at the package level, outside of any function or block. It represents the building blocks that define what the program actually does. Supports type definitions, function definitions, variable declarations, and parser error recovery.

3.2.3 Type Declaration

`TypeDecl`	= **type** IDENT Type
`Type`	= SimpleType | StructType | ArrayType | SliceType
`SimpleType`	= **int** | **float32** | **float64** | **bool** | **string** | IDENT
`StructType`	= **struct** "" StructFieldDecls ""
`StructFieldDecls`	= ϵ | StructFieldDeclList
`StructFieldDeclList`	= StructFieldDecl | StructFieldDeclList StructFieldDecl
`StructFieldDecl`	= FieldDecl | FieldDecl
`FieldDecl`	= IDENTList Type | IDENTList Type Tag
`Tag`	::= LIT_STR
`ArrayType`	= "[" Expression "]" Type
`SliceType`	= "[]" Type

Supports four kinds of types: simpleType which are built-in types (int, float32, float64, bool, string) or identifiers, structType, composite types with named fields, arrayType for fixed-size collections [size]ElementType and sliceType for dynamic-sized collections []ElementType. For the structType, fields can have optional string tags for metadata, and multiple fields of the same type can be declared together.

3.2.4 Function Declarations

`FunctionDecl`	= **func** IDENT Signature Block
`Signature`	= "(" ParametersOpt ")" ResultOpt
`ParametersOpt`	= ϵ | ParameterList
`ParameterList`	= ParameterDecl | ParameterList "," ParameterDecl
`ParameterDecl`	= IDENT Type
`ResultOpt`	= ϵ | Type

Defines functions of the type: func name(signature) { body }. From signature it gives the function interface specification, the optional parameter list in parentheses, the name Type pairs and the optional return type.

3.2.5 Variable Declarations

`VarDecl`	= **var** VarSpecList
`VarSpecList`	= VarSpec | VarSpecList ";" VarSpec
`VarSpec`	= IDENTList AssignOp ExpressionList | IDENTList Type AssignOp ExpressionList | IDENTList Type
`IDENTList`	= IDENT | IDENTList "," IDENT
`ExpressionList`	= Expression | ExpressionList "," Expression
`AssignOp`	= "=" | "+=" | "-=" | "*=" | "/=" | "%=" | "&=" | "|=" | "^=" | "«=" | "»=" | "&^=" | ":="
`ShortVarDecl`	= IDENTList ":=" ExpressionList

The variable specifications come in three forms:

- Untyped with initialization: x, y = 1, 2
- Typed with initialization: x int = 5
- Typed without initialization: x int (zero value)

Additionally, shorthand declaration has type inference: x := 5 or x, y := 1, 2.

3.2.6 Expressions

`Expression`	= LogicalOrExpression
`LogicalOrExpression`	= LogicalAndExpression | LogicalOrExpression "||" LogicalAndExpression
`LogicalAndExpression`	= EqualityExpression | LogicalAndExpression "&&" EqualityExpression
`EqualityExpression`	= RelationalExpression | EqualityExpression "==" RelationalExpression | EqualityExpression "!=" RelationalExpression
`RelationalExpression`	= AdditiveExpression | RelationalExpression "<" AdditiveExpression | RelationalExpression ">" AdditiveExpression | RelationalExpression "<=" AdditiveExpression | RelationalExpression ">=" AdditiveExpression
`AdditiveExpression`	= MultiplicativeExpression | AdditiveExpression "+" MultiplicativeExpression | AdditiveExpression "-" MultiplicativeExpression
`MultiplicativeExpression`	= UnaryExpression | MultiplicativeExpression "*" UnaryExpression | MultiplicativeExpression "/" UnaryExpression | MultiplicativeExpression "%" UnaryExpression
`UnaryExpression`	= PrimaryExpression | "+" UnaryExpression | "-" UnaryExpression | "!" UnaryExpression
`PrimaryExpression`	= Operand PrimarySuffixList | CompositeLit | Type "(" Expression ")" | Type "." IDENT
`PrimarySuffixList`	= ϵ | PrimarySuffixList PrimarySuffix
`PrimarySuffix`	= "." IDENT | "[" Expression "]" | "(" ArgumentListOpt ")"
`Operand`	= Literal | IDENT | QualifiedIdent | "(" Expression ")"
`QualifiedIdent`	= IDENT "." IDENT
`CompositeLit`	= Type "" ElementListOpt ""
`ElementListOpt`	= ϵ | ElementList
`ElementList`	= KeyedElement | ElementList "," KeyedElement
`KeyedElement`	= Key ":" Expression
`Key`	= IDENT
`ArgumentListOpt`	= ϵ | ArgumentList
`ArgumentList`	= Expression | ArgumentList "," Expression
`Literal`	= LIT_INT | LIT_FLOAT | LIT_STR | LIT_BOOL

The operator precedence declaration follows the standard operator precedence declaration described as such, from highest to lowest:

1. Primary expressions: literals, identifiers, parentheses, etc.
2. Unary operators: +, -, ! (right-associative)
3. Multiplicative: *, /,
4. Additive: +, - (left-associative)
5. Relational: <, >, <=, >= (non-associative)
6. Equality: ==, != (non-associative)
7. Logical AND: && (left-associative)
8. Logical OR: || (left-associative)

The most basic expression forms are operand: literals, identifiers, expressions inside parenthesis or the added feature that includes simple and qualified identifiers. Another basic expression form is the composite literals that are used for initializing structs, arrays or slices. Other primary expression is function calls, it specifies between operands that are qualified identifiers which enable accessing exported identifiers from other packages. The last ones are for explicit type conversion and first-class method references and method expression.

3.2.7 Statements and Control Flow

`Block`	= "{" StatementList "}"
`StatementList`	= ϵ | StatementList Statement
`Statement`	= Expression | ShortVarDecl | VarDecl | TypeDecl | FunctionDecl | **return** Expression | **if** Expression Block | **if** Expression Block **else** Statement | **if** Expression Block **else if** Expression Block | **for** Block | **for** Expression Block | **for** ForClause Block | **switch** Expression "" CaseClauses "" | **break** | **continue**
`ForClause`	= InitStmt ";" Expression ";" PostStmt
`InitStmt`	= SimpleStmt | ϵ
`PostStmt`	= SimpleStmt | ϵ
`SimpleStmt`	= ShortVarDecl | Expression | IncDecStmt
`IncDecStmt`	= Expression "++" | Expression "−"
`CaseClauses`	= ϵ | CaseClauses CaseClause
`CaseClause`	= **case** Expression ":" StatementList | **default** ":" StatementList

For statements and control flow we have a block which represents a sequence of statements in { } braces. For statement types there are expression statements (x + y), declaration statements, control flows (if, for, switch, break, continue), and short variable declarations (x := 5).

The if statement has three forms: simple, while-like and c-style, so initStmt and PostStmt are optional. The simple statement are used in for loop clauses.

3.2.8 Grammar Notes

The Go language's grammar has the semicolon ";" as a terminal symbol; however, while doing the implementation this proved to introduce ambiguity to the grammar and raise shift/reduce and reduce/reduce warnings from the parser generator. For which we decided to not include this optional feature and not handle the semicolon as a terminal symbol.

For the syntax, the format used for better understanding inside the documentation is described as follows:

- Each production uses the format: `production_name = Expression`
- ϵ represents an empty production.
- Basic Operators:
  - | for alternatives.
  - [] for optional elements.
  - { } for repetition (zero or more).
  - ( ) for grouping.
- Terminal symbols are shown in **bold** for keywords, quoted for operators/punctuation, or token names in uppercase (IDENT, LIT_STR, etc.).

In general, the grammar implements supports a Go-like syntax with a simplified type system.

3.2.9 Ambiguity

As some production rules are inherently ambiguous (such as addition operations), precedence must be defined manually to disambiguate ambiguous production rules, in RPLY, this is defined within the *parsergenerator* class as a list of precedence rules for a given set of productions.

3.2.10 Defining production rules

In RPLY production rules are defined as a sequence of terminals (tokens) and non-terminals using the *ParserGenerator.production()* decorator the argument for this function is a string of the form: "$non-terminal\ :\ (tokens|nonterminals)^*$" and returns a list p containing all the elements matched by the right hand side of the production.
This list p can then be used to define a python function containing semantic actions for that production, such as printing the production or in our case returning a node of the parse tree.

3.3 Semantic Analyzer (STD)

Once the syntactic analyzer confirms that the token sequence forms a valid structure (generating the NLTK Parse Tree), the semantic analyzer's job is to validate that this structure has logical meaning. This implementation is encapsulated in **semantic.py** The analysis is performed using a **Visitor Pattern**. A **SemanticAnalyzer** class traverses the entire Parse Tree returned by the parser. Using the node labels (e.g., "FunctionDecl","VarDecl"), it dynamically calls specific validation methods. The core of this analyzer relies on two key components:

3.3.1 The Symbol Table

To perform semantic checks, we must track the identifiers (variables, functions) that exist in the program. This is handled by the **SymbolTable** class. A stack of dictionaries implements lexical scoping. Precisely, the symbol table is a stack of Python dictionaries where each dictionary represents one scope.

3.3.2 Key SDT Rules (Visitor Methods)

The semantic analyzer implements the SDT rules as visitor methods in the class *SemanticAnalyzer*. The visitor transits the NLTK tree obtained int he parsing stage, and send methods with the structure `visit_NodeLabel`. Id there isn't a specialized method, a generic visitor recursively processes the children.

3.4 Output

The program first outputs a token summary containing all tokens that were detected during the lexical analysis phase as well as a token count.
After the lexical analysis phase the program begins the syntactic analysis phase, if it is successful the program outputs to console a *parsing success* message, if it detects any out of place tokens (according to the language's grammar) the program will instead output the content and location of the misplaced token.
After the program finishes parsing the token stream it will begin the semantic analysis phase, if successful it will output an *SDT Verified!* message, indicating the program is semantically correct, if it detects a semantic error it will output an error message instead.
After the programs successfully runs it will produce a .txt file containing the token summary and a text representation of the source programs parse tree.

3.5 Error Handling

For this program we handle errors in two different stages of the process, first we handle errors detected during parsing using RPLY's built in error *rply.ParsingError*, this error is raised by the program when a token is found that matches no existing production and contains information regarding where in the source file the error was found, when a syntax error is detected this way the program stops.
Semantic errors are handled by the *semantic.py* module, it verifies that tokens are being used in the correct context.

4 Results

4.1 Test Cases

OP01	**Code test for Hello world program**: Lexer *Valid* Syntax *Valid* Semantic *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. • If lexycally correct, run the next stage syntactic analysis. • If syntactically correct, run the semantic analyzer. • Save token summary and parse tree to file hello.txt	• *Lexical phase*: Valid token stream; comments/whitespace ignored; line/column preserved. • *Syntactic phase*: Console shows `Parsing Success!`; parse tree appended to output file. • *Semantic phase*: Console shows `SDT Verified!`.

```
1/python.exe c:/Users/jocel/unam.fi.compilers.g5.05/unam/fi/compilers/g5/05/compiler/src/main.py
Enter the source file's name: ..\test\helloworld.go


The program is lexically correct

Tokens summary written to: C:\Users\jocel\unam.fi.compilers.g5.05\unam\fi\compilers\g5\05\compiler\src\helloworld.txt
```

Figure 1: Result of lexer phase

```
Starting Parsing (Syntactic)...
Parsing Success!

Parse tree appended to: C:\Users\jocel\unam.fi.compilers.g5.05\unam\fi\compilers\g5\05\compiler\src\helloworld.txt
```

Figure 2: Result of parser phase

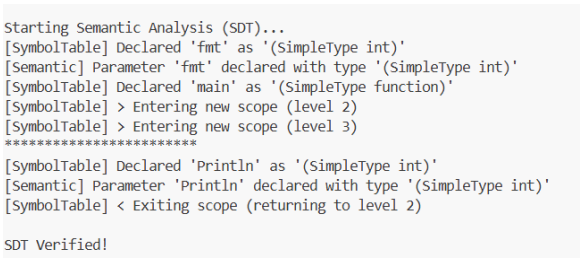


Figure 3: Result of semantic phase

Listing 1: Content of helloworld.txt

```
1
2
3  ------------- TOKEN SUMMARY ------------
4  Keywords (3): package import func
5  Identifiers (4): main fmt Println
6  Strings (2): "fmt" "Hello, World!"
7  Delimiters (6): ( ) { }
8  Operators (1): .
9
10 Total tokens: 16
11
12
13 ---------- PARSE TREE (NTLK STYLE) ----------
14 (SourceFile
15   (PackageClause (package ) (Identifier main))
16   (ImportDecls (ImportDecl (ImportSpec (path fmt))))
17   (TopLevelDecls
18     (TopLevelDecl
19       (FunctionDecl
20         (Identifier main)
21         (Signature (Parameters ) (Result ))
22         (Block
23           (StatementList
24             (ExprStmt
25               (CallExpr
26                 (QualifiedIdent
27                   (Identifier fmt)
28                   (Identifier Println))
29                 (ArgumentList (StringLiteral "Hello, World!")))))))))
```

OP02	**Code test for Fibonacci sequence program** Lexer *Valid* Syntax *Valid* Semantic *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. • If lexycally correct, run the next stage syntactic analysis and buid the parse tree. • If syntactically correct, run the semantic analyzer based on the parse tree. • Save token summary and parse tree to file hello.txt	• *Lexical phase*: Valid token stream; comments/whitespace ignored; line/column preserved. • *Syntactic phase*: Console shows `Parsing Success!`; parse tree appended to output file. • *Semantic phase*: Console shows `SDT Verified!`.

```
Enter the source file's name: ..\test\fib.go


The program is lexically correct

Tokens summary written to: C:\Users\jocel\unam.fi.compilers.g5.05\unam\fi\compilers\g5\05\compiler\src\fib.txt
```

Figure 4: Result of lexer phase for fib.go

```
  self.parser = self.pg.build()

Starting Parsing (Syntactic)...
Parsing Success!

Parse tree appended to: C:\Users\jocel\unam.fi.compilers.g5.05\unam\fi\compilers\g5\05\compiler\src\fib.txt
```

Figure 5: Result of parser phase for fib.go

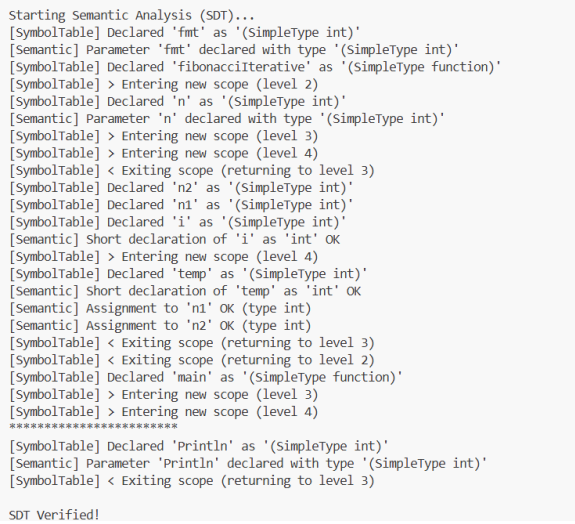


Figure 6: Result of semantic phase for fib.go

Listing 2: Content of fib.txt

```


------------- TOKEN SUMMARY ------------
Keywords (9): package import func if return var for
Identifiers (23): main fibonacciIterative n n2 n1 i temp fmt Println
Strings (1): "fmt"
Delimiters (20): ( ) { } , ;
Types (3): int
Operators (10): <= = := ++ + .
Numbers (5): 1 0 2 9

Total tokens: 71


---------- PARSE TREE (NTLK STYLE) ----------
(SourceFile
  (PackageClause (package ) (Identifier main))
  (ImportDecls (ImportDecl (ImportSpec (path fmt))))
  (TopLevelDecls
    (TopLevelDecl
      (FunctionDecl
        (Identifier fibonacciIterative)
        (Signature
          (Parameters
            (ParameterDecl (Identifier n) (SimpleType int)))
          (Result (SimpleType int)))
        (Block
          (StatementList
            (IfStmt
              (BinaryExpr
                (Identifier n)
                (Operator <=)
                (IntLiteral 1))
              (Block (StatementList (ReturnStmt (Identifier n)))))
            (DeclStmt
```

```
              (VarDecl
                (VarSpecList
                  (VarSpec
                    (IdentifierList (Identifier n2) (Identifier n1))
                    (SimpleType int)
                    (ExpressionList (IntLiteral 0) (IntLiteral 1))))))
            (ForStmt
              (ForClause
                (ShortVarDecl
                  (IdentifierList (Identifier i))
                  (ExpressionList (IntLiteral 2)))
                (BinaryExpr
                  (Identifier i)
                  (Operator <=)
                  (Identifier n))
                (IncDecStmt (Identifier i) (Operator ++)))
              (Block
                (StatementList
                  (ShortVarDecl
                    (ShortVarDecl
                      (IdentifierList (Identifier temp))
                      (ExpressionList (Identifier n1))))
                  (AssignStmt
                    (ExpressionList (Identifier n1))
                    (AssignOp =)
                    (ExpressionList
                      (BinaryExpr
                        (Identifier n1)
                        (Operator +)
                        (Identifier n2))))
                  (AssignStmt
                    (ExpressionList (Identifier n2))
                    (AssignOp =)
                    (ExpressionList (Identifier temp))))))
            (ReturnStmt (Identifier n1))))))
    (TopLevelDecl
      (FunctionDecl
        (Identifier main)
        (Signature (Parameters ) (Result ))
        (Block
          (StatementList
            (ExprStmt
              (CallExpr
                (QualifiedIdent
                  (Identifier fmt)
                  (Identifier Println))
                (ArgumentList
                  (CallExpr
                    (Identifier fibonacciIterative)
                    (ArgumentList (IntLiteral 9))))))))))))
```

OP03	**Code test for Operation program**: Lexer *Valid* Syntax *Valid* Semantic *Valid*
Program that does a simple assignment (a:= 2+2) in Go	
Steps to follow to perform the test	**Expected Result**
• Run the lexer on `operation.go`: • Inspect token stream order and categorize them. • If lexycally correct, run the next stage syntactic analysis (parser) and build and save the parse tree. • If syntactically correct, run the semantic analyzer. • Save token summary and parse tree to file operation.txt	• *Lexical phase*: Valid token stream; comments/whitespace ignored; line/column preserve; token summary appended to output file operation.txt. • *Syntactic phase*: Console shows `Parsing Success!`; parse tree appended to output file operation.txt. • *Semantic phase*: Console shows `SDT Verified!`, (type-correct arithmetic `int + int`; new identifier `a` declared in `main` scope).

```
Enter the source file's name: ..\test\operation.go


The program is lexically correct

Tokens summary written to: C:\Users\jocel\unam.fi.compilers.g5.05\unam\fi\compilers\g5\05\compiler\src\operation.txt
```

Figure 7: Result of lexer phase for operation.go

```
Starting Parsing (Syntactic)...
Parsing Success!

Parse tree appended to: C:\Users\jocel\unam.fi.compilers.g5.05\unam\fi\compilers\g5\05\compiler\src\operation.txt
```

Figure 8: Result of parser phase for operation.go

```
Starting Semantic Analysis (SDT)...
[SymbolTable] Declared 'main' as '(SimpleType function)'
[SymbolTable] > Entering new scope (level 2)
[SymbolTable] > Entering new scope (level 3)
[SymbolTable] Declared 'a' as '(SimpleType int)'
[Semantic] Short declaration of 'a' as 'int' OK
[SymbolTable] < Exiting scope (returning to level 2)

SDT Verified!
```

Figure 9: Result of semantic phase for operation.go

Listing 3: Content of operation.txt

```


------------- TOKEN SUMMARY ------------
Keywords (2): package func
Identifiers (3): main a
Delimiters (4): ( ) { }
Operators (2): := +
Numbers (2): 2

Total tokens: 13


---------- PARSE TREE (NTLK STYLE) ----------
(SourceFile
  (PackageClause (package ) (Identifier main))
  (ImportDecls )
  (TopLevelDecls
    (TopLevelDecl
      (FunctionDecl
        (Identifier main)
        (Signature (Parameters ) (Result ))
        (Block
          (StatementList
            (ShortVarDecl
              (ShortVarDecl
                (IdentifierList (Identifier a))
                (ExpressionList
                  (BinaryExpr
                    (IntLiteral 2)
                    (Operator +)
                    (IntLiteral 2)))))))))))
```

OP04	**Code test for Operation `x + *y`:** Lexer *Valid* Syntax *Invalid* Semantic *Not executed*
Go program with commented blocks (ignored by lexer) and an expression `x + *y` that is outside the accepted grammar, triggering a parser error.	
Steps to follow to perform the test	**Expected Result**
• Run the lexer on `syntaxerr.go`: • Inspect token stream order and categorize them. • If lexycally correct, run the next stage syntactic analysis (parser) • (*Stops on error*) Semantic analyzer (SDT) is not executed.	• *Lexical phase*: Valid token stream; comments/whitespace ignored; line/column preserve; token summary appended to output file syntaxerr.txt. • *Syntactic phase*: Console shows `Parsing error: Unexpected token '*' at line X, column Y` (error points to the unary `*` in `x + *y`). • *Semantic phase*: Not executed.

```
Enter the source file's name: ..\test\syntaxerr.go


The program is lexically correct

Tokens summary written to: C:\Users\jocel\unam.fi.compilers.g5.05\unam\fi\compilers\g5\05\compiler\src\syntaxerr.txt
```

Figure 10: Result of the lexer phase on syntaxerr.go

```
Starting Parsing (Syntactic)...
ERROR SINTACTICO: No se esperaba encontrar el Token '*' en la línea '20' columna '18'
Traceback (most recent call last):
  File "c:\Users\jocel\unam.fi.compilers.g5.05\unam\fi\compilers\g5\05\compiler\src\main.py", line 114, in <module>
    main()
  File "c:\Users\jocel\unam.fi.compilers.g5.05\unam\fi\compilers\g5\05\compiler\src\main.py", line 73, in main
    parse_tree = parser.parse(lexer.lex(source_code))
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\jocel\AppData\Local\Programs\Python\Python311\Lib\site-packages\rply\parser.py", line 60, in parse
    self.error_handler(lookahead)
  File "c:\Users\jocel\unam.fi.compilers.g5.05\unam\fi\compilers\g5\05\compiler\src\modules\parser.py", line 699, in error_handle
    raise ValueError(token)
ValueError: Token('OP_MUL', '*')
PS C:\Users\jocel\unam.fi.compilers.g5.05\unam\fi\compilers\g5\05\compiler\src>
```

Figure 11: Result of the parser phase on syntaxerr.go

OP05	**Code test for `x = "hola"` assignment**: Lexer *Valid* Syntax *Valid* Semantic *Invalid*
Go program where `x` is declared as `int` and then assigned a `string`.	
Steps to follow to perform the test	**Expected Result**
• Run the lexer on `type_mismatch.go`: • Inspect token stream order and categorize them. • If lexycally correct, run the next stage syntactic analysis (parser) • (*Stops on error*) Semantic analyzer (SDT) is not executed.	• *Lexical phase*: Valid token stream; comments/whitespace ignored; line/column preserve; token summary appended to output file operation.txt. • *Syntactic phase*: Console shows `Parsing Success!` and Parse tree appended to `main_type_err.txt` (NLTK style). • *Semantic phase*: `Parsing Success!` and then `SDT error... ⟨reason⟩`

```
Starting Parsing (Syntactic)...
Parsing Success!

Parse tree appended to: C:\Users\isaia\Downloads\unam.fi.compilers.g5.05-Semantic
s\unam\fi\compilers\g5\05\compiler\src\type_mismatch.txt

Starting Semantic Analysis (SDT)...
[SymbolTable] Declared 'main' as '(SimpleType function)'
[SymbolTable] > Entering new scope (level 2)
[SymbolTable] > Entering new scope (level 3)
[SymbolTable] Declared 'x' as '(SimpleType int)'

--- SEMANTIC (SDT) ERROR ---
Parsing Success!
SDT error... SDT Error: Cannot assign type 'string' to variable 'x' of type 'int'
```

Figure 12: Result of the parser and semantic phase

OP06	**Code test for `x = 10` without declaration**: Lexer *Valid* Syntax *Valid* Semantic *Invalid*
Go program assigns to `x` inside `main()` without a prior declaration.	
Steps to follow to perform the test	**Expected Result**
• Run the lexer on `undeclared.go`: • Inspect token stream order and categorize them. • If lexycally correct, run the next stage syntactic analysis (parser) • (*Stops on error*) Semantic analyzer (SDT) is not executed.	• *Lexical phase*: Valid token stream; comments/whitespace ignored; line/column preserve; token summary appended to output file .txt. • *Syntactic phase*: Console shows `Parsing error: Unexpected token '*' at line X, column Y` (error points to the unary `*` in `x + *y`). • *Semantic phase*: Not executed.

OP07	**Code test with operations** Lexer *Invalid* Syntax *Not executed* Semantic *Not executed*
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. • The next stages are not executed.	• 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 13: File with the tokens of fail.go

5 Conclusions

We consider that this project has successfully met the primary objectives established in the introduction. A robust syntactic analyzer was developed by defining a Context-Free Grammar for a subset of the Go Language and leveraging the RPLY library to generate an LALR(1) parser, thus achieving our first goal. This parser proved capable of correctly translating the token stream from the lexical phase into a structured Parse Tree.

Furthermore, the second objective was met by designing and implementing a SemanticAnalyzer module (semantic.py). This module successfully integrates Syntax-Directed Translation (SDT) rules using the Visitor Pattern. By traversing the parse tree and employing a scoped Symbol Table, the analyzer performs critical semantic checks, including the validation of variable declarations and type-checking on assignments and expressions.

Finally, the validation tests confirm that the implementation meets our third and most critical objective: the system now correctly differentiates between syntactic and semantic failures. As demostrated in the results, the compiler properly reports Parsing error... for syntactically malformed code, while correctly reporting Parsing Success! SDT error... for programs that are syntactically valid but semantically incoherent, such as those with type mismatches or undeclared variables. This successful integration of the parser and semantic analyzer provides a solid and validated foundation for the subsequent phases of the compiler.

During the development of this project, we realized that the grammar definition could be improved in further stages by minimizing shift/reduce and reduce/reduce conflicts to strengthen the grammar's robustness overall, even if the testing process proved to give us the expected results.

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] R. S. A. A.V. Abo M.S. Lam and J. D. Ullman, *Compilers : principles, techniques, and tools*. Pearson Education. Addison-Wesley-, 2007, ISBN: 0-321-48681-1.

[3] Cartagena99. "Apuntes de ininf2 módulo 4 unidad 4 tema 3." Accedido: 2 de noviembre de 2025. [Online]. Available: `https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2_M4_U4_T3.pdf`.

[4] D. M. Beazley. "Ply (python lex-yacc)." Accedido: 2 de noviembre de 2025. [Online]. Available: `https://www.dabeaz.com/ply/ply.html`.

[5] The Go Authors. "The go programming language specification." version go1.25, Accessed: Nov. 3, 2025. [Online]. Available: `https://go.dev/ref/spec`.