UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Computer Engineering

Compilers

Interpreter

Students:

ID: 319069437

ID: 319321674

ID: 319302149

ID: 319133756

ID: 422022569

Group: 05

Semester: 2025-1

Mexico, CDMX. November 2024

Índex

Introduction

In this project, we reach the final stage of developing a fully functional interpreter, culminating the work carried out in the previous phases. This final phase integrates the key components built earlier —specifically the lexer and the parser/semantic analysis modules—into a functional system. Together, these components enable the interpreter to bridge the gap between input code and executable actions, marking a significant milestone in the computational process

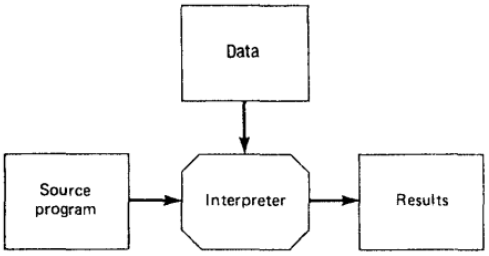
The lexer, responsible for breaking down the input code into manageable tokens, lays the foundation for the interpreter by performing lexical analysis. This step ensures that the raw text is segmented into meaningful units —such as keywords, identifiers, literals, and operators—which are essential for further processing. Building upon this, the parser performs syntactic analysis by organising the tokenized input into a syntax tree, which represents the logical structure of the program. This tree serves as a blueprint for the interpreter, outlining how operations are organised and executed.

By integrating these critical components, the interpreter evolves into a system capable of understanding the language's syntax and translating it into actionable operations. This process involves interpreting the syntax tree and executing the corresponding instructions, thereby transforming abstract code into functional output. Through this project, the theoretical and practical aspects of compiler design converge, demonstrating how individual components like lexers and parsers collaborate to achieve the overarching goal of interpreting and executing code.

Theoretical Framework

About an interpreter

An interpreter is a kind of translator (a *translator* inputs and then converts a source program into an object or target program) that processes an internal form of the source program and data at the same time. That is, interpretation of the internal source form occurs at run time and no object program is generated. The next figure illustrates this interpretative process.



Some interpreters analyse each source statement every time it is to be executed.

How does an interpreter work?
Interpreters typically work in two phases:

Parsing: The interpreter parses the source code and breaks it down into its component parts, that is, into tokens.

Execution: The interpreter executes the source code line by line. This includes executing the instruction in the source code, as well as managing memory and other resources.

Something important to clarify is that interpreters cannot generate machine code. This means that the source code is executed directly by the interpreter, and the interpreter must translate the source code into machine code on the fly.

What is constant folding?

It is a technique that evaluates constant expressions at compile time and replaces them with their computed value.

What are Compact Instructions?

The process of optimizing the set of instructions generated during compilation to make them more compact or efficient.

What is Bytecode Storage?

The storage of the compiled bytecode, which is the intermediate code that the Python Virtual Machine (PVM) executes.

What is Dead Code Elimination?

An optimization process that removes code that does not execute.

What is Literal Merging?

An optimization that combines identical literals into a single instance to reduce memory usage.

What is Bindings?

The association between a name and its value or memory address.

Interpreter Development

Input

The source code provided is taken to be executed. This code is typically provided in the following ways:

Source Code (.c): The (.c) file containing the code written by the user is the most common form of input. The interpreter reads this file and processes it line by line.

Interactive Input: In interactive mode, the interpreter receives lines of code interactively through the console or terminal. The user types commands or code snippets, and the interpreter evaluates them and displays the results immediately.

Intermediate code

1. **Source Code Analysis**
    - **Lexical Analysis:**

Theoretical explanation

PLY (Python Lex-Yacc) is an implementation of traditional lexical and syntactic analysis tools (Lex and Yacc) in Python. Lex is responsible for lexical analysis, where the input text is divided into tokens based on a set of rules.

Here is a detailed description of how PLY Lex is used to build a lexical analyzer:

To use PLY in a Python script, you first need to make sure the library is installed. Then, you should import the **ply.lex** module at the beginning of your script.

If you haven't installed PLY yet, you can do so using pip. Open a terminal or command prompt and run: pip install ply

At the beginning of your Python script, you should import the **ply.lex** module for lexical analysis: import ply.lex as lex

After importing the module, you can define your tokens and lexical rules.

List of Keywords: For reserved words we define a list containing all keywords that this lexical analyzer will accept, which belong to the C language. This list is important, as it includes the conditional structures, data types and functions that will be valid in the lexical analyzer, and in the future, in the compiler.

$$keywords = \{else,\ float,\ if,\ int,\ exp,\ sqr,\ printf\}$$

Token list: For this project, several types of different tokens were defined, and these are grouped into a list that will store them, which are:

KEYWORD: Are reserved words that have a special meaning and cannot be used as identifiers.

IDENTIFIER: Names used to identify variables, functions, data types, or other elements defined by the user in the program. An identifier must comply with the naming rules of the programming language and we will check this in its construction.

NUMBER: Represents a literal number.

PLUS: Represents the addition operator (+).

MINUS: Represents the subtraction operator (-).

TIMES: Represents the multiplication operator (*).

DIVIDE: Represents the division operator (/).

LPAREN: Represents the left parenthesis (**(**), used for grouping expressions or parameters.

RPAREN: Represents the right parenthesis (**)**), used to close a group of expressions or parameters.

OCURLB: Represents the left curly brace ({), used in code blocks.

CCURLB: Represents the right curly brace (}), used to close code blocks.

SEMIC: Represents the semicolon (;), used to terminate declarations or statements.

TYPE: Represents data types (int, float).

EQUALS: Represents the assignment operator (=).

NS: Represents the (#).

PRINT: Represents the function or command to print something to the standard output.

EXP: Represents the exponentiation operation, such as exp().

SQR: Represents the square root operation.

IF: Represents the conditional if statement, used for making decisions based on conditions.

ELSE: Represents the alternative part of an if statement, executed if the condition is false.

VAL_BOOL: Represents a boolean value (true or false).

OP_BOOL: Represents logical operators, such as AND, OR, NOT.

STRING: Represents string literals, typically delimited by quotes ("text").

DIRECTIVE: These are special instructions to the preprocessor in some programming languages. Directives are not part of the language itself, but rather control the compilation process.

Specification of tokens with regular expressions: For each type of token we must specify the regular expressions with which they will be formed. They are declared using the t_ prefix to indicate that it is a token.

There are the following regular expressions and its automatons:

KEYWORDS

$$\wedge[else|float|if|int|void|printf|exp|sqr]\$$$

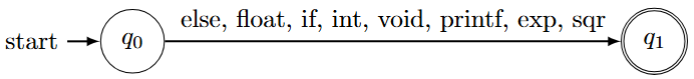


Figure 1: Automaton for recognising keywords

OPERATORS

$$\wedge[+ - */]\$$$

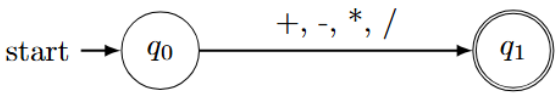


Figure 2: Automaton for recognising operators

PUNCTUATIONS

$$\wedge[(){};]\$$$

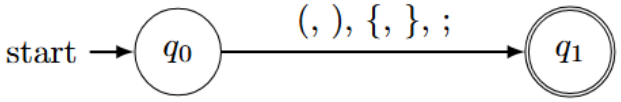


Figure 3: Automaton for recognising punctuations symbols

BOOLEAN OPERATORS

$$\wedge[<= \mid >= \mid == \mid < \mid >]\$$$

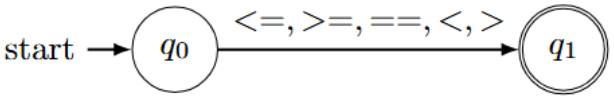


Figure 4: Automaton for recognising boolean operators

IDENTIFIERS

$$\wedge[a - zA - Z\_][a - zA - Z0 - 9\_] * \$$$

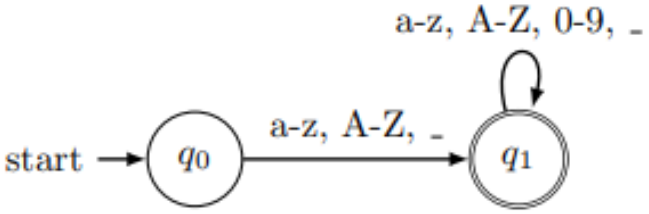


Figure 5: Automaton for recognising identifiers

NUMBERS

$$\wedge[\backslash d + (.\backslash d+)?]\$$

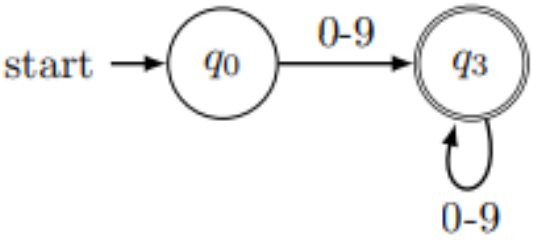


Figure 6: Automaton for recognising numbers

STRINGS

$$\wedge[\backslash"([\wedge\backslash\backslash n]|(\backslash\backslash.))*?\backslash"|\backslash'([\wedge\backslash\backslash n]|(\backslash\backslash.))*?\backslash']\$$$

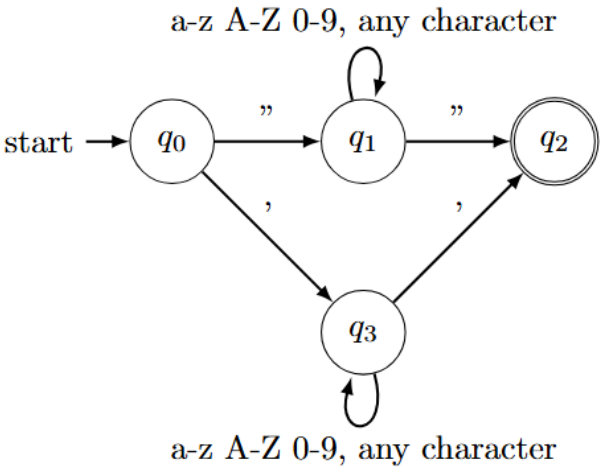


Figure 7: Automaton for recognising strings

LIBRARIES

$$\wedge[\backslash b(? : (stdio.h|math.h))\backslash b]\$$

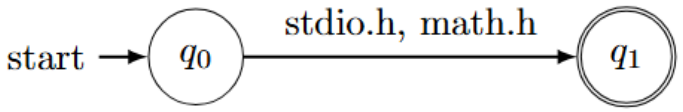


Figure 8: Automaton for recognising language libraries

DIRECTIVES

$$\wedge[\backslash b(? : (include))\backslash b]\$$

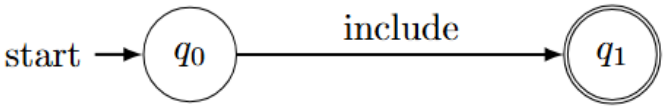


Figure 9: Automaton for recognising language directives

BOOLEAN VALUES

$$\wedge[\backslash b(? : (True|False))\backslash b]\$$

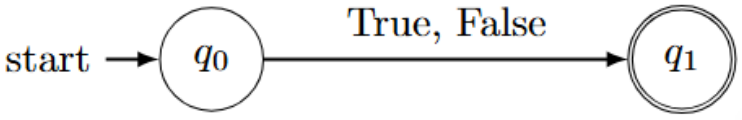


Figure 10: Automaton for recognising boolean values

TYPE

$$\wedge[\backslash b(? : (int|float))\backslash b]\$$

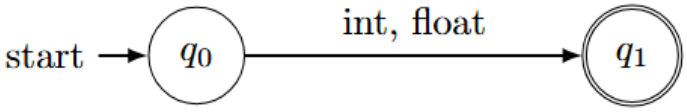


Figure 11: Automaton for recognising language data types

Discarded tokens: It is important to declare the "tokens" that will be ignored by the analyzer, as they are used only for formatting or providing additional information to the programmer and not for the program itself, such as comments, spaces, or tabs. When the analyzer identifies them, it will simply skip them. We declare this type of "special token" with the following regular expressions.

Line numbers: does not have knowledge of the number of lines in the code because it does not have a defined way to indicate a "new line." Therefore, we need to define it as another token, which we will call "newline," and we detect it with the following regular expression.

Error handling: The library provides us with a function that is activated when an element does not match any of the declared token regular expressions. In our case, we decided to send a message indicating to the user that the element will be ignored.

Build the lexer: To create a lexer, the library provides us with the lex.lex() function. When creating it, Python reads and uses the regular expressions we defined and constructs the lexer. Once this is done, it allows us to use other functions with the created lexer.

Code Analysis Function: The function responsible for analyzing the code it receives as a parameter, which in this case we named data, uses the input function and passes the information received to it, in this case using lexer.input(data). This function resets the lexer and stores a new input.

We create empty sets to store the tokens that belong to each of the designated types.

Next, we use a while loop, which starts by obtaining the next token. This is achieved with the lexer.token() function, which returns the next identified token. If no such token exists, the loop will terminate, indicating that the code being analyzed has finished. If the token exists, it is classified using a series of if statements, which compare the type with the different token types, and if they match, it is stored in the sets we created earlier.

Once this cycle is complete, which happens when all tokens have been classified and no more tokens are found, the function will proceed to print a classification of token types and the tokens recorded for each type.

File Reading Function: This function takes the path of a file as a parameter, opens it in read mode, and returns the string of what was read.

Main Function: As the main process, we start by printing a screen message asking the user whether they want to enter the code manually or prefer to use a file.

If the user chooses to type the code, they are prompted to enter multiple lines of text, which are stored in a list. This continues until the user presses Enter twice. To finish, all lines are concatenated and saved in the code variable.

On the other hand, if the user opts to use a file, they are asked to provide the file path. The program checks if the document exists, and if so, it uses the leer_archivo function, passing the path as a parameter, and stores the returned result in the code variable. If the file does not exist, the user is notified

- **Syntactic Analysis:**

After our lexer finishes identifying and classifying lexemes into tokens, this stream can be sent to the parser to begin the second phase of compilation and generate the parse tree.

Context-Free Grammar

For the construction of out parse tree, we proposed the following CFG:

$$program \rightarrow code \mid \ < empty >$$
$$code \rightarrow code\ statement \mid statement$$
$$statement \rightarrow declaration \mid assignment \mid prt \mid directives \mid ifst$$
$$directives \rightarrow NS\ DIRECTIVES\ LIBRARIER$$
$$declaration \rightarrow TYPE\ IDENTIFIER\ SEMIC \mid TYPE\ IDENTIFIER\ EQUALS\ expression\ SEMIC$$
$$assignment \rightarrow IDENTIFIER\ EQUALS\ expression\ SEMIC$$
$$prt \rightarrow PRINT\ LPREN\ expression\ RPAREN\ SEMIC$$
$$ifst \rightarrow IF\ LPAREN\ valbool\ RPAREN\ OCURLB\ program\ CCURLB$$
$$ifst \rightarrow IF\ LPAREN\ valbool\ RPAREN\ OCURLB\ program\ CCURLB\ ELSE\ OCURLB\ program\ CCURLB$$
$$expression \rightarrow expression\ PLUS\ term \mid expression\ MINUS\ term$$
$$term \rightarrow term\ TIMES\ factor \mid term\ DIVIDE\ factor$$
$$factor \rightarrow EXP\ LPAREN\ factor\ value\ RPAREN \mid SQR\ LPAREN\ factor\ RPAREN$$
$$value \rightarrow NUMBER$$
$$valbool \rightarrow VAL\_BOOL \mid LPAREN\ valbool\ RPAREN \mid expression\ OP\_BOOL\ expression$$
$$expression \rightarrow term$$
$$term \rightarrow factor$$
$$factor \rightarrow value \mid IDENTIFIER \mid LPAREN\ expression\ RPAREN \mid MINUS\ factor$$

For further analysis, we have defined the following rules for our program:

General Program Structure: The program can consist of code (a set of statements) or be empty (<empty>). Code represents a list of statements (rules 3 and 4).

Types of Statements: Declaration (declaration), assignment (assignment), print (prt), directives (directives), and conditional structures (ifst).

Declarations and Assignments: Declaration defines variables of a specific type and optionally initializes them (rules 11 and 12). Assignment performs a simple assignment of an expression to an identifier (rule 13).

Print Statement: The print statement (prt) allows printing an expression using the structure PRINT(expression) (rule 14).

Conditionals: Ifst defines if and if-else structures, which accept boolean expressions (valbool) as conditions and allow code blocks in {} (rules 15 and 16).

Expressions and Operators: Expression enables arithmetic operations (+, -) and term handles multiplication and division operations (*, /). Support is provided for functions like exponential and square root (rules 17-23). Valbool defines boolean conditions, including boolean operators (OP_BOOL).

Boolean Operators and Parentheses: Simple and compound boolean expressions (valbool) are allowed, as well as nested expressions with parentheses (rules 24-26).

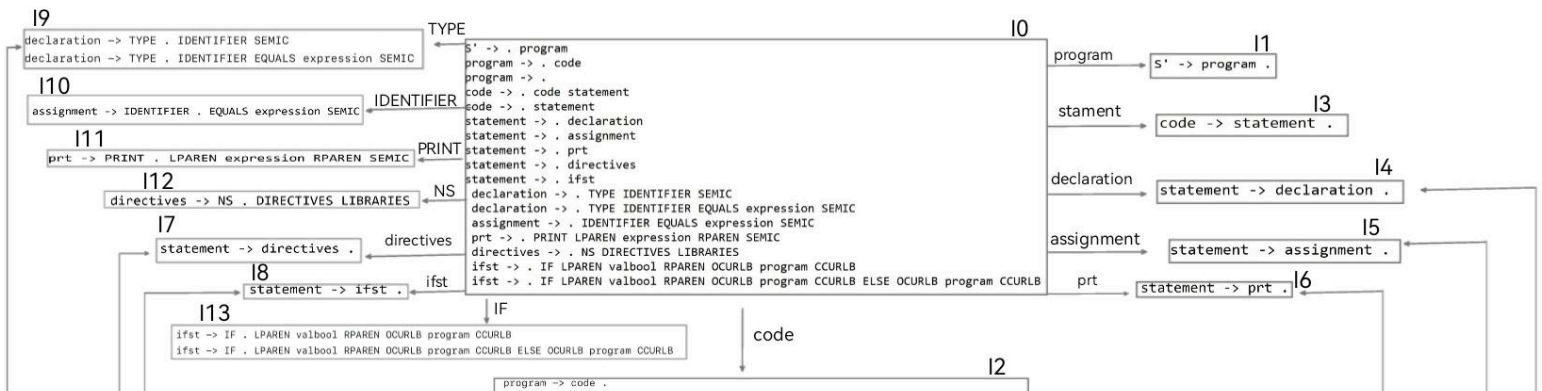
Expression Levels: The different levels of expression, term, and factor follow operation precedence (rules 27-32), handling operators and elements such as numbers, identifiers, and grouped expressions.

After determining the grammar that the parser will use during syntactic analysis, it is important to define the parsing method. From the early stages of this project, we used the PLY module in Python, which is an implementation of compiler construction tools lex and yacc. In the latter, the bottom-up LALR(1) parsing method is used, allowing us to subsequently obtain the canonical collection of LR(1) items, which, as a reminder, is the collection of canonical LR(0) items plus a lookahead.

We will need the augmented grammar, which is simply the grammar defined in the lexical analyzer section plus an additional start symbol production rule. We will not rewrite the entire grammar, only the first five production rules. We leave it to the reader to envision the entire grammar:

$$i.\quad program' \rightarrow program$$
$$ii.\quad program \rightarrow code \mid < empty >$$
$$iii.\quad code \rightarrow code\ statement \mid statement$$
$$iv.\quad statement \rightarrow declaration \mid assignment \mid prt \mid directives \mid ifst$$
$$v.\quad directives \rightarrow NS\ DIRECTIVES\ LIBRARIER$$
$$.$$
$$.$$
$$.$$

The canonical collection of LR(1) items is as follow:



The parsing table obtained by the canonical collection is as follow:

STATES	ACTION									
	TYPE	IDENTIFIER	PRINT	IF	NS	LPAREN	EQUALS	CCURLB	DIRECTIVES	$
0	s9	s10	s11	s13	s12					r2
1										
2	s9	s10	s11	s13	s12			r1		r1
3	r4	r4	r4	r4	r4			r4		r4
4	r5	r5	r5	r5	r5			r5		r5
5	r6	r6	r6	r6	r6			r6		r6
6	r7	r7	r7	r7	r7			r7		r7
7	r8	r8	r8	r8	r8			r8		r8
8	r9	r9	r9	r9	r9			r9		r9
9		s15								
10							s16			
11						s17				
12									s18	
13						s19				
14	r3	r3	r3	r3	r3			r3		r3

STATES	GOTO							
	program	code	statement	declaration	assignment	prt	directives	ifst
0	1	2	3	4	5	6	7	8
1								
2			14	4	5	6	7	8
3								
4								
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								

In both the Canonical Collection of LR(1) items and the parsing table, only certain states are considered, as the complete grammar produces a total of 73 states. Therefore, for illustrative purposes, we decided it would be convenient to represent only a few states.

- **Semantic Analysis:**

As the parse tree is constructed, semantic rules are applied to ensure the meaning of our program and to prevent actions that are incorrect within the context of the programming language.

Semantic analysis is integrated directly into the syntax parsing phase using the Yacc parser generator. Semantic rules are added to the parsing rules to check and enforce language constraints, such as variable types and declarations.

The parser not only identifies the syntactic structure of the code but also performs semantic checks during parsing. This is achieved by embedding semantic rules within the parsing functions, which manage both syntax validation and the meaning of each construct.

Syntax-Directed Translation

To ensure the semantic meaning of the statements that make up our program, we have defined the following semantic rules in some production rules to achieve the SDT.

As a note, we define p as a statement within the programming language. E.g. $int\ a = 3 + 5;$

For *declaration rules:*

$$var\_type = p[1]$$
$$var\_name = p[2]$$
$$if\ var\_name\ in\ symbol\_table:$$
$$print("Semantic\ error")$$
$$if\ len(p) > 4\ and\ p[3] == ' = ':$$
$$if\ p[4] == none:$$
$$print("Error:\ asignación\ de\ valor\ nulo\ a\ una\ variable")$$
$$elif\ (p[1] == int):$$
$$valueNum = abs(p[4])$$
$$if\ (valeNum - int(valueNum)\ != 0):$$
$$print("Error:\ asignación\ de\ flotante\ a\ variable\ de\ tipo\ entero")$$
$$else:$$
$$symbol\_table[var\_name] = \{'Identifier': var\_name, 'Type': var\_type, 'Value': int(p[4])\}$$
$$else:$$
$$symbol\_table[var\_name] = \{'Identifier': var\_name, 'Type': var\_type, 'Value': p[4]\}$$
$$else:$$
$$symbol\_table[var\_name] = \{'Identifier': var\_name, 'Type': var\_type, 'Value': none\}$$

For *assignment rules:*

$$var\_name = p[1]$$

```
if var_name not in symbol_table:
    print("Error: Variable ya declarada")
else:
    valueNum = abs(p[3])
    if (symbol_table[var_name]['Type']) == 'int':
        if (valueNum - int(valueNum) != 0):
            print(f"Error: No se puede asignar un flotante en una variable entera")
        else:
            symbol_table[var_name]['Value'] = int( p[3])
    else:
        symbol_table[var_name] = p[3]
```

For *expression_plus:*

```
if((p[1] != None) and ((p[3] != None))):
    p[0] = p[1] + p[3]
else:
    print(f"Error: No se pueden sumar variables nulas")
```

For *expression_minus:*

```
if((p[1] != None) and ((p[3] != None))):
    p[0] = p[1] - p[3]
else:
    print(f"Error: No se pueden restar variables nulas")
```

For *term_times:*

```
if((p[1] != None) and ((p[3] != None))):
    p[0] = p[1] * p[3]
else:
    print(f"Error: No se pueden multiplicar variables nulas")
```

For *term_div:*

```
if((p[1] != None) and ((p[3] != None))):
    p[0] = p[1] / p[3]
else:
    print(f"Error: No se pueden dividir variables nulas")
```

For *factor_exp:*

```
if((p[3] != None) and ((p[4] != None))):
    p[0] = p[3] ** p[4]
else:
```

```
print(f"Error: No se pueden elevar a la potencia variables nulas")
```

For *factor_sqr:*

```
if(p[3] != None):
    p[0] = math.sqrt(p[3])
else:
    print(f"Error: No se pueden elevar a la potencia variables nulas")
```

For *values_num:*

```
p[0] = p[1]
```

For *value_bool:*

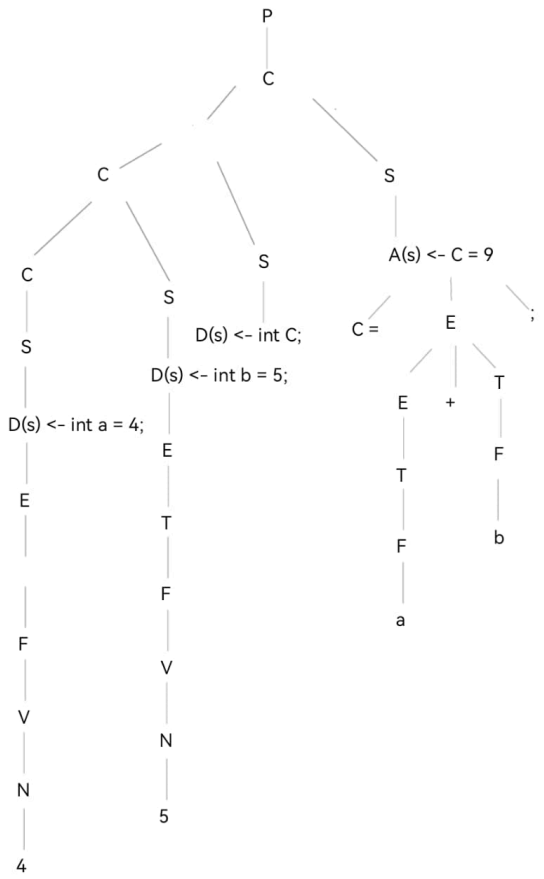
```
if (p[1] == 'True' or p[1] == 'False'):
    Val = p[1]
elif p[2] == ' < ':
    Val = p[1] < p[3]
elif p[2] == ' > ':
    Val = p[1] > p[3]
elif (p[2] == ' <= '):
    Val = p[1] <= p[3]
elif (p[2] == ' >= '):
    Val = p[1] >= p[3]
elif (p[2] == ' == '):
    Val = p[1] == p[3]
else:
    Val = p[2]
p[0] = Val
```

For *factor_value*:

```
if p[1] == ' - ':
    p[0] = (- 1) * p[2]
elif p[1] == '(':
    p[0] = p[2]
elif not str(p[1])[0].isnumeric():
    var_name = p[1]
    if not var_name in symbol_table:
        print(f"Error: La variable '{var_name}' no existe declarada.")
    else:
        ref_val = symbol_table[var_name]['Value']
        p[0] = ref_val
else:
    p[0] = p[1]
```

How does syntactic and semantic analysis work with an input string?

Bottom-up-like parse tree



The tokens generated for the input presented in the lexical analysis section are:

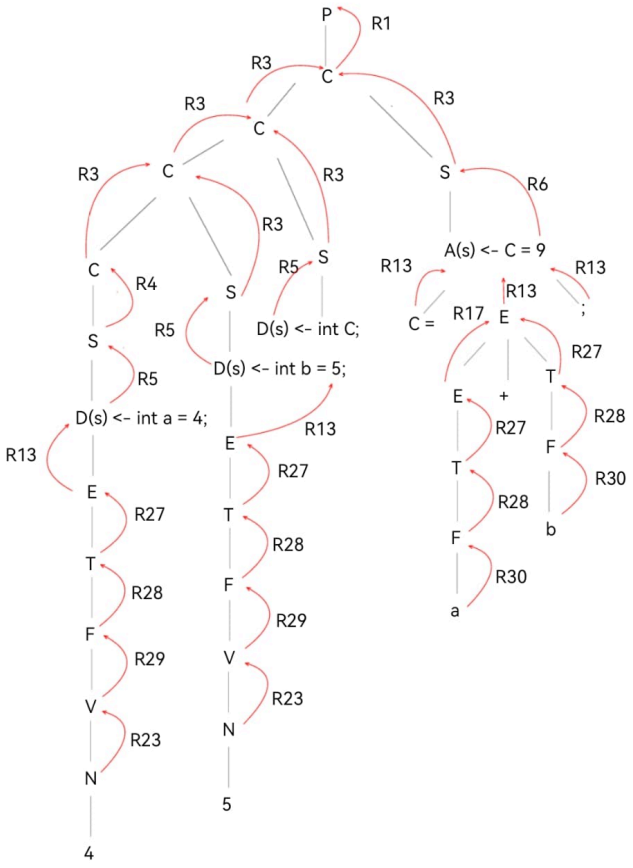
The parsing table for the input is as follow:

Input	Stack	State	Action
TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER SEMIC TYPE IDENTIFIER EQUALS expression SEMIC $	$	0	shift and go to state 9
IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER SEMIC TYPE IDENTIFIER EQUALS expression SEMIC $	$ TYPE	9	shift and go to state 15

EQUALS expression SEMIC TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER SEMIC TYPE IDENTIFIER EQUALS expression SEMIC $	$ TYPE IDENTIFIER	15	shift and go to state 21
expression SEMIC TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER SEMIC TYPE IDENTIFIER EQUALS expression SEMIC $	$ TYPE IDENTIFIER EQUALS	21	shift and go to state 38
SEMIC TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER SEMIC TYPE IDENTIFIER EQUALS expression SEMIC $	$ TYPE IDENTIFIER EQUALS expression	38	shift and go to state 53
TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER SEMIC TYPE IDENTIFIER EQUALS expression SEMIC $	$ TYPE IDENTIFIER EQUALS expression SEMIC	53	reduce using rule 12 shift and go to state 9
IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER SEMIC TYPE IDENTIFIER EQUALS expression SEMIC $	$ TYPE IDENTIFIER EQUALS expression SEMIC TYPE	9	shift and go to state 15
EQUALS expression SEMIC TYPE IDENTIFIER SEMIC TYPE IDENTIFIER EQUALS expression SEMIC $	$ TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER	15	shift and go to state 21
expression SEMIC TYPE IDENTIFIER SEMIC TYPE IDENTIFIER EQUALS expression SEMIC $	$ TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER EQUALS	21	shift and go to state 38
SEMIC TYPE IDENTIFIER SEMIC TYPE IDENTIFIER EQUALS expression SEMIC	$ TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER EQUALS expression	38	shift and go to state 53

$			
TYPE IDENTIFIER SEMIC TYPE IDENTIFIER EQUALS expression SEMIC $	$ TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER EQUALS expression SEMIC	53	reduce using rule 12 shift and go to state 9
IDENTIFIER SEMIC TYPE IDENTIFIER EQUALS expression SEMIC $	$ TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER EQUALS expression SEMIC TYPE	9	shift and go to state 15
SEMIC TYPE IDENTIFIER EQUALS expression SEMIC $	$ TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER	15	shift and go to state 20
TYPE IDENTIFIER EQUALS expression SEMIC $	$ TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER EQUALS expression SEMIC TYPE IDENTIFIER SEMIC	20	reduce using rule 11

Representation of the use of the simplified rules



2. Bytecode Generation

Once the code has been syntactically and semantically analysed, the next step is bytecode generation. In this phase, the AST is transformed into a set of bytecode instructions.

The conversion of the AST to bytecode involves taking each of these AST nodes and mapping them to a lower-level instruction (bytecode) that can be executed.

AST Traversal (Traversal of the AST)

The first step in bytecode generation is traversing the AST. The traversal is done in a pre-order fashion, where the current node is processed before its children (for expressions, for example, the operands are evaluated first, followed by the operation).

Mapping AST Operations to Bytecode Instructions
For each AST node, a set of bytecode instructions is generated that corresponds to the operation represented by that node. This set of instructions will be executed sequentially.

- **Compact Instructions:**

The bytecode size was optimised by compressing the instructions and eliminating redundancies, significantly reducing the amount of code that needs to be interpreted by the interpreter. This not only saves disk space but also improves performance, as it decreases the amount of data the interpreter needs to read and process.

Despite making the instructions more compact, we maintain full functionality. For example, a compact instruction can perform multiple operations in a single step, such as loading a value onto the stack and performing an additional operation with that value, all in one instruction.

Additionally, we have simplified the processing. Compact instructions reduce the number of processing cycles required to execute each operation. This means that instead of performing several intermediate steps, a compact instruction executes in a single step, making the PVM process much more efficient.

- **Bytecode Storage:**

The bytecode is stored to be used without the need to recompile or reinterpret the source code each time. It is stored in a way that allows the interpreter to access and execute it directly.

Automatic Storage: Python automatically stores the compiled bytecode in a directory called __pycache__ within the same directory as the source code file. The bytecode is saved with the name <module_name>.cpython-<version>.pyc, where <version> refers to the Python version used to compile the code (e.g., cpython-38 for Python 3.8).

Accessing Stored Bytecode: When the program is run again, Python checks whether there is an existing .pyc file for the script. If the bytecode exists and is up-to-date (i.e., the source

code has not been modified since the last compilation), Python loads and executes this bytecode directly instead of recompiling the source code.

3. **Bytecode Optimization**
    - **Constant Folding:**

When the interpreter encounters expressions involving only constants, such as arithmetic operations or string concatenation, it evaluates them at compile-time and replaces the expression with the result. Constant Folding is implemented in intermediate code generation. The goal is to avoid redundant calculations at runtime by evaluating them before interpreting the code. When the interpreter executes the code, it no longer evaluates these expressions because they have been replaced by their value.

- **Dead Code Elimination:**

The interpreter analyzes the program's flow and removes any code that cannot be executed due to its location or prior conditions. This reduces the size of the generated bytecode and improves efficiency. When the interpreter does not perform a prior optimization, it can avoid executing unreachable parts by processing the program's dynamic flow.

- **Literal Merging:**

Instead of storing multiple copies of the same literals in the symbol table, Python merges them into a single entry. This reduces memory usage and improves access to literals at runtime. When the interpreter detects that a literal (such as a number or a string) is repeated in various parts of the code, it generates a single instance of the literal in the symbol table and reuses that instance in the bytecode.

4. **Execution by the interpreter**

The PVM (Python Virtual Machine) is responsible for interpreting and executing the bytecode. Although bytecode is a step closer to actual execution, the PVM still needs to translate it into operations that the operating system and hardware can execute. The PVM is the layer that enables Python to be an interpreted language, as it is not compiled into machine code all at once but is executed step by step.

- **Execution:**

Interpretation: The PVM reads and executes the bytecode instructions one by one. Each bytecode instruction represents an operation, such as assigning a value to a variable, performing a mathematical operation, making a function call, etc.

Stack and Variables: During execution, the PVM uses an execution stack to manage temporary values and operations. Each time the bytecode requires calculating an expression or storing a value, the PVM places the values onto the stack and performs the operations.

Function Calls: When instructions to call functions or methods are encountered, the PVM also manages the function context (variable creation, etc.) and handles the unpacking and packing of the necessary variables in the execution environment.

Interaction with the Operating System: Although the PVM interprets the bytecode, it ultimately needs to interact with the operating system to perform operations involving input/output, such as reading a file or displaying something on the screen.Python has a bindings layer that translates these interactions into system calls via the C interface.

- **Bindings :**

These bindings allow Python to call functions and use data structures from these external libraries as if they were native Python functions or types.

In simpler terms, bindings enable Python to "talk" to other libraries or the operating system, which are written in languages closer to the hardware, like C, without having to rewrite all the code in Python.

How do bindings work?

System Calls: Many operations in Python, such as reading or writing files, accessing the network, or interacting with hardware, require the ability to make system calls. However, Python is a high-level language and does not have direct access to these operations. To handle this, Python uses C libraries designed to interact with the operating system.

C Interface (C API): Python is designed to integrate closely with C. The Python C API provides functions that allow Python modules to call C functions, access data in C memory, and handle C data structures.

- Python code calls functions through the C interface.
- The Python C interface translates these calls into something the operating system can understand (system calls or calls to other C libraries).
- The results are then returned to Python in a format the language can process.

C Extensions: When a Python module needs to interact with C libraries (for example, a module offering low-level system functionality or using high-performance algorithms written in C), this module uses bindings to translate functions and data between Python and C.

Using ctypes or Cython

Python provides several ways to interact with C code. Some common ones include:

- ctypes: A standard Python module that allows you to call functions in shared libraries (such as .dll files on Windows or .so files on Linux) directly from Python. ctypes provides a way to declare C data types and use C functions in Python code.
- Cython: An extension of Python that allows you to write code that compiles to C and then create bindings between C code and Python. Cython is commonly used to improve performance in Python.

Example Bytecode

Input : a = 5

b = 10

c = a + b

Bytecode:

Line	Position	Operation	Symbol Table Index	Value
2	0	LOAD_CONST	1	(5)
	2	STORE_FAST	0	(a)
3	4	LOAD_CONST	2	(10)
	6	STORE_FAST	1	(b)
4	8	LOAD_FAST	0	(a)
	10	LOAD_FAST	1	(b)
	12	BINARY_ADD		
	14	STORE_FAST	2	(c)
5	17	LOAD_FAST	2	(c)

Explanation of Columns:

Line: Refers to the line number in the source code where the instruction originates.

Position: The offset (index) in the bytecode sequence where this instruction is located.

Operation: The specific bytecode operation being executed.

Symbol Table Index: The index in Python's internal symbol table or constant pool, referring to variables, functions, or constants.

Value: The actual value involved in the operation, such as the variable, constant, or function being referenced.

Explanation of Bytecode:

LOAD_CONST 1 (5): Loads the constant value 5 onto the stack.

STORE_FAST 0 (a): Stores the value on top of the stack into the variable a.

LOAD_CONST 2 (10): Loads the constant value 10 onto the stack.

STORE_FAST 1 (b): Stores the value on top of the stack into the variable b.

LOAD_FAST 0 (a): Loads the value of the variable a onto the stack.

LOAD_FAST 1 (b): Loads the value of the variable b onto the stack.

BINARY_ADD: Adds the two values on top of the stack and pushes the result back onto the stack.

STORE_FAST 2 (c): Stores the result on top of the stack into the variable c.

LOAD_FAST 2 (c): Loads the value of the variable c onto the stack.

Error Handling

Error Handling is the process in which our program detects errors, reports them to the user, and then applies recovery strategies to try to handle the errors.

We can classify error types into lexical, semantic, and syntactic errors. The first, as we recall, occurs when the lexical analyser encounters a character or sequence of characters in the source code that does not match any defined token rule. Syntactic errors arise when the identified sequence of tokens does not follow the defined grammatical rules. Lastly, semantic errors occur when a syntactically valid construction makes no sense within the semantic context of the program.

Now then, firstly, lexical errors are handled through the *t_error* function provided by YACC, which notifies the user that a character was not recognised as a valid token and is therefore skipped to continue the analysis.

Next, if the parser encounters an error while attempting to reduce a production, we have a syntactic error. In this case, the *p_error* function is invoked to handle it, providing information about the nature of the error.

Finally, if the parser encounters a valid production that makes no sense within the program's context, it must be handled as a semantic error. This type or error is handled

explicitly within the grammatical rules. For example, let us consider a simple semantic error: attempting to use an undeclared variable.

Suppose the input is '*a = 20;*'. The processing steps for this string begin in the lexical analyser, which will identify the tokens used as valid. Then, the syntactic analyser will recognise the string as 'IDENTIFIER EQUAL expression SEMIC', according to grammatical rules, making it possible to construct the parse tree.

The next step is to verify if the string is semantically correct, which involves identifying the type of expression it represents —in this case, an assignment. Within the function that allows us to verify assignments, we need to check the symbol table to ensure that the variable exists within the structure. If it does exist, then the string is semantically valid; otherwise, it indicates an attempt to use a variable that has nos been previously declared, which is an error.

Something similar happens with other types of semantic errors. For instance, in an attempt to assign a floating-point value to a variable declared as an integer, the symbol table is first checked to confirm that a certain variable *x* is an integer. Then, if the difference between the absolute value of the variable and its value parsed as an integer is not zero, we can conclude that the subtraction yields a decimal value. This indicated an attempt to assign a floating-point value to a variable declared as an integer, which is another type of semantic error.

Our interpreter can identify and handle around 12 semantic errors, such as variable redeclaration, assignment of null values to variables, or operations involving variables with null values.

Output

The result of executing the source code can manifest in several ways:

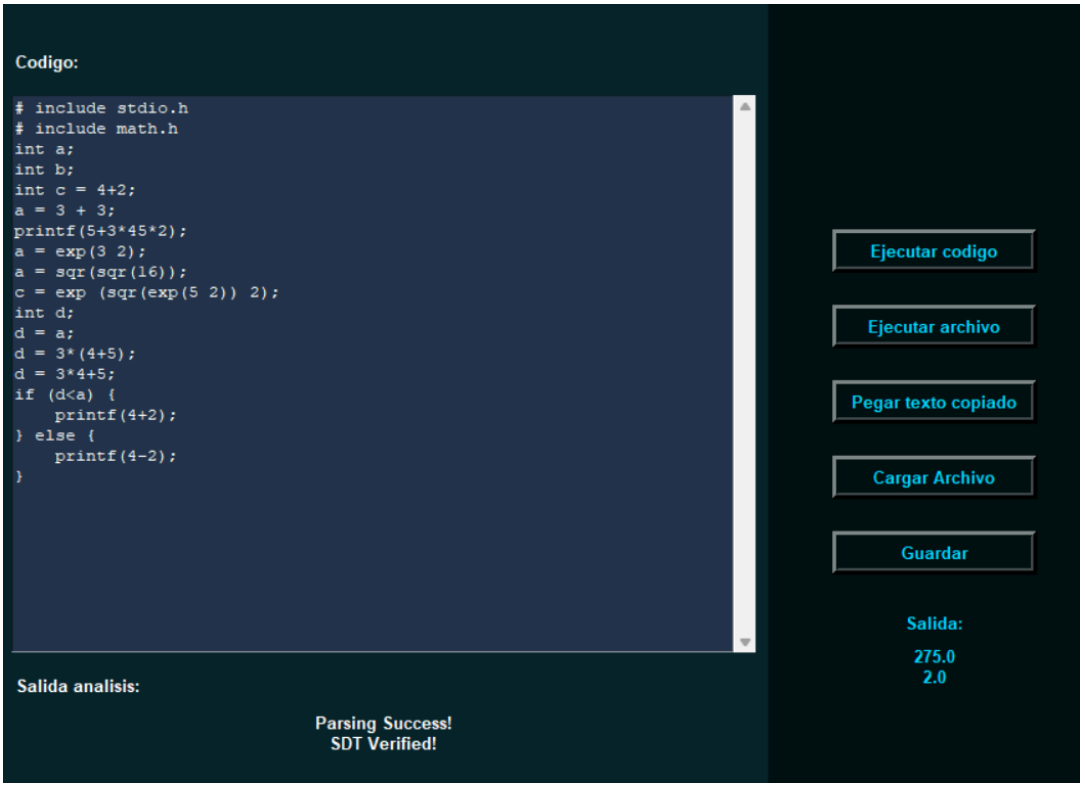
Standard Output (stdout): The main output of the interpreter is the screen, where the results of the code execution are displayed.

Errors and Exceptions: When an error occurs in the code (such as an exception or a syntax error), the interpreter generates an error message that is displayed in the console.

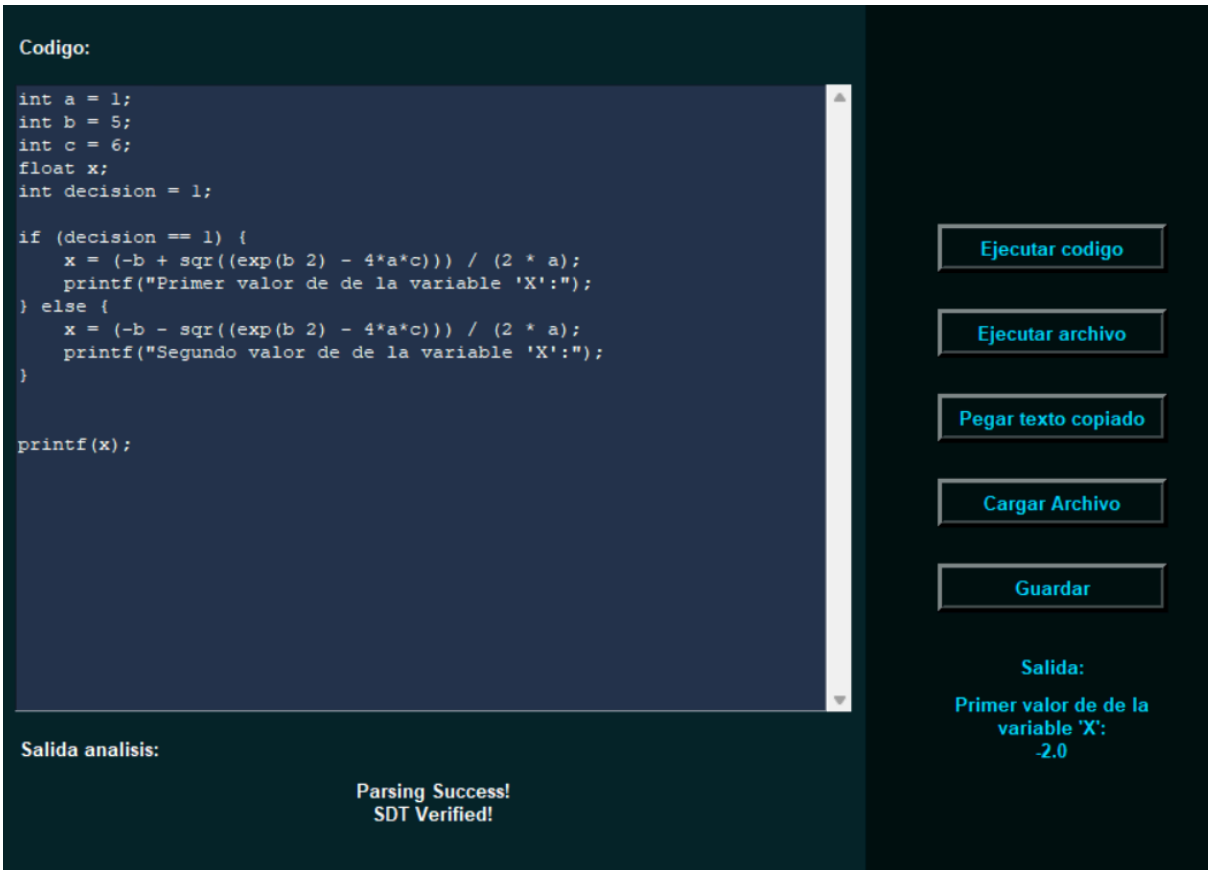
Results

Example 1:

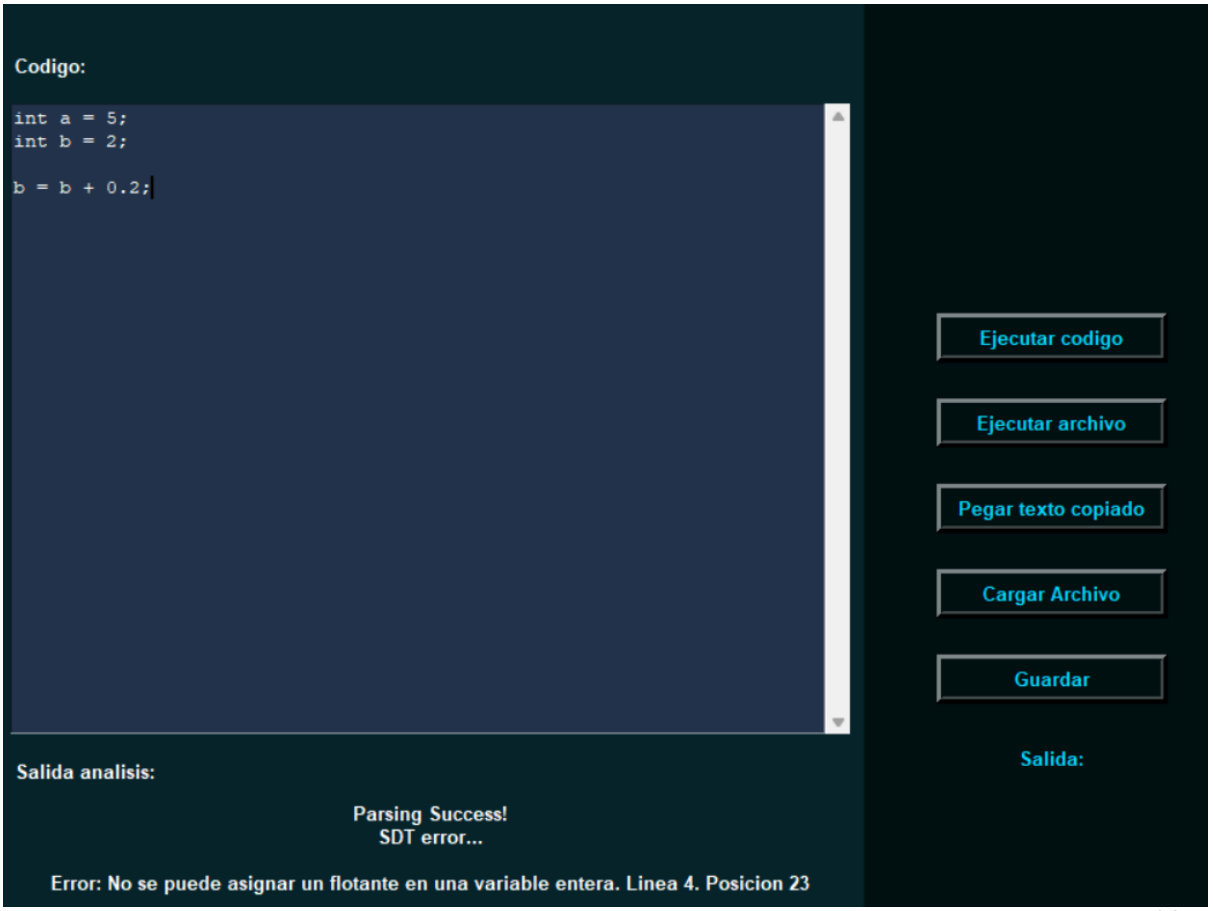
Example 2:



Example 3:



Example 4:



Conclusions

This project represents the culmination of our efforts to develop a fully functional interpreter, integrating foundational components such as the lexical analyser and the parser/semantic analysis modules. Together, these elements form a cohesive system that bridges the gap between abstract input code and its practical execution.

The lexical analyzer performs the task of breaking down input code into manageable tokens through lexical analysis. This initial step ensures that raw text is segmented into meaningful units such as keywords, identifiers, literals, and operators, preparing it for further processing. Meanwhile, the parser/STD utilizes these tokens to construct a syntax tree that reflects the program's logic and flow, acting as a detailed blueprint for the interpreter to organize and execute operations.

Beyond integration, this project successfully implemented an interpreter that introduced critical enhancements, including the generation and optimization of compact instruction sets. Techniques such as constant folding, dead code elimination, and literal merging, were implemented to reduce code size and improve performance. The interpreter flourishingly managed complex language constructs, such as declarations, expressions, and conditions, while effectively handling lexical, syntactic, and semantic errors.

In conclusion, this project not only consolidates the essential components of interpreter design but also demonstrates the impact of optimisation techniques on improving system efficiency. By achieving a balance between theoretical concepts and practical implementation, this work highlights the collaborative interplay of individual modules in building a robust and functional interpreter.

Bibliography

- R. S. Alfred V. Aho, Monica S. Lam and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston: Pearson Education, 2006.
- J.-P. Tremblay and P. G. Sorenson, *The Theory and Practice of Compiler Writing*. BS Publications, 1982.
- M. Omer, "Compilers vs. Interpreters: How They Work?," Aug. 07, 2023. https://www.linkedin.com/pulse/compilers-vs-interpreters-how-work-monzer-omer
- *ast — Abstract Syntax Trees*. (2024). Python Documentation. https://docs.python.org/3/library/ast.html
- *dis — Disassembler for Python bytecode*. (2024). Python Documentation. https://docs.python.org/es/3/library/dis.html
- python. (2017). *cpython/InternalDocs/interpreter.md at main · python/cpython*. GitHub. https://github.com/python/cpython/blob/main/InternalDocs/interpreter.md