GoCompiler

Your code may compile, but is it correct?
We'll tell you.

Team 05

Meneses Calderas Grecia Irais
Miramón Pérez Jocelyn
Membrilla Ramos Isaias Iñaki
Pérez Osorio Luis Eduardo
Salgado Cruz Emiliano Roman

RESULTS

STAGES FOR HELLOWORLD.GO

Lexer

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


The program is lexically correct

Tokens summary written to: C:\Users\j
ocel\unam.fi.compilers.g5.05\unam\fi\
compilers\g5\05\compiler\src\hellowor
ld.txt
```

Parser

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

Parse tree appended to: C:\User
s\jocel\unam.fi.compilers.g5.05
\unam\fi\compilers\g5\05\compil
er\src\helloworld.txt
```

Semantic

```
5\05\compiler\src\m[SymbolTable] Declared 'fmt' as '(Simpl
eType int)'
[SymbolTable] Declared 'fmt' as '(SimpleType int)'
[Semantic] Parameter 'fmt' declared with type '(SimpleType
 [SymbolTable] Declared 'fmt' as '(SimpleType int)'
[Semantic] Parameter 'fmt' declared with type '(SimpleType
[SymbolTable] Declared 'fmt' as '(SimpleType int)'
[Semantic] Parameter 'fmt' declared with type '(SimpleType
 int)'
[SymbolTable] Declared 'main' as '(SimpleType function)'
[SymbolTable] > Entering new scope (level 2)
[SymbolTable] > Entering new scope (level 3)
**********
[SymbolTable] Declared 'Println' as '(SimpleType int)'
[Semantic] Parameter 'Println' declared with type '(Simple
Type int)'
[SymbolTable] < Exiting scope (returning to level 2)

SDT Verified!
```

HELLOWORLD.TXT

unam > fi > compilers > g5 > 05 > compiler > src > 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!")))))))))
```

RESULTS

STAGES FOR SYNTAXERR.GO (SYNTAX ERROR)

Source

unam > fi > compilers > g5 > 05 > compiler

```
1   package main
2
3   func main() {
4       var x int = 5
5       var y int = 10
6       var z int = x + *y //
7   }
8
```

Lexer

```
er/src/main.py
Enter the source file's name: ..\test\sy
ntaxerr.go


The program is lexically correct

Tokens summary written to: C:\Users\joce
l\unam.fi.compilers.g5.05\unam\fi\compil
ers\g5\05\compiler\src\syntaxerr.txt
```

Parser

```
ERROR SINTACTICO: No se esperaba encontrar
el Token '*' en la línea '6' columna '18'
Traceback (most recent call last):
8'
Traceback (most recent call last):
Traceback (most recent call last):
  File "c:\Users\jocel\unam.fi.compilers.g5
.05\unam\fi\compilers\g5\05\compiler\src\ma
in.py", line 115, in <module>
    main()
  File "c:\Users\jocel\unam.fi.compilers.g5
.05\unam\fi\compilers\g5\05\compiler\src\ma
in.py", line 73, in main
    parse_tree = parser.parse(lexer.lex(sou
rce_code))
                 ^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^
```

SYNTAXERR.GO

unam > fi > compilers > g5 > 05 > compiler > src > syntaxerr.txt

```
1
2
3   ------------ TOKEN SUMMARY ------------
4   Keywords (5): package func var
5   Identifiers (7): main x y z
6   Delimiters (4): ( ) { }
7   Types (3): int
8   Operators (5): = + *
9   Numbers (2): 5 10
10
11  Total tokens: 26
12
```

RESULTS

STAGES FOR TYPE_MISMATCH. GO

Lexer

```
pilers/g5/05/compiler/src/main.py
Enter the source file's name: ..\
test\type_mismatch.go


The program is lexically correct


Tokens summary written to: C:\Use
rs\jocel\unam.fi.compilers.g5.05\
unam\fi\compilers\g5\05\compiler\
src\type_mismatch.txt
```

Parser

```
Starting Parsing (Syntactic)...

Parsing Success!

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

Semantic

```
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 'st
ring' to variable 'x' of type 'int'
```

TYPE_MISMATCH.TXT

unam > fi > compilers > g5 > 05 > compiler > src > type_mismatch.txt

```
1
2
3   ------------ TOKEN SUMMARY ------------
4   Keywords (3): package func var
5   Identifiers (4): main x
6   Delimiters (4): ( ) { }
7   Types (1): int
8   Operators (1): =
9   Strings (1): "hola"
10
11  Total tokens: 14
12
13
14  ---------- PARSE TREE (NTLK STYLE) ----------
15  (SourceFile
16    (PackageClause (package ) (Identifier main))
17    (ImportDecls )
18    (TopLevelDecls
19      (TopLevelDecl
20        (FunctionDecl
21          (Identifier main)
22          (Signature (Parameters ) (Result ))
23          (Block
24            (StatementList
25              (DeclStmt
26                (VarDecl
27                  (VarSpecList
28                    (VarSpec
29                      (IdentifierList (Identifier x))
30                      (SimpleType int)
31                      (ExpressionList )))))
32              (AssignStmt
33                (ExpressionList (Identifier x))
34                (AssignOp =)
35                (ExpressionList (StringLiteral "hola")))))))))
```

ADVANTAGES AND DIFFERENTIATORS

Modular Design

The system follows a modular architecture, facilitating maintenance and extensibility

Easy debugging

Each phase of the program handles errors in a way that can be easily identified and corrected by the user.

Transparent Execution

Every phase of the compiler provides a summary of it's results, ensuring traceability throughout the compilation process.

RPLY

Both the lexer and parser in our project are implemented using RPLY, a powerful Python library designed for parser generation. RPLY employs an efficient LALR(1) parsing algorithm, ensuring high performance and reliable analysis.

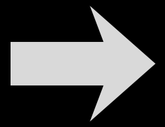
RPLY serves as a simplified implementation of PLY and is based on the traditional lex and yacc compiler construction tools.

https://rply.readthedocs.io/en/latest/

LEXER

- Implemented using RPLY's lexer generator
- REGEX based token detection
- Breaks down the source code into tokens that can be easily digested by the parser.
- Outputs a token summary for visualization and further debugging.

Lexical errors can be detected during the lexer’s execution so that only lexically correct source code is passed to the parser.

A = 2  TOKEN(Identifier, 'A') TOKEN(OP_EQ, "=") TOKEN(LIT_INT, '2')

A = 2+1 ✓

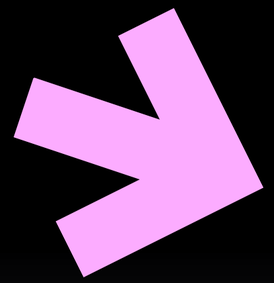
A # 2+1 X

Source code tokenization

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```



```
Token('KW_PACKAGE', 'package')
Token('IDENT', 'main')
Token('KW_IMPORT', 'import')
Token('LIT_STR', '"fmt"')
Token('KW_FUNC', 'func')
Token('IDENT', 'main')
Token('PUNC_LPAREN', '(')
Token('PUNC_RPAREN', ')')
Token('PUNC_LBRACE', '{')
Token('IDENT', 'fmt')
Token('OP_DOT', '.')
Token('IDENT', 'Println')
Token('PUNC_LPAREN', '(')
Token('LIT_STR', '"Hello, World!"')
Token('PUNC_RPAREN', ')')
Token('PUNC_RBRACE', '}')
```

PARSING

Our parser uses Rply and nltk.Tree. Rply analyzes the sequence of tokens and generates an abstract syntax tree (AST), while nltk.Tree allows us to represent the syntax tree in a hierarchical and manipulable way in Python.

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

Total tokens: 16
```

```
---------- PARSE TREE (NTLK STYLE) ----------
(SourceFile
  (PackageClause (package ) (Identifier main))
  (ImportDecls (ImportDecl (ImportSpec (path fmt))))
  (TopLevelDecls
    (TopLevelDecl
      (FunctionDecl
        (Identifier main)
        (Signature (Parameters ) (Result ))
        (Block
          (StatementList
            (ExprStmt
              (CallExpr
                (QualifiedIdent
                  (Identifier fmt)
                  (Identifier Println))
                (ArgumentList (StringLiteral "Hello, World!")))))))))
```

VarDecl

```
var n2, n1 int = 0, 1
```

```
(VarDecl
  (VarSpecList
    (VarSpec
      (IdentifierList (Identifier n2) (Identifier n1))
      (SimpleType int)
      (ExpressionList (IntLiteral 0) (IntLiteral 1))))))
```

Parameters

```
func fibonacciIterative(n int) int {
```

```
(Signature
  (Parameters
    (ParameterDecl (Identifier n) (SimpleType int)))
  (Result (SimpleType int)))
```

ShortVarDecl

```
for i := 2; i <= n; i++ {
```

```
(ShortVarDecl
  (IdentifierList (Identifier i))
  (ExpressionList (IntLiteral 2)))
```

For

```
(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))))))
```

If

```
(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))))))
```

arithmetic operations

```
(BinaryExpr
  (Identifier n1)
  (Operator +)
  (Identifier n2))))
```

SEMANTIC ANALYSIS

VISITOR + SYMBOLTABLE

This stage ensures that the parsed code is not only syntactically correct but also semantically coherent.

We leverage the following components:

- **Visitor Pattern**
- **SymbolTable**

- **Semantic Correctness**
- **Scope Handling**
- **Type Consistency**

UNDER THE HOOD
THE LOGICAL ENGINE

How our compiler “understads what the code actually means

VISITOR PATTERN

“The Inspector”

- Walks the tree
- Recognizes meaning
- Enables growth

SYMBOL TABLE

“The Memory”

- Keeps declarations
- Tracks types and scope
- Prevents misuse

Thank **You**

-