Universidade Federal de Pernambuco

Centro de Informática

Bachelor's Degree in Computer Engineering

Bleach: A programming language aimed for teaching Compilers

Victor Miguel de Morais Costa

Undergraduate Thesis

Recife, Pernambuco

October , 2024

Universidade Federal de Pernambuco
Centro de Informática

Victor Miguel de Morais Costa

Bleach: A programming language aimed for teaching Compilers

Work presented to the Undergraduate Course in Computer Engineering at the Informatics Center of the Federal University of Pernambuco as a partial requirement to obtain the Bachelor's degree in Computer Engineering.

Concetration Area: Compilers, Interpreters, Programming Languages
Advisor: Leopoldo Teixeira

Recife, Pernambuco
October, 2024

Dedico este trabalho a Deus e minha família.

Acknowledgments

Andrea,
Alvaro,
Maria Dolores,
Laura,
Leopoldo Motta Teixeira,
Fernando Castor, Gustavo Carvalho,
Daniel Perazzo, Fernando Macedo, Lucas Ambrosio, Matheus Teotônio, Riei Joaquim, Victor Hugo, Victor Ximenes, Zênio Ângelo
José Carlos da Silva Cruz, Pedro Nogueira Coutinho, Zilde Souto Maior Neto
José Victor Silva Cruz, Lucas Santana, Marcos Oliveira,
Cintia,
Noriaki Kubo,

Large fingers pushin' paint...
You're God and you got big hands...
The colors blend...
The challenges you give man...

Seek my part... devote myself...
My small self... like a book amongst the many
on a shelf...

Sometimes I know, sometimes I rise
Sometimes I fall, sometimes I don't
Sometimes I cringe, sometimes I live
Sometimes I walk, sometimes I kneel
Sometimes I speak of nothing at all...
Sometimes I reach to myself, hear God...

— "Sometimes", Pearl Jam, 1995

Resumo

Nos cursos de bacharelado em Ciência da Computação ou Engenharia da Computação, espera-se que os alunos tenham contato com uma disciplina de Compiladores de nível introdutório. Devido à grande amplitude e complexidade dos temas abordados nesta disciplina, os professores responsáveis tendem a conduzi-lá sob um ponto de vista mais teórico.

Esta abordagem costuma proporcionar aos alunos uma compreensão mais profunda dos conceitos fundamentais desta área e prepará-los melhor para uma carreira orientada para a pesquisa científica. No entanto, tende a limitar a experiência prática dos conceitos ensinados, fazendo com que os alunos muitas vezes se sintam desconectados das aplicações dessa área, causando um menor engajamento e motivação entre estes. Diante disso, desde a década de 90, novas metodologias, que mesclam teoria e prática, começaram a surgir. Na maioria delas, os alunos devem implementar um compilador/interpretador para uma linguagem de programação de fins didáticos, normalmente definida pelo professor responsável.

Devido às desvantagens mencionadas da abordagem tradicional e às limitações que algumas das linguagens propostas na abordagem mista apresentam, a proposta atual visa apresentar uma nova linguagem de programação chamada Bleach, juntamente com uma implementação de seu interpretador. Tal linguagem pretende ser utilizada como ferramenta em disciplinas introdutórias de Compiladores que optem por seguir uma abordagem de ensino mais voltada para a prática incremental, priorizando a flexibilidade e objetivando solucionar, ou pelo menos mitigar, as desvantagens de ambas as abordagens mencionadas.

Palavras-chave: Compiladores; C++; Educação; Software Educacional; Interpretadores; Linguagens de Programação.

Abstract

In Bachelor's degrees in Computer science or Computer engineering, students are expected to have contact with an introductory-level Compilers course. Due to the large breadth and depth of topics in this subject, professors and instructors tend to conduct it from a more theoretical point of view.

This approach tends to provide students a deeper understanding of the fundamental concepts of this field and better prepare them better for a research oriented career. However, it tends to limit the hands-on experience of the taught concepts, making students often feel disconnected from real-world applications of this area, causing less engagement, excitement and motivation among them. Given this, since the 90s, new methodologies, which combined theory and practice, began to emerge. In most of them, students we required implement a compiler/interpreter for a programming language for teaching purposes, normally defined by the responsible professor or instructor.

Due to the aforementioned disadvantages of the traditional approach and the limitations that some of the proposed programming languages in the mixed approach present, this proposal aims to present a new programming language called Bleach along with an implementation of its interpreter. This language is intended to be used as a tool in introductory-level Compilers courses whose responsible professors opt to follow a teaching approach more focused on incremental practice, prioritizing flexibility and aiming to solve, or at least mitigate, the disadvantages of both mentioned approaches.

Keywords: Compilers; C++; Education; Educational Software; Interpreters; Programming Languages.

List of Tables

List of Figures

Contents

Chapter 1

Introduction

"If you were to give me wings, I would fly for you. Even if this entire land were to sink underwater. If you were to give me a sword, I would stand up and fight for you. Even if the entire sky were to pierce you with its light."

– NORIAKI KUBO

1.1 Background and Motivation

In Computer Engineering and Computer Science programs, Compilers course are essential for development of students' knowledge in this area, since it provides the basis of how programming languages are executed by computers.

Despite its importance, this particular course is often considered difficult for students to grasp due to its complexity and abstract nature. Due to this issue, efforts have been made since this topic started being taught at universities, in order to build learning tools focused on theoretical aspects. However, in certain scenarios there is still a lack of practical tool that are capable of allowing students to get their hands dirty and work with compiler components.

Given this context, it becomes clear that there exists a gap when it comes down to learning tools focused in practical aspects. Therefore, the opportunity to design and implement a specialized programming language to address these educational challenges emerged.

1.2 The Problem

When analyzing the structure of initial and traditional Compilers courses, it is noticeable that they often rely too much on theoretical lectures and, in some cases, on code examples that fail to engage and motivate undergraduate students that are having their first contact with compilers design and implementation.

Given this past scenario, changes started to being made in order to offer a better and more fruitful learning experience through the creation of different "toy" programming languages. Unfortunately, even though the intentions were good, when analyzing them in retrospect, it is safe to conclude that most of them are either too simple and uninteresting to demonstrate real-world compiler concepts or too complex and overwhelming for students to work with in an educational setting that usually spans across just one academic semester.

By examining these circumstances and the history behind programming languages and compilers development, it became clear that currently there is a lack of programming languages that could effectively used in balancing educational clarity and practical experimentation. Having this obstacle in mind, this thesis seeks to offer a new solution for this problem, by focusing more on teaching just the essential concepts and giving the students the chance to practice as much as they can, instead of spending too much time on theoretical aspects of this field.

1.3 Objectives

The main goal of this undergraduate thesis is to present a programming language specifically designed and implemented for use in undergraduate Compilers courses at Computer Engineering and Computer Science programs.

The language aims to offer a practice-oriented understanding of core compiler and interpreter phases such as lexical analysis, syntax parsing, semantic resolution, and interpretation.

Finally, by allowing the students to implement and interact with these components in an active learning environment, Bleach seeks to create a better connection between theoretical knowledge and practical application.

1.4 Scope

It is very important to highlight to the reader that this undergraduates thesis' focus is exclusively on the presentation, implementation and evaluation of a programming language that was designed to simplify the understanding of compiler phases, particularly lexical analysis, syntax analysis, semantic resolution and interpretation. In other words, this thesis aims to defend a new point of view where teaching interpreters implementation instead of compilers is a reasonable and valuable approach that is supposed to not overwhelm students who are having their first contact with the subject.

Therefore, the implementation of the programming language presented here will not address advanced compiler topics, such as: type systems, code optimization, register allocation, or machine code generation, since each one of them is complex enough to be studied by several weeks.

Instead, the language implementation presented here will emphasize core concepts in an accessible way to undergraduate students.

1.5 Contributions

The main contribution of this thesis is the design and implementation of a new programming language specifically tailored for teaching Compilers courses.

The language's simplicity allows students to experiment with real compiler phases without being overwhelmed by the complexities of full-scale language implementations.

On top of that, the language provides detailed feedback at each stage of the interpreting pipeline, helping students to have a more solid comprehension of this area. Also, the implementation of the language has its own test suite, that can, and should be, used by the teaching staff during the course.

1.6 Thesis Structure

Ultimately, this thesis is structured in the following format: Chapter 2 provides a literature review that discusses essentials theoretical concepts of the programming language design and implementation field. Chapter 3 is dedicated to provide a historical context about how the teaching of this subject evolved during the last 60 years and also to show studies that highlight

the importance of a practice-oriented approach in Compilers courses. This chapter also provides an overview of existing programming languages that attempt to solve the same problem exposed in this chapter. Then, Chapter 4 presents Bleach, the programming language proposed as a new solution for the mentioned issue. The chapter goes deep into the features that the language has and gives a brief overview of its standard library. After this, an in-depth analysis of how each component of the implemented interpreter is presented and, it ends with a discussion about the challenges when designing and implementing the language, as well as the trade-offs made during these two processes. Chapter 5 goes into the evaluation aspect of the language, by explaining how its test suite works. Then, as a statement of it expressiveness, a link to a GitHub repository containing implementations of famous algorithms and data structures is presented to the reader. Concluding the chapter, a brief feature comparison between Bleach and its competitors is shown, proving that Bleach is, indeed, a viable option to choose when teaching Compilers courses. At last, Chapter 6 concludes this thesis by reflecting on the contributions that were made and opportunities of future work and enhancements regarding Bleach.

Chapter 2

Theoretical Review

"Do not live bowing down. Die standing up."

– NORIAKI KUBO

2.1 Introduction

This chapter is dedicated to provide a literature review about the Programming Language Design and Compiler/Interpreter Implementation fields. The conceptual content presented here is heavily influenced by [1], [2] and [3].

2.2 What is a Compiler and How does it work?

To simply put it simple, a compiler is a software whose responsibility is to translate a program written in a certain programming language into a another program written in another one, without modifying the meaning of the original program, that is, its semantics. However, despite this simple description, compilers are usually complex and large software systems composed of several components which interact with each other during the mentioned translation process, commonly known as compiling.

The importance of compilers in both computer science and real-world applications is immeasurable. The key points below illustrate some of the most prominent impacts of this creation:

- **Bridging High-Level Languages to Machine Code:** As previously mentioned, the purpose of a compiler is the translation between programs written in different program-

ming languages. Usually, in practice, such translation happens from a high-level programming language, commonly more human-readable and that provides several abstractions, into a programming language that is closer to the hardware and, therefore, more machine-readable. Given this fact, it is safe to say that a compiler provides several layers of abstractions that allow developers to build complex software that impact people's life around the world since its conception.

- **Impact on Portability:** One of the major features of compilers is their portability. In practice, this means that code once a program, written in a high-level programming language, passes through the compiling process, the resulting machine-code can be used in different hardware platforms by simply targeting different machine architectures, such as: x86, ARM, RISC-V, MIPS and others. This allows the same code base to be re-used in independent systems without the need to rewrite the code base in order to target each different architecture.

- **Programming Languages Evolution:** In the current context, it is crucial to reiterate that compilers and programming languages are deeply tied. In practice, the evolution of one serves as a trigger to the evolution of the other and vice-versa. Compilers permit language designers to experiment on new ideas that cross different programming paradigms, such as the procedural, object-oriented, functional, aspect-oriented, concurrent and several others. Furthermore, the evolution of existing programming languages and creation of new ones may allow the enhancement of existing compilers and creation of new ones for niche areas.

- **Impact on Software Development:** Without the existence of compilers, the development of software applications would become far more difficult since there would be no bridge uniting codebases written in high-level languages and hardware systems where such codebases are executed. Therefore, this would limit the growth of technology-based companies.

Given the impacts emerged from the creation of compilers, now it is time to take a deep dive into how this particular type of software works.

As already explained, a compiler is a complex software system and due to this, must be structurally organized in components responsible for executing a single task in the compiling procedure. According to [2], compilers are being implemented since 1955. During these early

years of development, a compiler was viewed as a tool that had to understand the program written in the source language and translate it, without altering its meaning, to a target architecture. The distinction between these two tasks, made computer scientist adopt the following structuring when it came down to the components of a compiler: each compiler should have a front-end and a back-end.

In conformity with what has been exposed, the front-end was responsible for understanding the program written in the source language, while the back-end's responsibility lied on the process of mapping programs to machines. As the reader might have already thought, there must be some kind of link between the front-end and the back-end. That is where the Intermediate Representation (IR) comes in. The front-end must encode the needed information about the source program in some way, so it can be properly used by the back-end when it starts the mentioned mapping process later on. The Intermediate Representation is the entity that contains such information generated by the front-end. It is considered the definitive representation of the source code that will be transformed into machine-code by the back-end.

In short, it can be said that the front-end's responsibility is guarantee that the source code is well-formed and also translate it to the intermediate representation. On the other side, the back-end's duty is translate such intermediate representation into the machine-code of a specific machine architecture, respecting the physical limitations of the target hardware.

As the decades passed on, the process of designing and implementing a compiler became more sophisticated as the structure of such system became increasingly more complex and robust.

Shifting the focus of compilers' implementation to a classroom environment, it is safe to expect that students are capable of implementing more sophisticated compilers that have the following structure, which is not that far from those used in the industry:

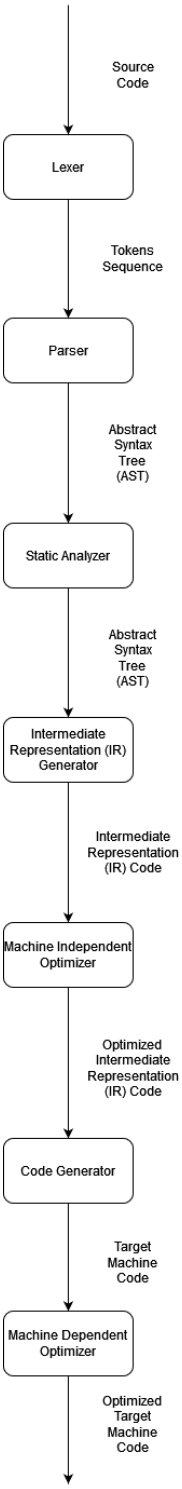

Figure 2.1: A detailed compiler structure

It is also completely plausible to simplify this task by asking the students to implement a compiler that follows the following simplified structure:

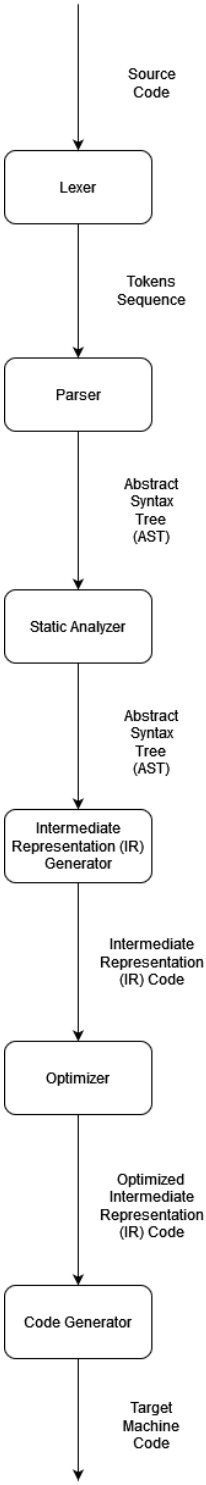

Figure 2.2: A simplified compiler structure

Section 2.4 is dedicated to provide an in-depth overview of each of the compiler's components from the simplified structure presented in the image above.

2.3 What is an Interpreter and How Does it Distinguishes Itself From a Compiler?

In order to explain what is an interpreter, it is necessary to recall the reader what is a compiler and what constitutes the compiling process.

A compiler is a program that takes as input a specific programming language and translates it to another, but without executing it. This step is delegated to the users, who must execute it by themselves.

In essence, compiling means translating code written in a source programming language to a target programming language (that is usually in a lower level, when compared to the source one). Whether the compiler generates machine code or bytecode, it is still performing a compiling process.

Once the reader has remembered what a compiler does and what compiling is, an interpreter is a program that takes as input a specific programming language and immediately executes it.

Keith Cooper and Linda Torczon provide another point of view to the discussion about the distinctions between compilers and interpreters in [2].

According to them, a compiler is a type of software responsible for taking as input a program written in a certain source programming language and producing as output a representation of this very same program as a sequence of the instructions available at the target machine. Normally, the source programming language can be C, C++, Fortran, Java, and ML, while the "target" programming language is just the instruction set of a specific computer architecture (ARM, RISC-V, MIPS, x86, etc).

An interpreter, in contrast, receives as input a program written in a certain programming language, but its output is the result of executing the code written in the source language. Some language that fall into this category are: Perl and Scheme.

The images shown below are good examples that illustrate the input and the output of both compilers and interpreters:

Figure 2.3: Compilers and Interpreters

Finally, it is also highlight that there are some programming languages, such as Java and

Python, whose implementations combine both compiling and interpreting. Considering Java as example, its implementation compiles the program written in the language to a bytecode format. Then, once the program is in the bytecode format, the Java Virtual Machine (JVM) executes the Java application (bytecode format) by running this bytecode inside the JVM (which can also be viewed as an interpreter for this lower-level representation). A similar process is followed by the Python programming language.

With this, it is reasonable to conclude that compilers and interpreters have a lot in common, due to the fact that they perform almost the same set of tasks, such as:

- Analysis of the input program written in the source language in order to figure out if it is a valid one or not.
- Construction of internal models that are used to better understand the structure and meaning (semantics) of the input program.
- Determination of where to store values during the execution of the input program.

2.4 Detailed Overview of Compiler's Components

2.4.1 Lexer

The lexer is the first component present in the compiling process of a compiler. It is responsible for executing a process called lexing, which is also known as lexical analysis.

As earlier elucidated, the input of a compiler is the source code of a program written in a specific programming language. The content present inside the source code is viewed by the compiler as a large string, a linear sequence of characters, that has no meaning for it at all at this stage. The lexer's goal is to transform this sequence of characters that is the source code into a new representation that is more abstract from the compiler's perspective, so it can be passed on to the next component, the parser.

To perform such procedure, the lexer takes in this stream of characters and group them together forming a sequence of entities that convey the idea of a "word". These entities are called tokens.

Making an analogy to linguistics, one can say that the lexer's purpose is to group letters (characters) into words (tokens). Given this analogy, it is essential to inform the reader that,

like words, tokens can have different lengths and different meanings. Some examples are listed below to better illustrate a few cases:

- Single-Character Tokens: `(`, `{`, `+`, `-`, `*`, `/`, etc.
- Double-Character Tokens: `==`, `!=`, `>=`, `<=`, etc.
- Multi-Character Tokens: `123` (a number literal), `"hello"` (a string literal), `true` (a boolean literal), `aVariable` (an identifier), etc.

It is also important to mention that certain characters have no meaning for the lexer during the lexical analysis and, therefore, are completely ignore in the process, such as: whitespace characters and characters that represent comments in the programming language in which the source file was written.

By the end of the lexical analysis, the sequence of characters is transformed into a sequence of tokens by following the syntax rules of the source programming language.

The figure below shows, in an simple example, the input and the output of this component:

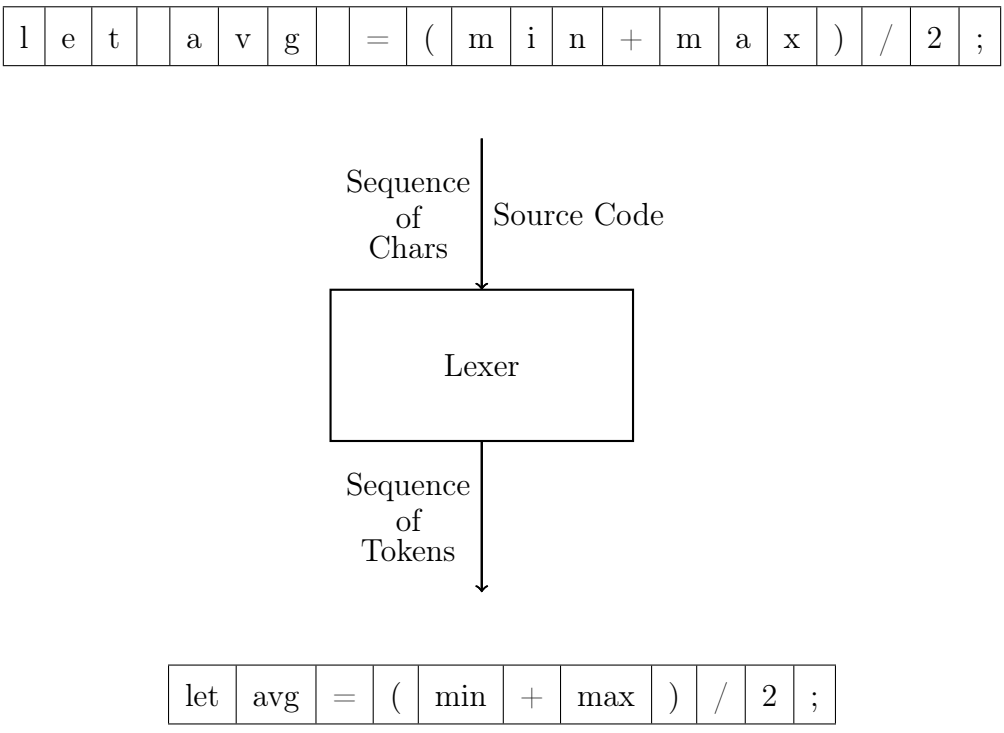

Figure 2.4: Input and Output of a Lexer

As seen in the example above, the lexer receives a sequence of characters, which represents the raw source code, and gives as the result a series of chunks of characters called tokens. Making

another analogy with linguistics, one can think of tokens as the "words" and the "punctuation" that constitutes a programming language.

Going further, it is also possible to say that the lexer's job is execute a traversal through the sequence of characters received and group such characters into the smallest sequences that represent something in the source programming language. Each of these resulting sequences of grouped characters is called a lexeme.

Therefore, in the example presented above, the sequence of lexemes produced from analyzing the sequence of characters received as input was: `let`, `avg`, `=`, `(`, `min`, `+`, `max`, `)`, `/`, `2`, `;`

In conclusion, lexemes are just raw substrings presented in the source code.

However, a token is much more than a lexeme. It generally contains other pieces of information that are useful in further stages of the compiling process and also in error report.

Usually, each token has a token type associated. For example, the lexeme `2` usually has `NUMBER` as its token type, while `+` often has `PLUS` as its token type. In this context, it is relevant to highlight that the amount of token types in a programming language basically depends on the amount of keywords (reserved words) that the language has, the amount of operators at the user's disposal, possibilities of literal values of different data types and, finally, the quantity of delimiters and separators.

Certain types of lexemes have literal values associated with them. numbers, booleans and strings, for example, fall into this category. In these cases, the lexer is able to convert the textual representation of the value into the living runtime entity that might be used by the compiler on later stages of the compiling process.

Last but not least, a compiler must also report errors when it encounters a lexical error during the lexing process, such as: an invalid number literal, an unterminated string, an invalid character that does not follow certain rules of the programming language. In order to report these errors, it is desired that the compiler display as much information as possible to the user about what went and, most importantly, where things went wrong. Having this in mind it is essential to store information about the location of each lexeme in the source code (the line it has appeared and, if possible, its column also).

Now it is time to analyze the functioning of a lexer from an implementation perspective.

The main part of a lexer's code is a loop and variables that are used as "pointers" to keep track of the beginning and ending of each lexeme present in the raw source code file. The lexer then starts working at the first character from the source code and, by using the mentioned

"pointers" along with the lexical rules of the source programming language. For the reader that may not be familiar with it, the term "lexical rules" denotes a set of rules that define how characters are grouped to form lexemes in a programming language .Then, it discovers to which lexeme the current character belongs to and consumes any other following characters that also are part of the lexeme. When the end of such lexeme is found, information about its location and literal value are generated and grouped along with the lexeme in order to build and yield the token itself.

This process starts again, in the next iteration of the loop, at the first character that has not been consumed in the generation of the previous token and continues to traverse the source code and generate the corresponding tokens. The lexer continues to do this until it reaches the end of the file. When this happens, it sends the sequence of produced tokens to the next component of the compiler, the parser.

And that is it for lexers. As explained above its implementation is fairly simple and can be done by the programmer without much trouble. In spite of that, it is possible to delegate this task to certain tools, such as: Lex [4] and Flex [5]. For the sake of exposition and learning, they work in the following way:

1. **Rule Writing:** When using such tools, the first step is to write the acceptable patters of characters of the source programming language through the use of regular expressions. These regular expressions define the lexemes that are useful and will be used to generate tokens. One example of rule could be "A sequence of digits must be treated as an integer number."

2. **Lexer Generation and Execution in the C programming language:** Once the rules mentioned in the previous step have been written, the user can run Lex/Flex and the output will be a C program, the lexer that scans the input source code, identifies lexemes and generates the corresponding tokens based on the rules provided.

2.4.2 Parser

The Parser is the second component present in the compiling process of a compiler. Its job is receive the sequence of tokens generated by the lexer in the previous stage, analyze if the provided sequence of tokens is valid or not by following the syntax rules of the source programming language, the Context-Free Grammar (CFG) and, finally, generating a new representation of

the user's source code called Abstract Syntax Tree (AST). It can be said that, while the lexer performs a lexical analysis, the parser performs syntax analysis.

Going further, this component executes this transformation between the two mentioned representations of an user's code by performing two tasks simultaneously: While the parser is analyzing whether the received token stream represents a valid source code or not through the inspection of the source language's CFG, it also starts generating an hierarchical structure that resembles the idea of "traversing" a CFG. This hierarchical structure is called AST and each of its nodes represent a different construct of the source programming language. Examples of constructs are: variable declaration statements, if statements, loop statements, expressions, function declarations, and many others. In general, it is safe to say that the amount of node types depends on the expressiveness and amount of features of the source programming language.

Therefore, the core roles of a parser can be summarized as shown below:

- **Syntax Verification:** The parser must guarantee that the received sequence of tokens is following the rules of the source language's CFG.
- **AST Building:** It also needs to build this tree representation of the source code, which is based on the token stream yielded by the lexer and the CFG rules. This newly formed tree representation is responsible for representing the syntactical constructs of the input program.
- **Error Reporting:** Finally, the parser is supposed to report any occurrences of syntax errors that might be found during the parsing process, so the user can correctly identify and fix them.

When it comes down to the implementation of this component, there two main classes of techniques responsible for grouping different algorithms that employ different ideas. Such classes are commonly called Top-Down and Bottom-Up:

- **Top-Down:** The techniques that are in this category function with the same principle: Start the parsing process from the topmost rule present in the CFG of the source programming language and attempt to break down the input (token stream) into smaller substructures, by following the CFG rules.
  - **Recursive Descent parsing:** This technique is considered to be the most simple and intuitive one compared to all the others presented here. Nevertheless, it is

relevant to mention recursive descent parsers are fast, robust and can handle error reporting very well. Furthermore, according to Nystrom in [3], this technique is applied in several famous programming language implementations, such as: GCC, V8 (Google Chrome's JavaScript Runtime) and Roslyn (Microsoft's C# compiler written in C#).

Since it is a technique that belongs to the Top-Down class of parsing techniques, this approach starts the parsing procedure from the topmost (or outermost) rule from the source language's CFG. Then, it starts "traversing" the CFG, navigating through subexpressions all the way down until it reaches the leaves of the AST (that is, until it reaches a sequence only composed of tokens or it finds a syntax error).

The simplicity and intuitiveness of the recursive descent parsing comes from the fact that when implementing it, every rule of the language's CFG becomes a function and the body of the rule can be easily translate into code as shown in the table below:

Grammar notation	**Code representation**
Non-Terminal Symbol	Call to the function that represents that rule
Terminal Symbol	Code to match and consume a token
`|`	`if`, `switch` statement
`*`, `+`	`for`, `while` loop
`?`	`if` statement

Table 2.1: Translation between CFG's notation and its equivalent code implementation.

The "recursive" term in recursive-descent parsing comes from the fact that if the non-terminal symbol of a rule's head is also present in the same rule's body, then when the translation to code happens, recursive function calls will happen depending on the source code provided as input to the compiler.

Its advantages are the easiness and simplicity to implement it by hand and, due to that it is an excellent option to deal with programming languages that have simple CFGs and also for educational purposes.

On the other hand, its disadvantages are the fact that it cannot handle left-recursive CFGs (those which have rules where non-terminal refers to itself at the start of the same rule) and it can become increasingly more complex to extend and maintain as the CFG of the programming language becomes larger and more complex.

– **LL parsing:** This technique is considered to be more methodical when compared

to the previous one. Such approach reads the token stream from left to right and produces a leftmost derivation of the input by using K tokens of lookahead. Thus, if a parser is called LL(1), then it means that it has 1 token of lookahead. If it is LL(2), it has 2, and so on.

The advantages of this procedure is that it is predictive parsing, hence there is no need to execute backtracking, which would slow down the parsing process as a whole, and it is more than enough to deal with programming languages whose grammars are simple and have no left-right recursion, which makes it a good candidate when implementing a project for a Compilers course at college.

The disadvantages of it is that this technique is, unfortunately, limited to CFGs that are LL(k). Which means that, in these scenarios, there is a necessity of first rewriting the grammar to be LL(k) and, only then, implementing the parser itself with this approach.

- **Bottom-Up:** The techniques that are in this category function with the same principle: Start the parsing process by examining the received token stream and trying to construct the AST by working from the terminal symbols of the language's CFG all the way up to the start symbol of the topmost CFG rule.

  - **LR parsing:** This technique is very similar to the previously explained LL parsing. It starts out by processing the received sequence of tokens from left to right, however it produces at rightmost derivation (instead of the leftmost one) in reverse using K tokens of lookahead.

    The advantages of this procedure is that it is more robust and powerful when compared to LL parsing, due to the fact that it can deal with a wider variety of CFGs, including those that are left-recursive. Moreover, such approach is considered to be a good choice when implementing machine-yielded parsers.

    The only downsides of this approach are the its complexity of implementation when compared to the Recursive-Descent and the LL approaches, and its defying nature when it comes to debugging.

In [3], Nystrom mentions other exotic parsing techniques, such as: LALR parsing, SLR parsing, Earley parsers, Pratt parsing, packrat parsing, parser combinators and the Shunting-Yard algorithm.

There are also tools that automate the process of building a parser. A few examples are: ANTLR [6], Bison [7] and Yacc [8]. All of these tools operate, in essence, in the same way, with a few peculiarities and twists for each one. Basically, they take the CFG of the source programming language as input and produces as output a parser that can recognize valid and invalid programs written in that language. Such tools are very useful because they automate the burdening process of writing a parser by hand, especially when dealing with large CFGs.

Lastly, in order to better illustrate what a parser does, the image below shows what the parser might generate for a very simple program:

First, consider the following sequence of tokens received from the lexer:

let	seven	=	2	*	3	+	1	;

Figure 2.5: Example of token sequence provided as input to the parser.

The output generated by the parser would be something very similar to the following AST:

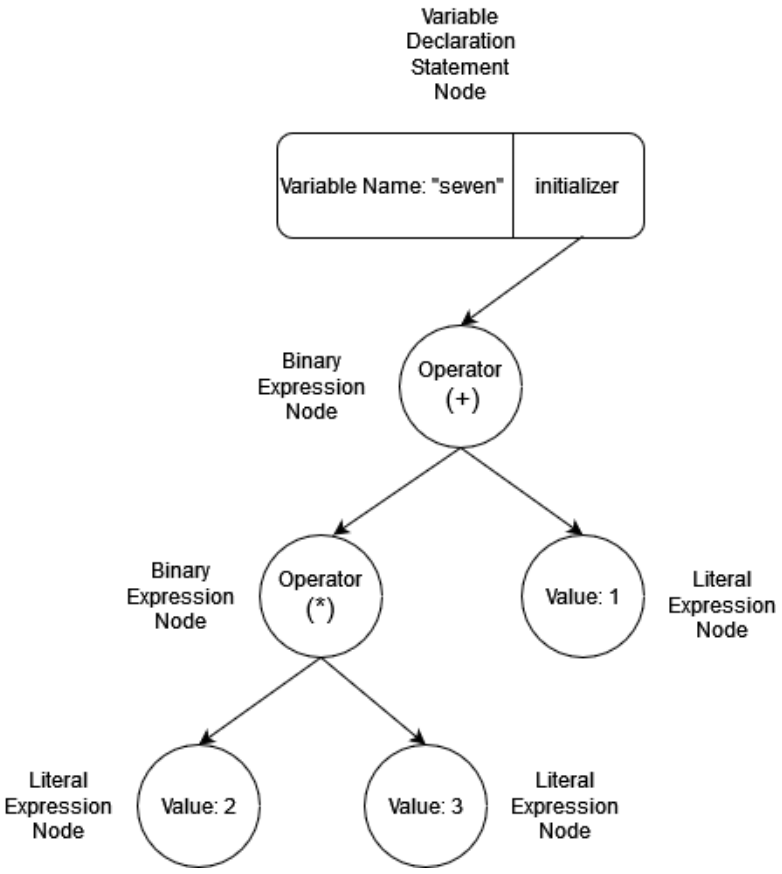

Figure 2.6: Example of AST generated as output by the parser.

2.4.3 Static Analyzer

The Static Analyzer is the third component present in the compiling process of a compiler. It is responsible for performing a task called static analysis. Static analysis essentially means examining the source code without executing it, hence the name. Delving into it, the static analyzer's task is certify that the source code stick to correctness rules defined by the source programming language design and principles.

Such correctness rules include and are not limited to:

- **Scope and Name Resolution:** This set of rules is responsible for ensuring that variables, functions and classes/types are declared before they are used. In addition, it is also in charge of making the correct bindings between an identifier use and its corresponding declaration, considering the scopes presented in the program's source code.

- **Type Checking:** Is an action performed by the static analyzer responsible for ensuring that functions, expressions and variables are properly used. For example: a value of type `int` cannot be multiplied by a value of type `bool`, if in a certain function with one parameters expects a value of type `string`, a value of type `int` cannot be passed as an argument in a call to such function.

- **Detection of Syntax Errors:** Although the parser is responsible for detecting such class of errors, some of these are easier detected during the static analysis stage. For example, a static analyzer can easily detect the misuse of the `return` keyword outside of a function (default or anonymous) or method. It can also detect the misuse of the `break` and `continue` keywords outside of loop statements. There are more examples related to these types of scenarios, but the two mentioned above are good and common examples.

- **Detection of Undeclared Variables and Functions:** As the rule's name suggests, a static analyzer is capable of recognizing the usage of variables and functions that have not been previously declared by the developer. It is also worth mentioning, that certain static analyzers take a step beyond and warn the user about variables and functions that have been declared but were never used in the program's source code.

Moreover, it is very important to remember the reader that the enforcement of such rules are performed on the Abstract Syntax Tree (AST) provided by the parser in the previous compiling

stage, and the output of the static analyzer component is the same, the Abstract Syntax Tree (AST).

Finally, yet importantly, it is essential to highlight the fact that in mature compilers (Clang, GCC, javac, Roslyn, rustc, etc.) there is usually more than one static analyzer component where each one of them is responsible for receiving different representations of the program's source code and, consequently, enforcing different types of rules according to the philosophy of the source language.

For instance, there are static analyzers that are able to identify unused variables from just traversing the generated AST from the parser.

Last but not least, there are static analyzers that only operate on certain representations of the user's source code. Some static analyzers operate on Intermediate Representation (IR), which is a different type of representation of the source code. It is worth mentioning that this one is more abstract and it is designed to be easier to analyze compared to an AST as it removes details that are too specific of the source programming language. Analyzers that act on this type of representation can perform more complex analysis, such as Control Flow Analysis (whose focus is on detecting unreachable code and ensuring the non existence of infinite loops) and Data Flow Analysis (whose focus is on tracking how values flow between variables).

2.4.4 Intermediate Representation (IR) Generator

After the previous phases, it is time for the IR generator comes into play. This component receives as the AST from the Static Analyzer and transforms it into another representation of the user's source code that is generally closer to machine-code or bytecode, which, by consequence, makes it also closer to the hardware.

Before explaining with more details what an IR really is and how it works it is essential to remind the reader that the compiler is divided into two big components: the front-end (responsible for dealing with the source programming language) and the back-end (responsible for dealing with the target computer architecture).

Given this structure, there usually exist some kind of code representation that works as a bridge between the front-end and the back-end. Given this motivation, the IR normally is not tied neither to the source programming language and nor to the machine-code of the target architecture or bytecode. By using an IR to link the front and back-ends, the user can implement multiple compilers targeting different architectures with less effort.

The image below shows the huge different that the use of an IR generator component makes inside a compiler:

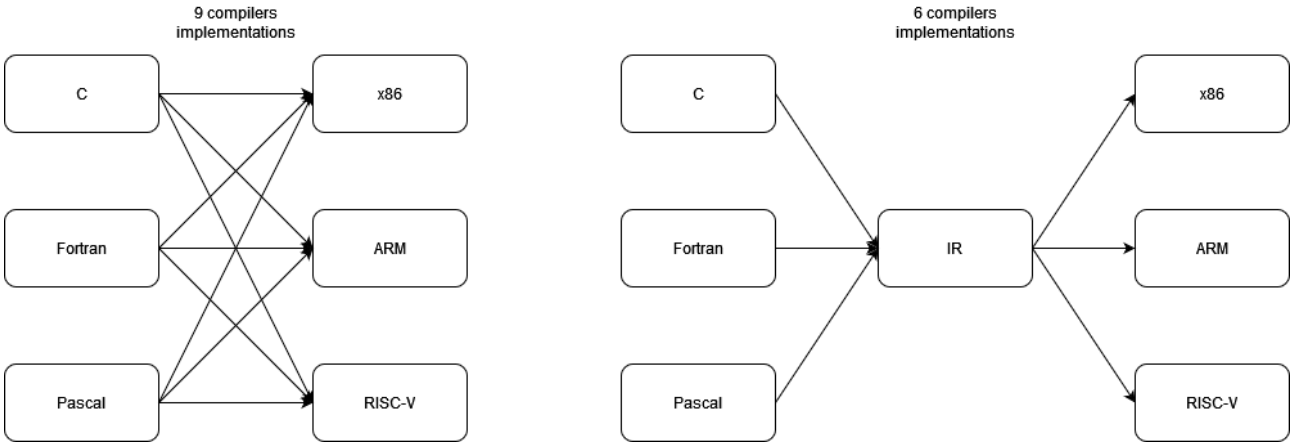

Figure 2.7: Impact of IR in multiple compilers implementations.

Basically, the developer implements only one front-end for each source language that produces the IR. Then only one back-end for each target architecture. Now, one is able to combine those to get every possible combination of source language and target architecture.

2.4.5 Optimizer

After the previous component emits the code in the intermediate representation (IR) format, it is time for the compiler to make the program more efficient in terms of execution time and memory usage without changing the meaning of the program (i.e., its semantics). For that purpose, the optimizer component comes in.

Before providing an overview of optimization techniques, it is important to inform the reader that this subfield of programming languages design and compilers implementation is still very active, with new discoveries and improvements being made in a regular basis. Therefore, this subsection will focus on the most well-known and popular optimization techniques.

Optimization techniques can be divided into two different groups: Machine-Independent Optimization Techniques and Machine-Dependent Optimization Techniques. The first group consists of techniques that can be applied to the user's program in the IR format, this is why they are called machine-independent. The second group is made of optimization techniques that require information about the target hardware architecture in order to be applied, hence the name machine-dependent.

Now, it is time to take a look at these two groups in more details:

- **Machine-Independent Optimization Techniques:**

    1. **Constant Folding:** This optimization consists of checking whether there is an expression in the code that always evaluate to the same value. If that is the case, then, the compiler can perform such evaluation at this stage and replace the code related of the expression with the computed resulting value.
    2. **Constant Propagation:** This technique is responsible for identifying constants in the code and then replacing the use of such constants with their respective values throughout the code.
    3. **Common Subexpression Elimination (CSE):** The aim of this optimization is identify expressions that are computed several times with the exact same operands inside the code and, then, replace all of them with a single computation, making the program faster by not requiring the computation of the same expression multiple times.
    4. **Dead-Code Elimination:** This optimization technique aims to eliminate code that has no effect on the behavior of the program, such as: declared variables that are not used and code placed after a `return` statement.
    5. **Inlining:** This one's goal is to remove function calls from the program and include the body of the function where the call to such function was made. By doing this, the overhead related to a function call is reduced, making the program faster.
    6. **Loop Invariant Code Motion:** This optimization technique, as the name suggests, move operations that generate the same result in every iteration of a loop to the outside of it. By performing this action, redundant computations are eliminated, making the program run faster.
    7. **Strength Reduction:** Such optimization approach is responsible for replacing expensive operations, from a computation point of view, with cheaper ones that do not alter the program's behavior.

- **Machine-Dependent Optimization Techniques:**

    1. **Instruction Scheduling:** This optimization consists of reordering instructions of a particular CPU architecture in order to make a better usage of its instruction pipeline and avoid possible stalls. This sort of optimization takes into account the

latency of each instruction and also the availability of finite physical resources, such as registers.

2. **Register Allocation:** This optimization is responsible for improving the assignment process of variables and produced temporary values to the limited amount of registers from a CPU. Since a CPU has a limited number of registers, this optimization tries to find out which variables and values should be kept in registers and which should be stored in memory, with the purpose of enhancing the performance of the program (it is worth remembering the reader that an access to a register is faster than the access to the memory). Therefore, it is essential to find out which variables and values are more used in the program and store them in registers instead of the memory.
3. **Instruction Selection:** The goal of this optimization technique is, given the IR representation of the source code, choose the most efficient set of instructions from the target hardware architecture to perform the required operations.
4. **Branch Prediction Optimization:** This optimization's objective is alter the code configuration in order to minimize the probability of a branch misprediction occurs. Normally, this is achieved by rearranging the code in such way that execution paths that are frequently executed are placed closer and also in a way that matches the CPU's prediction policy.
5. **Vectorization:** This technique's purpose is convert scalar instruction into vector instructions in order to leverage the SIMD (Single-Instruction, Multiple Data) capabilities of modern CPUs. By doing this, the execution time of the user's program is usually reduced.

Finally, as seen above, there are two different groups of optimizations that a compiler can perform. Therefore, it is very common that the pipeline of a compiler has two different optimizers, one right after the IR Generator (that deals with machine-independent optimizations) and another right after the Code Generator (responsible for machine-dependent optimizations).

2.4.6 Code Generator

Once the compiler has performed the machine-independent optimizations to the adopted IR, it is time to generate code for the target CPU architecture. This task is executed by the

Code Generator component.

In this context, it is important to highlight to the reader that the generated code is just Assembly code of a specific hardware architecture, whether it is x86, ARM, MIPS, RISC-V or another.

When it comes down to the implementation of this component, two possible paths can be taken by the implementer: one can generate code for an existing architecture, that is a real CPU, or for a virtual one.

If the first way is taken, then the compiler generates machine code for a real CPU and, after that, an executable is yielded and, finally, can be load into the memory of the system by the operating system. The advantage of this approach is that the native code generated is extremely fast since it leverages the most of the architecture that it is targeting. However, the downsides might be intimidating, in order to achieve such speed, one often needs to have a deep knowledge about the architecture that the implemented compiler has selected. Also, once the selection is made, the compiler is tied to such architecture. This means that, if the compiler targets the RISC-V architecture, then it cannot work on a x86 or ARM device.

According to [3], as a mean to deal with these disadvantages, Martin Richards and Niklaus Wirth developed their compilers to generate code for a virtual machine, instead of a real one. In practice, this means that their compilers produced code for an hypothetical and idealized machine rather than for a specific real CPU chip.

Advancing into this topic, the intention behind designing these synthetic instructions of a virtual machine is very clever: It allows the compiler developer to create instructions that are linked to the semantics of the source programming language without needing to worry about the peculiarities of any CPU architecture.

2.4.7 Virtual Machines

As seen above, a virtual machine is one of the possible approaches when it comes to the code generation phase of a compiler.

If the developer has chosen to built a compiler that emits bytecode, then there is still one task left to be done. Since, the compiler yields bytecode, no CPU architecture is able to understand what this representation actually means.

In this case, there are two options available again:

- Implement a "mini-compiler" that performs a translation between the generate bytecode

and the target machine architecture. In this scenario, it is important to pay attention to two facts: First, the produced bytecode will be essentially acting as a sort of intermediate representation since a translation between it and the target chip will need to happen. Second, the developer is again tied to the underlying hardware architecture. For each computer architecture, a new "mini-compiler" will have to be constructed.

- Implement a Virtual Machine (VM). In simple words, this is a program that emulates a chip that is able to fully understand the bytecode generated by the compiler. As one can already infer, virtual machines have their own trade-offs. Executing bytecode in a virtual machine is slower than directly executing native code in a computer architecture, due to the fact that each bytecode instruction must be simulated in software every time it executes instead of being executed by the hardware itself. On the other hand, by implementing a virtual machine, one gains a lot in terms of simplicity and portability. The need to translate the bytecode generated by the compiler to the machine code of a certain architecture is eliminated and say that the virtual machine was implemented in C or C++, then the implemented programming language can run in any platform that has a C or C++ compiler installed in it.

2.4.8 Runtimes

If the developer has opted to make the code generator directly produce machine code for a certain computer architecture, then the last step is tell the underlying operating system of the computer to load the executable file in the memory and execute it.

On the other hand, if the developer has opted to make the code generator yield bytecode, than the last step is start up the virtual machine, send the generated bytecode to it and execute the VM program.

In both of the mentioned cases, there is usually a need to have an environment that whose responsibility is provide support while the program has started its execution. Such piece of software is an essential part of the programming language implementation called Runtime.

The amount and type of responsibilities assigned to a runtime often varies according to the nature of the implemented programming language. However, some responsibilities are the same across different types of programming languages:

- Interface with the Operating System.

- Memory Management.
- Execution of Built-In Functions.
- Process and Thread Management.
- Type Checking (for Dynamically-Typed Languages).

2.5 Shortcuts and Alternate Routes

This section is dedicated to present an overview of a few alternate ways of implementing a programming language besides the structured route exposed in the previous section.

This is by no means an exhaustive list of possible approaches to be taken. Instead, it provides an overview of the most common and simple ones.

2.5.1 Single-Pass Compilers

A single-pass compiler is simple type of compiler that operates by interleaving the parsing, static analysis and code generation steps inside the parser component in order to generate the output code.

These kinds of compilers function without ever needing to use an AST or another sort of IR.

This simplicity, however, comes with a cost. The language design becomes severely restricted due to the way a single-pass compiler works. The developer does not have any kind of data structure to store information about the program and once some part of the code is parsed, it won't be re-visited later on, hence the name single-pass compiler. Essentially, this means that every time a statement or expression is parsed, the compiler needs to have enough information in order to correctly compile it.

The technique that enables single-pass compilers is known as "syntax-directed translation". In this approach, the developer establishes a link between a component of the programming language's grammar, one that outputs code, and an action to be executed. Then, when the parser starts its job, as soon as it identifies a piece of grammar that has an action associated, it executes such action, emitting the output code, step by step.

According to [3], both C and Pascal were firstly designed with the mentioned limitation in mind. It is worth remembering the reader that, at the time of these languages' conception,

memory was very scarce. At certain times, compilers were not even able to deal with a single source file, let alone an entire program.

2.5.2 Tree-Walk Interpreters

A tree-Walk interpreter is a simple kind of interpreter that usually works by executing code right after the parser generates the AST.

In this scenario, a tree-walk interpreter runs the user's source code by executing a traversal on the received AST and performing different types of actions by taking into account the type of node that it is currently on.

Due to its simplicity, this kind of programming language implementation is very common in student projects and toy languages. However, it is not often used in general-purpose language, since the nature of the code execution makes it slow, with one exception, according to [3], being the earlier versions of Ruby, which were implemented as tree-walk interpreters up to version 1.9, when the language implementation was changed to a bytecode virtual machine.

2.5.3 Transpilers

A transpiler, also known as source-to-source compiler or transcompiler, is special type of compiler. The idea behind it is very simple: Instead of translating the program written in a source programming language to some kind of intermediate representation (AST, IR, CFG, etc), it translates the program to another source programming language.

Going further into its functioning, the developer writes the front-end components of a compiler as usual. Nevertheless, in the compiler's back-end, the programmer generates a string of valid code in another programming language that is as high-level as the source language, while preserving the meaning and semantics of the original program.

Finally, the programmer can use the compiler for the chosen high-level target language in order to delegate the heavy lifting of the back-end stage to this compiler.

One of the most common use cases of a transpiler occurs in the context of web browsers. Nowadays, JavaScript is, by far, the most common choice to run code in it. However, JavaScript might not provide the best coding experience from certain developers' point of view. Due to that, a growing increase in the number of transpilers that have JavaScript as their target language has been observed, as can be seen in [9].

Chapter 3

Context and Purpose

"I wonder if I can keep up with... the speed of a world you're not in."

– NORIAKI KUBO

3.1 Introduction

This chapter is dedicated to provide the historical context on how the way of teaching undergraduate Compilers courses evolved since the 70s, as well as a brief overview of surveys that demonstrate the positive impact that more practice-oriented approaches brings to students. Such factors are what motivated the creation of the Bleach programming language. Furthermore, a subsection dedicated to recall the purpose of Bleach is provided. Then, an analysis about how Bleach distinguishes itself from other proposed programming languages that also aim to be used in Compiler courses is presented. In the end, the chapter provides a detailed description about the context Bleach is inserted in, which includes: the target audience, educational environment, typical use cases and the educational benefits provided by it.

3.2 Historical Evolution of Compilers Courses

Looking back at the history of compiler construction [10] and the history of programming languages [11], it can be said that the way of teaching an undergraduate Compilers courses evolved as follows.

During the 1970s, undergraduate Compilers courses used to have a heavy focus on the theoretical aspects of the field, such as formal languages, automata theory, lexing techniques

and parsing techniques. During that period, access to computational resources were still scarce, even in an academic environment. However, students were still able to implement toy compilers in different tastes of Assembly or in an early high-level language like Fortran [12]. Still, the pillars of this course emphasized a deep understanding of the theoretical and formal aspects mentioned above.

In the 1980s, even though the focus of these courses remained rooted in theoretical aspects (lexical analysis, syntax analysis and the fundamental of code generation and optimization), a gradual shift from theory to practice started to happen. Thanks to the spread of languages like C [13], C++ [14] and Pascal [15], more practical implementations of compilers were allowed to be made. Therefore, courses started to add more practical projects. In such projects, students were usually asked to write a compiler (or parts of one) in these new languages. Last, but not least, the rise of UNIX [16], Lex [4] and Yacc [17] allowed students to implement components of a compiler with more ease. This improvement led to more sophisticated course projects.

In the 90s, such courses had already become more oriented to hands-on experience thanks to the improvements achieved in the previous decade. The birth of languages like Java [18] and the increase in computational power of personal computers allowed students to tackle more complex compiler projects. Courses started to assign to students more complicated tasks like the implementation of entire compilers for small languages or significant compiler components, such as code generators and optimizers. Moreover, the textbook [1], popularly known as "The Dragon Book", became the standard reference in this field.

In the 00s, this trend of giving more focus to the practical experience went on, with a focus on real-world applications and the use of modern programming languages. Furthermore, courses began to add more advanced topics to their syllabus, such as just-in-time compilation, garbage collection, and runtime environments. It's safe to say that the way of teaching Compilers became more holistic, covering not just compiler theory and implementation but also language design and performance optimization. During this period, Alfred Aho published [19], an article in which he reflects on how the way of teaching undergraduate Compilers courses has evolved through the years and also how he changed his way of teaching such course. According to him, it is still possible to teach it in a way that provides educational benefits and satisfaction to the students. The latest approach adopted by him consisted on presenting the basics of the most important topics (lexical analysis, parsing, semantic analysis, intermediate code generation, runtime environments, resource management and target code generation). At the same time,

Aho asked his students to implement a compiler and provided the specifications of the source and target languages for the compiler they were implementing. He also noticed that the students weren't excited about implementing someone else's language. Thus, he took a bold move and assigned his students the task of working in smalls team to define their own language and then build a compiler for it. Such approach, according to him, made the students much more engaged and excited.

Finally, in the 2010s and 2020s, the teaching of this course continued the trend of emphasis on balancing theory and practice. Such factor influenced the way Compilers course were taught, as professors and instructors began to integrate more diverse and flexible learning materials, including video lectures, interactive simulations, and online coding platforms. Also, the COVID-19 pandemic played an important role in this change, since it forced educators to rethink how compilers could be taught in a remote manner, which led to rise of new assisting tool like virtual laboratories, collaborative coding environments and online assessment tools. Consequently, such courses became more modular, allowing students to learn at their own pace and apply their knowledge in a way that felt meaningful to them.

3.3 Reports on More Practical Ways of Teaching Compilers Courses

In this section, a few reports that discuss different approach to teach an undergraduate Compilers course are presented, along with their main findings.

In 2015, Lasseter [20] proposed a new strategy to present the fundamental concepts of an undergraduate Compiler Construction course. The suggested approach defends that the concept of an Interpreter serves as an effective conceptual framework that can be used to educate students about the later stages of the compilation pipeline, specially the semantic analysis and code generation phases. As reported by the author, this line of action is useful in the way that it not only unifies the major theoretical concepts of this undergraduate course, but is also helpful when it comes to the implementation of semester-long compiler construction project. At Hobart & William Smith Colleges, the university where Lasseter works, the undergraduate Compiler Construction course is a one-semester, upper-division elective, offered every other year. The course addresses both software engineering aspects as well as the major concepts of a traditional Compilers course (lexing, parsing, semantic analysis and code generation). Moreover, it already

adopts a more practice-oriented approach where every student must present a working compiler by the end of the 15-week semester. When it comes to the course project itself, Lasseter opted to use the Tiger programming language [21], originally developed by Andrew Appel for his project-oriented "Modern Compiler Implementation" suite of C, ML and Java textbooks [22], [23], [24]. During the years in which the author was responsible for this course, he noticed a worrying pattern: Even though the students did well in the first stages of compilation (lexing and parsing), the vast majority of them started to struggle with the semantic analysis stage. According to the students, the combination of the underlying concepts of type checking, the details of the famous visitor pattern when traversing an AST and the interaction between this design pattern and a real-world semantic analysis implementation is just too overwhelming. Given this scenario, Lasseter chose to schedule individual meetings with this students where he introduced the idea of a language's interpreter as a touchstone to better explain the last two stages of the compilation pipeline. His investigation led to the fact that such approach proved to be the to be most uniformly effective among the students with whom he discussed. In light of this discovery, professor Lasseter reorganized his lectures introducing these specific compiler phases to include the interpreter structure as a reference point. Finally, he came to the conclusion that adding lectures that presented the concepts behind the interpreter framework resulted on better understanding of the students when it came down to the semantic analysis and code generation phases of the compilation process, indicating that the idea of an interpreter has a substantial educational value in this field.

In 2016, Kundra et al. [25] proposed a novel approach called "Case-based and Project-based Learning" to teach a Compilers Design course to B.Tech third year students of a Delhi University (India) college. The responsibles for implementing such methodology based themselves in the definition of what is Traditional Learning present in [26] and used the definition and core ideas of Case-based Learning present in [27] in order to propose their own method. According to the authors, this proposed method combines 4 different pedagogical models, which makes this approach unique and effective: Didactic Teaching [26], Problem-Based Learning [28], Cognitive Apprenticeship Model [29] and Project-Based Learning [30]. To implement it, they divided one core project of this class into several sub-projects with the goal of improving the practical experience of the students when it came down to the task of designing a compiler. To evaluate the effectiveness of this approach, students were asked to complete a survey that grasped on their perceptions about the upsides of such method. Such survey was analyzed using the following

statistical tools: frequency estimation and chi square test of association. According to the paper, the results showed that the proposed approach indeed had a positive impact on students since it enhanced their learning, critical thinking, engagement, communication skills and team work. In the end, the authors came to the conclusion that the outcome of the survey analysis proved that both Case-based and Project-based learning are suitable for teaching concepts of compilers implementation.

In 2021, Robert Nystrom, a software engineer that works at Google on the Dart language, published the "Crafting Interpreters" book [3]: An extensive guide about not only implementing two different types of interpreters for Lox, a full-featured programming language designed by the author, but also a walkthrough that guides the readers on how they can design their own programming language. The book assumes that its the reader first contact with this field. Therefore, it covers each concept needed in an approachable style while also providing every line of code needed to implement the two types of interpreters covered. As the author himself admits in the first chapter of the book, "Crafting Interpreters" is not meant to be as rigorous as other references, such as [1] or [2] when it comes to the theoretical foundation of programming languages. Instead, as suggested earlier, its purpose is to be lighter in theory, while still introducing the history and core concepts of programming languages implementation. Moreover, the author stands by the point of view that the best way of learning a new subject is by practicing and experimenting. He even says that his goal is that every reader finish his book with a solid intuition about how real programming languages work. The book also aims that the reader will feel more comfortable when reading more advanced and theoretical books later on, such as [1], [2] or [31]. It's important to keep in mind that the book's focus is on building interpreters, which are simpler and more accessible to understand when compared to compilers. Furthermore, this way of teaching allows the students to have a better comprehension of vital concepts previously mentioned, like lexing/scanning, parsing, abstract syntax trees, semantic analysis and runtime execution without the daunting complexity of optimization and code generation phases that most compilers require. This approach brings a lot of benefits for students and professors. With respect to the students, they are usually able to build a deeper and stronger foundation about how programming languages work since the project-based nature of the book makes them apply what they learn immediately. This solid foundation is, without a doubt, of great use to those who wish to delve deeper into the field. As for the professors, the engaging writing style of the book tends to keep students engaged and motivated during

the whole duration of the course, which leads to a better retention and a more rewarding learning experience. Last but not least, it's important to highlight the fact that the "Crafting Interpreters" book is increasingly gaining more popularity and recognition due to its excellence when it comes to conveys the subject of study. It's purchase page on Amazon [32] is full of reviews praising its quality. Searching for the keywords "crafting interpreters" [33] shows that there are more than 2600 public repositories implementing the Lox language presented in the book, which is also a testimonial of the popularity and quality of such material.

3.4 Existing Toy Programming Languages

Before taking a deep dive in what makes Bleach an actual and viable option to be used in a classroom environment, it's important to mention that there are several programming languages created with the purpose of being a supplementary tool for teaching undergraduate Compilers course that have opted for a more practice-oriented procedure.

In this section, some of the most popular programming languages designed with this motivation will be discussed in chronological order. The intention with this is to provide information about their origin, historical context and features.

- **COOL (Classroom Object-Oriented Language):** This is a object-oriented and statically-typed programming language [34] created in the 90s by Alexander Aiken and his colleagues at the Stanford University. It was designed with the intention to be used as a tool in an educational setting in universities. Even though COOL is simple and small, it has an interesting variety of features besides the ones mentioned above, such as: basic class types (Bool, Int, IO, Object and String), automatic garbage collection and many others that can be found at "The COOL Reference Manual" [35].

  According to Aiken, a compiler project is normally the most complex software project that most undergraduate students will complete. Therefore, the use of a full-featured programming language (any language that has a substantial number of users, like C, C++, Pascal, etc) does not fit into the scope of such project. In practice, the usual approach was to select a subset of these popular languages and ask the students to implement them. However, according to Aiken, this method is too demanding for the teaching staff, since several steps must be taken after choosing a language for a Compilers project. For example: a detailed specification of the project itself must be written, any supporting

software that might be used must be developed, tested and documented beforehand. Finally, the project itself must also be implemented by the teaching staff *a priori* since, according to Aiken, such effort is required to guarantee that the project is complete, consistent and tractable.

It was with all the efforts mentioned above to build a framework for a compiler project in mind that Aiken created COOL. Ultimately, it is worth mentioning that COOL is not only a programming language, but rather a freely available and portable compiler project. COOL has being distributed since its creation with the goal of helping professors from other institutions to leverage from the efforts of the project developers and the many students who have written COOL compilers in order to structure their own Compilers courses following their own tastes. COOL has been adopted at several relevant institutions, such as University of California, Berkeley [34] (where it was first introduced), Stanford University [36], University of Illinois Urbana-Champaign [37] and University of Michigan [38].

- **MiniJava:** This programming language [39] is a subset, as its name suggests, of the Java programming language. MiniJava was created by Andrew Appel and Jens Palsberg during the 90s and was first presented as part of [23]. This subset of Java contains its most core features, such as: classes, attributes/fields, methods, logical control structures (if-else) and loop control structures (while). However, since it was tailored for an educational environment, it doesn't contain Java's more advanced features, like: exceptions, generics, inheritance, interfaces and lambda functions. More details about which features MiniJava exactly keeps can be found at [39] and [40].

  Its purpose is to be used as a teaching tool in Compilers courses. Since it is a subset, as previously mentioned, it allows the students to focus on the fundamental concepts of this subject without overwhelming them with the complexities of a full-fledged programming language like Java. A proof of its success is that MiniJava has been adopted in several prestigious universities worldwide as part of their computer science curriculum, particularly in courses related to compiler construction and programming language implementation, such as University of Washington [41], University of California, Los Angeles [42], among others, like the Loyola University of Chicago [43].

- **C* (C Star):** The C* (C Star) programming language [44] is a programming language

created in 2017 by Christoph M. Kirsch as a component of the Selfie Open-Source Software Project [45] [46].

According to Kirsch, the creation of C*, as well as the Selfie project as a whole, was inspired by seminal works made in different sub-fields from Computer Science, such as: Algorithms [47] [48], Compilers [49], Computer Architecture [50], Operating Systems [51], Programming Languages [13] [52], Systems Engineering [53] [54] [55] [56] and Theory of Computation [57]. In addition, it was motivated by the challenge of teaching the basics of this field to larger audiences that are not specifically majoring in Computer Science.

Regarding adoption of C*, the language has been adopted by the University of Salzburg at Austria [45] and by the Czech Technical University (CTU) at Czech Republic [58].

C* has support for 5 different types of statements (assignment, if-then-else, while loop, function call, return). Related to this, the language also implements the following native built-in functions: `exit`, `malloc`, `open`, `read` and `write` (it is important to notice the absence of `close` and `free`). Regarding to keywords, C* has just 6: `int`, `while`, `if`, `else`, `return` and `void`. Talking about functions, C* allows them to have parameters, local variables and a return value. With respect to operators, C* has support to the most common arithmetic (`+`, `-`, `*`, `/`, `%`) and comparison (`==`, `!=`, `>`, `>=`, `<`, `<=`) operators. However, it does not support bit-wise nor boolean operators. There are just 2 data types present in C*: signed integer (`int`) and pointer to signed integer (`int*`). It's worth mentioning that C* has support for integer, character and string literal however. Finally, when it comes to the access of dynamically-allocated memory, the only way of doing so is through the use of the dereferencing operator (`*`).

- **ChocoPy:** ChocoPy is a programming language created in 2019 and first presented at Padhye et al. [59] with the goal of being used as a teaching tool in an undergraduate course on compilers and programming language. It is a subset of Python 3.6 that uses static type annotations in order to ensure compile-time type-safety. The project stands out due to its complete specification, which uses a formal grammar, typing rules and operational semantics as described in [59]. More technical information about ChocoPy can be found at ChocoPy's official website [60].

  According to the authors of [59], the purpose of creating ChocoPy was based on the following 5 major reasons:

- **Familiarity:** The staff behind the project wanted to develop a language that was a subset of a widely used programming language, hence, prioritizing the familiarity of the students with it. With this reasoning in mind, Python [61] was chosen.
- **Expressiveness:** The professionals responsible for ChocoPy wanted to create a language that allowed students to write non-trivial programs in it. Particularly, an Object-Oriented paradigm with enough complexity got chosen to be supported. This option was made in order to illustrate important language implementation concepts, such as optimized code generation and type-checking at compile-time.
- **Formally Specified:** In order to make a robust tie the theoretical concepts presented in an undergraduate Compilers course with their application in an implementation project, the authors opted for a language whose syntax, type-checking rules and operational semantics were all formally specified. With these needs kept in mind, Python was a good alternative.
- **Modern Target Language:** Instead of targeting an old or too complex Assembly language. Too make the code generation less intimidating, the instructors instead opted for targeting the 32-bits RISC-V Instruction Set Architecture [62].
- **Re-usability:** One of the major reasons for ChocoPy's creation was the democratization of artifacts that could be freely used by professors, instructors, teaching assistants and students from different institutions offering Compilers courses.

The language started out being used at the UC Berkeley, as stated in [59], and since then has been adopted by several other institutions, such as: TU Delft [63], UC San Diego [64] and NYU [65]. Finally, the ChocoPy language specification was also used as a reference at IIT Bombay [66].

As previously stated, ChocoPy was designed to be a subset of Python. In practice, this means that every valid ChocoPy program that does not result in a runtime error is also a valid Python program. Therefore, the result observed when executing the said ChocoPy program is equal to the one when executing the same program in a Python Interpreter. When it comes down to language features, ChocoPy has support for statements, expressions, assignments and control-flow structures (conditionals and loops). When dealing with evaluation of an expression, the result will always have one of the following types: a boolean, an integer, a list, a string, an user-defined class or the special value `None`. Un-

fortunately, according to the authors of [59], the language does not have support for dict type, first-class functions and reflective introspection. It is also worth mentioning every expression in ChocoPy is statically typed. On top of it, variables (both global and local ones) and class attributes are also statically typed and have explicit type annotations. Ultimately, type annotations are also present in function and method declarations in order to signal the parameters' and return value's type. More information about ChocoPy's features can be found in [59] and in [67].

3.5 Revisiting Bleach's Purpose

As stated in the summary and previous sections of this document, Bleach is a programming language intended to be used as a tool by teachers, instructors and teaching assistants responsible for undergraduate courses in Compilers, ubiquitous in Bachelor's degrees in Computer Science and Computer Engineering.

The inspiration to create Bleach came from the fact that even though there are new approaches that combines theory and practice to expose the contents of the course's syllabus, there are still worrying issues when it comes down to the learning experience of the students, as reported by Lasseter in [20]. With regards to this particular issue raised, the approach proposed by Nystrom in [3] seems to be a good and innovative way of teaching Compilers. To reinforce the benefits of a more practical line of action, it's worth remembering that, as seen in the previous section, the "Case-based and Project-based" approach proposed by Kundra et al. in [25] ratifies the idea that a project-based direction for the course is, indeed, a good option when prioritizing the students' experience. On the other hand, such a approach must be executed with carefulness since a similar approach proposed by Aho in [19] may be unfeasible, especially in scenarios where there is a lack of teaching assistants or classes with too many students.

Keeping all of these mentioned points in mind, Bleach aims to be an alternative programming language that seeks to be used in undergraduate Compilers course as a means to engage and motivate students during the learning process of this subject, while also providing enough theoretical foundation for the students in this matter. In addition, it also relieves professors, instructors and teaching assistants from the burden of dealing with dozens of programming languages created by class students.

3.6 Scenario Overview

In this section, the context in which Bleach is expected to be used, who will be using it, and how it fits into the educational environment of undergraduate Compilers courses are all presented with more details.

- **Target Audience:** Bleach's target audience is divided into two different, but related groups: The first one is composed of undergraduate students that are enrolled in Compilers courses, which are part of Computer Engineering and Computer Science programs. These students are expected to have fundamental knowledge about programming languages, algorithms, data structures, but lack the more foundational understanding needed to implement compilers or interpreters. The second one is composed of professors, instructors and teaching assistants who are interested in providing a more incremental and hands-on experience to their students when it comes to compilers and interpreters design.

- **Use Cases and Educational Benefits:** Bleach was designed for use within undergraduate Compilers courses. The language is intended to be used as a practice tool, so the students can apply the learned core compiler concepts during the classes in order to construct a programming language implementation that resembles those that widely used by them, such as C, Python and JavaScript, for example. Through the use of Bleach, students can incrementally build an interpreter or compiler for the Bleach language by completing assignments that involve the implementation or extension of a lexer, a parser, a static analyzer, a IR generator, an optimizer, a code generator and, finally, a virtual machine or runtime. In short, this methodology is expected to provide enough practical experience in this course for the students.

Chapter 4

Bleach

"Bullet, Claw, Battle Flag, Short Sword. With my fingers bent, I wait for you."

– NORIAKI KUBO

4.1 Introduction

This chapter is dedicated to provide a detailed explanation about the Bleach programming language. Firstly, a brief overview of Bleach's features will be presented by showcasing a few programs written in the language. Then, a detailed breakdown of each of its components and features will be shown to the reader. Afterwards, the challenges encountered during the development of this project will be presented, as well as decisions and trade-offs made. Finally, a comparison between Bleach and the previous languages introduced will be made, as well as a discussion about how Bleach can be properly used to its fullest in an undergraduate Compilers course.

4.2 Bleach Overview

As previously mentioned, this section is dedicated to showcase a few of Bleach's features through the exposition of simple, yet useful, programs to the reader. It's expected that the reader has some familiarity programming with another languages, such as C [13], C++ [14], JavaScript [68], Lox [3], Python [61] and Ruby [69], so the similarities between Bleach and these languages make themselves more evident.

```
function greet(){
  print "Hello, World!";
}

greet(); // "Hello, World!"
```

Figure 4.1: Bleach's simplest program: The famous "Hello, World!" program.

```
function factorial(n){
  if(n == 0){
    return 1;
  }else{
    return n * factorial(n - 1);
  }
}

print factorial(5); // 120
```

Figure 4.2: A Bleach program that shows the usage of recursive functions, if-else statements and arithmetic operators to calculate the factorial of a number.

```
function main(){
  let numbersInfo = [];
  for(let i = 0; i <= 10; i = i + 1){
    if(i % 2 == 0){
      numbersInfo.append(i + " is even!");
    }else{
      numbersInfo.append(i + " is odd!");
    }
  }
  print numbersInfo;
}

main();
```

Figure 4.3: A Bleach program that shows the usage of for loops and the native `list` type in order to build a list that contains some information about numbers from 0 to 10.

```
function quadraticEquationSolver(a, b, c){
  let delta = std::math::pow(b, 2) - 4 * a * c;
  let x1 = (-b + std::math::sqrt(delta))/(2 * a);
  let x2 = (-b - std::math::sqrt(delta))/(2 * a);

  return [x1, x2];
}

print quadraticEquationSolver(1, 0, -4); // [2, -2]
```

Figure 4.4: A Bleach that uses a function which receives the coefficients of a quadratic equation to compute its roots.

```
class Shape{ // Base Class
  method init(name){
    self.name = name;
  }

  method str() {
    return "This is a " + self.name + ".";
  }

  method area(){ // To be overridden by subclasses
    return 0;
  }
}

class Circle inherits Shape{ // Derived Class
  method init(radius){
    super.init("Circle"); // Call the Base Class constructor
    self.radius = radius;
  }

  method area(){
    return std::math::pow(self.radius, 2) * 3.14159;
  }
}

class Triangle inherits Shape{ // Derived Class
  method init(base, height) {
    super.init("Triangle"); // Call the Base Class constructor
    self.base = base;
    self.height = height;
  }

  method area(){
    return (self.base * self.height) / 2;
  }
}

let c = Circle(5);
let t = Triangle(3, 7);

// This is a Circle. It has an area of 78.539749999999998 units.
print c + " It has an area of " + c.area() + " units.";
// This is a Triangle. It has an area of 10.5 units.
print t + " It has an area of " + t.area() + " units.";
```

Figure 4.5: A Bleach program that illustrates its Object-Oriented features, such as: classes, instances, attributes, methods, overriding and inheritance.

```
function writeToFile(){
  let userInput = std::io::readLine();
  std::io::fileWrite("output.txt", "w", userInput, true);

  userInput = std::io::readLine();
  std::io::fileWrite("output.txt", "a", userInput, false);

  return;
}

writeToFile();
```

Figure 4.6: A Bleach program that shows its capability with dealing with user-input and writing to `.txt` files.

4.3 Bleach Features

4.3.1 Introduction

Bleach is a high-level and dynamically-typed programming language designed with the intent to be used as a teaching tool in undergraduate Compilers course. The motivation behind its creation lies in the fact that according to the studies mentioned in section 3 and in my personal experience, a more incremental and practice-oriented approach when teaching this course is also more rewarding to the teaching staff and to the students, especially when it comes to grasp the core concepts of it. Last but not least, as it will be shown in this chapter and in the next one, Bleach is a more complete language when compared to Cool and MiniJava, which might capture students' interest in implementing a programming language that resembles those they are used to utilizing in their daily lives.

4.3.2 Data Types

Bleach has 5 built-in data types made available to the user. Such types are divided into scalar types and compound types.

Scalar types are types that can represent a single value. In Bleach, there are 3 scalar data types: `bool`, `nil` and `num`.

- `bool`: This data type is used to represent one of two possible values: `true` or `false`. Such values are used to express logical conditions and to control the execution of certain parts of a program's source code.

In this context, it's important to mention that Bleach takes inspiration from Ruby and other modern programming languages and implements the concept of "truthy" and "falsey" values in order to evaluate the truthiness or the falseness of values when they are being evaluated inside `if` statements, loops (`while`, `do-while`, `for`) and ternary operators (`condition ? expression_1 : expression_2`).

Essentially, this means that, in Bleach, values of any type (built-in or user-defined) can be used in places where a value of type `bool` is expected. Moreover, Bleach opted to follow the same convention of Ruby in this matter, which says:

- **"Falsey" values:** `false`, `nil`.
- **"Truthy" values:** Any other value that is not `false` nor `nil`.

- `nil`: This one is an old acquaintance for most programmers. The `nil` type has just one possible value: the `nil` value. In short this value conveys the idea of "no value" or the idea of "absence of a value".

  In other programming languages, this concept appears in other forms, like: `NULL`, `nullptr`, `null`, `None`, `nil`, and others. Since Bleach has some influence from Ruby, it adopted `nil`.

- `num`: In favor of simplicity, Bleach has only one type to represent numbers: the `num` type. This type can be used to represent both integers and floating-point numbers.

  Behind the scenes, the `num` type is implemented with a double-precision floating-point type, which allows Bleach to cover a lot of territory when it comes to numerical values while still keeping the simplicity initially envisioned for the language.

  Finally, with respect to numerical literals, Bleach has support for only basic integer and decimal literals, such as: `23`, `42`, `3.14159`, `2.71828`, `-10.4`.

Compound types are types that can group multiple values into one. In Bleach, there are 2 compound types: `list` and `str`.

- `list`: Taking inspiration from Python [61], Bleach has this type. Here, the `list` type represents an linear-sequence of elements. As previously stated, Bleach is a dynamically-typed language, just like Python. Thus, the `list` type can store values of different types with no issues. Moreover, Bleach has support for `list` literals, just as the ones shown

below:

```
let l0 = [0, 1, 2, 3, 2.71, 3.14159];
let l1 = ["hello", "there"];
let l2 = [];
let l3 = [false, "Brazil", 9.98, nil];
```

Figure 4.7: Examples of `list` literals in Bleach.

Finally, as in Python, the `list` type in Bleach has the following useful methods that makes the student's life easier when working with this type.

- `append`: Method responsible for adding one value of any type to the end of the `list` value it was called on. Returns `nil`.
- `clear`: Method responsible for deleting every element that is currently stored inside the `list` value it was called on. Returns `nil`.
- `empty`: Method responsible for checking whether the `list` value it was called on currently has values stored inside it or not. Returns a `bool` value.
- `fill`: Method responsible for resizing the `list` value it was called on to a provided size and also filling all of its indexes with a provided value of any type. Returns `nil`;
- `getAt`: Method responsible for returning the value present at the provided index. Returns a value of any type.
- `pop`: Method responsible for deleting and returning the last element present inside a `list` value, if any. Returns a value of any type.
- `setAt`: Method responsible for setting the value stored at the provided index to the value that was provided. Returns `nil`.
- `size`: Method responsible for checking and returning the current amount of elements that the `list` value it was on called on currently has. Return a `num` value.

Last but not least, it's important to mention that any misuse of the methods presented above will result in a runtime error during the program's execution.

- `str`: Also taking inspiration from Python [61], Bleach has this type. The `str` type represents an indexed-sequence of characters (a string), typically used to store and manipulate text. Just as the `list` type, Bleach also has support for literal values of this type:

```
let s0 = "Ichigo Kurosaki";
let s1 = "Grimmjow Jaegerjaquez";
let s2 = "Bazzard Black";
let s3 = "Jugram Haschwalth";
```

Figure 4.8: Examples of `str` literals in Bleach.

There are other aspects about the `str` type that must be mentioned.

First of all, it is important to recall that it's a sequence type. In short, it means that values of this type can be indexed. In a string, indexing usually allows the programmer to access individual characters from the value, which, in the Bleach programming language, are also values of `str` type.

Also, as seen in Figure 4.8, in Bleach, literals values of this type are always enclosed by double quotes (`""`).

Finally, as in Python, this type has the following useful methods available:

  - `empty`: Method responsible for checking whether the `str` value it was called is equal to `""` or not. Returns a `bool` value.
  - `find`: Method responsible for trying to figure out if the provided sub-string (a `str` value) exists within the `str` value the method was called on. Returns a `num` value which denotes the index at which the sub-string appears, otherwise returns `-1`.
  - `length`: Method responsible for checking and returning the current amount of characters (which are also values of type `str`) that the `str` value it was on called on currently has. Returns a `num` value.
  - `split`: Method responsible for generating a `list` where each value is of `str` type. This method receives as its unique argument a value of `str` type that works as the separator. Returns a `list` value.
  - `substr`: Method responsible for retrieving a sub-string from the `str` value it was called on. The method receives two arguments of `num` type that work as the start

and end delimiters. Returns a `str` value.

4.3.3 Comments

Even though it is recommendable that a programmer writes code in a way that is readable and understandable, there are certain scenarios where extra explanation about what certain parts of a program's source code is actually doing is necessary.

In these specific cases, programmers usually leave comments in their code that the compiler/interpreter will ignore but people reading their code may find helpful.

Given this, Bleach takes inspiration from C [13] when it comes to support for single-line comments and multi-line comments.

- **Single-Line Comments:** In Bleach, a single-line comment starts with two slashes (`//`), and the comment continues until the end of the line.
- **Multi-Line Comments:** Bleach also has support for multi-line comments. A multi-line comment has a beginning and an ending. The former is denoted by a (`/*`), while the later is denoted by a (`*/`). Everything written in-between is considered a comment and, thus, is ignored by the interpreter at runtime.

```
1 // This is a single-line comment.
2
3 /*
4 This  is
5 a multi-line
6 comment.
7 */
```

Figure 4.9: Examples of `str` literals in Bleach.

4.3.4 Variables

A variable, as in most programming languages, can be viewed as just a name associated with a storage location in memory. Such storage location is responsible for holding data that can be changed during the execution of a program.

In Bleach this is not different. Variables still have this main functionality. However, they have some particularities that might differ from variables in other programming languages. Such peculiarities are listed below:

- In Bleach, all variables are mutable. In other words, once a variable is declared, any amount of assignments are allowed to be performed on this declared variable.
- A direct consequence of the particularity explained above is that there is no concept of constants in Bleach. There are just mutable variables, as previously mentioned.
- In Bleach, to declare a new variable, the `let` keyword must be used:

```
let s = "A value of type 'str'";
```

Figure 4.10: Example of variable declaration in Bleach.

- In Bleach, if the declaration of a variable does not have an initializer, then, by default, the variable will store the nil value:

```
let someVariable;
print someVariable; // nil
```

Figure 4.11: Example of variable declaration without an initializer in Bleach.

- Since Bleach is a dynamically-typed programming language, variables don't have types associated with them. Instead, it's the values stored inside them that have types. The most important consequence of this fact is that a variable can hold a value of different types at different points in time:

```
let a = "hello";
print a; // "hello"

a = nil;
print a; // nil

a = 3.14 + 2.71;
print a // 5.85
```

Figure 4.12: Example demonstrating the fact that, in Bleach, variables do not have types associated.

Bleach also has support for global variables, those that have been declared outside of all functions, methods or classes, making them accessible from any part of the program's source code. One point that makes Bleach kind of unique is the fact that the programmer is allowed to re-declare a global variable anytime:

```
let pi = 3.14159;
// Write some code in the global scope.
let pi = 9.51413;
```

Figure 4.13: Example demonstrating re-declaration of global variables in Bleach.

As expected from most programming language, Bleach supports local variables, which are those declared within a specific block of code, such as a function, method, an `if-elif-else` statement or a loop statement (`while`, `do-while`, `for`). In this regard, it's important to remember the reader that the scope of a local variable is limited to the block in which it is defined, meaning it can only be accessed and used within that block. Furthermore, once the block of code finishes executing, the local variable is normally destroyed, and its memory is released.

```
function foo(){
  let bar = "This is a local variable";
  print bar; // "This is a local variable"

  return;
}

foo();
```

Figure 4.14: Example demonstrating the usage of local variables in Bleach.

As previously shown, variables are declared with the usage of the `let` keyword. Besides, after declaring a variable, the user is allowed to assign different values of different types at different points in time to the declared variable. To do that, the assignment operator (`=`) comes into play. However, before showing examples of code snippets that make use of this operator, it's very important to mention two semantic details of assignment in Bleach:

- In Bleach, an assignment is an expression, not a statement. This means that every assignment produces a value.

```
let foo;
print foo = 20; // 20
```

Figure 4.15: Example showing the behavior of an assignment expression.

- Bleach allows the user to assign a value to more than one variable in an assignment expression.

```
let x = 20;
let y = 42;

x = y = 13;

print x; // 13;
print y; // 13;
```

Figure 4.16: Example showing the possibility of assigning a value to multiple variable in a single assignment expression.

Lastly, the Bleach programming language has support for variable shadowing. In practice, this means that when a variable is declared in an inner scope with the same name of another variable, which was declared in an outer scope, the inner variable "shadows" or hides the variable declared in the outer scope. This means that, until this inner scope ends, every variable reference will "hit" the inner one, not the outer one. In this matter, it is important that in order to create a new scope, the user just needs to use curly braces ({}). The two examples presented below show how variable shadowing works in practice in Bleach:

```
let a = 42;
print a; // 42

{
  let a = "Hello, there!";
  print a; // "Hello, there!" --> Variable Shadowing in action.
}
print a; // 42
```

Figure 4.17: First example of variable shadowing in Bleach.

```
let foo = "hi";
print foo; // "hi"

function f(){
  let foo = 42;
  print foo; // 42;

  return;
}
f();

print foo; // "hi"
```

Figure 4.18: Second example of variable shadowing in Bleach.

4.3.5 Operators

Operators in a programming language are tools responsible for executing an operation on one or more operands in order to produce a result.

Operators are fundamental building blocks in every programming language, since they allow data manipulation, computations execution, values comparison, and much more.

This subsection is dedicated to introduce the operators Bleach grants to its users and how each of such operators behave. In the end, an operator precedence table will be shown.

- **Unary Operators:** These operators expects just one operand. Bleach has 2 operators that fall in this class.
  - **Arithmetical:**
    - **Negation** (-): Negates the value of an operand of type `num`.

```
let n = 42;
print -n; // -42
```

Figure 4.19: Example of usage of the "negation" operator (-).

- **Logical:**
  - **Not** (`!`): Inverts the value of an operand of type `bool`.

```
let b = true;
print !b; // false
print !!b; // true
```

Figure 4.20: Example of usage of the "not" operator (`!`).

- **Binary Operators:** These operators expects just two operands. Bleach has 13 operators that fall in this class.
  - **Arithmetical:**
    - **Addition** (`+`): This operator expects two operands, which can be of type `num` or `str`. However, the action performed by this operator at runtime depends on the types of the operands provided.

      If both operands are values of type `num`, then this operator adds the first (left) operand to the second (right) operand and returns the result (a value of type `num`).

```
print 2 + 3; // 5
print 2.71 + 3.14159; // 5.85159
```

Figure 4.21: First example of usage of the "addition" operator (`+`).

If both operands are values of type `str`, then this operator concatenates the second (right) operand to the first (left) operand and returns the result (a value of type `str`).

```
print "hello," + " there!"; // "hello, there!"
print "a" + "b"; // "ab"
```

Figure 4.22: Second example of usage of the "addition" operator (`+`).

If one operand is a value of type `num` and the other one is a value of type `str`, then the one that is a `num` is converted into its `str` representation and concatenated to the other operand. The operator then returns the result (a value of type `str`).

```
print 2 + "two"; // "2two"
```

Figure 4.23: Third example of usage of the "addition" operator (+).

If both operands are values of type `list`, then this operator concatenates the second (right) operand to the first (left) operand and returns the result (a value of type `list`).

```
print [1, 2, 3] + [4, 5, 6]; // [1, 2, 3, 4, 5, 6]
```

Figure 4.24: Third example of usage of the "addition" operator (+).

* **Subtraction** (-): This operator expects two operands of type `num`. It subtracts the second (right) operand from the first (left) operand, and returns the result (a value of type `num`).

```
print 5 - 3; // 2
```

Figure 4.25: Example of usage of the "subtraction" operator (-).

* **Multiplication** (*): This operator expects two operands of type `num`. It multiplies the first (left) operand by the second (right) operand, and returns the result (a value of type `num`).

```
print 1.5 * 4; // 6
```

Figure 4.26: Example of usage of the "multiplication" operator (*).

* **Division** (/): This operator expects two operands of type `num`. It divides the first (left) operand, also called dividend, by the second (right) operand, also called divisor, and returns the result of the division (a value of type `num`). It is worth mentioning that if the value of the divisor is `0`, then a runtime error will be thrown.

```
print 5 / 2; // 2
print 1 / 3; // 0.333333333333333
```

Figure 4.27: Example of usage of the "division" operator (/).

* **Remainder** (`%`): This operator expects two operands of type `num`. It divides the first (left) operand, also called dividend, by the second (right) operand, also called divisor, and returns the remainder of this division (a value of type `num`). It is worth mentioning that if the value of the divisor is `0`, then a runtime error will be thrown.

```
print 5 % 2; // 1
print 1 % 3; // 1
print -10 % 4 // -2
```

Figure 4.28: Example of usage of the "remainder" operator (`%`).

– **Comparison:**

* **Greater Than** (`>`): This operator expects two operands of type `num`. It checks if the first (left) operand is greater than the second (right) operand. If that is indeed the case, it returns `true`. Otherwise, it returns `false`.

```
print 5 > 2; // true
print 1 > 3; // false
```

Figure 4.29: Example of usage of the "greater than" operator (`>`).

* **Greater Than or Equal** (`>=`): This operator expects two operands of type `num`. It checks if the first (left) operand is greater than or equal to the second (right) operand. If that is indeed the case, it returns `true`. Otherwise, it returns `false`.

```
print 5 >= 2; // true
print -1 >= -1 // true
print 1 >= 3; // false
```

Figure 4.30: Example of usage of the "greater than or equal" operator (`>=`).

* **Lesser Than** (`<`): This operator expects two operands of type `num`. It checks if the first (left) operand is lesser than the second (right) operand. If that is indeed the case, it returns `true`. Otherwise, it returns `false`.

```
print 5 < 2; // false
print 1 < 3; // true
```

Figure 4.31: Example of usage of the "lesser than" operator (<).

* **Lesser Than or Equal** (<=): This operator expects two operands of type num. It checks if the first (left) operand is lesser than or equal to the second (right) operand. If that is indeed the case, it returns `true`. Otherwise, it returns `false`.

```
print 5 <= 2; // false
print 0 <= 0; // true
print 1 <= 3; // true
```

Figure 4.32: Example of usage of the "lesser than or equal" operator (<=).

– **Equality:**

* **Equal** (==): This operator expects two operands of the following built-in types (`bool`, `nil`, `num`, `str`). It checks whether the values are of the same type and, if that's the case, checks whether such values are the same. Returns `true` if both conditions are true. Otherwise, returns `false`.

```
print 2 == 2; // true
print 2 == (1 + 1); // true
print 2 == 3; // false
print "hello" == "hello"; // true
print "hello" == "hell"; // false
print 2 == nil; // false
print nil == nil; // true
print true == true; // true
print true == false; // false
print true == !!true; // true
```

Figure 4.33: Example of usage of the "equal" operator (==).

* **Not Equal** (!=): This operator expects two operands of the following built-in types (`bool`, `nil`, `num`, `str`). It checks whether the values are not of the same type and, if they are of the same type, it then checks whether such values are

not the same. Returns `true` if one (or both) conditions mentioned above are not satisfied. Otherwise, returns `false`.

```
print 2 != 2; // false
print 2 != (1 + 1); // false
print 2 != 3; // true
print "hello" != "hello"; // false
print "hello" != "hell"; // true
print 2 != nil; // true
print nil != nil; // false
print true != true; // false
print true != false; // true
print true != !!true; // false
```

Figure 4.34: Example of usage of the "not equal" operator (`!=`).

– **Logical:**

* **And** (`and`): This operator returns `true` if, and only if, both operands are "truthy" values. Otherwise, it returns `false`. This operator performs short-circuiting whenever possible.

```
print 5 and 2; // true
print 5 and false; // false
print false and nil; // false
```

Figure 4.35: Example of usage of the "and" operator (`and`).

* **Or** (`or`): This operator returns `true` if one of its operands is a "truthy" value. Otherwise, it returns `false`. As the operator above, this one also performs short-circuiting whenever possible.

```
print 5 or 2; // true
print 5 or false; // true
print false or nil; // false
```

Figure 4.36: Example of usage of the "or" operator (`or`).

- **Ternary Operator:** A ternary operator is, as its name suggests, an operator that expects three operands. Bleach has just one operator that falls in this class (`condition ? expression_1: expression_2`), the one the reader is probably familiar with from C and C++.

  This operator provide another way of executing conditional operations. It is used to evaluate a condition and return one of two values based on whether the condition evaluates to `true` or `false`.

  The three operands expected by the ternary operator can be of any type (built-in or user-defined). The first one is evaluated by the ternary operator with respect to its truthiness of falseness. If the value is "truthy", then the operator returns the second operand. Otherwise, it returns the third operand.

```
1 print 2 == 2 ? "2 is equal to 2" : "2 is not equal to 2"; // "2 is
    equal to 2"
```

Figure 4.37: Example of usage of the "ternary" operator (`condition ? expr_1: expr_2`).

- **Bleach's Operators Precedence Table:**

Precedence	Operators
1	!, - (unary)
2	*, /, %
3	+, - (binary)
4	>, >=, <, <=
5	==, !=
6	and
7	or
8	? :
9	= (assignment)

Table 4.1: Bleach's Operator Precedence Table.

4.3.6 Control-Flow Structures

Control-Flow structures are entities that allow the programmer to dictate in what order and by how many times instructions or code blocks are executed.

These entities are responsible for allowing branching mechanisms, decision-making and repetition in a program. The main consequence of this is that a program can have a more dynamic and sophisticated behavior.

In Bleach, there are 3 classes of control-flow structures:

- **Conditional Statements:** These statements are known to execute a block of code based on the fact that certain condition(s) are satisfied. Bleach has support for `if` statements, which may be followed by any amount of `elif` statements and a unique `else` statement, in this order. Bleach also provides support for nesting `if` statements, which gives more flexibility to the user.

```
let number = 15;

if(number % 3 and number % 5){
  print "Fizz Buzz"; // Executes this statement.
}elif(number % 3){
  print "Fizz";
}elif(number % 5){
  print "Buzz";
}else{
  print number;
}
```

Figure 4.38: First example that shows the usage of `if`, `elif` and `else` in Bleach.

```
let number = 10;

if(number > 0){
  if(number % 2 == 0){
    // Executes this block of code.
    print "The number " + number + " is positive and even!";
  }else{
    print "The number " + number + " is positive and odd!";
  }
}else{
  print "The number " + number + " is negative!";
}
```

Figure 4.39: Second example that shows the usage of `if`, `elif` and `else` in Bleach.

It is important to mention that the `if`, `elif` and `else` statements require a block after it. Otherwise, a syntax error will be thrown by the interpreter. This happens because such statements create a new local scope.

- **Loop statements:** These ones are used to allow the repetition of a code block based on a certain condition. Bleach has support for 3 different types of loop statements: `while`, `do-while` and `for`. Just as in conditional statements, Bleach also allows the nesting of loop statements.

```
let counter = 10;

while(counter > 0){
  counter = counter - 1;
}

print counter; // 0
```

Figure 4.40: Example that shows the usage of a `while` loop in Bleach.

```
let counter = 1;

do{
  counter = counter - 1;
}while(counter > 1);

print counter; // 0
```

Figure 4.41: Example that shows the usage of a `do-while` loop in Bleach.

```
let number;

for(number = 10; number > 0; number = number - 1){}

print number; // 0
```

Figure 4.42: Example that shows the usage of a `for` loop in Bleach.

It is important to mention that these statements also require a block after them. Otherwise, a syntax error will be thrown by the interpreter. This happens because they also create a new local scope, just as the conditional statements.

- **Jump statements:** Jump statements are used by the programmer to alter the execution flow of the source code by "jumping" to a different part of it. Bleach has support for 3 types of jump statements: `break`, `continue` and `return`.

```
let numbers = [];
for(let i = 0; i < 10; i = i + 1){
  if(i == 5){
    break;
  }
  numbers.append(i);
}
print numbers; // [0, 1, 2, 3, 4]
```

Figure 4.43: Example that shows the usage of a `break` statement in Bleach.

```
let numbers = [];

for(let i = 0; i <= 10; i = i + 1){
  if(i % 2 == 1){
    continue;
  }
  numbers.append(i);
}

print numbers; // [0, 2, 4, 6, 8, 10]
```

Figure 4.44: Example that shows the usage of a `continue` statement in Bleach.

```
function foo(){
  print "alpha";

  return;

  print "omega"; // Not reachable due to the "return" keyword.
}

foo(); // "alpha"
```

Figure 4.45: Example that shows the usage of a `return` statement in Bleach.

4.3.7 Functions

Functions, as is widely known, are fundamental building blocks in programming. They are reusable blocks of code created with the intention of executing a specific job. Therefore, they allow programmers to organize, reuse, and manage code in a more organized manner.

Functions can take inputs, process them and also return a value. As mentioned above, they are very useful because they are helpful when it comes down to breaking complex problems into smaller and simpler chores.

The code snippet below shows an example of a function declaration statement in Bleach:

```
function functionName(parameter1, parameter2, parameter3){
  // Code to be executed
  return value;
}
```

Figure 4.46: Example of function declaration statement in Bleach.

Still regarding function declaration statements, it is relevant to mention that if the programmer omits the `return` statement from it, then when such function is called at runtime it will, by default, return the `nil` value.

In Bleach, function calls are no different than in the most famous programming languages. The programmer is expected to call it by its name and provide the necessary arguments, according to its respective declaration statement. The code snippet below shows how to properly call a function in Bleach.

```
function subtract(a, b){
  return a + b;
}

let x = 5;
let y = 3;
let answer = subtract(a, b);

print answer; // 2
```

Figure 4.47: Example of a function call in Bleach.

As happens in most dynamically-typed programming languages, when a function call does not comply with its respective declaration statement, a runtime error is thrown.

It is very important to mention that functions in Bleach have certain limitations: They don't have support optional parameters, nor for default value for its parameters, and also cannot be overloaded.

Going further, in Bleach, functions are first-class values. Basically, this means that a function is not different from any other value in Bleach. This fact implies in the consequences listed below:

- Functions can be assigned to variables.
- Functions can be passed as arguments to other functions.
- Functions can be returned from other functions.
- Functions can be stored inside data structures.

Finally, Bleach also has support for anonymous functions, also known as lambda functions. As the name suggests, these are functions that do not have a name attached to them (i.e., they are declared without a name). Usually, famous programming languages have syntax mechanisms that permit such a thing. Bleach opted for taking a mixed approach by using the `lambda` keyword, used in Python, as well as an "arrow" token (`->`). When it comes to semantics, anonymous functions are just as powerful as the default ones.

The code snippet shown below demonstrates how to use an anonymous function in Bleach:

```
1  let add = lambda -> (a, b){ return a + b; }
2
3  let answer = add(10, 15);
4
5  print answer; // 25
```

Figure 4.48: Example of an anonymous function usage in Bleach.

4.3.8 The Object-Oriented Paradigm in Bleach

As previously stated, Bleach is a multi-paradigm programming language, since it has support both the procedural and the object-oriented paradigms.

When it comes to the object-oriented paradigm, Bleach has support to the following features of it, such as: classes, attributes/fields, methods, instances, the `self` keyword and inheritance. These features allow the reader to build modular, reusable, and maintainable code by organizing software into objects, entities that are capable of representing both data and behavior.

The rest of this subsection is dedicated to provide a more in-depth overview about each of the mentioned features:

- **Classes:** As is common knowledge among undergraduate CS students, in the object-oriented paradigm, a class is basically a "blueprint" for creating instances/object. Regarding this, since Bleach is a dynamically-typed language, it made sense for it to take

the same route as Python: It does not require explicit type information about attributes and methods. Furthermore, there is a lot of flexibility when it comes to additions, because the programmer can add new attributes and methods at runtime to instances of a class, as will be shown later in this subsection. In order to declare a new class, the user must use the `class` keyword followed by the name of the declared class and curly braces (`{}`):

```
// The declaration of a class called "Person".
class Person{
  method init(name, age){
    self.name = name;
    self.age = age;
  }

  method greet(){
    return "I am " + self.name + " and I have " + self.age + " years.";
  }
}
```

Figure 4.49: Example of a class declaration in Bleach.

- **Instances/Objects:** When it comes to instances/objects, Bleach took the same approach that was taken by Lox and Python. An instance/object is a concretization of a class. In other words, an instance is an entity created at runtime that has the structure and behavior defined by its class declaration. However, it is important to remember that each instance of a class can store its own unique data. In essence, an instance of a class has its own copy of the class's attributes, but they share the class's methods.

  Since Bleach took the same approach to deal with instances/objects as Lox, it is safe to say that the following quote written by Robert Nystrom in [3] also applies to Bleach: "Instances are loose bags of data and you can freely add fields to them as you see fit using normal imperative code."

- **The `self` keyword:** The keyword `self` serves for a very specific purpose inside a class: It's a reference to the current instance/object of the class. This keyword allows methods within a class to access the specific instance's attributes and other methods.

In addition to that, the `self` keyword assists to clarify when local variables (parameters that receive passed arguments in a method call, for example) have the same name as instance variables.

Finally, this keyword permits methods to return the current instance, which allows method chaining: a scenario where multiple methods can be called in sequence on the same instance.

The code snippet shown below demonstrates a more sophisticated use of the `self` keyword:

```
class Counter{
  method init(){
    self.count = 0;
  }

  method increment(){
    self.count = self.count + 1;
    return self;
  }
}
let counter = Counter();
counter.increment().increment().increment();
print counter.count; // 3;
```

Figure 4.50: Example of a more refined use of the `self` keyword in Bleach.

- **Attributes:** As seen in the item above, attributes don't need to have type annotations attached to them. Instead, the attributes' types are determined when values are assigned to them at runtime. On top of that, attributes of a class can store values of different types at different points in time. As in most programming languages that are object-oriented, in order to access the attributes of a class, one must use the "dot" (`.`) notation, as displayed below:

```
class Square{
  method init(length){
    self.length = length;
  }
}

let square = Square(5);

print square.length; // 5

square.area = square.length * square.length;
print square.area; // 25

square.perimeter = 4 * square.length;
print square.perimeter; // 20
```

Figure 4.51: Example that illustrates the usage of attributes of a class in Bleach.

Falar sobre a implicação semântica de ter um método e um atributo com o mesmo nome em uma instância de uma classe.

- **Methods:** In Bleach, methods of a class don't have type annotations in their signature, just like functions. The types are checked during runtime when the method is called, which can potentially throw a runtime error if a value of an unexpected type is passed as an argument. In order to declare a method of a class in Bleach, the programmer can use the same syntax as in a function declaration, just needing to replace the `function` keyword by the `method` keyword. To call a method of a class, it is required to use the "dot" notation (`.`) combined with the function calling convention (`()`), as exhibited below:

```
class Dog{
  method bark(){
    return "Owf Owf!";
  }
}

let dog = Dog();

print dog.bark(); // "Owf Owf!"
```

Figure 4.52: Example that illustrates the usage of methods of a class in Bleach.

Last but not least, since default functions and anonymous functions are values in Bleach, it is completely possible to add new methods to an instance of a class by using them, as demonstrated below:

```
class Dog{
  method bark(){
    return "Owf Owf!";
  }
}

let dog = Dog();

dog.louderBark = lambda -> (){ return "WOOF!"; };
print dog.louderBark(); // "WOOF!"
```

Figure 4.53: Example that illustrates the addition of a new method to an instance of a class during runtime.

Falar sobre os métodos especiais "init" e "str". Ambos podem ser omitidos na declaração de uma classe. O que acontece quando eles são omitidos? Quais as implicações disso?

- **Inheritance and the `super` keyword:** In Bleach, as in other popular languages like C++, Java and Python, inheritance allows the user to create a new class based on an existing class, extending or modifying its functionality.

Bleach follows a very similar implementation of inheritance to that of Python with just one major difference: For simplicity purposes, Bleach only supports single inheritance whereas Python has support for multiple inheritance.

In order to create a class that inherits from another, the use of the `inherits` keyword is required as it will be shown below.

Lastly, taking inspiration from Python, Bleach also has support for the `super` keyword. This one is used in order to refer to the methods of a superclass (i.e., the base class) from within a subclass (i.e., the derived class). To put it in another way, the `super` keyword allows a subclass invoke methods that were defined in its superclass (if any), even if such methods were overridden inside the subclass.

The code snippet below shows how to properly use inheritance in Bleach:

```
class Animal{
  method init(name){
    self.name = name;
  }
  method str(){
    return self.name + " makes a sound.";
  }
}
class Dog inherits Animal{
  method init(name, breed){
    super.init(name);
    self.breed = breed;
  }
  method str(){
    return self.name + " is a " + self.breed + " and barks.";
  }
}

let dog = Dog("Thor", "Rottweiller");
print dog; // "Thor is a Rottweiller and barks."
```

Figure 4.54: Example that shows the usage of inheritance in Bleach.

4.3.9 Bleach Native Functions

In case the reader is not familiar with this concept: native functions (built-in functions, intrinsic functions) are functions that are implemented in the underlying language (C++ in this particular case) of the platform or runtime environment rather that in the programming language itself (Bleach).

Native functions are often organized in as a part of the language's standard library and can also perform tasks in a more efficient way than user-defined functions.

As a way of achieving clarity and organization, the native functions of Bleach are organized into namespaces. A namespace is a concept that allows the logical grouping of functions, variables and other identifiers to prevent naming conflicts.

Before getting to an overview of these functions, it is important to recall the reader that any misuse of them will result in a runtime error being thrown.

Bleach's native functions are organized into the following namespaces:

- `std::chrono`
  - `std::chrono::clock`: Native function that does not take any arguments and calculates the current time in seconds as a `num` value since the epoch (which usually is 00:00:00 UTC on January 1, 1970) and returns that value.
- `std::io`
  - `std::io::fileRead`: Native function responsible for receiving a `str` value that represents the path (absolute or relative) to a `.txt` file, reading its contents and returning them as a `str` value.
  - `std::io::fileWrite`: Native function that receives a `str` value representing a path (absolute or relative) to a `.txt` file, a `str` value representing the opening mode of the file to be read (which can be "a" for append or "w" for write), another `str` value representing the content to be written to the file and a `bool` value signaling whether or not a newline must be inserted at the end of such content in the file.
  - `std::io::readLine`: Native function responsible for receiving user input from the terminal. It returns such input as a value of type `str`.
  - `std::io::print`: Native function responsible for receiving a variable number of arguments and outputting them to the console/terminal. Such arguments can be of

any type (built-in or user-defined). This function returns `nil`.

- `std::math`
  - `std::math::abs`: Native function responsible for receiving a value of type `num` and returning the absolute value of the received one, which is also of type `num`.
  - `std::math::ceil`: Native function responsible for receiving a value of type `num` and returning the smallest integer number that is bigger than the provided value. This function returns a value of type `num`.
  - `std::math::floor`: Native function responsible for receiving a value of type `num` and returning the largest integer that is smaller than the provided value. This function returns a value of type `num`.
  - `std::math::log`: Native function responsible for receiving two values of type `num` and computing the logarithm of the second argument with respect to the base represented by the first argument. The basis of the logarithm (the first argument) must be a positive number different than `1`, while the argument (the second argument) must be a positive number. If these conditions are not met, then a runtime error will be thrown.
  - `std::math::pow`: Native function responsible for receiving two values of type `num` and computing the exponential function where the first argument is the base and the second argument is the exponent. This function returns a value of type `num`.
  - `std::math::sqrt`: Native function responsible for receiving a value of type `num` and computing its square root. This function returns a value of type `num`. Passing a negative number as an argument will result in a runtime error.
- `std::random`
  - `std::random::random`: Native function responsible for receiving two values of type `num` as its arguments and returning a random `num` value that is between the two provided values. If the first argument is bigger than the second one, then a runtime error will be thrown.
- `std::utils`

- `std::utils::ord`: Native function responsible for receiving a value of type `str` of length `1` and returning its respective ASCII number. This functions returns a value of type `num`. Passing a string with length different than `1` will result in a runtime error.
- `std::utils::strToNum`: Native function responsible for receiving a value of type `str` that represents a number as its unique argument. The function converts the `str` value into its respective `num` value and, then, returns such value of type `num`. If the `str` value does not represent a number, a runtime error is thrown.
- `std::utils::strToBool`: Native function responsible for receiving a value of type `str` that represents a boolean as its unique argument. The function converts the `str` value into its respective `bool` value and, then, returns such value of type `bool`. If the `str` value does not represent a boolean, a runtime error is thrown.
- `std::utils::strToNil`: Native function responsible for receiving a value of type `str` that represents the `nil` value as its unique argument. The function converts the `str` value into this value and, then, returns it. If the `str` value does not represent `nil`, a runtime error is thrown.

4.4 Bleach Interpreter Components Breakdown

This section is dedicated to provide a detailed breakdown of each component that is part of the first interpreter for the Bleach programming language presented here.

First of all, a brief overview of this interpreter will be presented. Such overview's purpose is to show how each of its components interact with each other when interpreting a `.bch` file.

Then, a comprehensive analysis about each of the introduced components will be made.

4.4.1 Bleach Tree-Walk Interpreter Overview

First of all, it is important to remember the reader that the implementation of the Bleach language presented in this subsection is a Tree-Walk Interpreter, a type of interpreter that was previously presented in Section 2.2 of this document. Also its underlying language is C++. The box-diagram exhibited below shows how such sort of interpreter works with more details:

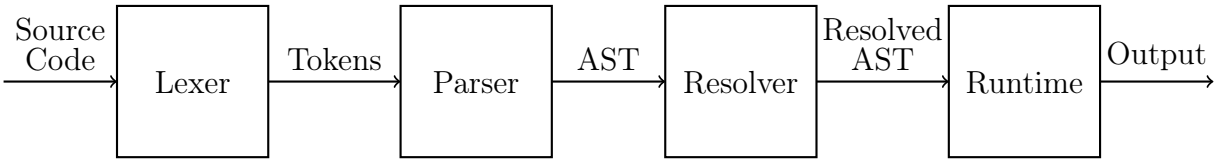

Figure 4.55: Box-Diagram representing the phases of the Bleach Interpreter.

4.4.2 Lexer/Scanner

When it comes to the lexer/scanner, it's worth remembering the reader that its goal is to transform the received source code (which is viewed by it as a large string of characters) into a sequence of tokens that will be fed to the parser of the interpreter. Such transformation performed by the lexer is known as lexical analysis.

When it comes to the possible tokens to be generated during the lexical analysis, these are the following types of tokens recognized by the lexer:

- **Single-Character Tokens:**

Token Type	Token
LEFT_PAREN	(
RIGHT_PAREN	)
LEFT_BRACKET	[
RIGHT_BRACKET	]
LEFT_BRACE	{
RIGHT_BRACE	}
COMMA	,
DOT	.
COLON	:
SEMICOLON	;

Table 4.2: Single-Character Tokens - Pt. 1

Token Type	Token
QUESTION_MARK	?
PLUS	+
MINUS	-
STAR	*
SLASH	/
REMAINDER	%
BANG	!
EQUAL	=
GREATER	>
LESS	<

Table 4.3: Single-Character Tokens - Pt. 2

- **Double-Character Tokens:**

Token Type	Token
ARROW	->
EQUAL_EQUAL	==
EQUAL_EQUAL	!=
GREATER_EQUAL	>=
LESS_EQUAL	<=

Table 4.4: Double-Character Tokens recognized by the lexer of Bleach's Interpreter.

- **Multi-Character Tokens:**

Token Type	Token
IDENTIFIER	`(a-z|A-Z|_)(a-z|A-Z|0-9|_)*`
NUMBER	`[0-9]+(.[0-9]+)?`
STRING	`"(any ASCII character)*"`

Table 4.5: Multi-Character Tokens recognized by the lexer of Bleach's Interpreter.

- **FILE_END Token:** This token is a special one that the Bleach interpreter supports. It is responsible for signaling the end of a `.bch` file. This token has an empty string as its lexeme. Its addition will be discussed on section 4.5.

When talking about tokens, specially identifier tokens, it is important to provide the table of keywords of a programming language. In Bleach's case, this table is shown below:

`and`	`break`	`class`
`continue`	`do`	`elif`
`else`	`false`	`for`
`function`	`if`	`inherits`
`lambda`	`let`	`method`
`nil`	`or`	`print`
`return`	`self`	`super`
`true`	`while`	

Table 4.6: Table with all keywords available in Bleach.

Regarding the implementation of the lexer itself, the road taken to achieve such goal was to build a hand-written lexer.

The implemented hand-written lexer can successfully recognize string literals and number literals. It can also ignore whitespace characters, single-line comments and multi-line comments. On top of that, the lexer is able to successfully distinguish token that share the same initial characters on their lexemes by using the concept of lookahead and "maximal munch", both previously presented. The "maximal munch" concept is also used to identify tokens that have a keyword as their lexemes from tokens that have just an identifier as their lexemes.

Lastly, the lexer is capable of identifying invalid characters, unterminated string literals, unterminated multi-line comments and misuse of the `':'` character.

4.4.3 Parser

When it comes to the parser, it's worth remembering the reader that its goal is to transform the received sequence of tokens into an Abstract Syntax Tree (AST) that will be fed to the resolver of the interpreter. This conversion executed by the parser is known as syntax analysis.

Regarding the possible AST nodes to be generated during the syntax analysis, they can be divided into two groups of nodes (expressions and statements). These groups are structured as displayed below:

Expr	**Stmt**
Assign	Block
Binary	Break
Call	Class
Get	Continue
Grouping	DoWhile
LambdaFunction	Expression
ListLiteral	For
Literal	Function
Logical	If
Self	Print
Set	Return
Super	Var
Ternary	While
Unary	
Variable	

Table 4.7: All possible types of AST nodes in the Bleach programming language.

With respect to the parsing technique that is behind the parser component of the implemented Bleach interpreter, the chosen one was the Recursive-Descent parsing technique. With this in mind, it's important to present the BNF-grammar of Bleach:

```
             program ::= statement* EOF
           statement ::= block | breakStmt | classDeclStmt | continueStmt | doWhileStmt
                       | exprStmt | forStmt | funcDeclStmt | ifStmt | printStmt
                       | returnStmt | varStmt | whileStmt
               block ::= "{" statement* "}"
           breakStmt ::= "break" ";"
        classDeclStmt ::= "class" IDENTIFIER ( "inherits" IDENTIFIER )? "{" methodDeclStmt* "}"
       methodDeclStmt ::= "method" method
              method ::= IDENTIFIER "(" parameters?  ")" block
         continueStmt ::= "continue" ";"
          doWhileStmt ::= "do" block "while" "(" expression ")" ";"
             exprStmt ::= expression ";"
              forStmt ::= "for" "(" ( varDecl | exprStmt | ";" ) expression?  ";" expression?  ")" block
         funcDeclStmt ::= "function" function
             function ::= IDENTIFIER "(" parameters?  ")" block
           parameters ::= IDENTIFIER ( "," IDENTIFIER )*
               ifStmt ::= "if" "(" expression ")" statement ("elif" "(" expression ")" statement )*
                       ( "else" statement )?
            printStmt ::= "print" expression ";"
           returnStmt ::= "return" expression?  ";"
           varDeclStmt ::= "let" IDENTIFIER ("=" expression )?  ";"
            whileStmt ::= "while" "(" expression ")" block
           expression ::= assignment
           assignment ::= ( call "." )?  IDENTIFIER "=" assignment | ternary
              ternary ::= logic_or ( "?" expression ":" expression )*
             logic_or ::= logic_and ( "or" logic_and )*
            logic_and ::= equality ( "and" equality )*
             equality ::= comparison ( ( "==" | "!=" ) comparison )*
           comparison ::= term ( ( ">" | ">=" | "<" | "<=" ) term )*
                 term ::= factor ( ( "+" | "-" ) factor )*
               factor ::= unary ( ( "*" | "/" | "%" ) unary )*
                unary ::= ( "!" | "-" ) unary | call
                 call ::= primary ( "(" arguments?  ")" | "." IDENTIFIER )*
            arguments ::= expression ( "," expression )*
              primary ::= "true" | "false" | "nil" | NUMBER | STRING | IDENTIFIER | "super" "." IDENTIFIER
                       | "(" expression ")" | "[" ( expression ( "," expression )* )?  "]"
lambdaFunctionExpr ::= "lambda" "->" "(" parameters?  ")" block
```

In other words, the grammar of Bleach is defined by the set of rules that have been just presented. Such rules are the entities that control how expressions and statements are formed in this particular programming language.

For example, by analyzing the BNF-Grammar presented, it becomes clear to the reader the precedence of the binary arithmetical operators. The operators `"*"`, `"/"` and `"%"` have the same precedence among them, but have more precedence when compared to the operators `"+"` and `"-"`, which also have the same precedence among them.

Going further, as mentioned previously, the parser is implemented with a top-down approach called Recursive-Descent parsing. In the case of the implemented interpreter, this parser is hand-written, since, as explained in Chapter 2, this particular parsing technique is basically a direct translation of a grammar to code that makes use of recursive functions, which makes its implementation straight-forward.

Concerning the parsing phase and the AST generation, as explained in Chapter 2, a recursive-descent parser starts the parsing process from the outermost grammar rule, which in Bleach's case is the `program` rule, and executes the process already explained when it comes to a top-down recursive descent parsing strategy.

One of the most important aspects of parsing is error reporting. In the parsing phase, the kind of error that is reported is commonly know as syntax error.

Following the guidelines presented in Chapter 2, to deal with syntax errors, the parser component of the implemented interpreter implements the ideas of "panic mode" and "error recovery". When a parser enters such mode, it needs to keep parsing process going until and continue to report more valid syntax errors until it reaches the end of the tokens sequence. To achieve this goal, the parser has a synchronization mechanism that looks for "synchronization points" from which the parser can continue the parsing process without issues. When in synchronization the parser ignores any tokens until it finds one that is a "synchronization point" and, in Bleach's case, such points are between statements.

4.4.4 Resolver

The resolver is a component of the interpreter responsible for performing a semantic analysis through the AST generated by the parser in the previous phase. To do that such traversal on the AST the Visitor design pattern is used, since it is better suited for this type of task, as explained previously in Chapter 2.

Even though the resolver performs a semantic analysis pass over the source code, it is a simple and straight-forward one. This pass is in charge of discovering where variables were declared and guarantee that they are properly used within their scope. This component is able to resolve variable references by traversing the AST generated by the parser and associating each variable with its corresponding environment/scope inside which it was defined.

As a way to make these associations correctly the resolver has a stack of environments/scopes in which the the environment/scope at the top of the stack is the current environment/scope being analyzed. For instance, the resolver component always creates a new environment/scope and pushes it to the top of the mentioned stack every time it starts the visiting of one of the following types of AST node: `LambdaFunction`, `Block`, `Class`, `DoWhile`, `For`, `While`. When the visiting of such nodes is ended, the environment/scope at the top of the previously mentioned stack is popped and the AST traversal continues. Still on this aspect, the resolver is responsible for providing the runtime the information necessary to find the correct variable to which a certain identifier is referring to in the chain of environments created at runtime.

Diving the deeper in this respect, the resolver is also able of reporting different types of errors related to variable declaration and usage.

- The resolver is able to find a variable re-declaration inside the same environment/scope and report it as an error.
- The resolver is also able to identify that a variable is being used in its own initializer expression and report it as an error.

On top of that, the resolver component of the Bleach interpreter is capable of catching other errors, such as:

- Usage of the `self` keyword outside of a class declaration statement.
- Usage of the `super` keyword outside of a class declaration statement.
- Usage of the `super` keyword inside a class declaration statement that is not a subclass from another class.
- Usage of the `break` keyword outside of any loop statement (`for`, `do-while`, `while`).
- Usage of the `continue` keyword outside of any loop statement (`for`, `do-while`, `while`).
- Usage of a class as the superclass of itself.

- Usage of the `return` keyword outside of a function, anonymous/lambda function or method.

- Usage of the `return` keyword inside the `init` method of a class declaration statement.

4.4.5 Runtime

The runtime is the last component of the interpreter. In short, this component is responsible for executing statements, evaluating expressions, performing control-flow, managing environments/scopes with their variables and respective values and also throwing runtime errors whenever required. From a different point of view, the runtime can be viewed as the engine responsible for executing code accordingly with the semantics of the implemented programming language.

As previously explained the implemented interpreter for the Bleach programming language presented in this document is a Tree-Walk one. This type of interpreter works, as earlier explained in Chapter 2, by executing a recursive traversal on the AST and performing different actions depending on the type of the AST's node that is currently being visited. All types of AST's node were also previously presented in Table **??**.

Dealing with the first group of AST nodes, the ones that represent expressions, the behavior of the runtime will fundamentally be the same: it will visit the node that represents an expression, then evaluate the expression, finally it will produce the value from said expression and will move on to continue the recursive traversal of the abstract syntax tree. However, the evaluation of these AST expression nodes consistently varies accordingly with the specific expression in question. A rundown of these specific behaviors regarding each type of expression node is presented beneath:

- **Assign:** When visiting this kind of expression node, the interpreter first evaluates the RHS (right-hand side) of this expression node. This evaluation will produce a value that must be assigned to the LHS (left-hand side) of this very same node. However, the evaluation of the LHS is a bit different than the one made with respect to the RHS. This one evaluates to a location in-memory in which the value obtained through the evaluation of the RHS will be stored. In order to find out the correct memory location, the runtime makes use of information previously computed by the resolver about variable resolution and environments. By default, this visiting returns the value from the RHS evaluation.

- **Binary:** When visiting this type of expression node, the interpreter first evaluates its left operand and, then, the right one. Finally, it performs the operation depending on the type of operator that this node stores and returns the result of such operation.

- **Call:** When visiting this sort of expression node, the interpreter first evaluates the callee of this node in order to figure out if it is a valid one (a function, an anonymous/function, a native function, a method, a class, or a method from the `list` and `str` built-in types). Once this is done, the interpreter evaluates each of the expressions that represent the arguments of the call and store such values appropriately. Finally, it makes the call to the callee passing the evaluated arguments. This sort of expression node returns the same thing the function that has been called returns.

- **Get:** When visiting this kind of expression node, the interpreter first evaluates the expression representing the entity on which the property is trying to be accessed. If such expression evaluates to an instance of a class, a value of type `list` or a value of type `str`, then the interpreter tries to find the value associated to the property and, finally, returns it.

- **Grouping:** When visiting this type of expression node, the interpreter just evaluates and, ultimately, returns the produced value by the nested expression within this type of node, since a Grouping node essentially represents a pair of parentheses.

- **LambdaFunction:** When visiting this kind of expression node, the interpreter just creates a runtime representation of the anonymous/lambda function by using the visited expression itself along with the current environment/scope that is in, since it will serve as the closure of such anonymous function.

- **ListLiteral:** When visiting this sort of expression node, the interpreter creates the runtime representation of a list. Then it evaluates each expression present inside the ListLiteral expression and appends the result to the runtime portrayal of the mentioned list. In the end, it returns such list.

- **Literal:** When visiting this type of expression node, the interpreter just evaluates the value associated with the literal and returns it.

- **Logical:** This expression node is a basically a specialization of the "Binary" node because it can only perform the following operations: `and`, `or`. Still, it evaluates the left

operand first and, according to the left value and the operator itself, tries to perform a short-circuiting. If that's not possible, it evaluates the right operand, produces the corresponding value by following the rules of Boolean Algebra and, finally, returns the produced value.

- **Self:** When visiting this kind of expression node, the interpreter uses it and the token that represents that particular `self` keyword to figure out at which environment the required variable is at (remember that the `self` is a name that references the current instance of a class, making it, in practice a variable). Once at such environment, the interpreter returns the instance bound to such name.

- **Set:** The visiting process of this expression node is very similar to that of the "Get" node. In this particular expression node, the interpreter first evaluates the expression representing the value that works as the RHS. Then, the interpreter evaluates the expression that represents the entity that contains the attribute that is about to receive the generated RHS value. If the evaluation of the expression that represents an entity results in a valid value (an instance of a class), then the interpreter searches for the attribute inside such value (which will result in a LHS value) in order to perform the binding between them. Finally, this visiting process returns the RHS value.

- **Super:** When visiting this sort of expression node, the interpreter first retrieves the superclass of the class whose "Super" node is being currently visited. Once this is finished, the interpreter then retrieves the instance of the class whose "Super" node is being visited. Finally, the interpreter searches and returns the superclass version of the method required in the "Super" expression node.

- **Ternary:** When visiting this particular expression node, the interpreter first evaluates its first operand, known as condition. If the result produced by such evaluation is `true`, then the ternary operator evaluates and returns its second operand, the one between `"?"` and `":"`. Otherwise, this operator evaluates and returns its third operand, the one that appears after `":"`.

- **Unary:** When visiting this type of expression node, the interpreter first evaluates its unique operand, commonly known as right operand. After this, it performs the operation, given the operator, on the operand, produces and, at last, returns the corresponding value.

- **Variable:** When visiting this type of expression node, the interpreter uses the expression node itself, which represents a variable usage, combined with the variable's token to figure out at which environment the required variable is at. Then, the runtime accesses that environment, retrieves and returns the value associated to the required variable.

Now the focus is turned to the second group of AST nodes, the one that contains nodes representing different kinds of statement, which can be called statement nodes. However, before showing each one, it is important to recall the reader that statements do not produce values. Keeping this in mind, there exists the following types of statement nodes:

- **Block:** When visiting this kind of statement node, the interpreter first stores the current environment (the implementation of the concept of lexical scope). Then, sets the current environment of the interpreter to another that has been created and whose enclosing one is the one that has been on use before the interpreter started the visit to this statement node. On this new environment current environment, the interpreter executes the statements present in the "Block" node one-by-one from top to bottom. After finishing this, it restore the environments to the initial configuration before starting the visit of this type of statement node.

- **Break:** When visiting this sort of statement node, the interpreter just throws an instance of a "Break" entity that will be caught by the runtime when visiting any loop statement node (`do-while`, `for`, `while`) and dealt with by it. This entity is responsible for simulating the behavior of loop interruption at runtime caused by the use of a `break` statement.

- **Class:** When visiting this type of statement node, the interpreter first checks whether the node has an expression that might evaluate to a superclass. If that's the case, such expression is properly evaluated and its result is stored for later use. After this, the interpreter creates a binding at the current environment it is in between the class name and the `nil` value. If the "Class" statement node indeed has a superclass associated to it, a nesting environment is created and a binding between the name `super` and the runtime representation of the superclass is generated. Then, for each method declared inside this statement node, its respective runtime representation is generated and stored inside the runtime representation of the class in a way that associates the name of the method and its respective runtime depiction. At last, the runtime portrayal of the class is populated with the needed information.

- **Continue:** When visiting this kind of statement node, the interpreter just throws an instance of a "Continue" entity that will be caught by the runtime when visiting any loop statement node (`do-while`, `for`, `while`) and dealt with by it. This entity is responsible for simulating the behavior of interrupting an iteration of a loop and going to the next iteration by the use of a `continue` statement.

- **DoWhile:** When visiting this sort of statement node, the interpreter first saves the previous environment it was in, so it can restore it, once it has finished the execution of the "DoWhile" statement. After this, the interpreter makes the current environment store a new environment whose enclosing environment is the one that has just been saved and starts executing the statements present inside the block of the "DoWhile" statement, always checking whether the condition of such statement still evaluates to `true` after the execution of all statements present inside its block. Once this condition evaluates to `false`, the interpreter interrupts the execution of this loop and restores the initial environment that was previously stored.

- **Expression:** When visiting this type of statement node, the interpreter simply evaluates the expression present inside it, which results in the generation of a value.

- **For:** When visiting this kind of statement node, the interpreter first saves the previous environment it was in, so it can restore it, once it has finished the execution of the "For" statement. After this, the interpreter makes the current environment store a new environment whose enclosing environment is the one that has just been saved. Then, the interpreter executes the initializer statement present inside the "For" statement. Once this is done, it evaluates the expression that represents the condition of the "For" statement at beginning of each iteration. If the evaluation results in a "truthy" value, then the statement(s) present inside the block of the "For" statement, including the last special statement, usually called "increment statement". This goes on until the expression that represents the condition evaluates to a "falsey" value. As soon as this happens, the visiting of this node is complete.

- **Function:** When visiting this sort of statement node, the interpreter just creates the runtime representation of the function and creates a binding between the name of the function and its corresponding runtime portrayal in the current environment the runtime is currently in.

- **If:** When visiting this type of statement node, the interpreter first evaluates the expression associated with the "If" branch within this type of node. In case the produced value is a "truthy" one, the statement(s) associated with this clause are executed and the interpreter finishes this node visiting. If that's not the case, then the interpreter will check for the possible presence of "Elif" branches in this node (which are stored in a sequential manner). If they are indeed present, the interpreter evaluates the expression associated with each one in order until it finds one expression that, when evaluated, produces a "truthy" value, when this happens, the statement(s) associated to such clause are executed and this visiting is also terminated. Finally, the interpreter checks for the presence of an "Else" clause. If it exists, then its statement(s) are executed and this node visiting ends.

- **Print:** When visiting this kind of statement node, the interpreter first evaluates the expression that is stored inside it. After this, it outputs the produced value to the standard output (terminal).

- **Return:** When visiting this sort of statement node, the interpreter checks whether there is an expression present inside such statement. If there is, then this expression is evaluated and the produced value is wrapped inside an entity that is thrown and will be caught by the caller of the function, anonymous function, native function, method. If there is not such expression, then the `nil` value is wrapped and thrown instead.

- **Var:** When visiting this type of statement, the interpreter checks whether there is an initializer expression for the variable declaration statement. If there is, then such expression is evaluated and the generated value is stored. Otherwise, by default the value `nil` is stored. At last, the interpreter performs a binding inside the current environment it is in between the name of the variable present inside the node and the respective value that was previously saved.

- **While:** When visiting this kind of statement, the interpreter first saves the previous environment it was in, so it can restore it, once it has finished the execution of the "While" statement. After this, the interpreter makes the current environment store a new environment whose enclosing environment is the one that has just been saved. Then, the interpreter starts to properly execute the "While" statement by at each iteration checking whether the condition of such statement evaluates to `true` and then executing

its block of statements if that is indeed the case. Once this condition does not hold anymore, the interpreter does not execute this block of statements.

4.5 Challenges, Decisions and Trade-Offs

This section is dedicated to report the challenges faced when designing Bleach and implementing its Tree-Walk Interpreter as well as the decisions made, the reasoning behind them and trade-offs that rose from such decisions.

For structuring purposes, this section is organized as a list of challenges. The first group of challenges is all about language design while the second one is about interpreter implementation challenges.

- **Language Design Challenges, Decisions and Trade-Offs:**
  - **Concept of "Truthy" and "Falsey" values**: Since popular programming languages like Ruby, Python, JavaScript and TypeScript started adopting this idea that every value has an associated notion of truthiness or falseness, it felt right to also adopt this philosophy in order to make Bleach modern and flexible. The question that rose from this adoption was: Which values should be considered 'truthy' and which should be considered 'falsey'? For the sake of design's simplicity and easiness when it came down to Bleach's implementation, the road taken by Ruby was also followed: the values `false` and `nil` are considered "falsey", while every other one is considered "truthy".
  - **Inserting the concept of `nil` in Bleach:** This is undoubtedly an idea that may raise questions, specially since the creator of the concept itself, Tony Hoare, regrets inventing it, calling such a thing "a billion dollar mistake" [70]. However, since Bleach and its interpreter implementation are heavily inspired by the "Crafting Interpreters" book [3], it took the same road as the language presented in that book simply because in a dynamically-typed language, which is Bleach's case, trying to exclude the idea of `nil` is more annoying than just allowing it. Thus, due to simplicity, it was adopted.
  - **Absence of a `char` type:** The reader might have questions about the absence of this built-in data type in Bleach. The answer is: simplicity and familiarity. By not

adding a new data type to the language, its type system becomes smaller. Moreover, the `str` type is enough to represent a such a type, since values of the `char` type can be viewed as values of the `str` that have length equal to `1`. In addition, it is important to remember that Bleach is dynamically-typed, which means that types are determined at runtime. Therefore, having a smaller type system reduces the amount of checks the runtime system must perform. Ultimately, popular dynamically-typed languages such as Python, Ruby and JavaScript adopted this same approach, thus, following it seems reasonable since this is a familiar pattern for students that had already contact with the mentioned languages.

- **A unique type (`num`) to represent numbers:** This design decision was taken having in mind simpleness and ease of use. One could argue that having a single numerical type has several disadvantages, like loss of integer precision, reduced performance and floating-point arithmetic issues. Even though all of these claims are correct, it is very important to remember that precision and optimized performance are not the main goals of Bleach. The main goal of the language proposed in this document is simplicity and easiness. Therefore, having just the `num` type reinforces this philosophy since having this unique type enables: consistency, unified arithmetic operations and unneeded type coercion or conversion.
- **Permission to re-declare global variables:** This decision has both pros and cons but, in the end, the pros seem to have a bigger impact than the cons. The major con is the visible inconsistency between local and global variables, since the local ones cannot be re-declared while global ones can. However, it is worth remembering the reader that the interpreter implemented for this language has two operating modes (one that activates an interactive session, a REPL, and another that reads and executes the content present inside a `.bch` file). Usually, in a REPL, the user might have difficulties on keeping track of which variable have been already declared and which have not. Following this reasoning, in order to make the user experience better when using the REPL, global variable re-declaration has been allowed.
- **Adoption of single inheritance instead of multiple inheritance:** This design decision was mainly motivated by clarity, simplicity and maintainability. Bleach was designed to be an educational tool, thus, there is encouragement to make it beginner-friendly and more accessible for students. With this in mind, dealing with

single inheritance is much easier than with multiple inheritance. In this vein, it is valuable explaining to the reader that single inheritance makes the whole hierarchy of classes less complicated to understand. A class can only have one "parent" class, with the exception of the base class. This structure makes inheritance more predictable and avoids the complexities of multiple inheritance, such as the "Diamond Problem" [71].

- **Tree-Walk Interpreter Implementation Challenges, Decisions and Trade-Offs:**
  - **Lexer Implementation:** When it came to the building of the Lexer component of the interpreter implementation, different strategies could have been adopted. In particular, a hand-written Lexer was the preferred one. The rationale behind this decision is that, even though there are tools capable of automatically generate a lexer given a set of rules defined by regular expressions (such as: Flex [5] [72], JFLex [73]), implementing one from scratch is not a hard task. Basically, all the programmer needs to do is deal with basic string processing and conditional logic, knowledge that is expected from CS students that are taking their undergraduate Compilers course. Moreover, demanding the students to implement the lexer themselves might be better a better approach to put their theoretical knowledge to practice instead of asking them to use semi-transparent automated tool solutions like Flex or JFlex, that abstract the nitty-gritty details of a lexer implementation.
  - **Parser Implementation:** Regarding the choice of which way to follow when implementing the parser component, the most important factor taken into account was accessibility, easiness and simplicity. With these reasons in sight, the parser was implemented as a Top-Down Recursive Descent Parser. According to Nystrom [3], this kind of parser is intuitive and easy to understand, since it works as direct translation of the programming language grammar rules into functions/methods, easy to implement given the reason presented before and also simple to debug. The combination of these aspects make it a good choice for educational purposes, which is Bleach's main goal. One could argument that there are other kinds of parsers that could be used, such as LL(0) parser, LL(K) parser, LR(0) parser, LR(1) parser, LALR parser, among others. However, none of these approaches has a focus on education and learning. Besides, Nystrom affirms in [3] that, even though, Recursive-Descent

parsers are simple, they should not be underestimated since they are widely used in several popular programming language implementations: GCC, V8 (A JavaScript Runtime present in the Chrome browser) and the Roslyn Compiler (A C# compiler written in C#). Last but not least, there are tools that allow the automatic creation of parsers thanks to the use of tools like: ANTLR [6], Bison [7] or Yacc [17] [8]. However, the use of such instruments might abstract important theoretical concepts of parsing theory. Therefore, the avoidance of such methodology.

– **Interpreter or Compiler? Why specifically a Tree-Walk Interpreter?** The decision to implement an interpreter instead of a compiler for Bleach was again influenced by simplicity and easiness of learning for the students. Implementing Bleach as a compiler instead of an interpreter makes the students invest their time in topics that, although important in this area, could be skipped, due to their intrinsic complexity and breadth and richness, such as: Intermediate-Representation (IR) transformation, code generation, optimization and type systems. On top of that, both the optimization and the type systems topics are still objects of current research when it comes to the Programming Languages field. Given this, it is important to consider if it is really worth to address these topics when teaching an undergraduate Compilers course. And even if that is the case, it is essential to reflect which concepts of these topics could be taught to undergraduate CS students without compromising their learning experience.

  Given that the interpreter road was taken, the decision about which type of interpreter should be implemented was still open. To address this situation, a Tree-Walk Interpreter was chosen. The reasoning behind this selection was rather simple: easiness to understand the topics being presented from the students point of view and focus on the language design, instead of worrying about the complex mechanisms needed to transform the source code into machine-code or byte-code. The easiness aspect comes from the fact that a Tree-Walk Interpreter works by performing a traversal through the AST generated by the parser and executing certain actions depending on the type of AST node that is being visited, by the time CS students took their Compilers course they should have already taken their Algorithms and Data Structures course, which addresses Trees, and therefore this makes it easier to follow the way such kind of interpreter works. Lastly, by focusing on language design

it is expected that the students can focus more on the language semantics (how it behaves and what it can express), which for a beginner in this field might be more rewarding in terms of learning.

4.6 What Makes Bleach Shine and How It Can Be Used In a Classroom Environment

First of all, to extract Bleach's full potential, it is important to understand the conditions imposed in a classroom environment. In this context, highlighting the period of time allocated to an academic semester in Compilers courses is essential.

Given the fact that, normally, an academic semester spans through 10 to 16 weeks depending on the internal organization of the university, for the sake of simplicity, it will be assumed that the academic semester spans through 15 weeks.

Now, below it is defined a proposal of how Compilers courses that opt to use Bleach could be structured considering the assumed time frame:

- **Weeks 1 - 2 (Introduction to Programming Languages and Interpreters):**
  - Provides an overview of programming languages design and implementation, by making a comparison between compilers and interpreters.
  - Offers a review from certain topics related to Theory of Computation, for instance: Regular Expressions, Finite Automata (DFA and NFA) and Context-Free Grammars (CFG).
  - Introduction of the Bleach programming language to the students.
- **Weeks 3 - 4 (Lexing):**
  - Provides an explanation about what is the purpose behind a lexer and what exactly lexical analysis is, while exposing related concepts, like: lexemes and tokens. Takes a deep dive into how a lexer actually works, supplying the theoretical baggage needed to implement one later on.
  - Proposes the implementation of a lexer for the Bleach programming language.

- **Weeks 5 - 6 (Parsing):**
  - Provides an explanation about what is the purpose behind a parser and what exactly syntax analysis is, while exposing related concepts, like: ASTs, Bottom-Up and Top-Down parsing. Later on, explains the most important types of parsing (Recursive-Descent, LL(k) and LR(k)), while focusing in the first one, since it is the simplest of them all, but also robust, fast and reliable.
  - Provides a more in-depth explanation about CFGs, if the need arises.
- **Weeks 7 - 8 (Tree-Walk Interpretation):**
  - Provides an explanation about how a tree-walk interpreter works in practice. This can be done by reviewing the concepts of trees and their respective types of traversal. Explanation about how each node in a syntax tree represents a different construct of a programming language, focusing on Bleach.
  - Introduction to the Visitor design pattern, which is very useful to deal with the traversal of syntax trees
  - Proposition of an assignment that deals with the initial implementation of the Bleach tree-walk interpreter, by first dealing with literal values, unary operators and binary operators.
- **Week 9 (Variables, Scopes and Environments):**
  - Provides an explanation about variable declaration statements, initializer of a variable, scopes (a conceptual idea), blocks (its visual representation) and environments (its respective implementation).
  - Explains how to manage variables that are stored in different scopes and the concept of variable shadowing.
  - Proposition of a practice assignment to extend Bleach, so it can support the taught concepts.
- **Week 10 (Control Flow Structures):**
  - Provides an explanation about how control flow structures work behind the scenes. Explanation of how implement the most basic of these structures (`if` statement and `while` loop).

  - Proposition of a programming assignment to extend Bleach in order for it support such structures.
  - Proposition of bonus challenges that make the students think about concepts that have not been directly taught but can be learned through different types of references, such as: Addition of support for `for` and `while` loops (as well as support for the `break` and `continue` statements) and also for `else if` and `else` clauses as a mean to make the `if` statement more robust.

- **Week 11 (Functions):**
  - Provides explanation about how function declaration statements and function calls work behind the scenes. As well as how closures work and how one can implement such concepts.
  - Proposition of an assignment to extend Bleach to support function declaration statements, function calls and closures.
  - Proposition of a bonus challenge where students need to implement anonymous (lambda) functions.

- **Week 12 (Standard Library):**
  - Provides an explanation about what are standard libraries and native functions of a programming language, by providing a few examples of those present in languages like C and Python.
  - Proposition of an assignment where students are asked to implement simple native functions for Bleach, following Bleach's documentation or other ideas.

- **Week 13 (Resolving):**
  - Provides explanation about what is static analysis, its role and its importance in the development of a programming language implementation, focusing on dynamically-typed languages, like Bleach, where such analysis is simpler and compared to statically-typed languages.
  - Provides explanation about the importance of name resolution and how it can lead to unexpected bugs in during the interpretation of the AST (i.e., the runtime).

  - Proposition of an assignment where students must implement their own Resolver component (a very simple component in an interpreter, where name resolution occurs).

- **Week 14 (Object-Oriented Features):**
  - Provides explanation about the essential object-oriented features present in Bleach: classes, instances, attributes, methods, the `self` keyword and single inheritance.
  - Proposition of an assignment where students must extend Bleach in, so it can support all of the presented concepts and features.
  - Proposition of a challenge asking students to implement static methods for classes and multiple inheritance instead of single inheritance.

- **Week 15 (Projects Presentation and Final Exam):**
  - In the last week of the academic semester, the students must present their tree-walk interpreters, as well as the extra challenge assignment that they were able to solve.
  - Finally, apply final assessments and conduct project discussions.

Teaching a Bleach implementation that employs the idea of Tree-Walk Interpreters can provide several benefits to the students, such as:

- **Simplicity:** Generally speaking, a tree-walk interpreter is much simpler to implement than a full-compiler (it does not deal with the IR generation, optimization and code generation phases). Also, it does no go to deep into type systems, which can have their own courses given their complexity and vastness. Therefore, this might be ideal for students who are having their first contact with the world of programming language design and implementation without overwhelming them with the low-level concepts and implementations mentioned above.

- **Fast Execution and Iteration:** By implementing a tree-walk interpreter, students are able to see immediate results of their own work, since an interpreter essentially executes code from the generated AST, as explained in Chapter 2, making the connection between language syntax and runtime behavior more evident, while also motivating students since they can see the results of their work early on the course.

- **Hands-On Experience:** This approach enable students to obtain hands-on experience implementing a full interpreter, giving them a concrete understanding of how language constructs like variables, expressions, statements, control flow structures, functions, OOP features and simple static analysis work in practice.
- **Flexibility for Future Modifications:** This approach also encourages the students to experiment with language features (once students they have grasped the core concepts of tree-walk interpreters implementation) by adding new features, including but not restricted to the ones suggested earlier, which allows for creativity and deeper learning.
- **Solid Foundation for Future Learning:** By going through this methodology, it is expected that the students will be prepared to deal with more complex topics, such as bytecode interpreters, type systems and compilers with a deeper understanding of language runtime.

Chapter 5

Evaluation

"There is no pain as long as I keep my eyes on the balance scale."

– NORIAKI KUBO

5.1 Introduction

This chapter is dedicated to present to the reader a series of different approaches that were taken in order to test the Bleach programming language and the Tree-Walk Interpreter implemented for it from different perspectives. Regarding these perspectives, the following approaches were taken:

- **Test Suite:** For the reader that is not familiar with this concept: A Test Suite is a carefully tailored collection of test cases and test scripts implemented in order to verify that a software program or a specific module of a program is behaving as expected.
- **Implementation of famous Algorithms and Data Structures:** As a way to better demonstrate Bleach's expressiveness and prowess, this section is dedicated to present to the reader implementation of famous Algorithms and Data Structures commonly taught in an undergraduate Computer Science degree.
- **Comparison of Bleach with ChocoPy, Cool and MiniJava:** Regarding comparison with its predecessors, this section is dedicated to compare Bleach with them in terms of language features.

5.2 Bleach's Test Suite

As mentioned above, one of the different ways to evaluate Bleach was through the creation of a test suite that verifies the correctness of the Tree-Walk Interpreter implementation. In this particular scenario, this test suite is composed of both test cases and a test script responsible for executing the test cases automatically.

The Test Suite is made of 44 test cases and a shell script that executes these tests in an automated manner. These test cases work are divided into 3 groups (tests responsible for Expression nodes, tests responsible for Statement nodes and tests responsible for some of the Bleach Native Functions). All types of AST nodes previously presented in Table **??** are covered. Moreover, from these 44 test cases there are 2 of them that cover native functions from the following namespaces: `std::math` and `std::utils`.

Each test case is composed by a Bleach file (`.bch` extension) representing the Bleach program that will be executed by the interpreter, a file with the extension `.bch.expected`, representing the expected output for that corresponding `.bch` file and, finally, a corresponding `.bch.log` file that represents the actual output generated by the execution of such `.bch` file. Finally, the difference between the `.bch.expected` and the `.bch.log` is computed. If there are no differences, then it means that the particular test case passed. Otherwise, it did not.

All this process is automated by a shell script called `bleach_test_pipeline.sh` that automatically executes each Bleach program and compares its expected output file with the produced output file. In the end, this script provides a simple metric displaying how many of the test cases have passed from the total number and it also shows which test cases did not pass.

More detailed information about how to execute Bleach's test suite can be found at Bleach's official GitHub repository [74].

5.3 Implementing Famous Algorithms and Data Structures in Bleach

The main goal here is show the expressiveness and prowess of Bleach by demonstrating that it is possible to implement famous algorithms and data structures that are usually taught in an undergraduate "Introduction to Algorithms" course. These implementations, along with a few tests cases, can be found at Famous A & DS implementations in Bleach

5.4 Comparing Bleach with ChocoPy, Cool and MiniJava

In this section, a comparison between Bleach, ChocoPy, Cool and MiniJava is made in terms of the features available in these programming languages.

Feature	Bleach	ChocoPy	Cool	MiniJava
Type System	Dynamically-Typed	Statically-Typed	Statically-Typed	Statically-Typed
Built-In Types	`bool`, `nil`, `num`, `list`, `str`	`int`, `bool`, `None`, `list`, `str`	`Int`, `Bool`, `String`	`int`, `int[ ]` `boolean`, `void`
Arithmetical Operators	`+`, `-`, `*`, `/`, `%`	`+`, `-`, `*`, `//`, `%`	`+`, `-`, `*`, `/`	`+`, `-`, `*`
Comparison Operators	`>`, `>=`, `<`, `<=`	`>`, `>=`, `<`, `<=`	`<`, `<=`	`<`
Equality Operators	`==`, `!=`	`==`, `!=`, `is`	`=`	No support
Logical Operators	`and`, `or`, `!`	`and`, `or`, `not`	`not`	`&&`, `!`
Assignment Operator	`=`	`=`	`<-`	`=`
Ternary Operator Support	Yes	Yes	No	No
If Statement Support	`if`, `elif`, `else`	`if`, `elif`, `else`	`if`, `else`	`if`, `else`
Loops Support	`for`, `do-while`, `while`	`while`, `for in` (only in lists and strings)	`while`	`while`
Break/Continue Statements Support	Yes, Yes	No, No	No, No	No, No
Function Declaration Support	Yes (with the `function` keyword)	Yes (with the `def` keyword)	Yes (only method declaration within classes)	Yes (only method declaration within classes)
Anonymous Function Support	Yes (with the `lambda` keyword)	No	No	No
Comments	Single-line (`//`) and Multi-line (`/**/`)	Single-Line (`#`)	Single-line (`--`) and Multi-line (`**`)	Single-line (`//`) and Multi-line (`/**/`)
Language Paradigm	Object-Oriented and Interpreted	Object-Oriented and Compiled	Object-Oriented and Compiled	Object-Oriented and Compiled
Inheritance	Single	Single	Single	Single

Table 5.1: Comparison between Bleach, ChocoPy, Cool and MiniJava.

Chapter 6

Conclusion and Future Work

"Even without a form, we will never stop walking."

– TITE KUBO

6.1 Conclusion

This undergraduate thesis was written to present Bleach, a programming language whose major purpose is serve as an educational tool in undergraduate Compilers courses present in Computer Engineering and Computer Science programs. The language intends to lower the gap between theoretical concepts and practical implementation, by offering the students an engaging and interesting mechanism to apply what the core concepts related to compilers (lexical analysis, syntax analysis, static analysis, interpretation and runtime).

The implementation of the Bleach programming language consisted of building a lexer component, a parser component, a semantic resolver and, finally, the interpreter component itself, which can also be referred to as the language runtime. The development of the mentioned components enables the students to have a better grasp of the internals of programming language implementations. Last but not least, by implementing Bleach using a tree-walk interpreter approach, essential, but complex topics about compiler are simplified and the other ones that are even more difficult are delegated to advanced courses about programming languages, type systems and optimizing compilers.

When it comes down to design challenges, the hardest part when designing Bleach was finding the optimal balance between complexity and simplicity. The major aim was to create a programming language that was interesting for the students and robust enough to cover

key compiler phases, while still being approachable for students with limited experience in programming language design and implementation.

Lastly, it is expected that Bleach's establishment into Compilers courses provides students a complete, intuitive, interesting, and hands-on approach to understand the most relevant concepts related to programming language design and implementation. By allowing students to experiment with Bleach, it is anticipated that they can solidify their knowledge in this area. As for the teaching staff, Bleach offers an incremental, modular and robust approach for programming assignments or whole projects that directly reinforce the taught theoretical material.

6.2 Future Work

This section is dedicated to present directions for future development and research that build upon the work presented in this thesis.

First of all, it is important to recognize that, even though Bleach solved the problem it was designed to, it must not be considered, by no means, a silver bullet to it. Instead, unfortunately, Bleach has some limitations that could be object of improvement and research in order to extend it and make it even more robust, so it can better solve the problem it was proposed to.

With this in mind, the aspects below are limitations that could be worked on as a means to deliver a more fruitful and pleasant experience to both professors and students:

- **Allowance of constants through the use of the `const` keyword**: JavaScript, a famous dynamically-typed programming language allows the user to define constants in their programs so that they don't support assignments after the declaration of this constant. Adding such feature might be an interesting idea in order to make students think more about the concept of environments/scopes and resolving.

- **Extension of Bleach so it can support indexing on values of type `list` and `str`:** Unfortunately, the implementation of this feature could not be properly achieved given the deadline to deliver this thesis. On the other hand, a workaround was the usage of the built-in methods for both of these types: `setAt` (which works as an assignment to a certain index of a `list` or `str` value) and `getAt` (which works as a way to retrieve the element stored at a certain index of a `list` or `str` value). Although, the behavior is exactly the same, it is important to recognize the syntax may not have the best taste when

compared to an indexing option. This might be a good gap in Bleach's implementation that can taught the students more about concepts of parsing, resolving, runtime execution and runtime errors.

- **Addition of a `for ... in ...` structure, similar to Python:** The addition of such loop structure in Bleach can be fruitful, since it can used to teach students about syntactic sugar in the programming languages they are already familiar with, as well as about the desugaring process, common in several languages.

- **Enhancement of the Resolver component of the Bleach Interpreter:** Despite the fact that the Resolver component is already able to detect certain static errors, it could be augmented in order to detect new kinds of static errors, such as:

  - Detection of unused variables.
  - Detection of function overloading or re-declaration.
  - Detection of unreachable code.

- **Increasement of the quantity of test cases present in Bleach's Test Suite:** At the moment this thesis was delivered, Bleach's Test Suite was composed of 44 test cases that covered every single type of AST node, as well as some native function from Bleach's standard library. However, there is still more room to cover. For example, all of these mentioned tests deal with cases where the Bleach programs are correct and produce an output that is compared to expected ones. It is desirable to implement a battery of test cases that deal with incorrect Bleach programs and check whether the errors being thrown by the interpreter are indeed the expected ones.

- **Improvement of Bleach's REPL in order to make it more user-friendly:** In order to provide a better user experience when it comes to Bleach's REPL, it would be a great idea to allow line breaks when declaring functions, classes, anonymous functions, if statements and loops, so the readability of the written code could be improved.

- **Spreading of an opinion survey about Bleach to the target audience:** Since Bleach is a tool created to help the teaching and learning process in an undergraduate Compilers course, it seems reasonable to get feedback from those involved in this scenario.

With this in mind, an opinion pool could be created as a means to get input from professors, instructors and teaching assistants regarding the viability, upsides and drawbacks from using Bleach as a teaching tool in introductory Compiler courses.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] K. D. Cooper and L. Torczon, *Engineering a compiler*. Morgan Kaufmann, 2022.

[3] R. Nystrom, *Crafting interpreters*. Genever Benning, 2021.

[4] M. E. Lesk and E. Schmidt, *Lex: A lexical analyzer generator*, vol. 39. Bell Laboratories Murray Hill, NJ, 1975.

[5] Vern Paxson, *Flex: Fast Lexical Analyzer Generator*, 1987. Accessed: 2024-09-21.

[6] Terence Parr, *ANTLR (ANother Tool for Language Recognition)*, 2023. Accessed: 2024-09-21.

[7] The GNU Project, *Bison: The Yacc-compatible Parser Generator*, 2023. Accessed: 2024-09-21.

[8] Stephen C. Johnson, *Yacc: Yet Another Compiler-Compiler*, 1975. Accessed: 2024-09-21.

[9] J. Ashkenas and C. Community, "List of languages that compile to javascript." `https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js`, 2024. Accessed: 2024-10-05.

[10] Wikipedia contributors, "History of compiler construction," 2024. [Online; accessed 7-September-2024].

[11] Wikipedia contributors, "History of programming languages," 2024. [Online; accessed 7-September-2024].

[12] F. lang Community, "The fortran programming language, 2018 release." `https://fortran-lang.org/`, 2024. Accessed: 2024-08-30.

[13] B. W. Kernighan and D. M. Ritchie, *The C programming language*. prentice-Hall, 1988.

[14] B. Stroustrup, *The C++ Programming Language*. USA: Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2000.

[15] N. Wirth, "The programming language pascal," *Acta informatica*, vol. 1, pp. 35–63, 1971.

[16] D. M. Ritchie and K. Thompson, "The unix time-sharing system," *Bell System Technical Journal*, vol. 57, no. 6, pp. 1905–1929, 1978.

[17] S. C. Johnson *et al.*, *Yacc: Yet another compiler-compiler*, vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[18] Oracle, "The java™ language specification." `https://docs.oracle.com/javase/specs/jls/se17/html/`, 2021. Accessed: 2024-08-30.

[19] A. V. Aho, "Teaching the compilers course," *ACM SIGCSE Bulletin*, vol. 40, no. 4, pp. 6–8, 2008.

[20] J. H. Lasseter, "The interpreter in an undergraduate compilers course," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pp. 168–173, 2015.

[21] A. W. Appel, "The tiger programming language." Documentation provided in the "Modern Compiler Implementation" series of books, 1998. Educational language for compiler courses.

[22] A. W. Appel, *Modern Compiler Implementation in C, 1st Edition*. Cambridge University Press, 1997.

[23] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java, 2nd Edition*. Cambridge University Press, 2002.

[24] A. W. Appel, *Modern Compiler Implementation in ML, 1st Edition*. Cambridge University Press, 2004.

[25] D. Kundra and A. Sureka, "An experience report on teaching compiler design concepts using case-based and project-based learning approaches," in *2016 IEEE Eighth International Conference on Technology for Education (T4E)*, pp. 216–219, IEEE, 2016.

[26] B. A. Altman, "Workplace bullying: Application of novak's (1998) learning theory and implications for training," *Employee Responsibilities and Rights Journal*, vol. 22, pp. 21–32, 2010.

[27] V. L. Golich, "The abcs of case teaching," *International Studies Perspectives*, vol. 1, no. 1, pp. 11–29, 2000.

[28] C. E. Hmelo-Silver, "Problem-based learning: What and how do students learn?," *Educational psychology review*, vol. 16, pp. 235–266, 2004.

[29] V. P. Dennen and K. J. Burner, "The cognitive apprenticeship model in educational practice," in *Handbook of research on educational communications and technology*, pp. 425–439, Routledge, 2008.

[30] J. W. Thomas, "A review of research on project-based learning," tech. rep., Autodesk Foundation, San Rafael, CA, 2000.

[31] S. Muchnick, *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[32] Amazon, "Crafting interpreters by robert nystrom [paperback]." `https://www.amazon.com/Crafting-Interpreters-Robert-Nystrom/dp/0990582930`, 2021. Accessed: 2024-09-02.

[33] GitHub, "Github public repositories related to the "crafting interpreters" book." `https://github.com/search?q=crafting+interpreters&type=repositories`, 2024. Accessed: 2024-09-02.

[34] A. Aiken, "Cool: A portable project for teaching compiler construction," *ACM Sigplan Notices*, vol. 31, no. 7, pp. 19–24, 1996.

[35] A. Aiken, "The cool reference manual." `https://theory.stanford.edu/~aiken/software/cool/cool-manual.pdf`, 2011. Accessed: 2024-09-05.

[36] S. University, "Stanford university - cs143 - compilers course page." `http://web.stanford.edu/class/cs143/`, 2024. Accessed: 2024-09-05.

[37] U. of Illinois Urbana-Champaign, "University of illinois urbana-champaign (fall 2022) - cs426 - compiler construction course page." `https://courses.grainger.illinois.edu/cs426/fa2022/`, 2022. Accessed: 2024-09-05.

[38] U. of Michigan, "University of michigan - eecs483 (2018) - compiler construction course page." https://dijkstra.eecs.umich.edu/eecs483/index.php, 2018. Accessed: 2024-09-05.

[39] C. U. Press, "Modern compiler implementation in java: the minijava project." https://www.cambridge.org/resources/052182060X/, 2024. Accessed: 2024-09-03.

[40] C. U. Press, "Bnf for minijava." https://www.cambridge.org/resources/052182060X/MCIIJ2e/grammar.htm, 2024. Accessed: 2024-09-03.

[41] U. of Washington, "University of washington cse401 and cse m 501 (2024) - compiler construction course project page." https://courses.cs.washington.edu/courses/cse401/24sp/project/, 2024. Accessed: 2024-09-03.

[42] J. Palsberg, "Ucla cs132 (fall 2012) - compiler construction course page." https://web.cs.ucla.edu/~palsberg/course/cs132/F12/, 2012. Accessed: 2024-09-03.

[43] L. U. of Chicago, "Loyola university of chicago (fall 2018) - comp 271 minijava project page." https://pld.cs.luc.edu/courses/271/fall18/mnotes/compiler.html, 2018. Accessed: 2024-09-03.

[44] C. M. Kirsch, "Selfie and the basics," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 198–213, 2017.

[45] U. of Salzburg, "Selfie | an educational software system of a tiny self-compiling c compiler, a tiny self-executing risc-v emulator, and a tiny self-hosting risc-v hypervisor.." http://selfie.cs.uni-salzburg.at/, 2017. Accessed: 2024-09-07.

[46] C. Kirsch, "Selfie | official github repository." https://github.com/cksystemsteaching/selfie, 2017. Accessed: 2024-09-07.

[47] D. E. Knuth, *The Art of Computer Programming, Volumes 1-4*. Addison Wesley, 2011. 4 volumes.

[48] N. Wirth, *Algorithms + Data Structures = Programs*. Prentice Hall, 1976.

[49] N. Wirth, *Compiler Construction*. Addison Wesley, 1996.

[50] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th ed., 2011.

[51] J. Liedtke, "Toward real microkernels," *Communications of the ACM*, vol. 39, no. 9, pp. 70–77, 1996.

[52] M. Richards and C. Whitby-Strevens, *BCPL: The Language and its Compiler*. Cambridge University Press, 2009.

[53] E. W. Dijkstra, "The structure of the "the"-multiprogramming system," *Communications of the ACM*, vol. 11, pp. 341–346, May 1968.

[54] A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[55] N. Nisan and S. Schocken, *The Elements of Computing Systems: Building a Modern Computer from First Principles*. MIT Press, 2005.

[56] G. J. Sussman and H. Abelson, *Structure and Interpretation of Computer Programs*. MIT Press, second edition ed., 1996.

[57] M. Sipser, "Introduction to the theory of computation," *ACM Sigact News*, vol. 27, no. 1, pp. 27–29, 1996.

[58] Czech Technical University in Prague, "Czech technical university (2024/2025) - bi-ccn - compiler construction course page." `https://bilakniha.cvut.cz/en/predmet7514106.html#gsc.tab=0`, 2024. Accessed: 2024-09-07.

[59] R. Padhye, K. Sen, and P. N. Hilfinger, "Chocopy: A programming language for compilers courses," in *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, pp. 41–45, 2019.

[60] R. Padhye, K. Sen, and P. Hilfnger, "Chocopy project official website." `https://chocopy.org/`, 2019. Accessed: 2024-09-05.

[61] P. S. Foundation, "Python programming language, version 3.12.5." `https://www.python.org/`, 2024. Accessed: 2024-09-02.

[62] A. Waterman, D. A. Patterson, Y. Lee, and K. Asanović, *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA Version 2.0*. RISC-V Foundation, version 2.0 ed., 2014.

[63] T. Delft, "Tu delft - cs4200 (2020) - compiler construction course page." `https://tudelft-cs4200-2020.github.io/`, 2020. Accessed: 2024-09-05.

[64] U. S. Diego, "Uc san diego - cse231 (winter 2021) - advanced compiler design course page." `https://ucsd-cse231-w21.github.io/`, 2021. Accessed: 2024-09-05.

[65] N. Y. University, "New york university - csci-ga 2130 (2020) - compiler construction course page." `https://cs.nyu.edu/dynamic/courses/schedule/?semester=spring_2020&level=GA&day=M#csci-ga2130-001-desc`, 2020. Accessed: 2024-09-05.

[66] U. Khedker, "Sclp: A language processor for a small c-like language." `https://www.cse.iitb.ac.in/~uday/sclp-web/`, 2020. Accessed: 2024-09-05.

[67] R. Padhye, K. Sen, and P. Hilfnger, "Chocopy v2.2: Language manual and reference." `https://chocopy.org/chocopy_language_reference.pdf`, 2019. Accessed: 2024-09-05.

[68] ECMA International, "Javascript," 2023. ECMAScript Language Specification, Edition 13.

[69] Y. Matsumoto, "Ruby programming language," 2023. Version 3.2.

[70] T. Hoare, "Null references: The billion dollar mistake," 2009. Presentation at QCon London 2009.

[71] Wikipedia contributors, "The diamond problem," 2024. [Online; accessed 20-September-2024].

[72] Wikipedia contributors, "Flex (lexical analyser generator)," 2024. Accessed: 2024-09-21.

[73] G. Klein, "Jflex user's manual," *Available on-line at www.jflex.de*, 2010.

[74] Victor Miguel de Morais Costa, "The bleach programming language official github repository," 2024. Accessed: 2024-09-28.