Vrije Universiteit Amsterdam

Bachelor Thesis

# Design and Implementation of an Interpreter for a Small Procedural Programming Language in Python

**Author:** Berkay Ilhan Kush (2688924)

*1st supervisor:* Prof. Dr. Jasmin Christian Blanchette
*2nd reader:* Anne Baanen

*A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science*

June 23, 2023

## Abstract

In this paper, we present the design and implementation of a correctly functioning interpreter written in Python. The interpreter is capable of executing programs in a small custom procedural programming language. This paper provides a comprehensive explanation of the inner workings of our interpreter, covering the entire process from plain text programs to executing them and producing the desired output. Moreover, we answer the research question of whether Python is suited for writing an interpreter for a static programming language**.**

# Contents

# 1 Introduction

Various programming languages have been created worldwide to serve different use cases in numerous fields and industries. For instance, in data science, Python and R are popular because they have a rich set of libraries and packages for data manipulation, analysis, and visualization. Meanwhile, JavaScript is a powerful language for web development due to its ability to create interactive and dynamic websites. The list does not end here. There are hundreds of other programming languages that serve specific purposes. The diversity of programming languages is one of the fascinating aspects of computer science.

Nevertheless, programming languages also share a common and general purpose: to provide instructions to tell a computer what to do. However, computers cannot process and execute programming languages immediately. Instead, they require a program called a compiler or an interpreter. The aim of this paper is to dive deeply into how one of these programs, namely an interpreter, works to execute the instructions from programming languages.

The ability to build interpreters and compilers and understand how they work can be advantageous in many ways. For instance, it can enhance one's comprehension of programming languages and how they operate under the hood. Such knowledge can help developers or programmers optimize their software's performance, making it more efficient and scalable while minimizing bugs and errors. Additionally, writing an interpreter or a compiler requires a deeper understanding of computer science concepts such as data structures, algorithms, and computer architecture, which can lead to improved skills in these areas. Lastly, the most widely used programming languages, including Python, Java, and C++, are interpreted or compiled. This emphasizes the significance of having extensive expertise in these topics for anyone aspiring to develop the next major programming language.

To better grasp the concepts discussed in the upcoming sections, it is essential to understand the distinction between interpreters and compilers. The main difference between interpreters and compilers, as shown in Figure 1, lies in their approaches to translating or executing code.

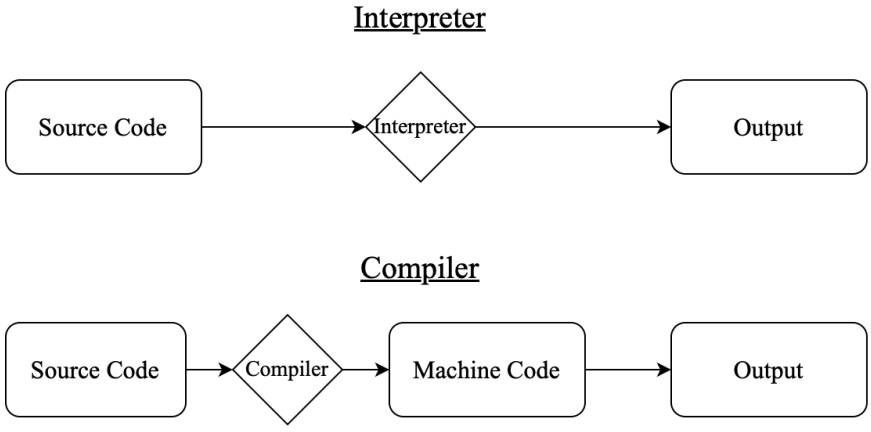


Figure 1: Visualization of how interpreters and compilers work

Interpreters directly execute programs, executing code line by line or statements one at a time. Compilers, on the other hand, translate the entire source code into some other form, such as machine code, bytecode, or high-level language, without executing it. The resulting form can be executed in different ways based on its nature. For instance, if the output is machine code, it can be executed by the target computer's hardware.

This paper also aims to address the research question: Is Python suitable for writing an interpreter for a static programming language? Why use Python for this purpose? Firstly, the main interest of this research question is to investigate the suitability of a dynamic programming language for developing an interpreter for a static programming language; thus, static languages have been excluded. Secondly, the paper includes several code blocks to enhance comprehension. Therefore, the provided code must be clear and concise. Python is a dynamic language with a beginner-friendly and English-like syntax, making it easy to write, read, and understand.

In the upcoming section, this paper will deliberate on various components of our interpreter's implementation. The subsequent section, The Compact Programming Language, will comprehensively elucidate the syntax and features of the programming language exclusively created for the interpreter. Furthermore, this segment will also delineate the typing rules and provide clear examples that illustrate the usage of this language. In the third section, we will delve into lexical analysis, where we will discuss how source code can be broken down into sequences of tokens. The fourth section, Parsing, will cover the process of creating an abstract syntax tree (AST) from the sequence of tokens generated in the lexical analysis stage. The fifth section, Semantic Analysis, will describe the semantic analysis process, which involves analyzing the meaning of a program to ensure it is structured correctly and has no static semantic errors. The sixth section, Program Execution, will delve into how the interpreter executes the AST generated from the parsing phase. In the discussion section, we will discuss and provide an answer to our research question. Finally, we will conclude by summarizing the main findings and discussing limitations and possible future work for Compact and the implemented interpreter. The complete source code for this project is available on GitHub [1].

# 2 The Compact Programming Language

To better understand the design and implementation of our interpreter, it is important to first discuss the programming language our interpreter translates. This section will explore Compact, a small procedural programming language explicitly created to be interpreted by our interpreter. Despite its conciseness, Compact has a clear and understandable syntax. Additionally, it offers the following features:

- Java-like syntax
- Support for integers, floating-point numbers, booleans, and strings
- Variable declarations and bindings
- Arithmetic, comparison, logical, and range expressions
- Conditional statements and loops
- Recursion and nested functions
- Built-in functions

Before we delve into each feature, it is good to discuss the type guidelines in Compact. Compact is a static programming language that ensures type safety by, for instance, checking that variables, the operands of an operation, and function arguments are of the correct data type. To perform type checking, Compact adheres to the Java type rules [2] with slight variations. One such difference is that, in Compact, multiplying a string by an integer repeats the string multiple times, which is not allowed in Java. Furthermore, Compact allows default values for function parameters, whereas Java does not allow this and uses method overloading to achieve the same effect.

Now, we will explore each feature in Compact, starting with variable declarations and bindings, which appear in the format depicted in Figure 2.

```
1     var(int) num1;
2     var(float) num2 = 2.1, num3 = 3.2;
3     var(str) name = "Berkay";
4     var(str) firstLetter = name[0];
5     var(int) num1; /* error */
```

Figure 2: A set of examples of variable declarations and bindings in Compact

To declare a variable in Compact, one must type the keyword `var`, followed by the data type enclosed in parentheses, and then the variable name. For instance, in Figure 2, line 1, the variable `num1` is declared with the data type `int` using the syntax `var(int) num1`. The semicolon at the end of each variable declaration statement is a must. Furthermore, Compact allows assigning a value to a variable while declaring it, as demonstrated in lines 2, 3, and 4 of Figure 2. It is also possible to declare and assign multiple variables of the same data type in a single line, as illustrated in line 2 of Figure 2. Lastly, as indicated by line 5, once a variable is declared, it cannot be declared again with the same name in the same scope to prevent ambiguity in the code and ensure clear variable naming conventions.

As previously mentioned, Compact supports arithmetic, comparison, logical, and range expressions. Of these, range expressions are particularly interesting and are formatted in the following way:

```
1     startIndex to endIndex (optionally) step incrementSize
```

These expressions specify a range of integers and are used in `for` loops. The `startIndex` and `endIndex` represent the inclusive lower and upper bounds of the range. The `step` keyword is optional and allows users to specify the increment size between each iteration. If the `step` keyword is omitted, the increment size defaults to 1.

In addition to the range expression, it is also important to know which operators are supported when working with arithmetic, comparison, and logical expressions. Figure 3 demonstrates the supported operators for these types of expressions.

| Arithmetic |
|---|
| + |
| - |
| * |
| / |
| // |
| % |

| Comparison |
|---|
| == |
| != |
| < |
| <= |
| > |
| >= |

| Logical |
|---|
| and |
| or |
| not |

Figure 3: Supported operators for arithmetic, comparison, and logical expressions

Compact guarantees the order of operations, allowing it to solve complex expressions. It can, for instance, evaluate the complex expression below, which has numerous arithmetic and logical operators with parentheses.

```
1     ((3 * 4) + (2 * 6) / (8 % 3)) == 13 and not(5 <= 3 or 8 > 10);
```

In Compact, a conditional statement is represented using `if`, `elseif`, and `else` blocks. The syntax for these blocks is represented in Figure 4.

```
1     if(5 > 3) {
2         /* statements */
3     } elseif(3 == 3) {
4         /* statements */
5     } else {
6         /* statements */
7     }
```

Figure 4: Syntax for conditional statements in Compact

An `if` block begins with the keyword `if`, followed by a condition inside parentheses. The condition can be any expression that evaluates to a boolean value. If the condition evaluates to true, the statements within the curly braces will be executed. Optionally, one or more `elseif` blocks and a single `else` block can follow an `if` block.

The final information to consider about conditional statements is that variables created outside the curly braces of the conditional blocks can be accessed within them. However, variables created inside the curly braces are limited to the scope of that block and cannot be accessed from an outer scope.

The Compact programming language also includes `while` and `for` loops. The syntax for `while` loops in Compact follows the standard syntax used in Java, but the syntax for `for` loops varies slightly from other programming languages. Figure 5 provides examples of how to construct a `for` loop in Compact.

```
1     for(var(int) i from 1 to 10) {
2         /* statements */
3     }
4
5     for(var(str) char from "Hello") {
6         /* statements */
7     }
```

Figure 5: Different ways of constructing a for loop in Compact

In Compact, a `for` loop starts with the keyword `for`, followed by a declaration of an integer or string variable. After that, the keyword `from` is added, which is then followed by either a range expression, a string, or a string variable. Additionally, similar to conditional statements, variables declared outside the curly braces of the loop constructs can be used within them, but variables declared inside the curly braces have a scope limited to that specific loop and cannot be accessed from an outer scope.

Defining functions in Compact is not that different from any other programming language. Figure 6 showcases how a function can be defined.

```
1     func(returnType) funcName(parameters) {/*statements*/}
```

Figure 6: Syntax for a function definition in Compact

The format for defining a function starts with the keyword `func`, followed by the function's return type. There are five possible return types in Compact: `int`, `float`, `bool`, `str`, and `void`. Only `void` functions do not require return statements within them. Next, the function's name is provided, followed by its parameters in parentheses. As previously discussed, it is also possible

to assign values to function parameters. Lastly, the function's body is enclosed within curly braces, containing the statements that will be executed when the function is called.

Compact also supports recursion, which allows for handling problems that can be broken down into smaller sub-problems.

What distinguishes Compact from some other programming languages like C, C++, and Java is its ability to have nested functions. Nested functions can be helpful for creating helper functions that are only needed in a specific context. The syntax for defining a nested function is the same as a regular function, except it is defined within the body of another function.

The last and final feature that we will discuss is the set of built-in functions in Compact. Here is the list of built-in functions in Compact:

- `print()` and `println()`
- `input()`
- `reverse()`
- `len()`
- `pow()`
- `typeof()`
- `toint()`, `tofloat()`, `tobool()`, and `tostr()`

The first two functions, `print()` and `println()`, display output to the console, with the former printing on the same line and the latter adding a new line after printing. The `input()` function is used for accepting user input. The `reverse()` function reverses the order of either characters or elements in a string. The `len()` function returns the length of a string. The `pow()` function computes the power of a number by a given exponent. The `typeof()` function is used to return the data type of a given variable or value. The remaining four functions, `toint()`, `tofloat()`, `tobool()`, and `tostr()`, convert data types.

To summarize, Compact is a small procedural programming language designed to be interpreted by our interpreter. It has a clear and understandable syntax, along with various features. Finally, it is a statically typed language, ensuring type safety by checking type compatibilities.

# 3 Lexical Analysis

To effectively work with source code, it must be converted into a different format. Although plain text is simple to navigate in an editor, it can quickly become unwieldy when attempting to interpret it as a programming language. Therefore, alternative representations are necessary to simplify the process. Before evaluating our source code, we will need to modify its representation twice, as shown in Figure 7.


Figure 7: The process of converting source code into an abstract syntax tree

This section will cover the first modification, which involves converting our source code to tokens. This process is called lexical analysis and is carried out by a lexer, which is also referred to as a lexical analyzer, a scanner, or a tokenizer. According to Pai T, Jayanthiladevi, and Aithal [3], in the context of compiler/interpreter construction, a lexer is a pattern recognizer that reads

a string of individual characters as input in the source program and groups them into sequences called lexemes by matching with the token pattern. The result is a stream of tokens.

There are a variety of lexer generators available, such as JFlex [4], Flex [5], and re2c [6]. JFlex is a tool for generating lexers for Java that provides support for Unicode. Flex, on the other hand, is a free and open-source lexical analyzer generator tool for C and C++ code for performing efficient pattern matching on input text files. Lastly, re2c is a free and open-source lexer generator for C, C++, Go, and Rust. These tools help speed up the implementation of a lexical analyzer by allowing programmers to specify patterns at a high level and relying on the generator to produce detailed code. However, the lexer for our interpreter was created from scratch rather than using a generator, which could provide more flexibility and clarity compared to utilizing a generator.

In the upcoming subsections, we will explore the concept of tokens and briefly discuss the `Token` class. Finally, we will conclude the third section by delving into the `Lexer` class in more detail.

## 3.1 Tokens

In the context of programming languages, a token refers to the smallest element or basic component of a program that is meaningful to compilers or interpreters. Token types include keywords, identifiers, numeric, boolean, and string literals, operators, and punctuators.

To make the explanation more concrete, consider Figure 8.

```
1     var(int) result = 0;
2     for(var(int) num from 1 to 10) {
3         result += num;
4     }
5     println(result);
```

Figure 8: An example program in Compact

Let us break this example program down by identifying the types of tokens it contains. Firstly, it contains numbers like 0, 1, and 10. The program also contains identifiers such as `result`, `num`, and `println`. Additionally, it includes keywords like `var`, `int`, `for`, `from`, and `to`. Finally, the program includes symbols such as "( ", ")", "=", ";", "{", "+=", and "}".

After grasping the idea of tokens, let us delve into the `Token` class. The class has four attributes that are used constantly: `type`, `val`, `line`, and `col`. Each instance of the token class is assigned a specific token type, which allows parsers to perform different actions based on the type. Figure 9 demonstrates the various token types that are used in our interpreter. Additionally, tokens typically have a non-null value associated with them. For instance, a boolean token can hold either a true or false value. The `EOF` token is the only one with a null value, indicating that there are no more commands to read. Lastly, tokens store their respective line and column numbers, providing information about where an error may have occurred in the program.

```
1     (EOF, IDENTIFIER, SEMICOLON, INT, FLOAT, BOOL, STR, PLUS, MINUS,
      MULTIPLICATION, ASSIGN, EQUALS, LESS_THAN, LEFT_PARENTHESIS,
      RIGHT_SQUARE_BRACKET, K_VAR, K_INT, K_AND, K_IF, K_FOR)
```

Figure 9: The possible token types

## 3.2 The Lexer Class

As stated earlier, our lexical analyzer was built from scratch and not generated. In this section, we will delve into the crucial parts of the implementation of the `Lexer` class. The constructor of the `Lexer` class is depicted in Figure 10.

```
1     def __init__(self, text)
2         self.__text = text
3         self.__char_pos = 0
4         self.__curr_char = self.__text[self.__char_pos]
5
6         self.__line = 1
7         self.__col = 1
```

Figure 10: Constructor implementation of the `Lexer` class in Compact

The `Lexer` class has five attributes. One is the `text` attribute, which holds the entire program as a string. The next two attributes keep track of the current character and its location in the lexer. Furthermore, the last two attributes show the current line and column number of the lexer in the input text. These attributes work together to represent the current state of the lexer and enable it to keep track of its position in the input text while processing each token.

There are two methods crucial for the lexer to work: `get_next_token()` and `advance()`. We will go through each one and finish this subsection.

The `get_next_token()` method is the lexer's core method responsible for returning the next token in the input text. It works by iterating over the input text one character at a time, and based on the current character, it decides what type of token to generate. For example, when the current character is a quotation mark ("), the lexer recognizes it as the beginning of a string and creates a string token, which is then returned. Other conditions the method checks and returns a token object for include keywords, identifiers, number and boolean literals, operators, and punctuators. Finally, when all characters in the input text have been processed, the lexer generates the `EOF` token.

The `advance()` method is a private helper method used by the lexer to move to the next character in the input text. This method updates the current line and column numbers and sets the current character to the next in the input text. It also sets the current character to `None` if no more characters are left. This method is used in conjunction with `get_next_token()` to generate the appropriate tokens from the input text.

# 4 Parsing

In the previous section, we discussed how source code can be converted into a stream of tokens. This section will focus on the next step, which involves transforming those tokens into an abstract syntax tree, which is accomplished through a process called parsing, carried out by a parser. According to Aho et al.'s compiler model [7], the parser analyzes and verifies that a given sequence of tokens aligns with the grammar of the source code language. Moreover, the parser is expected to raise syntax errors when the tokens do not conform to the grammar. In the case of a syntactically correct program, the parser constructs a hierarchical representation, typically an abstract syntax tree, that captures the program's structure. Our parser functions in the same way. Figure 11 illustrates the functionality of our parser in a simplified manner.

```
1     var(int) result;
```

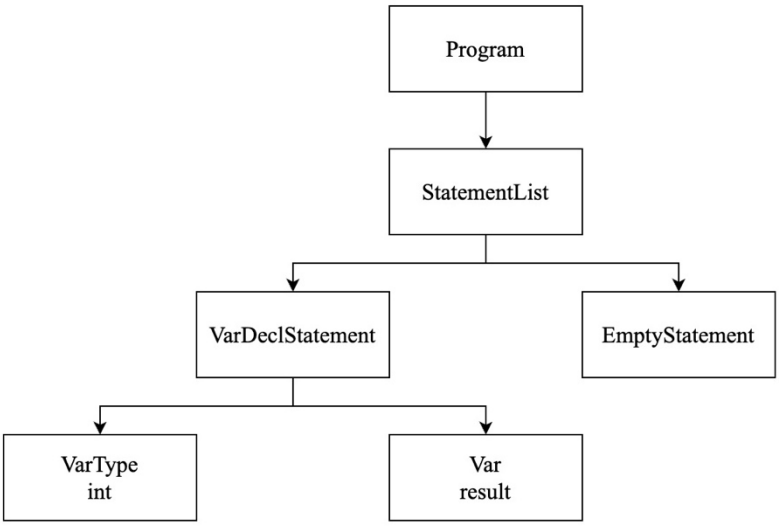


Figure 11: Transition of the one-line program `var(int) result;` to its AST representation

Similar to lexer generators, numerous parser generators are available that can automate the process of generating parsers based on a formal grammar specification. These parser generators take a grammar as input and produce the corresponding parser code in a target programming language. One popular parser generator is ANTLR [8], which supports ten languages, including Python. It can automatically build parse trees and generate tree walkers to navigate those trees and execute application-specific code. It is commonly used in academia and industry to create various languages, tools, and frameworks. Grammatica [9] is another popular parser generator for C# and Java. It uses LL(k) grammars, which refer to a class of context-free grammars in which a top-down parsing strategy is employed, using k tokens of look-ahead to determine the production rule to apply during parsing. Moreover, the tool facilitates the creation and testing of the parser during runtime, enabling debugging prior to commencing the actual source code development process.

In conclusion, parser generators are highly effective tools that generate parsers for even the most complex grammars. In addition, they significantly save time when writing an interpreter. Nonetheless, a parser was built from scratch for our interpreter. There are several important reasons for this choice. Firstly, manually implementing a parser allows for a deeper understanding of the benefits, limitations, and problem-solving capabilities of parser generators, as suggested by Thorsten Ball [10]. Building a parser from scratch also provides greater flexibility in dealing with specific language features and meeting specific requirements.

According to Aho et al., there are three types of parsers based on their approach to handling grammars: universal, top-down, and bottom-up [7, Section 4.1]. While universal parsers can handle any grammar, they are not commonly used in compilers due to their computational inefficiency. Top-down parsers start from the root and work toward the leaves, while bottom-up parsers start from the leaves and work toward the root. In both cases, the parser reads the input symbols from left to right, processing them individually. Our parser is a top-down, or more specifically, a recursive descent parser [7, Section 2.4.2]. It uses mutually recursive procedures that correspond to each nonterminal in the grammar to process the input.

In the following subsections, we will delve more into the essential components of the parsing process. Firstly, we will discuss the concept of context-free grammars and their crucial role in defining the syntax rules of a programming language. Next, we will explore ASTs. Finally, we will dive into the implementation of our parser.

## 4.1 Context-Free Grammars

The grammar of a programming language plays a crucial role in defining the syntax rules that govern its structure. It specifies how the various elements of the language, such as statements, expressions, and declarations, can be combined to form valid programs. In the parsing process, the grammar serves as a guide for the parser to determine the syntactic correctness of the input source code.

Our interpreter uses a context-free grammar to define the syntax of the programming language it interprets. A context-free grammar consists of terminals, nonterminals, a start symbol, and production rules [7, Section 4.2.1]. Terminals are typically the tokens generated by the lexical analyzer. Nonterminals, on the other hand, are variables that represent sets of strings and establish a hierarchical structure within the language.

Within a context-free grammar, one nonterminal is designated as the start symbol, where the grammar derivation begins. Lastly, production rules specify how terminals and nonterminals can be combined to form strings. The proper structure of production rules is crucial for maintaining a well-defined programming language.

To make the explanation more concrete, let us discuss Figure 12.

```
1     program = statement_list ;
2     statement_list = statement, statement_list | empty_statement ;
3     ...
4     return_statement = K_RETURN, [ logical_expr ] ;
5     arithmetic_expr = term, { ( PLUS | MINUS ), term } ;
```

Figure 12: The subset of Compact's grammar rules

Here are the key points to note about Figure 12:

- Figure 12 includes three terminals (`K_RETURN`, `PLUS`, and `MINUS`), eight nonterminals (`program`, `statement_list`, `statement`, `empty_statement`, `return_statement`, `logical_expr`, `arithmetic_expr`, and `term`), one start symbol (`program`), and four production rules.
- The symbols "=", ";", ",", "|", "{", "(", "}", ")" are not considered terminals, as they hold a special meaning in the grammar, which we will discuss later. However, they can be terminals if represented differently. For instance, to use "(" as a terminal, it must be replaced with `LEFT_PARENTHESIS`.
- Each production rule includes a nonterminal (referred to as the head), the symbol "=", and a body consisting of zero or more terminals and nonterminals. The body components dictate how the strings of the nonterminal at the head can be constructed.

There are various notations that express context-free grammars, such as BNF, EBNF, and ABNF. Our grammar is expressed using EBNF, which is an extension of BNF. EBNF introduces additional syntax and constructs that enable the expression of more complex grammars. The EBNF symbols that are used in our grammar and their meanings are as follows:

- "=": Indicates that the nonterminal on the left can be derived or expanded into the sequence of terminals and nonterminals on the right.
- ";": Denotes the end of a production rule.
- "',": Indicates concatenation, instructing that the elements should be combined in a sequence.
- "|": Serves as the alternation operator, allowing the head to be derived from either the sequence to the left or the sequence to the right of the symbol. This provides multiple possibilities for a grammar rule.

- "[ ]": Implies that the elements enclosed may or may not be present.
- "{ }": Represents repetition, implying that the elements within the curly braces can be repeated zero or more times.
- "( )": Groups elements within a production rule, clarifying grammar rules.

## 4.2 Abstract Syntax Trees

Syntax-directed interpreters evaluate expressions as soon as a specific language construct is recognized by their parser. These interpreters are appropriate for basic language applications and typically only require one pass over the input. However, more complex languages require a data structure—an intermediate result—that parsers can construct and interpreters can utilize for accurate evaluation. Some of the most common data structures an interpreter's or compiler's parser can output are parse trees and abstract syntax trees.

A parse tree, also known as a concrete syntax tree, is a representation of the structure of a language construct. It helps us understand how the parser recognized and analyzed the language, and it illustrates how the grammar's start symbol generates a specific string in the programming language [7, Section 2.3.2]. Consider the parse tree in Figure 13.

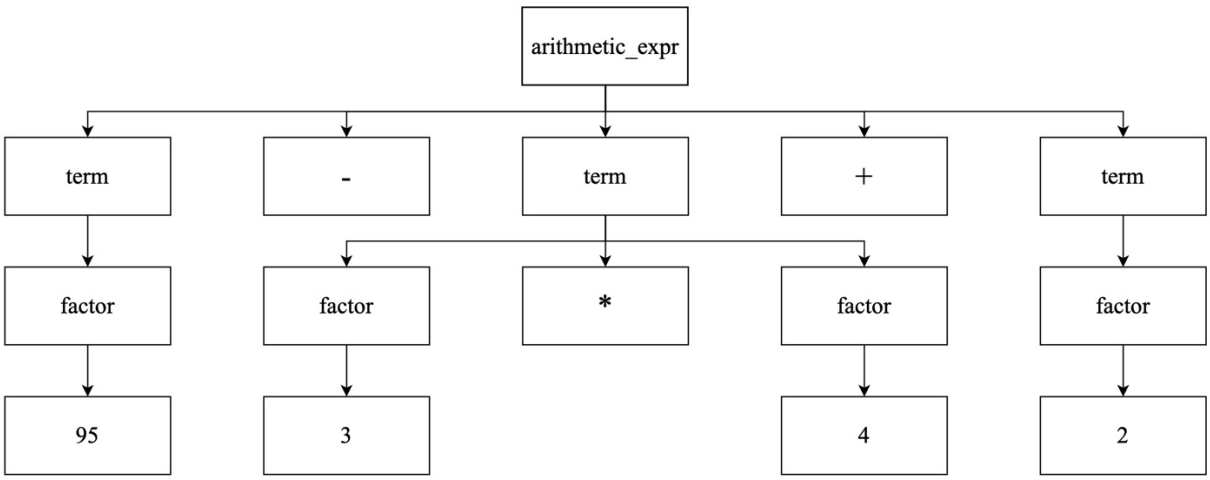


Figure 13: The parse tree for the expression "95 - 3 * 4 + 2"

The parse tree captures the sequence of rules the parser applies to identify the input. Interior nodes correspond to nonterminals, such as arithmetic_expr, term, or factor, indicating the application of grammar rules. Leaf nodes, on the other hand, represent tokens.

An abstract syntax tree is a type of tree that represents the abstract syntactic structure of a language construct. The word abstract in AST refers to the fact that some details that are visible in the source code are not included in the AST. For instance, semicolons, brackets, and parentheses might not be part of the AST, depending on the programming language and the parser. Consider the AST shown in Figure 14.

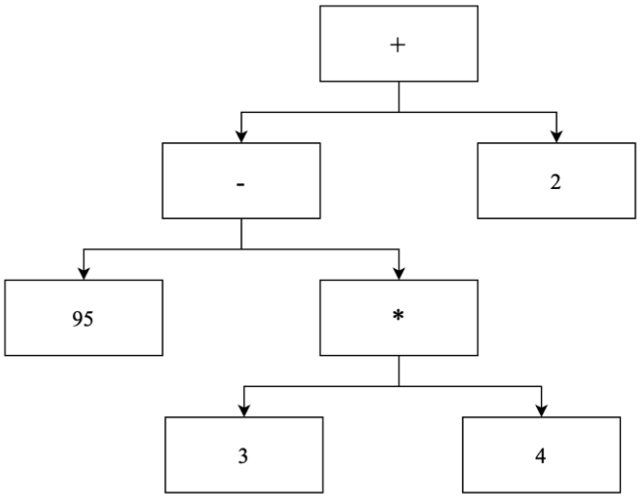


Figure 14: The AST for the expression "95 - 3 * 4 + 2"

In contrast to parse trees, ASTs do not utilize interior nodes to depict grammar rules, nor do they include all the specific syntax details like parentheses or every keyword. Furthermore, ASTs are more compact compared to parse trees when representing the same language construct. By referring to Figure 14, we can also observe that nodes with higher precedence appear lower in the tree.

Our parser generates an AST as an intermediate result. The benefit of using an AST is that it captures the essential elements of syntax without being overly specific, making it more concise and comprehensible.

To wrap up this subsection, let us examine the AST node depicted in Figure 15. A `VarNode` instance represents a variable in our programming language. Nodes like this exist for different rules of the grammar, each storing the necessary information for the interpreter to evaluate them. For instance, the `VarNode` class has two attributes. The `token` attribute stores the reference to the token, which is helpful in cases of errors during the semantic analysis or interpretation. It allows the users to easily identify the line and column number of the token causing the error. The `val` attribute extracts the variable's name from the token.

```
1     class VarNode(AST):
2         def __init__(self, var_token):
3             self.__token = var_token
4             self.__val = var_token.val
```

Figure 15: Implementation of the `VarNode` class

## 4.3 The Parser Class

Having discussed the role of a grammar and the concept of an AST, we can now explain the implementation of our parser. First, let us discuss the two attributes of the `Parser` class: `lexer` and `curr_token`. The `lexer` attribute is passed to the parser to retrieve tokens from the input source code. Meanwhile, the `curr_token` attribute helps the parser keep track of the token that is being processed. It is set by calling the `get_next_token()` method of the lexer and is only modified by the `eat()` method of the parser.

As stated in Section 4, our parser is a recursive-descent parser, which means that the parsing program consists of a set of methods, one for each nonterminal. Therefore, every

production rule in our grammar needs to be translated into its own method. We will illustrate the key points of converting a production rule to its method in Figure 16.

```
1     def __return_statement(self):
2         """
3         rule: return_statement = K_RETURN, [ logical_expr ] ;
4         """
5         return_token = self.__curr_token
6         self.__eat(Token.K_RETURN)
7
8         if self.__curr_token.type_ != Token.SEMICOLON:
9             return ReturnStatementNode(return_token,
10                                       self.__logical_expr())
11        return ReturnStatementNode(return_token)
```

Figure 16: Method implementation for the return statement rule in the parser

Within the method in Figure 16, we follow the structure of the production rule. The initial expectation is to come across the `K_RETURN` token. To ensure that the current token is indeed `K_RETURN`, we validate it using the `eat()` method. If the validation is successful, the `eat()` method consumes the current token, allowing the method to proceed to the next part of the production rule after the comma. The key takeaway is that when a nonterminal method encounters a terminal, it uses the `eat()` method to validate and consume that terminal token.

The next expectation is to come across the "[ ]" symbol, indicating that the content inside is optional. The best way to represent this in code is by using conditional statements. This situation is represented in line 8 of Figure 16, where the method checks if the `logical_expr` nonterminal is present. If the method encounters the `logical_expr` nonterminal, it calls its method in line 10. Essentially, whenever a method encounters a nonterminal, it will call its corresponding method. It is also important to note that nonterminal methods always return an AST node.

Finally, we will discuss the method that starts the parsing process, namely the `parse()` method. The `parse()` method starts the process by calling the nonterminal method for the start symbol (`program`). This method, in turn, invokes other methods for nonterminals that appear in its production rules. Then, this leads to a mutually recursive process where each nonterminal method calls other nonterminal methods. Once the parsing process is complete, the method verifies if the entire input has been parsed in accordance with the grammar rules. If it has, the method returns the created AST.

# 5 Semantic Analysis

Although a program may exhibit grammatical accuracy and a successfully built AST, it can still contain significant errors. These errors pertain to the program's interpretation and meaning beyond its structure. For instance, consider the given example below.

```
1     var(int) k = "hello";
```

In this situation, the parser will not show an error because the statement is grammatically correct. However, in Compact and many other programming languages, assigning a string to an integer variable will result in an error. This section aims to explain the methods to catch and report such errors to ensure the program's correctness and reliability.

In programming, semantics refers to the program's intended meaning. The meaning of a program helps enforce rules. These rules are known as semantic rules and are split into two categories: static and dynamic semantics [11, Section 4].

Dynamic semantic checks are conducted during the execution of programs after calling the `interpret()` method in the `Interpreter` class. These checks include verifying that there is no division by zero or ensuring that the string index is within bounds. On the other hand, the `SemanticAnalyzer` class performs static semantic checks, which are the main focus of this section, prior to creating an instance of the `Interpreter` class and calling the `interpret()` method. In Compact, the static semantic rules are as follows:

- Variables must be declared before they can be used.
- Variables involved in operations must have compatible types.
- Functions must be defined before they are invoked.
- Variables and functions declared within a specific scope are accessible only within that scope and any nested inner scopes.
- There cannot be two identifiers (variables and functions) with the same name in the scope to which they belong.
- Function calls must provide the correct number of arguments, and their types must align with the formal parameters in the function declaration.
- A non-default parameter must not follow a default parameter.
- Return statements can only appear within function definitions, and the return type of a function must match the declared return type.
- The use of continue and break keywords is restricted to within-loop constructs only.

The purpose of these rules is to ensure that the program is accurate and reliable in its meaning.

How does our semantic analyzer check that our static semantic rules are followed? It does this by traversing and examining the AST generated during the parsing phase. The process of traversing the AST is similar to the process of constructing it. The traversal starts at the root node, known as the `ProgramNode`, and proceeds by moving to different nodes based on the program's structure being analyzed. During the traversal, the semantic analyzer gathers information about variable declarations, function definitions, and other language-specific constructs. This information is stored in a convenient data structure called a symbol table. A symbol table acts as a repository of knowledge, helping the analyzer detect semantic rule violations. If a violation is detected, the semantic analyzer generates an error message that provides information about the nature and location of the issue in the source code.

In the following subsections, we will explore the process above in more detail. Firstly, we will provide a more detailed explanation of symbol tables' role and give details about the `SymbolTable` class. Secondly, we will examine the `SemanticAnalyzer` class to observe the enforcement of rules at the implementation level.

## 5.1 Symbol Tables

As mentioned, a symbol table is a crucial data structure used by semantic analyzers to manage and track information about variables, functions, and other language-specific constructs within their respective scopes. Specifically, symbols contain this information and are held within a symbol table.

There are, in total, seven symbol classes implemented: `BuiltInTypeSymbol`, `BuiltInFuncSymbol`, `VarSymbol`, `ConditionalSymbol`, `RangeSymbol`, `LoopSymbol`, and `FuncSymbol`. An example of what the `VarSymbol` class stores can be seen in the code snippet in Figure 17.

```
1     class VarSymbol(Symbol):
2         def __init__(self, name, type_):
3             super().__init__(name, type_)
```

Figure 17: Implementation of `VarSymbol`

The `VarSymbol` class inherits from the base `Symbol` class, which provides the two common attributes `name` and `type_`. The `name` attribute represents a variable's name, and the `type_` attribute stores an instance of the `BuiltInTypeSymbol`. Besides these two common attributes, each symbol class may have its own extra attributes based on its unique function. For instance, the `FuncSymbol` class has details about the function's parameters and default parameter count.

Now that we have discussed the implemented symbols, let us delve into the implementation details of the `SymbolTable` class. All symbols associated with a symbol table instance are stored in the `symbols` attribute (a Python dictionary). In this dictionary, each symbol is linked to its name as the key, and its corresponding symbol object is the associated value. This allows for efficient lookup and retrieval of symbols.

In addition to storing symbols, the class also stores information about the current scope's name, level, and outer scope, which points to the surrounding scope if it exists. These details help the analyzer check for the correct use of variables and functions. To visualize this, Table 1 showcases the three symbol tables that will be created for the example program depicted in Figure 18.

```
1     func(int) findMax(var(int) num1, var(int) num2) {
2         if (num1 > num2) {
3             return num1;
4         }
5         return num2;
6     }
7
8     var(int) num1 = 5, num2 = 10;
9     var(int) max = findMax(num1, num2);
10    println(max);
```

Figure 18: The `findMax` program that finds the maximum of two integers

Table 1: Symbol Tables and the information they store for the `findMax` program

| **Scope Name** | **Scope Level** | **Outer Scope** | **Symbol names in each scope** |
|---|---|---|---|
| Global | 1 | None | int, float, bool, str, print, println, input, reverse, len, pow, typeof, toint, tofloat, tobool, tostr, findMax, num1, num2, max |
| findMax | 2 | Global | num1, num2, if |
| if | 3 | findMax | |

The `SymbolTable` class has two primary responsibilities: adding and retrieving symbols. These functionalities are performed using the `add_symbol()`, `add_built_in_symbols()`, and `get_symbol()` methods. We will briefly review each method before moving on to the next subsection, starting with the adder methods.

The implementation of the `add_symbol()` method is as follows:

```
1     def add_symbol(self, symbol):
2         self.__symbols[symbol.name] = symbol
```

The method takes a symbol object as input and inserts it into the symbol table using the symbol's name as the key.

The `add_built_in_symbols()` is used only in the constructor of the `SemanticAnalyzer` class. Its purpose is to populate the first instance of the symbol table, global scope, with pre-defined built-in symbols, namely built-in types and functions, as shown in Table 1.

Finally, we will discuss the `get_symbol()` method. Compared to adder methods, `get_symbol()` is more complex because scopes are involved. As can be seen from Figure 19, the `get_symbol()` method is responsible for looking up a symbol by its name. The method begins by verifying whether the symbol is present in the current scope and returning it if it is. However, if the symbol is not present, the method checks the value of `check_outer_scope`. If `check_outer_scope` is true, the search continues in the outer scopes until the symbol is found or `None` is returned, indicating that the symbol is not present.

In addition, it is important to note that if the requested symbol is not found within the current scope and the current scope is a conditional or loop (checked from lines 6 to 12 in Figure 19), the method will search for the symbol in at least one outer scope, regardless of the value of `check_outer_scope`. This is due to the fact that Compact closely follows Java's rules, and in Java, it is not allowed to have two variables with the same name, where one is inside a loop or conditional statement and the other is outside. The additional search helps validate whether this rule is violated. If a violation regarding this is detected, the semantic analyzer will raise an error.

```
1     def get_symbol(self, name, check_outer_scope=True):
2         if self.__check_symbol(name):
3             return self.__symbols[name]
4
5         if (not check_outer_scope) and (
6             not (
7                 self.__scope_name.startswith("if")
8                 or self.__scope_name.startswith("elseif")
9                 or self.__scope_name.startswith("else")
10                or self.__scope_name.startswith("while")
11                or self.__scope_name.startswith("for")
12            )
13        ):
14            return None
15
16        if self.__outer_scope is not None:
17            return self.__outer_scope.get_symbol(name,
18                                                 check_outer_scope)
```

Figure 19: Implementation of the `get_symbol()` method

## 5.2 The Semantic Analyzer Class

The semantic analyzer is a crucial component of our interpreter design that performs static semantic analysis. Its primary task is to ensure the correctness of the program's semantics based on the rules defined by the programming language. In this section, we will delve into the crucial parts of the implementation of the `SemanticAnalyzer` class.

The main and most used attribute of the class is `curr_symbol_table`, which keeps track of the current symbol table. It begins with the global scope, serving as the program's initial scope, and is updated every time a new scope is introduced. Moreover, upon exiting the current scope, `curr_symbol_table` transitions to the outer scope of the current scope, ensuring accurate evaluation.

Now, we will cover how the `SemanticAnalyzer` class checks for static semantic rules by traversing an AST tree. First, we will explain the process of traversal; then, we will discuss how the analyzer ensures the first and second static rules outlined in Section 5.

As previously discussed in Section 5, the traversal starts at the root node, known as the `ProgramNode`, and proceeds by moving to different nodes based on the program's structure being analyzed. However, how does the semantic analyzer determine which node to move to next? For instance, how does it know if the next node to move to is a `VarNode` or a `NumberNode`? Every AST node in the interpreter has a corresponding method in the `SemanticAnalyzer` class, with the naming convention of `visit` followed by the AST node's name. This convention is due to `SemanticAnalyzer` inheriting from `ASTNodeVisitor`, a visitor class that handles the execution of node methods.

The `ASTNodeVisitor` class has a method named `visit()`, which is responsible for executing the appropriate node method based on the type of the given AST node (`ast_node`). Thus, whenever the `SemanticAnalyzer` class comes across an AST node, it calls the `visit()` method. This method then calls the correct node method, ensuring that the AST traversal proceeds correctly.

Now, let us explain how our semantic analyzer ensures the enforcement of the following static rules:

- Variables must be declared before they can be used.
- Variables involved in operations must have compatible types.

The selected rules are widely applicable semantic rules in programming languages. In addition, the concepts that apply to these rules are also used in other static semantic rules outlined in Section 5.

Whenever a variable is declared, it is added to the current symbol table. The semantic analyzer prevents the usage of variables before their declaration by verifying their existence in both the current scope and any preceding scopes. This is illustrated in the code on lines 3 to 5 of Figure 20.

```
1     def visitVarNode(self, ast_node):
2         var_name = ast_node.val
3         variable_symbol=self.__curr_symbol_table.get_symbol(var_name)
4
5         if variable_symbol is None:
6             self.__error(f'Identifier "{var_name}" not found',
7                          ast_node.token)
8     ...
```

Figure 20: Variable lookup and declaration check in the SemanticAnalyzer class

In order to perform type checks, our semantic analyzer needs to be aware of the type rules and have access to the types of variables and expressions. The type rules are implemented in a file called `type_checking.py`. An example rule from that file is illustrated in Figure 21. The `SemanticAnaylzer` class uses the `TypeChecker` class to access the rules in this file.

```
1     def check_assignment_statement(var_type, var_val_type,
2                                    var_val_token):
3         if var_type != var_val_type:
4             TypeChecker.__error(
5                 f'Cannot assign "{var_val_type}" to {var_type}"',
6                 var_val_token)
```

Figure 21: Implementation of the type rule for assignment statements

The semantic analyzer gathers information about variable and expression types from the return values of AST node methods during the traversal of the AST. The return type of one node is passed on to the next node. For instance, as shown in Figure 22, the return type of a `NumberNode` can be used to determine the resulting type of a binary operation. This resulting type can then be used in subsequent operations to enforce type rules or determine the resulting type of more complex expressions. In other words, the type information obtained from one node influences the types of subsequent nodes.

```
1     def visitNumberNode(self, ast_node):
2         if isinstance(ast_node.val, int):
3             return BuiltInTypeSymbol(Token.K_INT)
4
5         return BuiltInTypeSymbol(Token.K_FLOAT)
6
7     def visitBinaryOpNode(self, ast_node):
8         return TypeChecker.check_binary_op(
9             ast_node.op_token,
10            left_node_type=self.visit(ast_node.left_node).name,
11            right_node_type=self.visit(ast_node.right_node).name)
12
13    def visitAssignmentStatementNode(self, ast_node):
14        TypeChecker.check_assignment_statement(
15            var_type=self.visit(ast_node.left_node).name,
16            var_val_type=self.visit(ast_node.right_node).name,
17            var_val_token=ast_node.right_node.token)
```

Figure 22: An example of type propagation and rule checking

In conclusion, by propagating type information from one node to another and performing rule checking at appropriate points using the `TypeChecker` class, our semantic analyzer ensures adherence to Compact's type rules.

# 6 Program Execution

The execution phase is the last step in processing source code. During this phase, expressions such as "2 * 2" are evaluated to give 4, and statements like `print("Hello World!")` are executed to print `Hello World!` to the output console.

There are numerous interpreter designs for assessing source code. In Section 4.2, we discussed one approach known as syntax-directed interpreters, which execute the source code directly without using an intermediate representation or translating it to another language. Another option is to use tree-walk interpreters, which execute the source code after parsing it into an AST and performing possible static analysis. The interpreter evaluates each node while traversing the AST, performing the actions each node indicates. Our interpreter is also a tree-walking interpreter. Additionally, there is another approach named just-in-time (JIT) compilation that blurs the line between interpreters and compilers and combines the two

together [10, Section 3.2]. This method generally involves parsing source code, building an AST, and converting it into bytecode, which is then compiled into machine code before execution.

Since many interpreter designs are available, choosing one requires careful consideration of factors such as performance requirements and language complexity. For instance, syntax-directed interpreters are relatively easy to implement but can become challenging to maintain as the language grows in complexity. Tree-walking interpreters, although potentially slower than the other two, are straightforward to construct, expand, and comprehend [10, Section 3.2]. This makes them ideal for student projects. Finally, the JIT compilation approach offers improved speed, but its implementation can be more intricate.

Now that we have familiarized ourselves with the concept of tree-walking interpreters, let us be more specific and briefly discuss how our tree-walking interpreter executes programs. To execute a program, the interpreter traverses the AST generated in the parsing phase, similar to how it is traversed by the `SemanticAnalyzer` class. During this traversal, the interpreter performs different actions for each encountered node type. For instance, when it encounters a `NumberNode` representing an integer or a float value, it returns the corresponding value. Moreover, if the interpreter encounters a `VarDeclStatement` node, it stores that variable's information in a program stack. Once the interpreter completes the traversal of the AST and executes all necessary actions for each node, the program's execution is considered complete.

In the following subsections, we will explain this process in more detail. We will start by discussing what a program stack is and why it is necessary for our interpreter. Then, we will delve into the implementation details of the `Interpreter` class, the main class responsible for dynamic semantic checks and program execution.

## 6.1 Program Stack

At the interpreter level, using a Python dictionary (known as a hash map in other programming languages) is an excellent way to store information about variables, functions, and their respective values. However, the problem is that having just one dictionary to store information poses a challenge for supporting scopes. To address this issue, there needs to be a separate dictionary for each scope. The question then arises: How can these multiple dictionaries be efficiently organized and accessed?

The solution lies in using a stack. Here is how the stack data structure is used by our interpreter:

- As the interpreter traverses the AST, it detects new scopes and pushes a new stack frame onto the stack.
- After encountering a new scope and pushing the corresponding stack frame, the interpreter proceeds to execute the statements within that scope. During this execution, the interpreter adds information about variables and functions to the stack frame associated with that scope.
- Once the execution of statements within a scope is finished, the stack frame representing that scope is popped from the stack. This action ensures proper management of memory resources and avoids conflicts with variables from other scopes.

The stack used by our interpreter is called the program stack, also often referred to as the call stack. It is implemented as a class named `ProgramStack`. The class itself is nothing special and follows a standard implementation for a stack in Python. However, the frames that it stores make it interesting.

The class `StackFrame` is used to implement stack frames. Similar to the `SymbolTable` class, the `StackFrame` class also stores information about the current scope's name, level, and outer scope. As we will soon discover, their methods are also quite similar. In addition to scope information, stack frames hold information about variables and functions, which are kept in two dictionaries: `variables` and `functions`. The `variables` dictionary maps variable names to their values.

Meanwhile, the `functions` dictionary maps function names to their respective stack frames, parameter names, and bodies. This information is crucial for the `visitFuncCall()` method to execute functions. For instance, normally, the `visitFuncCall()` method lacks access to the statements contained within the called function. Nonetheless, with the `functions` dictionary, this information can be retrieved.

The main tasks of the `StackFrame` class are to set and get variable and function values. Its getter methods, namely `get_var()` and `get_func()`, are responsible for searching for a requested variable or function (referred to as the `key`) within both the current and outer scopes. If the `key` is found, its corresponding value is returned. However, if it is not found, `None` will be returned.

When assigning values to variables and functions, it is important to determine whether the value is being assigned during their declaration. If it is, the current frame will create a new entry for that identifier with its assigned value, as demonstrated in Figure 23.

```
1     curr_stack_frame.variables[var_name] = var_val
2
3     curr_stack_frame.functions[func_name] = {
4         "stack frame": func_frame,
5         "param names": param_names,
6         "body": ast_node.body
7     }
```

Figure 23: Setting values for variables and functions during their declaration

However, when updating the value of an existing variable, a different approach must be followed. Let us assume a scenario where there is a need to change the value of a global variable inside a function. The `set_var()` method plays a crucial role in such cases. It thoroughly searches the current and outer scopes to locate the desired variable and update its value accordingly. By using this method, variables can be modified accurately, regardless of whether they are defined in the current scope or an outer scope.

To summarize this section, with the program stack in place, our interpreter can effectively manage variables and functions, ensuring accurate accessibility and modification.

## 6.2 The Interpreter Class

The `Interpreter` class plays an important role in the execution of a program written in Compact. It interprets and executes the AST produced by the parser, carrying out the instructions and producing the desired output. In this section, we will explore the key components and functionality of the `Interpreter` class.

The `Interpreter` class has two frequently used attributes: `PROGRAM_STACK` and `ast`. `PROGRAM_STACK` represents the program stack, and `ast` holds the AST the `Parser` class generates.

In the rest of this subsection, we will begin by discussing the `interpret()` method. Afterward, we will conclude the section by explaining the different categories of AST node methods that work together to execute Compact programs.

One of the main methods of the `Interpreter` class is the `interpret()` method, and it serves as the starting point for traversing the AST. The execution begins at the `ProgramNode` and continues until every node in the AST has been visited. Once the interpreter completes the traversal of the AST and executes all necessary actions for each node, the method returns the outcome of the executed program. As shown in line 4 of Figure 24, the method calls the `visit()` method. Similar to the `SemanticAnalyzer` class, the `Interpreter` class also inherits from the `ASTNodeVisitor` class, allowing it to use the `visit()` method to call appropriate node methods.

```
1     def interpret(self):
2         if self.__ast is None:
3             return ""
4         return self.visit(self.__ast)
```

Figure 24: Implementation of the `interpreter()` method

We can separate the methods of AST nodes into four categories:

- Category 1: Methods that evaluate expressions and return their value. For instance, the `visitBinaryOpNode()` method can evaluate expressions such as "1 + 2" and return the result.
- Category 2: Methods that update the current stack frame's information. An instance of this category is the `visitAssignmentStatementNode()` method, which updates the value of a variable in the `variables` dictionary by assigning it a new value obtained from a method of category 1.
- Category 3: Methods that push a new stack frame and initiate the execution of statements for that particular frame. The `visitProgramNode()` method is an instance of this category, as it pushes the initial stack frame (global scope) on top of the program stack and starts the execution of the statements in the global scope.
- Category 4: This category consists of a single method, `visitFuncCallNode()`, that combines the functionalities of categories 1, 2, and 3. It pushes a new stack frame for the called function, updates its parameters and local variables, initiates the execution of its statements, and finally returns the result.

During the traversal of AST, the AST node methods from the same or different categories work together to execute Compact programs correctly.

# 7 Discussion

One of the main purposes of the paper is to address the research question: Is Python suitable for writing an interpreter for a static programming language? Let us start by discussing both the benefits and drawbacks of using Python. The advantages of using Python to write an interpreter for a static programming language are as follows:

- **Ease of development:** Python has a clear and concise syntax, making writing and understanding code easier. It has a reputation for accomplishing tasks with fewer lines of code. For instance, in a study conducted by Khoirom et al. [12], they analyzed the number of lines of code needed to implement the quicksort algorithm and the tic-tac-toe game in both Python and Java. The results indicated that Python consistently required less code than Java.
- **Community and resources:** Python has a large and active community of developers, providing a wealth of resources that make it easier to write an interpreter. For instance, Python features numerous built-in functions, simplifying the implementation of

equivalent built-in functions in a custom programming language. Additionally, many lexer and parser generators are available for Python, such as ANTLR. Another helpful resource is Graphviz's Python package [13], which can be used to debug ASTs visually.

As for the disadvantages, they are as follows:

- **Lack of static typing features:** Python is a dynamically typed language, meaning variables do not have pre-defined types. In addition, a function can return values of different types depending on the conditions within the function body or the values passed to it. Therefore, implementing a static programming language requires more time and effort. One must write additional type checks and errors and introduce them to their semantic analyzer, similar to the `type_checking.py` file for Compact.
- **Performance considerations:** Python is an interpreted language, generally making it slower than compiled languages such as C or C++. In addition, it is also slower than some interpreted languages. For example, in a test conducted by Horntvedt and Åkesson [14], the average execution time of the quicksort algorithm, the fizzbuzz game, and file reading operations in Python, Java, and JavaScript were analyzed. The results showed that Python took longer to execute compared to the other two languages. This is especially important to consider if high performance is the top priority for writing an interpreter for a static language.

Considering the advantages and disadvantages mentioned, we can reasonably say that Python is suited for writing an interpreter for a static programming language, despite its disadvantages. Python's clear and concise syntax makes it ideal for interpreter development. Furthermore, Python's vast standard library and its diverse third-party libraries provide help for almost every conceivable task, allowing developers to leverage existing functionality and focus on higher-level project requirements.

While Python may be slower than languages such as C, C++, and Java, the difference in speed is generally not noticeable for small to medium-level programs. Well, what about interpreters that execute large programs? It is true that for larger programs involving extensive computations, users may notice a difference in performance. However, several optimization techniques are available to enhance Python programs' overall speed. An example would be writing computation-heavy parts of a Python program in C for faster computation and speed. In fact, this is what happens in NumPy [15], where many array operations are implemented in C for improved performance.

As for the lack of static typing features in Python, workarounds are available for this drawback. Python has external type-checker tools that can identify type errors, such as passing the wrong type of argument to a function, returning a different type than declared, or assigning incompatible types to variables. One example is Mypy [16], which can perform static type-checking for Python code. It combines Python's expressive power and convenience with a powerful type system and compile-time type checking. With the already implemented static checks and errors from Mypy, the time and effort it takes to write additional type checks and errors for Python can be reduced.

Overall, Python offers several benefits that make it well-suited for writing an interpreter for a static programming language. Although Python has limitations in terms of static typing and performance, these drawbacks can often be mitigated or worked around. By leveraging Python's strengths and using optimization techniques where necessary, developers can create maintainable, manageable, and extendable interpreters for static languages using Python as the implementation language.

# 8 Conclusion

This paper has presented the design and implementation of an accurately functioning interpreter capable of executing programs written in our custom programming language. We have provided a comprehensive explanation of the inner workings of our interpreter, covering the process from plain text programs to executing them and producing the desired output. Additionally, we have discussed and provided an answer to our research question.

We will finalize our conclusion section by discussing the limitations and future work of our interpreter and Compact. The major limitation of Compact is the lack of features to solve complex or large problems. Compact supports four types: integers, floats, booleans, and strings. While these data types cover many common use cases, more complex data types such as arrays, lists, and hash maps are needed to handle advanced data manipulation tasks. Additionally, Compact does not support the object-oriented programming (OOP) paradigm, such as classes, inheritance, or polymorphism. This limits the ability to conceptualize real-world scenarios. Lastly, our interpreter and Compact lack the implementation of some common and advanced language constructs, such as exception handling and multithreading.

Regarding future work, the main focus will be to minimize the limitations discussed above in order to solve complex or large problems. For instance, we would like to extend our interpreter to support more complex data types such as arrays, lists, and hash maps. A good starting point would be to establish the syntax for these data types. Afterward, we plan to introduce new tokens, grammar rules, semantic rules, symbols, AST nodes, and new AST node methods for these data types to our interpreter. Furthermore, the introduction of new data types would require the inclusion of additional built-in functions. Python already has several built-in functions implemented for lists and dictionaries, which we expect to use in implementing our own versions of these functions.

# References


[1] B. I. Kush, "Compact Programming Language," *GitHub*, May 26, 2023. https://github.com/berkaykush/Compact-Programming-Language-in-Python (accessed Jun. 22, 2023).

[2] "Chapter 4. Types, Values, and Variables," *docs.oracle.com*. https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html (accessed April 30, 2023).

[3] V. Pai T, A. Jayanthiladevi, and P. S. Aithal, "A Systematic Literature Review of Lexical Analyzer Implementation Techniques in Compiler Design," *International Journal of Applied Engineering and Management Letters*, pp. 285–301, Dec. 2020, doi: https://doi.org/10.47992/ijaeml.2581.7000.0087.

[4] G. Klein, S. Rowe, and R. Décamps, "JFlex - JFlex The Fast Scanner Generator for Java," *jflex.de*. https://jflex.de/ (accessed May 4, 2023).

[5] W. Estes et al., "Westes/Flex: The Fast Lexical Analyzer - scanner generator for lexing in C and C++," *GitHub*, https://github.com/westes/flex (accessed May 4, 2023).

[6] P. Bumbulis et al., "re2c," *re2c.org*. https://re2c.org/ (accessed May 4, 2023).

[7] A. V. Aho et al., *Compilers: Principles, Techniques, & Tools*. Pearson, 2007.

[8] T. Parr, "ANTLR," www.antlr.org. https://www.antlr.org/index.html (accessed May 16, 2023).

[9] P. Cederberg et al., "Grammatica," *grammatica.percederberg.net*. https://grammatica.percederberg.net/ (accessed May 16, 2023).

[10] T. Ball, *Writing an Interpreter in Go*. 2020.

[11] M. L. Scott, *Programming Language Pragmatics*. Amsterdam: Elsevier/Morgan Kaufmann Pub., 2009.

[12] S. Khoirom et al., "Comparative analysis of Python and Java for beginners," *Int. Res. J. Eng. Technol*, vol. 7, no. 8, pp. 4384-4407, 2020.

[13] S. Bank et al., "Graphviz," *graphviz.readthedocs.io*, 2013. https://graphviz.readthedocs.io/en/stable/ (accessed Jun. 16, 2023).

[14] T. Åkesson and R. Horntvedt, "Java, Python and Javascript, a comparison," 2019.

[15] C. R. Harris et al., "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, 2020, doi: https://doi.org/10.1038/s41586-020-2649-2.

[16] J. Lehtosalo et al., "mypy - Optional Static Typing for Python," *mypy-lang.org*. https://mypy-lang.org/ (accessed Jun. 22, 2023).