**Bulgarian Diploma Thesis**

**Nulascript**

**Michael Bozhilov, ID#000000000**

**Student:** Michael Bozhilov **, Date: 30.11.2023**

*signature*

**Supervisor:**   **, Date:**

*signature*

**Department of Computer Science, AUBG**

**Blagoevgrad, 2023**

1. Table of Contents

[1. Introduction: 4](#_Toc152266286)

[1. Overview of the project 4](#_Toc152266287)

[2. Motivations for Dynamic Typing 4](#_Toc152266288)

[3. Motivations for a Python-like Parser 4](#_Toc152266289)

[4. Innovation 4](#_Toc152266290)

[2. Specification of the Software Requirements and their Analysis 5](#_Toc152266291)

[1. Purpose 5](#_Toc152266292)

[2. Scope 5](#_Toc152266293)

[3. Functional Requirements 5](#_Toc152266294)

[3.1 Dynamic Typing System 6](#_Toc152266295)

[3.2 Referencing and Dereferencing 6](#_Toc152266296)

[3.3 Syntax 6](#_Toc152266297)

[3.4 Control Flow 7](#_Toc152266298)

[3.5 Functions and Closures support through First-Class Citizens Assignment 7](#_Toc152266299)

[3.6 Supporting String Literals 7](#_Toc152266300)

[3.7 Supporting Integer Literals 7](#_Toc152266301)

[3.8 Supporting Boolean Literals 8](#_Toc152266302)

[3.9 Printing 8](#_Toc152266303)

[4 Non-functional Requirements 8](#_Toc152266304)

[4.1 Performance 8](#_Toc152266305)

[4.2 Cross-Platform Compatibility 8](#_Toc152266306)

[4.3 Extensive Test Suites 8](#_Toc152266307)

[4.4 Maintainability 8](#_Toc152266308)

[3. Design of the software solution 9](#_Toc152266309)

[1. The Lexer 9](#_Toc152266310)

[2. The Parser 11](#_Toc152266311)

[1. Nulascript and LL Grammar 14](#_Toc152266312)

[2. Understand LL Grammar 15](#_Toc152266313)

[3. LL Grammar Strengths 15](#_Toc152266314)

[4. Challenges 16](#_Toc152266315)

[5. Nulascript’s approach 17](#_Toc152266316)

[6. The interpreter 17](#_Toc152266317)

[6.1 Read-Evaluate-Print-Loop 18](#_Toc152266318)

[7. Object-Oriented Paradigms in Nulascript 20](#_Toc152266319)

[7.1 Safeguarding Implementation Details 20](#_Toc152266320)

[7.2 Safeguarding Implementation Details 21](#_Toc152266321)

[7.3 Power of Flexibility 21](#_Toc152266322)

[7.4 Building Hierarchies of Reusability 22](#_Toc152266323)

[7.5 Design Patterns 22](#_Toc152266324)

[7.6 Future Prospects 22](#_Toc152266325)

[8. Conclusion: Nulascript's OOP implementation 23](#_Toc152266326)

[9. Nulascript’s Syntax: A fusion of C++, Python and JavaScript Syntax for Seamless Productivity 23](#_Toc152266327)

[9.1 C++ Influence 23](#_Toc152266328)

[9.2 Syntax Interchangeability 24](#_Toc152266329)

[9.3 Nulascript’s Syntactic Mixture 24](#_Toc152266330)

[10 Lexical Precedence 24](#_Toc152266331)

[11 Functions as First Class Citizens 26](#_Toc152266332)

[11.1 Higher-Order Functions 26](#_Toc152266333)

[11.2 Abstraction and Modularity Without Supporting an OOP System 26](#_Toc152266334)

[11.3 Dynamic Behaviour 27](#_Toc152266335)

[11.4 Closures 27](#_Toc152266336)

[4. Implementation 27](#_Toc152266337)

[1. Lexer’s Core 27](#_Toc152266338)

[1.1 Handling Complex Tokens 30](#_Toc152266339)

[1.1.2 Identifier and Reserved Keywords Handling 31](#_Toc152266340)

[1.1.3 Integer Literal Handling 31](#_Toc152266341)

[1.1.3 Illegal Character Handling 32](#_Toc152266342)

[1.2 Abstract Syntax Tree 32](#_Toc152266343)

[2. Abstract Syntax Tree Implementation 33](#_Toc152266344)

[3. Parser Implementation 39](#_Toc152266345)

[3.1 Parsing Initialization Statements 39](#_Toc152266346)

[3.2 Parsing Return Statements 40](#_Toc152266347)

[3.3 Parsing Expression Statements 40](#_Toc152266348)

[3.4 Debugging Parsing of an Infix Expression 41](#_Toc152266349)

[3.5 Parsing References 42](#_Toc152266350)

[4. Evaluation 43](#_Toc152266351)

[4.1 Storage and Representation of Values 43](#_Toc152266352)

[4.2 Evaluation Function 46](#_Toc152266353)

[5. Testing: 48](#_Toc152266354)

[1. Automated Testing 48](#_Toc152266355)

[1.1 Unit Tests 48](#_Toc152266356)

[1.2 Functional Tests (End-to-End testing) 50](#_Toc152266357)

[2. Regressions 51](#_Toc152266358)

[3. Tested Functional Requirements 53](#_Toc152266359)

[3.1 Dynamic Typing System 53](#_Toc152266360)

[3.2 Referencing & Dereferencing 54](#_Toc152266361)

[3.3 Functions & Closures 54](#_Toc152266362)

[3.3 Syntax 55](#_Toc152266363)

[3.4 Control Flow 55](#_Toc152266364)

[4. Tested Non-functional requirements 56](#_Toc152266365)

[4.1 Maintainability 56](#_Toc152266366)

[4.2 Portability 57](#_Toc152266367)

[6. Results and Conclusion: 57](#_Toc152266368)

[1. Performance – Nulascript vs. Javascript (Node v18) 57](#_Toc152266369)

[2. Takeaways 60](#_Toc152266370)

[8. References: 62](#_Toc152266371)

# **1. Introduction:**

## Overview of the project

The creation of Nulascript stems from a deliberate effort to address the limitations perceived in existing programming languages and to explore the benefits of dynamic typing in the context of scripting languages. This thesis outlines the intentions behind the development of Nulascript, delves into the motivations that led to the choice of dynamic typing and the inclusion of references, and elucidates the significance of understanding parsing mechanisms in the language's design.

## Motivations for Dynamic Typing

The conventional paradigm of statically typed languages offers advantages in terms

of performance and error detection at compile-time. However, the inherent inflexibility

of static typing systems can impede the fluidity of the coding process. Nulascript, as

a dynamically typed language, prioritizes adaptability and ease of expression by

allowing variables to assume types dynamically during runtime. This design choice

aims to facilitate a more intuitive and flexible development experience, where the

coder can respond dynamically to evolving requirements.

## Motivations for a Python-like Parser

Having embarked on my professional journey with Python, a language renowned for its readability and emphasis on developer-friendly practices, I found myself naturally drawn to adopting a similar methodology in implementing the language's interpreter. Drawing from the experiences and principles ingrained in Python development, I made the deliberate decision to employ a top-down parser for the interpreter.

## Innovation

Nulascript**,** akin to statically typed C-level languages, employs references. References are a fundamental concept in computer programming, especially in low-level languages like C. Unlike pointers, which store memory addresses, references enable direct access to memory locations without duplicating actual values. This distinction is significant for efficiency, allowing manipulation of data in memory without unnecessary copying. In contexts such as high-level scripting languages operating on virtual machines and in deployment pipelines, the avoidance of data duplication and the ability to directly access memory locations can result in expedited execution and more cost-effective infrastructure management.

# **2. Specification of the Software Requirements and their Analysis**

## Purpose

Nulascript serves as a dynamically typed scripting language designed with a focus on flexibility, ease of use, and the introduction of references to create aliases for variables. The language aims to provide developers with the freedom to write code without the constraints of static type declarations, fostering rapid development and experimentation.

## Scope

The scope of the Nulascript project, within the confines of the current academic semester, centers around the design, development, and continuous evolution of a dynamically typed scripting language. The primary emphases lie in fostering flexibility, ensuring usability, and laying the groundwork for potential performance optimization and iterative development in the future. While the overarching goal is to create a language that aligns with scripting principles, it's important to note that the immediate focus may not extensively delve into fine-tuning performance or executing iterative updates.

Given the nature of this project as a senior thesis, the primary emphasis is on establishing the foundational elements of the scripting language. This includes defining the dynamic typing system, incorporating references for variable management, and crafting a syntax that promotes ease of use. While considerations for potential performance optimizations and iterative development remain on the radar, they are not mandatory focal points for the current timeline.

## Functional Requirements

Functional requirements for the Nulascript project outline the essential features and behaviors that the dynamically typed scripting language is expected to exhibit. These requirements are crucial for the language to fulfill its intended purpose and meet the needs of its potential users.

### 3.1 Dynamic Typing System

**def** a = 5;

a = "some-string"

Pseudo Code 1: Dynamic Typing System

* The language must support dynamic typing, allowing variables to be assigned without explicit type declarations.
* It should handle runtime type determination and adaptability.
* Potential Constraint: performance implications of dynamic typing for runtime evaluation.

### 3.2 Referencing and Dereferencing

**def** a = 5;

**def** b = &a;

log(a + \*b);

Pseudo Code 2: Referencing and Dereferencing

* The introduction of references as a feature for creating aliases for variables should be supported.
* Language should support dereferencing of references.
* Potential Constraint: References which are not garbage collected when referenced variables are deleted might lead to segmentation faults.

### 3.3 Syntax

**if** (someVar **is** **not** anotherVar) # python

# or

**if** (someVar != anotherVar) # c++ / javascript

Pseudo Code 3: Interchangeable Syntax

* The language should have a syntax that is intuitive and draws inspiration from established programming languages.
* Syntax elements should be consistent and familiar to developers with diverse language backgrounds
* Potential Constraint: Grammar is non-implementable due to its incompatibility with LL parsers; Learning curve introduced by any deviations from conventional syntax.

### 3.4 Control Flow

**if** (true) **else** { "hello world" };

**for** (**def** a = 5; 5 < 10; a + 2) {};

Pseudo Code 4: if statement and a for loop

* The language must include constructs for control flow, such as conditional statements (if-else) and loops (for).
* Control flow mechanisms should be expressive and flexible.

### 3.5 Functions and Closures support through First-Class Citizens Assignment

**def** closure = func(x) { func(y) { y + x } }

Pseudo Code 5: Closure declaration

* Nulascript must support the definition and invocation of functions
* Nulascript should support functions as first-class citizens.
* Functions should allow for parameters, return values, and support both built-in and user-defined functions.
* Potential Constraint: Variables stored within a closure’s environment are not garbage collected once the function goes out of scope=.

### 3.6 Supporting String Literals

"hello world"

Pseudo Code 6: String literal

* Nulascript should support the concatenation of strings using a designated operator or function.

### 3.7 Supporting Integer Literals

100042

Pseudo Code 7: Integer literal

* Nulascript should support arithmetic operations with numbers (language is to support only integers at the initial stage of its development)

### 3.8 Supporting Boolean Literals

true false

Pseudo Code 8: Boolean literal

* Nulascript should support logical comparisons with `true` and `false` values.

### 3.9 Printing

log("Hello World")

Pseudo Code 9: Logging function invocation

* Nulascript is required to facilitate printing to the standard output (stdout) by utilizing a built-in logging function.

## 4 Non-functional Requirements

### 4.1 Performance

* The Nulascript language should strive to achieve competitive performance compared to widely adopted scripting languages, with a specific focus on being comparable to or faster than JavaScript in typical use cases.

### 4.2 Cross-Platform Compatibility

* Nulascript should be designed and implemented to run consistently across multiple operating systems, ensuring compatibility with major platforms such as Windows, macOS, and various Linux distributions.

### 4.3 Extensive Test Suites

* The development of Nulascript should be supported by comprehensive and well-maintained test suites that cover a wide range of language features, functionalities, and potential edge cases. These test suites are essential for ensuring the robustness and reliability of the language.

### 4.4 Maintainability

* The codebase of Nulascript should adhere to best practices, coding standards, and maintainability guidelines. This ensures that the language remains maintainable and adaptable for future enhancements.

# **3. Design of the software solution**

The process of running **Nulascript** code involves several stages. At a high level, it starts with the user writing code in a text file. This code is then passed to a lexer, which breaks it down into tokens. These tokens are then used as input for a parser to construct an abstract syntax tree. Finally, the AST is interpreted to execute the code.

## The Lexer

A screenshot of a computer

Description automatically generated

Figure 1: Token definition and Lexer class

The lexer, also known as a lexical analyzer, is a core component of a compiler or interpreter. Its primary role is to examine the source code of a programming language and divide it into individual tokens. These tokens are the smallest meaningful units of code, like keywords, identifiers, operators, and literals. The lexer categorizes these tokens and sends them to subsequent stages in the compilation or interpretation process for further analysis, where the code's structure and meaning are further understood and processed. Through this systematic dissection, Nulascript’s lexer plays a pivotal role in enabling the computer to comprehend and execute the programmer's instructions with precision and accuracy.

Lexers serve as the initial gatekeepers in the complex realm of language parsing, employing a distinctive algorithm designed to swiftly and accurately distill raw source code into meaningful tokens. At the heart of this algorithm lies a dynamic stateful parsing approach, deftly navigating the intricate tapestry of characters in the input stream. By maintaining awareness of the current and impending positions within the code, the lexer adeptly maneuvers through the syntax, uncovering the semantics hidden within.

A notable feature is the algorithm's adaptive nature, guided by dynamic function pointers that selectively engage specific character validation routines. This dynamic function selection injects a layer of versatility into the algorithm, enabling it to tailor its behavior based on the nuanced requirements of distinct token types. Such adaptability not only streamlines the parsing process but also fosters an environment of modular code, enhancing maintainability.

Efficiency takes center stage in the lexer's algorithm, with a keen focus on recognizing and processing diverse language constructs. Specialized pathways within the algorithmhandle intricate elements, be it the extraction of string literals or the identification of extended tokens like integers and identifiers. The algorithm navigates through the input, deftly stepping over whitespace characters, tabs, and newlines, ensuring that the parsing endeavor remains unhindered by inconsequential whitespace, a strategic move to enhance the algorithm's overall agility.

This discriminating approach guarantees the precise identification of operators amidst the sea of characters, fortifying the algorithm against potential misinterpretations.

In the face of unexpected characters, the algorithm gracefully navigates through the syntax, marking anomalies with a token of the **ILLEGAL** variety. This robust approach to error handling serves as a sentinel, standing guard against syntax pitfalls and fortifying the algorithm's ability to gracefully navigate through imperfect code.

As the algorithm traverses the code, it bears witness to the conclusion of the source stream, acknowledging the end of the input through a designated token. This careful consideration for the end-of-file scenario ensures the algorithm's graceful denouement, completing its tokenization journey with a sense of closure.

Top of FormBottom of Form

## The Parser

The parser, also known as a syntactic analyzer, is another core component which primary task is to inspect the source code tokens generated by the lexer and arrange them into a hierarchical structure known as the syntax abstract tree (6). The tree servers as a representation of the source code which clearly defines the relationship between different tokens and their grammatical structure. The parser's role is to verify whether the code complies with the language's syntax rules and to generate a structured representation that can be further processed by the compiler or interpreter. Additionally, there exist various types of parsers. In the context of Nulascript, the tokens are processed by a Pratt parser, also referred to as a top-down operator precedence parser (2). Its parser primarily begins at the top of the parse tree, examining the initial symbol of the grammar, and proceeds linearly by inspecting the subsequent tokens to determine the appropriate path to take.

This strategy is particularly effective for simpler or moderately complex programming languages, aligning well with the typical code structure of these languages. In the context of a recursive descent parser, each non-terminal symbol in the grammar corresponds to a function in the code. These functions are tasked with recognizing and parsing the corresponding grammar rules, with the parser recursively invoking these functions to navigate through the input code.

This top-down parsing technique is specifically geared towards LL(k) grammars, where "LL" signifies "left-to-right, leftmost derivation," and "k" denotes the number of lookahead tokens. Recursive descent parsers are known for their hand-crafted nature, involving the manual creation of parsing functions based on the language's grammar (4).

Recursion is pivotal to the workings of recursive descent parsing (1). Parsing functions recursively call each other to handle nested constructs within the language, establishing a seamless alignment with the natural hierarchy of the language's syntax. For instance, a parsing function for expressions might invoke another for sub-expressions, maintaining the hierarchical order.

The distinction in parser types arises from the methodology of associating parsing functions with specific tokens. The core concept involves assigning two distinct sets of parsing functions to each token, depending on whether the token functions as a prefix or an infix.

Despite their strengths, recursive descent parsers have limitations (3). They are most effective for languages with simple to moderately complex grammars that adhere to LL(k) specifications. In cases of ambiguity or left-recursive grammars, alternative parsing techniques such as LR parsing may be more suitable. Error handling in recursive descent parsers is often straightforward, generating error messages when unexpected input occurs. While backtracking is a common strategy for resolving prediction errors, it's noteworthy that Nulascript, in this case, does not implement backtracking.

The fundamental concept behind Pratt parsing involves associating parsing functions with each operator in the language. These functions are responsible for parsing expressions that involve the respective operator. With each operator having left and right precedence, the Pratt parser seamlessly handles different levels of precedence (7).

In contrast to traditional recursive descent parsers, Pratt parsers exhibit greater flexibility in handling precedence by dynamically adapting to the current token's precedence. This adaptability makes Pratt parsers particularly effective for languages with intricate operator precedence rules.

Pratt parsers are renowned for their simplicity, efficiency, and ease of extensibility. They find common use in implementing expression parsers for programming languages and other systems requiring the parsing of mathematical or logical expressions.

A diagram of a function

Description automatically generated

Figure 2: AST Definition

In the process of parsing code, a crucial step involves transforming the raw source code into an Abstract Syntax Tree (AST), a hierarchical representation that captures the syntactic structure of the program. This AST is structured through a series of object-oriented classes that form a cohesive framework for understanding and manipulating the parsed information. At the top of this hierarchy is the abstract Node class, serving as the foundation for all AST nodes. This class includes essential methods such as tokenLiteral() and toString(), ensuring uniformity across the various nodes in the tree. The Node class is then extended into distinct categories, namely Statement, Expression, and Program, each catering to specific constructs within the programming language.

Within the Statement category, further specialization occurs through classes like ReturnStatement and LetStatement, encapsulating the syntax and semantics of return and let statements, respectively. On the Expression side, a variety of classes is defined to handle different expression types. For instance, there may be classes for IntegerExpression and ConditionalExpression, each responsible for representing and interpreting their respective expression forms.

This object-oriented approach to constructing an AST allows for a clear and modular organization of the code's abstract representation. As the parser encounters various statements and expressions in the source code, it instantiates the corresponding AST nodes, ultimately forming a structured and navigable tree that mirrors the program's syntax. This AST serves as a powerful intermediary for subsequent stages in the compilation or interpretation process, enabling efficient analysis and transformation of the code.

A screenshot of a computer

Description automatically generatedIn Figure 2, a pivotal component of the parsing process is depicted through the Parser class, representing an intermediate stage where the integration between lexical analysis and abstract syntax tree (AST) construction takes place. Within this class, the lexer is invoked to tokenize the raw source code, breaking it down into meaningful units. Subsequently, the parser meticulously processes these tokens, mapping them to their corresponding AST definitions based on the language's syntax rules. This crucial step involves instantiating instances of the Node class and its subclasses, such as Statement, Expression, and Program, as dictated by the detected constructs in the code.

Figure : Parser Class

### Nulascript and LL Grammar

In the ever-evolving landscape of programming languages, the choice of a specific grammar plays a pivotal role in shaping a language's readability, ease of parsing, and overall expressiveness. Nulascript, a language that has garnered attention for its clarity and efficiency, strategically incorporates LL grammar, a decision that profoundly influences its syntax, parsing methodology, and developer experience. In this exploration, we delve into the nuances of LL grammar, dissecting its merits, potential challenges, and the reasons behind Nulascript's alignment with this parsing paradigm.

### 2. Understand LL Grammar

LL grammar, standing for "left-to-right, leftmost derivation," is a form of context-free grammar that plays a fundamental role in predictive parsing. This parsing strategy involves traversing the input from left to right and constructing a leftmost derivation of the input sentence. The "LL" designation further signifies the number of lookahead tokens considered during parsing. For example, LL(1) indicates a single lookahead token.

The predictability inherent in LL parsing stems from the fact that, at each parsing step, the parser decides which production rule to apply based solely on the next input symbol. This predictive nature simplifies the parsing process, offering advantages in terms of speed and clarity. Nulascript's adoption of LL grammar aligns with these benefits, contributing to a parsing strategy that is both efficient and conducive to code readability.

### LL Grammar Strengths

#### Clarity

LL grammar's top-down parsing approach resonates with the human thought process. Reading code from top to bottom, left to right, aligns with how people naturally approach problem-solving and code comprehension. The LL parsing strategy allows for straightforward translation of high-level language constructs into parsing rules, fostering a clear mapping between the language's syntax and its representation in the parser.

#### 3.2 Ease of Implementation

LL parsers, especially recursive descent parsers, are known for their ease of implementation. This simplicity arises from the direct correspondence between grammar rules and parsing functions in the code. Nulascript benefits from this simplicity, which allowed the crafting of a parser that mirrors the language's grammar with minimal abstraction layers.

#### Efficiency in Parsing

LL parsers, being predictive, exhibit efficiency during the parsing process (5). The parser anticipates the next production rule without the need for extensive backtracking, resulting in faster parsing times. This efficiency aligns with Nulascript's commitment to providing a responsive and performant development experience.

#### Error Reporting

Error reporting in LL parsers tends to be straightforward. When unexpected input is encountered, error messages can be generated promptly, aiding developers in identifying and rectifying issues. Nulascript's LL parsing strategy facilitates clear and concise error handling, contributing to a more robust development experience.

### Challenges

While LL grammar brings numerous advantages, it is essential to acknowledge potential challenges and considerations that may arise in specific scenarios.

#### Complex Grammars

LL grammars are well-suited for languages with simple to moderately complex grammars adhering to LL(k) specifications. However, in cases of highly complex grammars, LL parsing might become impractical, necessitating alternative parsing strategies like LR parsing.

#### Left Recursion

Handling left recursion can be challenging in LL grammars. When a production rule directly or indirectly refers to itself from the left side, it can lead to parsing ambiguities (never generating a terminating symbol). Mitigating left recursion often involves restructuring the grammar, which may introduce additional complexity (8).

#### Limited Expressiveness

The predictive nature of LL parsing limits the expressiveness of grammars. Certain language constructs, especially those requiring extensive lookahead, might be cumbersome to express within the constraints of LL grammar.

### Nulascript’s approach

Nulascript's embrace of LL grammar is a deliberate choice rooted in the language's goals of clarity, simplicity, and developer-centric design. By aligning with LL parsing, Nulascript provides a parsing experience that mirrors the natural thought processes of developers, making code more accessible and readable.

The language's grammar, carefully crafted to adhere to LL specifications, ensures that the parsing process is both efficient and predictable. Recursive descent parsing, a manual and handcrafted approach, allows for a direct translation of grammar rules into code, minimizing abstraction and enhancing the clarity of the parser's implementation.

Nulascript acknowledges the potential challenges of LL grammar, particularly in handling left recursion and addressing the limitations on grammar expressiveness. However, these considerations are weighed against the language's emphasis on simplicity and readability, with careful design choices made to strike a balance between expressiveness and the benefits of LL parsing.

In conclusion, Nulascript's incorporation of LL grammar reflects a nuanced understanding of parsing strategies and their impact on the developer experience. By embracing the strengths of LL parsing while acknowledging its limitations, Nulascript navigates the terrain of language design with a commitment to providing developers with a language that is not only powerful but also a joy to read and write.

## The interpreter

An interpreter is, as the name implies, the underlying software which interprets the meaning of our code by walking the abstract tree we’ve just created. Interpreters provide a direct way to execute commands without building a machine code executable. Rather it executes the code on the fly which enables us to interactively run commands, if needed. At a high level, Nulascript code will run from a C++ binary, which breaks down into machine code and contains instructions on how to execute each step from the tree.

### Read-Evaluate-Print-Loop

The Read-Evaluate-Print-Loop (REPL) stands as a fundamental component of an interpreter, offering a dynamic and interactive environment for developers to experiment with code snippets in real-time. This iterative process of reading, evaluating, and printing results forms the backbone of interactive programming and is a defining feature of languages like Nulascript.

#### 6.1.1 Read-Evaluate-Print-Loop Cycle

At its core, the REPL cycle encapsulates the continuous loop of user interaction. The journey begins with the user input, which is read by the interpreter. This input is then evaluated, meaning that the interpreter processes the code and executes the corresponding actions. Following evaluation, the results are printed back to the user. This loop seamlessly repeats, providing an instant feedback loop for developers to refine their code incrementally.

In the context of Nulascript, the REPL cycle serves as a conduit for translating high-level code written in the language into actionable steps. The interpreter, embedded within a C++ binary, takes on the responsibility of orchestrating this cycle, ensuring that each stage is executed cohesively.

#### 6.1.2 Read-Evaluate-Print-Loop Cycle

One of the paramount advantages of a REPL environment is its interactivity. Developers can experiment with code snippets, test hypotheses, and iterate rapidly without the need to compile and execute an entire program. This immediacy fosters a creative and exploratory programming mindset.

Nulascript, by embracing the REPL paradigm, encourages developers to engage in a dialogue with their code. This interactive nature proves invaluable during the development and debugging phases, allowing for quick iterations and the identification of issues in real-time. The ability to test individual lines of code or small segments before incorporating them into larger projects enhances both the learning process and overall development efficiency.

#### Navigating the Abstract Syntax Tree (AST)

Within the REPL cycle, the interpreter navigates the abstract syntax tree (AST) created during the parsing phase. The AST serves as an intermediate representation of the code's structure, breaking it down into a hierarchy of nodes that capture the semantics of the program.

As the interpreter evaluates user input, it traverses the AST, executing each node's corresponding actions. This process involves interpreting the meaning behind the code, handling variables, executing functions, and managing control flow. The ability to dynamically explore and manipulate the AST within the REPL environment is a powerful tool for both understanding code behavior and refining its logic.

#### The Power of Incremental Development

The REPL environment empowers developers to adopt an incremental approach to code development. Rather than writing large chunks of code in isolation, developers can build and refine their programs incrementally, testing and validating each addition in real-time. This methodology aligns with the principles of agile development, allowing for rapid adaptation to changing requirements and continuous improvement.

Nulascript, by embracing the REPL philosophy, supports this iterative development model. Developers can witness the immediate impact of their code modifications, facilitating a more organic and responsive programming experience. This approach not only accelerates the development cycle but also contributes to the creation of more robust and maintainable code.

#### Learning and Exploration

The REPL is not merely a tool for code execution; it is also an educational resource. Developers, whether novices or seasoned professionals, can leverage the REPL environment to experiment with language features, explore libraries, and test hypotheses. This hands-on learning experience contributes to a deeper understanding of the language and its nuances.

In the context of Nulascript, the REPL becomes a playground for developers to probe the language's capabilities, experiment with different syntax constructs, and gain insights into the inner workings of the interpreter. This symbiotic relationship between developers and the REPL fosters a continuous learning environment, where discovery and experimentation go hand in hand.

#### Challenges and Considerations

While the REPL brings numerous benefits, it is not without challenges. Managing state and preserving the context across iterations, handling side effects, and ensuring a seamless user experience are among the considerations that demand careful attention. Nulascript's approach to addressing these challenges within the REPL cycle contributes to the language's overall reliability and user satisfaction.

#### The Interactive Tapestry of Nulascript

In essence, the Read-Evaluate-Print-Loop forms the interactive tapestry of Nulascript, weaving together the user's input, the interpreter's evaluation, and the immediate feedback loop. This dynamic and iterative process embodies not just a mechanism for code execution, but a philosophy of engagement and exploration. As Nulascript developers traverse the realms of code, the REPL stands as their steadfast companion, facilitating a journey of discovery, creativity, and continuous improvement.

## Object-Oriented Paradigms in Nulascript

Nulascript, a language designed with a commitment to clarity, extensibility, and maintainability, has been meticulously crafted using Object-Oriented Programming (OOP) paradigms. OOP serves as the guiding architectural framework, providing a structured and modular approach to software design. In this exploration, we delve into the core tenets of OOP manifested in Nulascript, unraveling the intricate tapestry of encapsulation, abstraction, and polymorphism that defines the language's foundation.

### 7.1 Safeguarding Implementation Details

At the heart of Nulascript's design philosophy lies the principle of encapsulation. This OOP concept involves bundling data and the methods that operate on that data within a single unit, commonly known as a class. Encapsulation acts as a protective barrier, concealing the internal details of a class and exposing only what is necessary for external interaction.

In Nulascript, classes act as encapsulation units, encapsulating data attributes and the operations that can be performed on them. This design choice promotes information hiding, minimizing dependencies between different parts of the codebase. By encapsulating implementation details, Nulascript enhances code modularity, reduces complexity, and mitigates the risk of unintended interference between different components.

### 7.2 Safeguarding Implementation Details

Abstraction is a cornerstone of OOP, allowing developers to model real-world entities in a simplified and relevant manner. Nulascript leverages abstraction to distill complex systems into manageable and understandable components, facilitating a more intuitive programming experience.

Classes in Nulascript serve as abstract representations of entities, capturing their essential properties and behaviors. Through abstraction, developers can focus on high-level concepts without being burdened by the intricacies of implementation. This not only enhances code readability but also promotes a modular and scalable architecture, laying the groundwork for building intricate systems with ease.

### Power of Flexibility

Polymorphism, a key OOP concept, empowers Nulascript to exhibit flexibility and adaptability in its codebase. The language embraces both compile-time and runtime polymorphism, allowing developers to write code that can work with objects of various types.

In Nulascript, polymorphism manifests through method overloading, enabling multiple methods with the same name but different parameter lists, and method overriding, where a subclass provides a specific implementation of a method defined in its superclass. This flexibility enhances code reuse, promotes extensibility, and facilitates the creation of versatile and adaptable software systems.

### 7.4 Building Hierarchies of Reusability

Inheritance, another vital aspect of OOP, finds its place in Nulascript's architecture, enabling the creation of hierarchical relationships between classes. Through inheritance, a class (subclass) can inherit attributes and behaviors from another class (superclass), fostering the reuse of code and the establishment of a clear and intuitive class hierarchy.

Nulascript utilizes inheritance judiciously to structure code in a way that reflects the inherent relationships between different entities. This not only streamlines code organization but also simplifies maintenance and promotes a modular design that accommodates changes with minimal disruption.

### 7.5 Design Patterns

Nulascript's embrace of Object-Oriented Programming (OOP) extends to the integration of well-established design patterns, with a particular focus on Dependency Injections (9). This pattern, exemplified by mechanisms like the Singleton pattern for ensuring a single instance of a class and the Observer pattern for managing dependencies between objects, significantly contributes to the language's architectural robustness.

### Prospects

As Nulascript continues to evolve, the adherence to OOP paradigms positions the language to embrace future challenges and opportunities. The modular and extensible nature of the codebase, facilitated by encapsulation and abstraction, ensures that Nulascript can adapt to changing requirements with agility.

The ongoing commitment to OOP principles serves as a guiding compass for the language's evolution, enabling developers to build scalable and sophisticated applications with confidence. The synergy between Nulascript's design philosophy and OOP paradigms lays the groundwork for a language that not only meets the needs of the present but also anticipates the demands of the future.

### Conclusion: Nulascript's OOP implementation

In conclusion, Nulascript's utilization of Object-Oriented Programming paradigms is akin to orchestrating a symphony of encapsulation, abstraction, polymorphism, and inheritance. Each note played by these OOP principles harmonizes to create a language that is not only syntactically elegant but also architecturally sound. As Nulascript resonates with the spirit of OOP, it invites developers into a realm where code is not just written but composed, where software development becomes an art form guided by principles that stand the test of time.

## Nulascript’s Syntax: A fusion of C++, Python and JavaScript Syntax for Seamless Productivity

Nulascript, a language designed with a keen focus on developer productivity and cross-boundary applicability, emerges as a harmonious blend of syntax elements from C++, Python, and JavaScript. This unique fusion aims to provide developers with a language that not only feels familiar across different programming paradigms but also allows them to hit the ground running quickly. In this exploration, we delve into the distinctive syntactic features of Nulascript, drawing parallels to its influential predecessors and elucidating how it strives to create a seamless coding experience.

### C++ Influence

Nulascript's syntax draws significant inspiration from C++, a language celebrated for its robust performance and versatility. The incorporation of C++ syntax elements instills a sense of familiarity, especially among developers engaged in systems programming or handling performance-critical applications.

To illustrate, the utilization of an asterisk (\*) for dereferencing echoes the explicit typing characteristic of C++. This choice not only aligns with C++ conventions but also enhances the clarity of variable utilization. Similarly, the use of the ampersand (&) for referencing variables mirrors the well-established practice in C++, making it instantly recognizable and accessible to developers accustomed to working with pointers and references.

### Syntax Interchangeability

The overarching goal of Nulascript is to enable developers to cross boundaries effortlessly and start coding with minimal friction. The syntax choices, drawn from the rich tapestry of C++, Python, and JavaScript, contribute to a language that feels both versatile and approachable. Whether you're crafting performance-sensitive algorithms, scripting for automation, or building dynamic web applications, Nulascript endeavors to be a versatile tool in your coding arsenal.

The interchangeability of **not** and **!**, the dual syntax for equality checks, and the blend of explicit typing with Pythonic readability showcase Nulascript's commitment to syntax harmony. Developers can leverage their existing skills from multiple domains, fostering a sense of familiarity and reducing the learning curve associated with adopting a new language.

### Nulascript’s Syntactic Mixture

In conclusion, Nulascript emerges as a syntactic symphony, harmonizing the strengths of C++, Python, and JavaScript to create a language that transcends boundaries and caters to diverse programming needs. By encapsulating the best elements of each language, Nulascript empowers developers to navigate seamlessly across domains, bringing their expertise to the forefront without being encumbered by syntactic idiosyncrasies.

As the programming landscape continues to evolve, Nulascript stands as a testament to the idea that a well-crafted syntax can be a unifying force, fostering a community where developers from different backgrounds converge to create innovative and efficient solutions. Whether you're a systems programmer, a data scientist, or a web developer, Nulascript beckons you into a realm where syntax is not a barrier but a bridge to boundless creativity.

## Lexical Precedence

Precedence is crucial in parsing because it determines the order in which operators are applied in an expression. In a programming language, expressions often involve multiple operators with varying levels of precedence, such as multiplication, addition, and logical comparisons. The precedence rules dictate which operations take precedence over others when evaluating an expression.

For example, in the expression 2 + 3 \* 4, the multiplication has higher precedence than addition. Without considering precedence, the expression would be evaluated left to right, resulting in 20. However, with precedence considered, the multiplication is performed first, yielding the correct result of 14.

In the context of a parser, precedence guides the construction of the abstract syntax tree (AST). The AST represents the hierarchical structure of expressions, ensuring that the nodes in the tree reflect the correct order of operations. Parsing functions use precedence to determine how to group and associate operands and operators within the AST, capturing the intended semantics of the code.

By incorporating precedence rules, parsers can accurately reflect the programmer's intent and enable the subsequent stages of compilation or interpretation to execute the code correctly. Precedence is an essential aspect of parsing that contributes to the accurate representation of the syntactic and semantic structure of programming language constructs.

In the context of Nulascript expressions, the **Precedence** enum defines different levels of precedence for operators. The levels are arranged from lowest to highest weight, indicating the order in which operators are evaluated.

1. **LOWEST**: This is the lowest precedence level, meaning operators at this level are evaluated last. In expressions, operators with this precedence are typically the ones that bind the least tightly.
2. **EQUALS**: The **EQUALS** level is higher than **LOWEST** but lower than subsequent levels. It likely represents equality and inequality operators, such as **==** and **!=**.
3. **LESSGREATER**: This level deals with comparison operators like **<** and **>**. Operators at this level have higher precedence than equality operators but lower than arithmetic operators.
4. **SUM**: This level represents addition and subtraction operators. Operators at this level are evaluated before those at lower precedence levels.
5. **PRODUCT**: This level corresponds to multiplication, division, and other similar arithmetic operations. Operators at this level have higher precedence than addition and subtraction.
6. **PREFIX**: Prefix operators, such as unary negation or logical negation, fall into this category. They have a higher precedence than arithmetic operators.
7. **CALL**: The highest precedence level in this enum likely represents function calls or method invocations. Operators at this level are evaluated first, binding most tightly.

## Functions as First Class Citizens

Functions as first-class citizens empower developers with tools for creating modular, reusable, and expressive code. This paradigm aligns well with both functional and imperative programming styles, providing a flexible and powerful foundation for building sophisticated software systems.

### Higher-Order Functions

Functions as first-class citizens enable the creation of higher-order functions, which are functions that take other functions as parameters or return functions as results. This facilitates a more modular and reusable code structure.

### Abstraction and Modularity Without Supporting an OOP System

By treating functions as first-class citizens, developers can create more abstract and modular code. Functions can be designed to encapsulate specific behavior and then passed around or composed to create more complex functionality. This promotes code reuse and maintainability.

### Dynamic Behaviour

The ability to pass functions as arguments allows for dynamic behavior in a program. For instance, callbacks and event handlers can be specified at runtime, providing flexibility and adaptability in response to changing conditions.

### Closures

Closures, a powerful concept enabled by first-class functions, allow functions to "close over" variables from their lexical scope. This helps in creating more flexible and expressive code by capturing and maintaining state (10).

# **4. Implementation**

Nulascript is crafted in C++ and designed to seamlessly operate across diverse platforms, including Linux, Windows, and Unix, ensuring versatility for users across different environments once compiled with a native compiler. Remarkably, the language's development adheres to a principle of self-sufficiency, abstaining from external libraries to maintain a streamlined and self-contained structure. The implementation intricacies are orchestrated through a set of makefile "phonies," orchestrating the invocation of CMake builds. These builds culminate in the creation of binaries, executable by end-users, providing a straightforward and efficient means to interact with the language. The choice of C++ as the programming language stems from its renowned speed and its unique capability to access memory directly through pointers, contributing to the language's performance and memory management characteristics.

## Lexer’s Core

**public**:

Lexer(**const** std::string& input);

Token getNextToken();

Snippet 1: Lexer’s public members

In the code snippet, we have two public methods. The first is a constructor named lexer, which takes a single argument: a reference to a std::string containing Nulascript code. The purpose of this constructor is to initialize an instance of the Lexerclass with the provided input.

The second public method is getNextToken(), which is used to retrieve the next token from the Nulascript code. This method plays a crucial role in the lexical analysis process, where the code is systematically broken down into individual tokens for further processing.

Lexer holds the following private variables:

**private**:

TokenLookusp tokenLookup;

std::string input;

Snippet 2: TokenLookup’s private members

At a high level, the TokenLookup class is responsible for handling the lookup of reserved keywords. Here's a brief excerpt from the class's implementation:

TokenLookup::TokenLookup() {

keywords = {

{"fn", FUNC},

{"let", LET},

{"true", TRUE},

TokenType TokenLookup::lookupIdent(**const** std::string& ident)

Snippet 3: TokenLookup core initialization

This constructor initializes an unordered map named keywords, which contains pairs of reserved keywords and their corresponding token types. This map serves as a lookup table for identifying and categorizing keywords in the code.

These private variables are used for storing the state of the lexer:  
 **int** pos;

**int** readPos;

**char** ch;

Let’s use the following example code let a = 5; and walk through how these variables would be set initially when initializing the lexer:

* **pos** (position) will be initialized as 0**:** The lexer begins processing at the first character of the input string, which is 'l' in the keyword 'let'.
* **readPos** (reading position) will be initialized as 1**:** This indicates that the lexer is ready to read the next character after the current character pointed to by pos, which is 'e' in 'let'.
* **ch** (current character) will be initialized as 'l'**:** This variable stores the character currently pointed to by pos, which is 'l' in 'let'.

Before we explore the additional features of the Lexer, let's first examine where the tokens are defined. These tokens are encapsulated in a struct called "Token," which is defined as follows:

**struct** Token {

TokenType type;

std::string literal;

};

Snippet 4: Token struct

The "literal" field in the Token struct, as the name suggests, stores the literal value of the token. For instance, if the token is '=' (an assignment operator), the "literal" field would contain the string '='. The "TokenType" of the token is determined by an enum declared in the same file:

**enum** TokenType {

ASSIGN,

// Other token types...

};

Snippet 5: TokenType enum

Consequently, in the case of a token with the literal value '=', the Token struct will hold the TokenType ASSIGN to represent this specific token.

Back to the Lexer’s implementation, now we’re going to go over the method is the core of the tokenization.

Token Lexer::getNextToken() {

Token currentToken;

skipOverWhitespace();

**switch** (ch) {

**case** '=':

currentToken = newToken(TokenType::ASSIGN, ch);

**break**;

**case** '+':

currentToken = newToken(TokenType::PLUS, ch);

**break**;

Snippet 6: Lexer’s getNextToken() implementation

First, it calls skipOverWhitespace() to skip any whitespace characters.

Then, it enters a switch-case statement based on the current character to identify and assign the appropriate token type. Finally, it returns the extracted token.

## Handling Complex Tokens

The handleComparisonOperators serves to differentiate between two possibilities: whether the character is a simple '<' or '>', indicating less than or greater than, or whether it's part of an extended equality comparison (e.g., '<=' or '>=').

To optimize the code and minimize branching, it first checks if the next character, obtained using peekNextChar(), is an '=' sign. If it is, the function treats the current character as part of an extended comparison. It temporarily stores the current character, advances to the next character using readChar(), and constructs a token literal that combines the two characters (e.g., "<=" or ">="). Finally, it creates a new token with the extendedType and this combined literal.

If the next character is not '=', indicating a simple comparison, the function creates a new token with the shortType and the current character, which represents 'less than' or 'greater than'.

So, in essence, this function helps distinguish between basic comparison operators and their extended counterparts whenever '<' or '>' characters are encountered in the input.

In the getNextToken() function, the default case is reached when the current character **ch** does not match any of the specific characters (e.g., '=', '+', '-', etc.) that have been explicitly handled in the preceding switch cases. This code block serves as a catch-all for characters that are not part of those specific categories.

### 1.1.2 Identifier and Reserved Keywords Handling

The default case of the switch checks if the current character is a letter. If it is, this typically indicates the start of an identifier, which could be a variable name or a keyword. If it's a letter, the code reads an extended token using the readExtendedToken(TokenType::IDENT) function. This function reads characters until it encounters a non-letter, effectively extracting the entire identifier.

The extracted literal (the identifier or keyword) is stored in currentToken.literal, and the lexer uses the tokenLookup mechanism to determine the appropriate token type for this identifier. The identified token type is assigned to currentToken.type.

### 1.1.3 Integer Literal Handling

If the current character is a digit, it's treated as the start of an integer literal. The lexer sets the token type to TokenType::INT to indicate an integer.

It then uses the readExtendedToken() function with currentToken.type as the argument to read and assemble the complete integer literal. This function reads characters until it encounters a non-digit character, effectively forming the full integer literal.

The integer literal is stored in currentToken.literal. As of the moment Nulascript support only integers.

### 1.1.3 Illegal Character Handling

If the current character does not match either of the above conditions (not a letter or a digit), it's considered an illegal character.

In this case, the code creates a new token with a type of TokenType::ILLEGAL and assigns the current character as the token's literal value.

## Abstract Syntax Tree

In the context of programming language development and interpretation, an Abstract Syntax Tree (AST) is a crucial data structure used to represent the hierarchical structure of the source code. When discussing the implementation of an interpreter, particularly one using a recursive descent parser, the AST plays a central role in the overall design.

Here are some aspects to consider when talking about the implementation of the AST in the context of an interpreter:

Representation of Code Structure: The AST serves as an intermediate representation of the source code. It captures the syntactic structure of the code in a hierarchical manner, where nodes in the tree correspond to language constructs such as expressions and statements.

Nodes and Relationships: The AST is composed of nodes, each representing a specific syntactic construct in the language. Nodes are interconnected to reflect the relationships between different parts of the code. For example, a binary expression node might have left and right child nodes representing the operands.

Building the AST during Parsing: As the recursive descent parser processes the input code, it simultaneously constructs the AST. Parsing functions are responsible not only for recognizing the grammar rules but also for creating AST nodes and organizing them into a tree structure.

Traversal and Evaluation: Once the AST is constructed, the interpreter typically traverses the tree to evaluate the code. Different types of nodes may have different evaluation strategies. For example, an expression node might be evaluated by recursively evaluating its subexpressions and applying the corresponding operation.

Abstracting Syntax: The term "Abstract Syntax Tree" emphasizes the abstraction of syntactic details. The AST captures the essential structure and meaning of the code while abstracting away specific lexical details. This abstraction facilitates further processing and analysis.

Support for Control Flow and Declarations: The AST accommodates various language features, including control flow constructs (e.g., if statements, loops) and variable declarations. Each type of statement or expression in the language corresponds to a distinct node in the AST.

## Abstract Syntax Tree Implementation

**class** Node

**class** Statement : **public** Node

**class** Expression : **public** Node

This code defines a basic hierarchy of classes for building an abstract syntax tree (AST) for a programming language. An AST is a hierarchical structure that represents the syntactic structure of code, making it easier to analyze and manipulate. Here's an explanation of each class:

**Node** is an abstract base class that serves as the root of the AST hierarchy. It contains two pure virtual functions, which means any class derived from Node must implement these functions.

**tokenLiteral():** This method is responsible for returning a string representing the token or value associated with a node. In other words, it provides the literal representation of the token that the node corresponds to.

**toString():** This method should return a string representation of the entire node. It is used to generate a string representation of the node and its sub-nodes for debugging and other purposes.

**Statement** is a derived class of Node. It represents statements in the programming language. A statement is a unit of code that performs some action or operation, like variable declarations, loops, or conditional statements.

While Statement inherits the tokenLiteral() and toString() functions from the Node class, it doesn't introduce any additional behavior beyond marking a class as a statement. Actually, the Statement class is always being used as a base class and objects are never initialized directly.

**Expression** is another derived class of Node. It represents expressions within the programming language. Expressions are code constructs that produce values, like arithmetic expressions or function calls.

Like Statement, Expression inherits the tokenLiteral() and toString() functions from the Node class without adding specific behavior. Either the Statement or Expression can be excluded, and we could utilize the Node as the foundational class for all high-level Statements and Expressions. However, the current implementation prevents the need for any unnecessary dynamic casting.

In summary, this code defines a framework for creating an abstract syntax tree. The Node class defines two essential functions for all nodes in the tree, and then two specialized classes, Statement and Expression, derive from it. While these derived classes don't introduce any new functions, they are typically used as a foundation for creating more specific node types that correspond to the various constructs of a programming language. These classes allow the representation and manipulation of the syntactic structure of code during parsing and analysis.

**class** Program : **public** Node

The Program class is designed to represent a program or script in the abstract syntax tree of the programming language. It inherits the basic structure and functionality of the Node class, allowing it to be part of the hierarchical tree structure. It stores a sequence of statements in the statements member and provides functions to retrieve the token literal and generate a string representation of the program, making it a key component for representing the high-level structure of code in the AST.

**class** Identifier : **public** Expression

This code defines a class named Identifier that inherits from the Expression class, which is part of an abstract syntax tree. It represents an expression in the abstract syntax tree corresponding to an identifier or variable name in the source code. It inherits the basic structure and functionality of the Expression class and contains the token and value members to store information about the identifier. The class provides functions to retrieve the token literal (typically the identifier's name) and generate a string representation of the identifier, making it a key component for representing variables and identifiers in the AST.

**class** LetStatement : **public** Statement

The LetStatement class represents a 'let' statement in the abstract syntax tree. It inherits the basic structure and functionality of the Statement class and contains data members to store information about the 'let' statement, including the associated Token, the variable name, and the assignment expression. The class provides functions to retrieve the token literal and generate a string representation of the 'let' statement.

**class** ReturnStatement : **public** Statement

The ReturnStatement class represents a 'return' statement in the abstract syntax tree. It inherits the basic structure and functionality of the Statement class and contains data members to store information about the 'return' statement, including the associated Token and the return value expression. The class provides constructors for both 'return' statements with and without return values and functions to retrieve the token literal and generate a string representation of the 'return' statement.

**class** ExpressionStatement : **public** Statement

The ExpressionStatement class represents a statement. It inherits the basic structure and functionality of the Statement class and contains data members to store information about the statement, including the associated Token and the expression it contains. The class provides a constructor for initializing the statement, and functions to retrieve the token literal and generate a string representation of the statement.

An expression statement refers to an expression employed in a context that anticipates a statement. In this scenario, the expression is assessed, and its outcome is disregarded. Consequently, it is meaningful exclusively for expressions that produce side effects, like performing a function or modifying a variable. For example, function calls, such as console.log("Hello"); and [1, 2, 3].forEach((i) => console.log(i));, are suitable instances of expression statements, as they trigger actions like printing to the console or iterating through an array with side effects.

**class** Conditional : **public** Expression

The Conditional class serves as the embodiment of conditional expressions or logic within the abstract syntax tree (AST) of the programming language. By encapsulating conditions, block statements for both true and false outcomes, and associated tokens, it plays a vital role in facilitating a structured representation of conditional logic in the AST. As of now, its current implementation support only else blocks, but potentially the class can be extended to store vector storing “else if” nodes.

**class** Function : **public** Expression

Stores function arguments and the code block parsed during function invocation.

**class** Invocation : **public** Expression

Records function invocations, parsing and storing argument values. This enables the use of actual values when invoking the function's body, replacing variables with their values.

**class** String : **public** Expression

Simple storage for string values.

**class** Integer : **public** Expression

Similar to the String class, but stores integer values instead of string values.

**class** Identifier : **public** Expression

Manages the identifier and the right-hand side expression, assigning the expression's value to the identifier during parsing.

**class** Identifier : **public** Expression

Stores the identifier pointing to the referenced value.

**class** Pointer : **public** Expression

Similar to the Reference class, storing identifiers that need to be dereferenced.

**class** Comment : **public** Expression

Stores only the token, aiding the parser in determining when to skip a line.

**class** BlockStatement : **public** Statement

Stores all the statements within a code block so that they can be later parsed. Imagine an else block of a for loop - the content of the else block is stored as a pointer to an object of class BlockStatement.

Before stepping into explaining how the ForLoop class is defined within the syntax tree, there are two important classes we haven’t mentioned yet.

**class** Infix : **public** Expression

**class** Prefix : **public** Expression

These two classes delineate nodes within the code corresponding to prefix and infix operators. In the context of expressions, infix operators are positioned between their operands, while prefix operators precede their operands.

For instance, consider the following example of a prefix operator:

\*foobar; # Here, we dereference the reference foobar.

And an example of an infix operator:

10 + 10; # Adding two numbers.

In the prefix node class, a single pointer is employed to store the expression, positioned on its right side. An illustration of this is found in expressions like **-5**, where the '-' is the prefix operator and **5** is the integer expression. Conversely, the infix class employs two pointers, one for the right expression and another for the left expression. An example of this is seen in the expression **20 / 20**, where **20** on the left is the left integer expression, '/' is the operator, and the other **20** on the right is the right integer expression.

**class** ForLoop : **public** Expression

The interesting thing for this particular class is the way we actually store the for-loop’s initialization header. In Nulascript you define a loop the following way:

**for** (**def** a = 5; a < 10; a + 2)

While in other languages this might be written as follows:

**for** (**let** a = 5; a < 10; a = a + 2)

In Nulascript, there's no requirement for us to explicitly specify adding a number to a defined variable through reassignment, as the entire expression is stored as an infix expression rather than an assignment expression. The conditional expression, much like comparing two values in other programming languages, is also treated as an infix operation and expression. A distinctive feature in Nulascript is that for loops always necessitate the initialization of a new variable, exclusively used within the loop and subsequently discarded. Therefore, a for loop in Nulascript always expects a LetStatement in its header.

## Parser Implementation

We have now reached the part where we begin exploring how statements are parsed in the Parser class. The parseStatement() function is a crucial component of this process. Let's explain the code:

Statement\* Parser::parseStatement() {

**switch** (currentToken.type) {

**case** TokenType::LET:

**return** parseLetStatement();

**case** TokenType::RETURN:

**return** parseReturnStatement();

**default:**

**return** parseExpressionStatement();

Snippet 7: Parsing statements

This method is responsible for parsing statements in the source code. It determines the type of statement based on the current token (the token being processed) and then delegates the parsing of that specific type of statement to dedicated methods.

### 3.1 Parsing Initialization Statements

The parsing of a **LetStatement** begins by initializing a **LetStatement** object with the current token. The parser then checks if the next token is an identifier, representing the variable name in the "let" statement. If an identifier is not found, the function returns **nullptr** to indicate a parsing error. If the identifier is successfully identified, an **Identifier** object is created and assigned to the **name** attribute of the **LetStatement**. The parser then checks for the presence of an assignment operator ("="). If it is missing, the function returns **nullptr**. Upon confirming the assignment operator, the parser advances to the next token, which in this case is the semicolon. The code proceeds to parse the expression on the right side of the assignment using the **parseExpression** function, and the resulting expression is assigned to the **value** attribute of the **LetStatement**. Finally, the code checks for the presence of a semicolon at the end of the statement, moving to the next token if found. The fully parsed **LetStatement** object is then returned.

### 3.2 Parsing Return Statements

This code is responsible for handling the parsing of "return" statements in a programming language. The method starts by creating a **ReturnStatement** object and initializing it with the current token. Subsequently, it advances to the next token using the **getNextToken** function. The code then parses the expression following the "return" keyword using the **parseExpression** function, and the resulting expression is assigned to the **returnValue** attribute of the **ReturnStatement**. Following this, it checks if the next token is a semicolon. If a semicolon is present, it moves the cursor to semicolon. Finally, the function returns the fully parsed **ReturnStatement** object, encapsulating the information about the "return" statement and the associated expression.

### 3.3 Parsing Expression Statements

The **parseExpressionStatement** function begins by creating an **ExpressionStatement** object and initializing it with the current token. The parser then proceeds to parse the expression using the **parseExpression** function, and the resulting expression is assigned to the **expression** attribute of the **ExpressionStatement**. After parsing the expression, it checks if the next token is a semicolon, which typically signifies the end of a statement in many programming languages. If a semicolon is found, the parser advances to the next token. The function then returns the fully parsed **ExpressionStatement** object, encapsulating the information about the expression to be executed as a statement.

The **parseExpression** method serves as a fundamental and shared functionality among the three methods mentioned. It plays a crucial role in recursively evaluating expressions, constituting the core logic for handling the parsing of expressions within the parser.

It starts by looking up the parsing method for the current token type in the **prefixParsingFunctions** map. If no function is found, it generates an error message, appends it to an error list, and returns **nullptr** (indicating a parsing failure).

If a parsing function is found, it calls it to parse the left expression. The result is stored in the **leftExpression** (left node expression) svariable.

The function then enters a loop that continues as long as the current token type is not **SEMICOLON**. Inside the loop, it looks up the parsing function for the next token's type in the **infixParsingFunctions** map. If no function is found, it returns the **left node expression**, indicating that the parsing is complete.

If an infix parsing function is found, it advances to the next token and calls the infix parsing function with the **left node expression** as an argument. The result is assigned back to the **leftExpression** variable.

The loop continues until it either reaches the end of the expression or the precedence of the current token is lower than the specified precedence level.

Finally, the function returns the parsed expression.

In summary, this code implements a recursive descent parser for expressions, handling both prefix and infix operators based on their precedence levels. The parsing process is driven by the precedence of operators and involves calling the appropriate parsing functions for prefix and infix operators. The result is a parsed expression tree.

### 3.4 Debugging Parsing of an Infix Expression

Let’s debug an infix expression (5 + 10) step by step:

The current token is of type Integer. We find the following parsing function: **parseInteger()**

This is the function which gets invoked to parse our current token to an expression.

Now, after we’ve parsed our integer expression and assigned it to a variable, we have to check if the next token defines an infix operation.

Within its execution, the method attempts to locate infix parsing functions for the upcoming token. Should it successfully find such a function, it invokes it, supplying the expression yielded by a preceding parsed prefix expression as an input. This iterative process repeats until the parser encounters a token with a greater precedence, signaling the conclusion of the infix expression parsing.

In this case, we call the found **parseInfix** function which is registered for the ‘+’ token and provide the parsed **Integer(Token(5))** expression to the function as an argument.

Then the result is reassigned to the initially created variable and returned. The returned value is an instance of the Infix class which holds to pointers to its left and right expression.

We can visualize it as follows:

**class** Infix : **public** Expression {

Token token = Token('+');

Expression\* left = Integer(5);

Expression\* right = Integer(10);

std::string op = '+';

Snippet 8: Infix initialization logical visualization

### 3.5 Parsing References

The parseReference function is responsible for parsing references.

It begins by creating a new Reference object named ref and initializes it with the current token. The Reference class represents references to identifiers in the language. The function checks if the next expected token is an identifier (IDENT) using the peekAndLoadExpectedToken function. Assuming the next token is an identifier, the function assigns the value of the literal of the current token (presumably the identifier's name) to the referencedIdentifier field of the ref object. It then advances to the next token using getNextToken() to prepare for subsequent parsing. Finally, the function returns a pointer to the fully parsed Reference object, encapsulating information about the parsed identifier reference.

In essence, references in Nulascript operate as powerful aliases, providing an alternate means to refer to identifiers. They serve as dynamic labels that seamlessly point to the same underlying data, enabling developers to manipulate and access variables using different names interchangeably.

## Evaluation

### 4.1 Storage and Representation of Values

In the dynamic landscape of programming languages, the smooth execution and interpretation of code rely heavily on advanced evaluators that boast resilient components dedicated to the management and storage of results. Central to this complex operation is the essential collaboration between a class responsible for representing data and guiding its interpretation by the REPL during the print step. These components collectively constitute the foundation of evaluators, offering a well-organized structure to handle a variety of data types, oversee variable scopes, and streamline the interpretation process of code.

In the context of an evaluator, Nulascript’s class which represents data is called a **Storage** class and the accompanying **Environment** class serve as fundamental components for managing and storing the results of evaluations. Here are several reasons why a **Storage** class is essential:

#### 4.1.1 Result Storage

* + The **Storage** class provides a standardized way to store the results of evaluations. Each subclass of **Storage** corresponds to a specific type of result, such as integers, booleans, functions, errors, and more.
  + The use of a common interface ensures that the evaluator can uniformly handle and manage diverse types of results.

#### 4.1.2 Abstraction of Data Types

* + Different types of data (e.g., integers, booleans, functions) can be abstracted into instances of the **Storage** class. This abstraction allows the evaluator to work with generic storage objects without directly dealing with the specifics of each data type.
  + It promotes a high level of abstraction, making the evaluator more flexible and extensible.

#### 4.1.3 Uniform Evaluation Interface

* + All subclasses of **Storage** share a common interface, including the **evaluate** method. This uniform interface allows the evaluator to interact with the stored results consistently, regardless of the underlying data type.
  + The **evaluate** method can be called on any **Storage** object to obtain a string representation of its value.

#### 4.1.4 Environment Management

* + The **Environment** class, which is a part of the storage system, facilitates the creation and management of environments. Environments are used to store and retrieve variables during evaluation.
  + The **Environment** class includes methods for setting, getting, and removing variables, contributing to the dynamic scoping and management of variable bindings.

#### 4.1.5 Nested Environments for Scoping

* + The **Environment** class supports nested environments, allowing for the creation of scopes. This is crucial for managing variable bindings and scoping rules during the evaluation of code.
  + Nested environments enable the implementation of lexical scoping, where variables are resolved based on their lexical context.

#### 4.1.6 Error Handling

* + The **Storage** class includes an **ErrorStorage** subclass, dedicated to representing errors during evaluation. This supports proper error handling mechanisms in the scripting language, providing meaningful error messages.

#### 4.1.7 User-Defined Functions

* + The **FunctionStorage** class is designed to represent user-defined functions. This allows the evaluator to store function definitions, including the function's arguments, code block, and associated environment.
  + The **evaluate** method for **FunctionStorage** can be implemented to handle function calls, supporting the execution of user-defined functions.

#### 4.1.8 Standard Library Functions

* + The **StandardFunction** class represents functions from a standard library. These functions can be used as built-in operations in the scripting language.
  + The **evaluate** method for **StandardFunction** executes the function's logic, allowing the evaluator to seamlessly integrate standard library functionality.

#### 4.1.9 Reference Handling

* + The **ReferenceStorage** class is designed to handle references to variables. This is crucial for managing variable assignments, especially in the context of nested environments.
  + The **evaluate** method for **ReferenceStorage** can retrieve the value of the referenced variable from the associated environment.

#### 4.1.9 Reference Handling

* + The design of the **Storage** class and its subclasses promotes extensibility. New types of data or storage classes can be added by introducing additional subclasses.
  + The modular structure allows developers to extend the capabilities of the evaluator by introducing new storage types without modifying the core evaluator logic.

In summary, the **Storage** class, in conjunction with the **Environment** class, provides a robust infrastructure for storing, managing, and interacting with the results of evaluations in a scripting language. It abstracts different data types, facilitates a unified interface, supports scoping and variable management, and enables seamless integration of user-defined and standard library functions. This design enhances the flexibility, modularity, and extensibility of the evaluator, making it a crucial component in the implementation of a scripting language interpreter.

### Evaluation Function

The evaluate function in Nulascript serves as a versatile interpreter, responsible for evaluating different types of nodes in the abstract syntax tree (AST). Its purpose is to execute and interpret various language constructs, such as expressions, statements, loops, and functions, by dispatching the evaluation to specific handler functions based on the type of the provided AST node.

The function checks the type of the input node and then delegates the evaluation to the appropriate handler function. For example, when encountering a node representing an integer or a boolean, it creates corresponding storage objects (IntegerStorage or BooleanStorage). Similarly, for prefix and infix expressions, it evaluates the expressions and performs the specified operations. The function also handles blocks of statements, conditionals, return statements, variable assignments, and references, among other language constructs.

Notably, the evaluate function plays a crucial role in managing environments, ensuring correct variable scoping and resolution. It interacts with the Environment class to retrieve, set, or modify variables as needed during the evaluation process.

Several key functionalities include handling functions and their invocations. When evaluating a function node, the function checks if the function body is empty, returning an error if so. For function invocations, it evaluates the function, processes the arguments, and then invokes the function with the provided arguments.

In the context of variables, the function appropriately manages assignments, references, and pointers. It handles variable assignments by evaluating the assigned expression and updating the environment with the new value. For references, it creates a ReferenceStorage object representing the referenced variable. Pointers are dereferenced to obtain the value they point to.

Furthermore, the function accommodates loop constructs, specifically the ForLoop node, by invoking the runForLoop function to execute the loop's block of code.

The evaluate function is designed to be comprehensive, covering a wide range of language features. It addresses potential error scenarios, such as encountering an unknown node type, and returns an error storage object in such cases.

In summary, the evaluate function is the pivotal component of the Nulascript interpreter, providing a versatile and extensible mechanism for the interpretation of diverse language constructs within the AST. It ensures proper scoping, handles variable assignments, manages control flow, and facilitates the execution of user-defined functions and loops.

# **5. Testing:**

Nulascript's commitment to software quality is exemplified by its meticulous testing strategy, which includes leveraging GitHub Actions for continuous integration and incorporating Google's C++ Testing Library to enhance the rigor of its unit testing processes. This comprehensive approach underscores Nulascript's dedication to creating a reliable and robust scripting language. In this extended discussion, we delve into the significance of automated testing, the role of GitHub Actions, the integration of Google's C++ Testing Library, and the broader implications for Nulascript's development lifecycle.

## Automated Testing

Automated testing has become a cornerstone of modern software development, offering an efficient means to validate code changes and ensure the stability of a project. In the case of Nulascript, the integration of automated testing practices is particularly crucial, given the dynamic and evolving nature of scripting languages. Unit tests, which target specific units of code, provide an effective mechanism to verify the correctness of individual components.

### 1.1 Unit Tests

By embracing automated unit testing, Nulascript can systematically evaluate the behavior of language constructs, parsing mechanisms, and execution procedures, ensuring that each unit performs as intended in isolation.

GitHub Actions plays a central role in Nulascript's automated testing workflow. GitHub Actions is a versatile CI/CD (Continuous Integration/Continuous Deployment) solution that seamlessly integrates with GitHub repositories, allowing developers to automate tasks such as building, testing, and deploying code. For Nulascript, GitHub Actions triggers the CI pipeline upon every git push, initiating a series of predefined workflows that orchestrate the execution of unit tests. This immediate feedback loop is invaluable, providing developers with prompt insights into the health of the codebase and helping them identify potential issues early in the development cycle.

The GitHub Actions workflow for Nulascript involves a Linux virtual machine hosted in the cloud, creating an isolated environment for testing. This environment ensures consistency and reproducibility, critical factors in reliable testing. GitHub Actions fetches the latest changes from the repository, builds the code, and then executes the suite of unit tests.

One notable aspect of Nulascript's testing strategy is the incorporation of Google's C++ Testing Library. The decision to use a well-established and widely adopted testing library underscores Nulascript's commitment to industry best practices. Google's C++ Testing Library provides a robust and extensible framework for writing unit tests in C++, offering a rich set of assertion macros and advanced testing features. This library aligns with Nulascript's C++ underpinnings, enabling seamless integration with the existing codebase and facilitating the creation of expressive and thorough unit tests.

Furthermore, Google's C++ Testing Library to Nulascript's testing suite brings several advantages. First and foremost, the library's rich set of assertion macros allows developers to express test conditions succinctly and clearly. This contributes to the readability of the tests, making it easier for developers to understand the expected behavior and logic being tested. Additionally, the advanced features of the library, such as parameterized tests and test fixtures, provide flexibility in designing comprehensive test suites that cover various scenarios and edge cases.

In conclusion, Nulascript's commitment to software quality is exemplified by its meticulous unit testing strategy, which incorporates Google's C++ Testing Library. The synergy between automated testing using GitHub Actions and the powerful capabilities of the testing library contributes to a robust and reliable development lifecycle. Nulascript's testing infrastructure not only identifies potential issues early but also fosters a collaborative and efficient development environment. As the language continues to evolve, this testing strategy serves as a foundation for maintaining code integrity, fostering innovation, and enhancing the overall reliability of Nulascript.

### 1.2 Functional Tests (End-to-End testing)

Nulascript embraces a comprehensive testing strategy that extends beyond unit testing, incorporating functional tests to validate the holistic functionality of its interpreter. These functional tests, implemented in Python using the built-in `subprocess` module and the `unittest` framework, assess the interpreter's behavior when executing actual script files. This multifaceted approach to testing, coupled with the seamless integration of GitHub Actions into the development workflow, showcases Nulascript's commitment to delivering a reliable and robust scripting language.

The functional testing framework in Nulascript revolves around a Python script named `interpreter.py`. This script serves a dual purpose: it functions as both a testing tool and a utility for executing Nulascript scripts from the command line. The key functionality lies in the `run\_interpreter` function within this script, which takes a script filename as input, executes the Nulascript interpreter on that script using a subprocess, and captures the output. This not only enables automated testing but also provides a convenient means of running scripts during development and debugging.

The functional tests are organized using Python's `unittest` framework, creating a structured and modular testing environment. The `TestInterpreter` class contains individual test methods, each corresponding to a specific Nulascript script located in the `examples` directory. For instance, there are tests for scripts involving loops, closures, logging, and conditionals. This approach ensures that a diverse set of language features is covered, allowing for a thorough evaluation of the interpreter's capabilities.

What sets functional testing apart is its focus on end-to-end testing, evaluating the interpreter's behavior when executing complete scripts. Unlike unit tests that isolate specific units of code, functional tests provide a more realistic assessment of the interpreter's performance in real-world scenarios. The expected output for each script is predefined, and the tests verify whether the actual output matches these expectations. This ensures that Nulascript not only correctly parses and evaluates individual language constructs but also functions as expected when executing complete and complex scripts.

A critical aspect of the functional testing strategy is the integration with GitHub Actions. Upon each git push, the GitHub Actions pipeline is triggered, initiating a series of predefined workflows. In this context, the CI pipeline not only builds the interpreter itself but also runs the suite of functional tests. This automation ensures a consistent and reproducible testing environment, allowing developers to receive immediate feedback on the impact of their code changes.

In conclusion, Nulascript's testing strategy is a holistic approach that combines unit testing and functional testing to ensure the correctness and reliability of its interpreter. The use of Python, GitHub Actions, and Google's C++ Testing Library underscores a commitment to industry best practices. By incorporating functional tests, Nulascript not only evaluates individual language constructs but also assesses the interpreter's behavior when executing complete scripts. This comprehensive testing framework, seamlessly integrated into the development workflow, positions Nulascript as a scripting language that prioritizes code quality, reliability, and continuous improvement.

## Regressions

The process of catching regressions, especially in a scripting language like Nulascript, is a crucial aspect of maintaining code reliability and preventing unintended side effects. Regressions can manifest in various forms, from unexpected behavior to performance issues, and addressing them promptly is essential for the overall health of the codebase.

One notable instance where regressions were identified in Nulascript involved the handling of uninitialized pointers in class objects. The code, originally developed on a Unix machine, assumed that uninitialized pointers in class objects would be falsy values, aligning with the behavior observed in that environment. However, when building binaries on Linux machines, an unexpected deviation occurred—uninitialized pointers were considered truthy values. This discrepancy in pointer evaluation led to failures in the code execution, as attempts to access uninitialized memory slots were resulting in unexpected behavior.

This regression scenario underscores the importance of cross-platform testing and highlights the challenges that can arise when code is developed and tested in one environment but deployed and executed in another. In this case, the realization that the code behaved differently on Linux machines compared to Unix machines was a crucial revelation. The ability to catch such platform-specific regressions early in the development cycle is a testament to the effectiveness of Nulascript's testing infrastructure, including both unit and functional tests.

The detection of this regression not only facilitated the correction of the specific issue but also prompted a deeper evaluation of the code's platform independence. I was able to address the inconsistencies in pointer evaluation, ensuring that the code behaved consistently across different platforms. This corrective action not only resolved the immediate regression but also fortified the codebase, making it more resilient and adaptable to diverse execution environments.

Another regression that was identified and addressed in Nulascript pertained to the precedence of operations. In this case, a seemingly negligible change in the codebase unexpectedly impacted the behavior of precedence, leading to failures in the expected order of operations. The significance of catching this regression lies in the fact that it was not an explicit modification to the precedence logic but a side effect of a seemingly unrelated alteration.

Regression testing in Nulascript played a pivotal role in exposing this subtle yet impactful change. The ability to identify regressions stemming from seemingly unrelated modifications is a testament to the thoroughness of the testing suite. It goes beyond verifying individual language constructs and assesses the holistic behavior of the interpreter, ensuring that modifications do not inadvertently introduce subtle bugs or alter the expected semantics of the language.

The swift detection and resolution of these regressions showcase the effectiveness of Nulascript's testing infrastructure, leveraging both unit and functional tests. By adopting a proactive approach to testing, Nulascript maintains a high level of code quality and reduces the likelihood of bugs making their way into the production environment. The combination of automated testing, continuous integration through tools like GitHub Actions, and a robust testing framework enables developers to catch regressions early in the development process.

Furthermore, the iterative nature of regression testing contributes to the evolution of Nulascript. As new features are introduced, and the codebase undergoes changes, the testing suite plays a critical role in ensuring that existing functionality remains intact. This iterative testing process fosters a development environment where potential developers can confidently introduce enhancements and modifications, knowing that the testing infrastructure will promptly identify and flag any regressions.

In summary, the instances where regressions were caught in Nulascript, particularly in the evaluation of uninitialized pointers and the precedence of operations, underscore the significance of rigorous testing practices. These regressions were not only promptly identified but were also used as opportunities to enhance the codebase, improving its platform independence and preserving the expected behavior of language constructs. The investment in comprehensive testing has not only prevented the introduction of subtle bugs but has also contributed to a more resilient, reliable, and adaptable scripting language. The lessons learned from these regression cases further reinforce the importance of a robust testing strategy in the ongoing development and maintenance of complex software systems.

## Tested Functional Requirements

### 3.1 Dynamic Typing System

In implementing the Dynamic Typing System for Nulascript, a notable achievement has been realized by allowing variables to be dynamically typed while minimizing the associated performance implications. The language successfully supports dynamic typing, enabling variables to be assigned without requiring explicit type declarations. This flexibility is particularly valuable for developers who can adapt their code dynamically during runtime, enhancing the language's adaptability. Despite the inherent challenges related to dynamic typing, a careful balancing act has been performed to mitigate performance concerns, ensuring that the runtime type determination is efficient. Performance constraints have been considered and addressed to a significant extent, resulting in a dynamic typing system that operates on par with and, in certain cases, even surpasses the speed of widely used dynamically typed languages like JavaScript. While acknowledging that it may not match the speed of statically typed languages, the achieved performance is notably less sluggish, offering a favorable compromise between dynamic typing flexibility and runtime efficiency.

### 3.2 Referencing & Dereferencing

The implementation of Referencing and Dereferencing in Nulascript has been successfully achieved, introducing references as a feature to create aliases for variables. However, the performance of referencing and dereferencing operations has not reached the anticipated speed due to the necessity for constant dynamic casting during runtime. The language has been designed to support dereferencing of references, but the process involves more dynamic casting than initially expected.

Notably, references have been implemented in a way that they point by identifier key rather than memory address. This design choice ensures that dangling storage objects are promptly collected when the main variable no longer points to them, preventing segmentation faults. While segmentation faults are currently not possible, it's essential to exercise caution when implementing deleting functionality in the future. The potential constraint of references not being garbage collected when referenced variables are deleted might resurface, prompting careful consideration and further refinement to maintain the language's robustness and reliability.

### 3.3 Functions & Closures

In the realm of Nulascript, the support for functions and closures has designated them solely as first-class citizen members rather than allowing them to be defined as standalones. While this architectural decision may initially seem unconventional, it has proven not to hinder their utility.

During the rigorous testing phase, where comparisons between JavaScript and Nulascript were conducted, thousands of closures were created and invoked to assess the language's performance and efficiency in handling such dynamic constructs. Though functions and closures are not defined as standalone entities, this design choice does not hinder their utility. In fact, it opens the door to a dynamic and versatile approach to programming. Leveraging functions and closures as variables means they can be assigned, passed as arguments, and returned from other functions, enhancing the expressive power of Nulascript. This paradigm aligns with the principles of functional programming, empowering developers to write concise and expressive code.

### 3.3 Syntax

A black rectangular object with white text

Description automatically generated

Figure 4: Execution of Nulascript code

In delving into the syntax realm of Nulascript, a notable innovation has been introduced to enhance readability and ease of use. Specifically, the syntax now allows the interchangeability of '==' for 'is' and '!=' for 'is not,' offering developers a familiar and versatile choice in expressing equality and inequality conditions. This adaptation aligns with the principles of language ergonomics, enabling developers to utilize syntax that resonates with their coding preferences and experiences.

It's essential to note, however, that the current implementation of Nulascript diverges from the traditional Pythonic syntax, particularly in the handling of code blocks. Unlike Python, which relies on indentation to define code blocks, Nulascript has opted for a structure where code blocks are not dependent on indentation. This decision provides a departure from Python's whitespace-sensitive approach, opting instead to align more closely with the syntax conventions of languages like C++ and JavaScript. While this shift may represent a deviation from one paradigm, it concurrently reinforces a commitment to coherence and consistency, especially for developers accustomed to the syntax styles prevalent in widely adopted programming languages

### 3.4 Control Flow

A distinctive approach to loops has been implemented, introducing a deviation from common syntax practices. One notable departure lies in the elimination of the need for reassignment within loops—a departure that challenges conventional programming paradigms. This unique twist on loop syntax reflects a personal interpretation, suggesting that this form of loop construction should be more widespread in programming languages. It represents a departure from traditional syntax but serves as a deliberate innovation to streamline code and enhance readability.

In testing the for loop, it demonstrated remarkable versatility and speed, proving to be on par with JavaScript and, in certain scenarios, even faster. However, it's essential to note that the current implementation lacks key functionalities, such as breaking out of the loop or skipping iterations using the **continue** keyword.

## Tested Non-functional requirements

### 4.1 Maintainability

In crafting Nulascript, I've been hands-on in testing and refining its modularity and maintainability. Once the evaluation functionality was in place, I put the language through its paces, pushing the boundaries to ensure that adding new functionalities remained straightforward and efficient.

Testing the modularity was a practical process. I systematically introduced new features, gauging how seamlessly they integrated with the existing codebase. The design choices made in Nulascript, especially in terms of Object-Oriented Programming (OOP) principles, proved their worth during these tests. I found it easy to extend and modify the codebase, a testament to the language's commitment to modularity.

With version control in play, I rolled back to previous versions, tweaking and refining the code. This hands-on approach allowed me to witness firsthand how easy it is to maintain and enhance Nulascript. The language's version control history acted as a valuable tool, providing a clear roadmap for improvements and optimizations.

One significant outcome of these tests was the confirmation that the for loop, despite its expedited introduction, didn't compromise the language's core principles. It's an extension that fits into the existing OOP design seamlessly. The commitment to maintaining clarity and simplicity in the codebase remained intact, allowing for the quick addition of new features without jeopardizing the language's foundational principles.

Overall, this testing phase underscored Nulascript's adaptability and ease of evolution. It's a language that not only upholds its design principles but also facilitates ongoing development and enhancement. As the architect, this firsthand experience has reinforced my confidence in Nulascript's capacity to be both dynamic and developer-friendly.

### 4.2 Portability

In thoroughly examining the compiled CMake binaries of Nulascript across a variety of machines at my disposal, the observed behavior revealed a striking consistency in terms of execution speeds. This extensive testing aimed to assess the language's performance across diverse hardware configurations and environments. I systematically ran the compiled binaries on various machines, ranging from different processors to varying memory capacities, and consistently found that Nulascript exhibited a stable and predictable execution speed.

This consistent behavior across multiple machines not only attests to the language's robustness but also underscores its portability and reliability. The fact that Nulascript delivers similar performance outcomes regardless of the underlying hardware variations indicates a level of optimization and efficiency in its implementation. This is particularly crucial for developers who may deploy Nulascript on a range of devices, ensuring that the language's performance remains reliable and predictable in diverse computing environments. Overall, this comprehensive testing of the compiled binaries serves as a testament to Nulascript's commitment to delivering consistent and dependable execution speeds, fostering a reliable foundation for developers across different platforms.

# **6. Results and Conclusion:**

## Performance – Nulascript vs. Javascript (Node v18)

In the domain of printing variables, Nulascript stands out for its commendable performance, maintaining remarkable efficiency with an impressively low average execution time of 0.01 seconds over 1000 print operations. This consistent and swift performance underscores the language's adeptness in handling repetitive tasks, emphasizing its proficiency in operations involving the output of variables. The negligible average time per print operation reinforces Nulascript's capability to swiftly and reliably execute tasks that involve the repeated display of variable values.

JavaScript, in comparison, demonstrates a slightly slower average execution time of 0.05 seconds over 1000 print operations. This variance in performance may be attributed to the interpretive nature of JavaScript, which can introduce additional overhead in handling printing operations. The contrast in execution times between Nulascript and JavaScript underscores the efficiency of Nulascript, particularly in scenarios where rapid and predictable variable output is a critical factor.

In creating closures and performing arithmetic operations, Nulascript exhibits noteworthy efficiency, maintaining an average execution time of 0.045 seconds over 1000 iterations. This performance highlights the language's capability to handle the creation of closures and associated arithmetic operations with commendable speed and consistency. Despite the absence of explicit optimizations, Nulascript's ability to maintain a consistently low average time per invocation suggests an intrinsic efficiency in managing complex language constructs.

JavaScript, in the same context, shows a slightly slower average execution time of 0.055 seconds over 1000 iterations. The dynamic nature of JavaScript and potential overhead in managing closures contribute to this difference in performance. While still performing efficiently, JavaScript's comparative speed indicates that Nulascript may have a slight edge in scenarios demanding frequent creation and invocation of closures.

Moving to conditional statements with arithmetic expressions, Nulascript impressively demonstrates efficient execution with an average time of 0.02 seconds over 1000 iterations and 0.15 seconds over 10000 iterations. This showcases the language's prowess in handling conditional constructs with arithmetic expressions, striking a balance between speed and consistency. The predictable performance across varying iteration counts reflects Nulascript's reliability in managing complex decision-making structures.

JavaScript, initially slower with an average of 0.035 seconds per invocation over 1000 iterations, experiences a slight increase in execution time to 0.05 seconds per invocation over 10000 iterations when compared to Nulascript. This trend suggests that JavaScript's Just-In-Time (JIT) compilation may play a role in the performance difference, with the language adapting dynamically to the execution patterns. While both languages demonstrate competence in managing conditional statements, Nulascript's consistently efficient performance across iterations remains notable.

In the realm of loops, Nulascript exhibits robust efficiency with consistently fast execution times. Across various iteration counts, including 1000, 10000, and 100000, Nulascript maintains low average execution times of 0.01 seconds, 0.06 seconds, and 0.40 seconds, respectively. This highlights the language's optimization for loop-intensive operations, demonstrating its ability to handle repetitive tasks with exceptional speed.

JavaScript, while still performing adequately, shows a slower average execution time in loop operations. With averages of 0.04 seconds for 1000 iterations, 0.09 seconds for 10000 iterations, and 0.36 seconds for 100000 iterations, JavaScript's performance in loops is comparatively less efficient than Nulascript. The impact of JavaScript's JIT compilation becomes apparent as the number of iterations increases, with the language adapting dynamically to improve performance over subsequent runs.

In conclusion, Nulascript proves to be a robust and efficient language, particularly in scenarios where predictability and speed are paramount. Its consistently low execution times across various operations underscore its intrinsic optimization, even in the absence of explicit optimizations.

To propel Nulascript into a competitive stance against statically typed languages and enhance optimization, a pivotal step involves the integration of a Just-In-Time (JIT) interpreter. This evolution entails the development of a dynamic interpreter capable of generating machine code on-the-fly during program execution. Unlike traditional interpreters, a JIT interpreter translates high-level Nulascript code into machine code just before its execution, leveraging runtime information to optimize performance. This dynamic approach allows for adaptive and context-specific optimizations, aligning more closely with the efficiency associated with statically typed languages.

Moreover, to address the computational overhead linked to dynamically casting pointers during the comparison of storage objects, a strategic adjustment involves exploring ways to harness type information effectively. Rather than relying on dynamic casting, the JIT interpreter can intelligently leverage type information for storage object comparisons. This shift in approach not only mitigates the overhead associated with dynamic casting but also introduces a more streamlined and efficient mechanism for type-aware comparisons. By doing so, Nulascript can enhance its efficiency and compete more effectively in performance-critical scenarios, further bridging the gap with statically typed languages.

A computer screen shot of a program

Description automatically generated

Figure : Executing a function which accepts an argument

## Takeaways

Reflecting on Nulascript, my foray into crafting a dynamically typed scripting language with references, it's evident that the journey has been one of both discovery and realization. The foundational concept of a scripting language geared toward flexibility and ease of use remains a guiding principle, and the introduction of references, allowing the creation of aliases for variables, adds a layer of abstraction that aligns with the spirit of scripting languages.

One of the key strengths of Nulascript lies in its pursuit of a dynamic typing system. The decision to embrace dynamic typing is rooted in the desire to provide developers with the freedom to write code without the constraints of static type declarations. This flexibility can be empowering, allowing for rapid development and experimentation. The introduction of references further enhances this flexibility, offering a mechanism for creating aliases that can simplify code and contribute to a more intuitive development experience.

The choice to incorporate a syntax reminiscent of established programming languages is a deliberate one, aiming to make Nulascript approachable for a broader audience. By drawing on familiar syntax elements, developers with diverse language backgrounds can transition more seamlessly into Nulascript. This decision, while intended to ease adoption, also poses challenges, as deviations from established syntax conventions may introduce a learning curve for those expecting strict adherence to certain language norms.

As the architect of Nulascript, I acknowledge that every language design involves a series of trade-offs and decisions. While dynamic typing provides freedom, it also introduces challenges in terms of error checking and catching potential issues at compile-time. The absence of a static type system may lead to runtime errors that could be mitigated with a more stringent type-checking approach. Striking the right balance between flexibility and error prevention is an ongoing consideration in the evolution of Nulascript.

The introduction of references as a central feature in Nulascript is both a distinctive choice and a potential point of contention. References offer a level of indirection that can simplify certain code constructs, yet they also introduce a layer of complexity. Developers accustomed to more straightforward variable management may find the concept of references less intuitive. Balancing the benefits of references with the potential cognitive load they introduce is a continuous exploration in optimizing the language for usability.

In assessing the performance benchmarks provided, it's apparent that Nulascript demonstrates competitive execution times in certain scenarios. The decision to compare its performance with JavaScript, a widely adopted scripting language, provides valuable context. However, the observed differences in execution times raise questions about the underlying optimizations and areas for further improvement. Optimizing the language implementation and addressing performance differentials can be pivotal in establishing Nulascript as a robust and efficient scripting language.

While Nulascript has achieved notable milestones, it's crucial to acknowledge the areas where support and decision-making may benefit from reassessment. The absence of certain language features and the potential complexities introduced by others underscore the iterative nature of language design. Embracing constructive feedback and actively soliciting community input can be instrumental in refining Nulascript's design choices and ensuring its continued evolution as a language that resonates with developers.

In conclusion, Nulascript represents a commendable attempt to carve out a niche in the dynamic scripting language landscape. Its emphasis on flexibility, references, and an accessible syntax aligns with the principles of scripting languages. Acknowledging the ongoing nature of language design, I recognize that certain decisions, such as the trade-off between dynamic and static typing, and the introduction of references, warrant careful consideration and potential refinement. Nulascript stands as a testament to the intricacies and challenges inherent in language design, and its journey is a continuous exploration toward a scripting language that strikes the right balance between innovation and developer-friendly usability.

# **References:**

1. <https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm>
2. <https://matklad.github.io/2020/04/13/simple-but-powerful-pratt-parsing.html>
3. <https://ycpcs.github.io/cs340-fall2016/lectures/lecture05.html>
4. <https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Syntax.html/node14.html>
5. <https://pages.cs.wisc.edu/~fischer/cs536.s05/lectures/Lecture19.4up.pdf>
6. <https://cseweb.ucsd.edu/~kube/cls/12.s13/Lectures/lec16/lec16.pdf>
7. <https://introcs.cs.princeton.edu/java/11precedence/>
8. <https://www.sciencedirect.com/topics/computer-science/left-recursion>
9. <https://www.researchgate.net/publication/340808780_Usage_of_Dependency_Injection_within_different_frameworks>
10. <http://adv-r.had.co.nz/Functional-programming.html#closures>
11. <https://craftinginterpreters.com/>
12. <https://github.com/python/cpython>
13. <https://interpreterbook.com/>