Michael Bozhilov

Senior Project I

Fall 23/24

# Nulascript

## Introduction:

### History of dynamically typed languages:

The evolution of dynamically typed languages and their interpreters has been driven by a variety of factors, including the need for increased developer productivity, adaptability to changing requirements, and a focus on creating more expressive and flexible programming models. Here are key aspects of this evolution:

1. Early Focus on Symbolic Computation (Lisp):

Dynamically typed languages emerged in the context of symbolic computation and artificial intelligence. Lisp, one of the earliest dynamically typed languages, prioritized symbolic expression manipulation, making it well-suited for AI research.

2. Object-Oriented Paradigm (Smalltalk):

The Smalltalk language, developed at Xerox PARC, introduced not only dynamic typing but also the concept of objects. This laid the foundation for the widespread adoption of object-oriented programming (OOP) and influenced subsequent dynamically typed languages.

3. Scripting Languages and Rapid Development (Perl, Python):

The advent of scripting languages like Perl and Python in the 1980s and 1990s emphasized rapid development and ease of use. Dynamic typing played a crucial role in simplifying the development process, allowing developers to focus on solving problems rather than dealing with complex type systems.

4. Web Development and JavaScript:

JavaScript, introduced by Netscape in 1995, was initially designed for client-side scripting in web browsers. Its dynamic typing and event-driven nature made it suitable for creating dynamic and interactive web pages. The rise of the web further popularized dynamically typed languages.

5. Dynamic Language Renaissance (Ruby, Groovy):

The popularity of dynamically typed languages experienced a resurgence with languages like Ruby and Groovy. Ruby, with its emphasis on developer happiness, and Groovy, a dynamic language on the Java Virtual Machine (JVM), demonstrated the appeal of dynamic features in modern development.

6. Dynamism in Functional Programming (Erlang, Lisp, Python):

Dynamically typed languages found a natural fit in the functional programming paradigm. Languages like Erlang and Lisp embraced dynamic typing, facilitating the development of scalable and concurrent systems. Python, with its support for functional programming constructs, further contributed to the adoption of dynamic languages in diverse domains.

7. Dynamically Typed Languages on the JVM (Groovy, JRuby):

The Java Virtual Machine (JVM) became a platform for dynamically typed languages, extending the benefits of dynamic typing to Java-based ecosystems. Groovy and JRuby leveraged the JVM's performance while introducing dynamic language features.

8. Web Development with PHP:

PHP, designed specifically for web development, employed dynamic typing to simplify server-side scripting. Its seamless integration with HTML and focus on web applications contributed to the popularity of dynamic languages in web development.

9. Data Science and Dynamic Typing (Python, R):

In the field of data science, dynamically typed languages, particularly Python and R, gained prominence. The flexibility of dynamic typing allowed for easier manipulation and analysis of data, contributing to the growth of data science and machine learning applications.

10. Balancing Safety and Expressiveness (Swift):

The introduction of Swift by Apple in 2014 marked an interesting blend of static and dynamic typing. While Swift is statically typed, it incorporates type inference and dynamic features, providing a balance between safety and expressiveness.

11. Innovations in Modern Languages (Julia):

Modern dynamically typed languages like Julia for scientific computing. These languages prioritize performance, safety, and expressiveness, demonstrating the continued evolution of dynamically typed paradigms.

12. Emphasis on Developer Experience (TypeScript, Flow):

Optional static typing in languages like TypeScript and Flow addresses the challenges of large-scale JavaScript development. While not strictly dynamically typed, these languages offer a middle ground, allowing developers to choose between dynamic and static typing based on their needs.

In summary, the evolution of dynamically typed languages and their interpreters has been driven by a desire for improved developer productivity, adaptability to different domains, and a focus on creating expressive and flexible programming models. From the early days of Lisp and Smalltalk to the modern innovations in languages like Julia and Rust, the trajectory of dynamically typed languages reflects a continuous quest for more effective and enjoyable programming experiences.

### What is Nulascript:

Similarly, **Nulascript** is a dynamically typed programming language and akin to C-level languages, it 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.

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, 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. Essentially, a lexer serves as the initial step in the process of translating human-readable source code into a format that a computer can comprehend and work with.

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. 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. Our 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 approach leads to efficient parsing. We'll delve into more detail about the parser's implementation as we continue.

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, our **Nulascript** code will run from a C++ binary, which breaks down into machine code and contains instructions on how to execute each step from our tree.

## Lexer:

In the realm of programming languages, Nulascript's Lexer adheres to a well-established convention. Functioning as a fundamental component in the code interpretation process, its primary job is to take a piece of code in the form of a string and break it down into smaller parts, which we call “tokens”. These tokens serve as the atomic units of code, encompassing everything from keywords and identifiers to operators and constants.

The Lexer's tokenization process serves as a crucial initial step in the broader process of code interpretation. It paves the way for subsequent stages in the software development lifecycle, such as parsing and interpretation. Through this systematic dissection, it plays a pivotal role in enabling the computer to comprehend and execute the programmer's instructions with precision and accuracy.

### Now, let’s dive into how Nulascript’s Lexer is implemented:

#### The Lexer class:

**public**:

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

Token getNextToken();

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 **Lexer** class 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.

Before jumping into the explanation what **getNextToken()** does, let’s first check what the Lexer holds as private variables.

**private**:

TokenLookup tokenLookup;

std::string input;

**int** pos;

**int** readPos;

**char** ch;

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) {

**auto** it = keywords.find(ident);

**if** (it != keywords.end()) {

**return** it->second;

}

**return** IDENT;

}

In the code, the default constructor TokenLookup is used to initialize an instance of the TokenLookup class. 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.

The lookupIdent() function within the TokenLookup class is responsible for examining a provided string to determine its token type. It does this by searching for the string in the keywords map. If a match is found, the function returns the associated TokenType. If no match is found, it returns IDENT, which represents an identifier - the name of a variable or function. This function is essential for identifying and categorizing tokens within the code based on the reserved keywords provided in the keywords map.

Now, back to this part:  
  
 **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 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 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 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;

};

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...

};

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

All the tokens Nulascript’s Lexer supports:

**EOF\_TYPE**: This token represents the end of the file or input stream and is used to indicate the end of parsing.

**ILLEGAL**: This token is used to represent an illegal or unrecognized input.

**IDENT**: Stands for "identifier" and is used to represent variable or function names in the source code.

**INT**: Represents integer values in the source code.

**ASSIGN**: Represents the assignment operator, used to assign or reassign a value to a variable.

**PLUS**: Represents the addition operator.

**MINUS**: Represents the subtraction operator.

**COMMA**: Represents a comma used to separate elements in a list or function arguments.

**SEMICOLON**: Represents a semicolon used to terminate statements.

**LPAR**: Stands for "left parenthesis" and is used to open a grouping, such as for function calls or mathematical expressions.

**RPAR**: Stands for "right parenthesis" and is used to close a grouping opened by LPAR.

**LBRACE**: Represents the left curly brace, used to open a block of code or a dictionary.

**RBRACE**: Represents the right curly brace, used to close a block of code or a dictionary opened by LBRACE.

**ASTERISK**: Represents the multiplication operator or a is used for dereferencing.

**DEREF**: Stands for "dereference" and is used to access the value pointed to by a reference or pointer.

**FUNC**: Indicates a function keyword, often used to define functions in the source code.

**BANG\_OR\_NOT**: Used for logical negation (‘!’ and “not” are interchangeable in Nulascript)

**SLASH**: Represents the division operator.

**LT**: Represents the less-than comparison operator.

**LOE**: Represents "less than or equal to" comparison operator.

**GT**: Represents the greater-than comparison operator.

**GOE**: Represents "greater than or equal to" comparison operator.

**LET**: Represents "let" keyword for defining functions and variables.

**IF**: Represents the if keyword, used to begin conditional statements.

**ELSE**: Represents the else keyword, used in conjunction with IF for conditional branching.

**TRUE**: Represents the boolean true value.

**FALSE**: Represents the boolean false value.

**RETURN**: Represents the return keyword, often used to exit a function and return a value.

**IS**: Is used for identity or equality checks.

Back to the Lexer’s implementation, now we’re going to go over all the function which handle appropriate token type assignment.  
  
**void** Lexer::readChar() {

**if** (readPos >= input.size()) {

ch = 0; // EOF

} **else** {

ch = input[readPos];

}

pos = readPos;

readPos = readPos + 1;

}

This function is responsible for updating both the current position and the reading position, as mentioned earlier. The noteworthy aspect here is how we signify the end of the input for the lexer, which is essentially our source code. We achieve this by assigning the value 0 to the EOF\_TYPE token type. You might wonder where this assignment is made. The truth is, we don't explicitly set a value for the token itself. Instead, in our getNextToken() function, we have a substantial switch-case construct that deals with all the possible characters in the code and assigns their corresponding token types.

Token Lexer::getNextToken() {

Token currentToken;

skipOverWhitespace();

**switch** (ch) {

**case** '=':

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

**break**;

**case** '+':

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

**break**;

**case** '-':

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

**break**;

**case** ',':

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

**break**;

**case** ';':

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

**break**;

**case** '(':

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

**break**;

**// …**

**break**;

**case** '/':

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

**break**;

**case** '<':

currentToken =

handleComparisonOperators(ch, TokenType::LT, TokenType::LOE);

**break**;

**case** '>':

currentToken =

handleComparisonOperators(ch, TokenType::GT, TokenType::GOE);

**break**;

**case** 0:

currentToken.literal = "";

currentToken.type = TokenType::EOF\_TYPE;

**break**;

**default**:

**if** (isLetter(ch)) {

currentToken.literal = readExtendedToken(TokenType::IDENT);

currentToken.type = tokenLookup.lookupIdent(currentToken.literal);

**return** currentToken;

} **else** **if** (isDigit(ch)) {

currentToken.type = TokenType::INT;

currentToken.literal = readExtendedToken(currentToken.type);

**return** currentToken;

} **else** {

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

}

}

readChar();

**return** currentToken;

}

This is pretty much the heart of the Lexer.

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

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

For example, if ch is '=', it creates a new token with a TokenType of ASSIGN.

If ch is '+', it creates a token with PLUS, and so on for various other characters.

The code handles various punctuation and operator characters commonly found in other programming languages.

If ch is 0, indicating the end of the file, it sets the token type to EOF\_TYPE.

If ch is not one of the recognized characters, it checks if it's a letter (part of an identifier) or a digit (part of an integer literal). If it's a letter, it reads an extended token (possibly a longer identifier) and determines the correct TokenType using a lookup. If it's a digit, it treats it as an integer and reads an extended token accordingly.

If none of the above conditions are met, it marks the token as ILLEGAL.

After determining the token type and possibly the token's literal value, the readChar() function is called to advance to the next character in the input.

Finally, it returns the extracted token.

In this code segment, we encounter a few tokens that require some special handling, particularly the comparison operator tokens.  
  
Token Lexer::handleComparisonOperators(**char** opChar, TokenType shortType,

TokenType extendedType) {

// reduce branching by not conditionally checking and inferring the extended

// type

**if** (peekNextChar() == '=') {

**char** savedCh = ch;

readChar();

**char** tokenLiteral[3] = {savedCh, ch, '\0'};

**return** newToken(extendedType, tokenLiteral);

}

**return** newToken(shortType, ch);

}

The Token Lexer::handleComparisonOperators(char opChar, TokenType shortType,

TokenType extendedType) function comes into play here. It 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.

**default**:

**if** (isLetter(ch)) {

currentToken.literal = readExtendedToken(TokenType::IDENT);

currentToken.type = tokenLookup.lookupIdent(currentToken.literal);

**return** currentToken;

} **else** **if** (isDigit(ch)) {

currentToken.type = TokenType::INT;

currentToken.literal = readExtendedToken(currentToken.type);

**return** currentToken;

} **else** {

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

}

**Identifier Handling**:

First, it 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.

**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.

**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.

## Parser:

Nulascript implements a recursive descent parser, a type of top-down parser, which means it starts with the highest-level grammar rules and recursively breaks them down into smaller, more specific rules. This approach is often used for simple or moderately complex programming languages because it maps well to the way code is typically structured in these languages.

In a recursive descent parser, each non-terminal symbol in the grammar corresponds to a function in the code. These functions are responsible for recognizing and parsing the corresponding grammar rules. The parser recursively calls these functions to parse the input code.

This top-down parsing technique used to recognize the structure of a programming language by applying a set of parsing functions in a recursive manner. This approach is often used for parsing LL(k) grammars, where "LL" stands for "left-to-right, leftmost derivation," and "k" represents the number of lookahead tokens. Recursive descent parsers are favored for their hand-crafted nature, as they involve manually writing parsing functions based on the grammar of the language.

In a recursive descent parser, parsing functions are created for each non-terminal symbol or production rule in the grammar. These functions correspond to different levels of abstraction or nesting in the language's grammar, and they mirror the hierarchical structure of the code. As such, the code of a recursive descent parser closely reflects the structure of the grammar rules, making it readable and easy to understand.

Recursion plays a fundamental role in recursive descent parsing. Parsing functions call each other recursively to handle nested constructs within the language. For example, a parsing function for expressions may invoke the parsing function for sub-expressions. This recursive nature aligns well with the natural hierarchy of the language’s syntax.

The difference in the type of parser is rooted in the method of associating parsing functions with specific tokens. The fundamental concept is that each token can have two distinct sets of parsing functions, determined by whether the token is positioned as a prefix or an infix.

However, recursive descent parsers have limitations. They are best suited for simple and moderately complex languages with LL(k) grammars. In cases where the grammar is ambiguous or left-recursive, other parsing techniques, such as LR parsing, may be more suitable. Error handling in recursive descent parsers is often simple, usually involving the production of error messages when unexpected input is encountered. Additionally, there can be instances of backtracking in cases where the parser makes incorrect predictions based on the input, but in the case of Nulascript, backtracking is not implemented.

## 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.

Ease of Transformation: The AST serves as a foundation for various transformations and optimizations. If additional language features or modifications are desired, developers can manipulate the AST, add new nodes, or apply transformations before or during the evaluation phase.

Error Reporting: The AST can also be a valuable tool for generating meaningful error messages. If an error is encountered during parsing or evaluation, information from the AST can aid in pinpointing the location and nature of the error in the original source code.

Persistence: In some cases, the AST might be persisted or serialized. This can be useful for scenarios where the interpreted code needs to be stored, transmitted, or analyzed outside the runtime environment.

In summary, the AST is a crucial component in the implementation of an interpreter, providing a structured representation of the code that facilitates parsing, evaluation, and various language-related tasks. Its hierarchical nature aligns well with the recursive descent parsing approach and contributes to the overall flexibility and extensibility of the interpreter.

Let’s proceed to Nulascript’s AST implementation:

**class** Node {

**public**:

**virtual** std::string tokenLiteral() = 0;

**virtual** std::string toString() = 0;

};

**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 function 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 function 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 in this snippet, 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 developers to represent and manipulate the syntactic structure of code during parsing and analysis.

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

**public**:

std::vector<Statement\*> statements;

std::string tokenLiteral() override;

std::string toString() override;

};

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 {

**public**:

Token token;

std::string value;

**public**:

Identifier(Token token);

std::string tokenLiteral() override;

std::string toString() override;

};

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 {

**public**:

Token token;

Identifier\* name;

Expression\* value;

**public**:

LetStatement(Token token);

std::string tokenLiteral() override;

std::string toString() override;

};

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 {

**public**:

Token token;

Expression\* returnValue;

**public**:

ReturnStatement(Token token, Expression\* returnValue);

ReturnStatement(Token token);

std::string tokenLiteral() override;

std::string toString() override;

};

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 {

**public**:

Token token;

Expression\* expression;

**public**:

ExpressionStatement(Token token);

// ExpressionStatement(Token token, Expression\* expression)

std::string tokenLiteral() override;

std::string toString() override;

};

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 {

**public**:

Token token;

Expression\* condition;

BlockStatement\* currentBlock;

BlockStatement\* elseBlock;

**public**:

Conditional(Token token);

std::string toString();

std::string tokenLiteral();

};

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.

To avoid repeating the same code all over again, I'll highlight some of the implemented expression classes.

**Function Class:**

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

**Invocation Class:**

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.

**String Class:**

Simple storage for string values.

**Integer Class:**

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

**Assignment Class:**

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

**Reference Class:**

Stores the identifier pointing to the referenced value.

**Pointer Class:**

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

**Comment Class:**

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

**BlockStatement Class:**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 a ForLoop is defined within the syntax tree, there are two important classes we haven’t mentioned yet.

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

**public**:

Token token;

Expression\* left;

Expression\* right;

std::string op;

**public**:

Infix(Token token, Expression\* left, Expression\* right);

Infix(Token token, Expression\* left);

Infix(Token token);

std::string tokenLiteral() override;

std::string toString() override;

};

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

**public**:

Token token;

std::string op;

Expression\* right;

**public**:

Prefix(Token token, Expression\* expression);

Prefix(Token token);

std::string tokenLiteral() override;

std::string toString() override;

};

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.

The last one class, but the least interesting is the **ForLoop** expression class.

**struct** ForLoopInitialization {

LetStatement\* variable;

Infix\* conditional;

Infix\* increment;

};

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

**public**:

Token token;

BlockStatement\* code;

// loop variables initialization

ForLoopInitialization definition;

**public**:

ForLoop(Token token);

std::string tokenLiteral() override;

std::string toString() override;

};

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 Class

**private**:

Lexer\* l;

Token currentToken;

Token peekToken;

std::vector<std::string> errors;

**Lexer\* l - t**his is a pointer to our Lexer object. The Parser needs access to the Lexer to request tokens from the source code during parsing. The Lexer is typically used to provide the parser with the current token and the next token (peeked token) from the source code.

**Token currentToken** - this member is of type Token and is used to store the currently processed token during parsing. The Parser will analyze this token to determine the structure and semantics of the source code. The currentToken is typically updated when the parser advances to the next token in the source code using the getNextToken() function.

**Token peekToken** - this member, also of type Token, is used to store the next token that will be processed by the parser. It is called the "peek token" because the parser can look ahead to the next token without consuming it. The peekToken is useful for making decisions about the parsing process based on upcoming tokens. It is typically updated in the getNextToken() function as well.

**std::vector<std::string> errors** - this is a vector used to store error messages encountered during the parsing process. If the parser encounters a syntax error or any other issue in the source code, it appends an error message to this vector. Later, the user or developer can retrieve these error messages for debugging or reporting issues in the source code.

In summary, these private data members are essential for the operation of the Parser class. The Lexer provides a stream of tokens for the parser to process, while currentToken and peekToken hold the currently analyzed tokens and the next token to be processed. The errors vector serves as a container for storing error messages, making it possible to report and handle issues in the source code.

**bool** isEqualToCurrentTokenType(TokenType tokenType);

**bool** isEqualToPeekedTokenType(TokenType tokenType);

**bool** peekAndLoadExpectedToken(TokenType tokenType);

Before exploring how statements are being parsed, we need to make sure we understand how the parser loads the tokens:

**bool** Parser::isEqualToCurrentTokenType(TokenType tokenType) {

**return** currentToken.type == tokenType;

}

**bool Parser::isEqualToCurrentTokenType(TokenType tokenType):** This function compares the type of the current token (stored in the currentToken member) to the specified tokenType. It returns true if the current token's type matches the specified type and false otherwise. This function is useful for making decisions based on the current token's type during parsing.

**bool** Parser::isEqualToPeekedTokenType(TokenType tokenType) {

**return** peekToken.type == tokenType;

}

**bool Parser::isEqualToPeekedTokenType(TokenType tokenType):** This function compares the type of the peeked token (stored in the peekToken member) to the specified tokenType. It returns true if the peeked token's type matches the specified type and false otherwise. Like the previous function, this one is used to make decisions based on the type of the upcoming token.

**bool** Parser::peekAndLoadExpectedToken(TokenType tokenType) {

**if** (isEqualToPeekedTokenType(tokenType)) {

getNextToken();

**return** **true**;

}

appendPeekError(tokenType);

**return** **false**;

}

**bool Parser::peekAndLoadExpectedToken(TokenType tokenType**): This function serves a more specific purpose. It compares the type of the peeked token to a specified tokenType. If they match, the function updates the currentToken to be the peeked token by calling getNextToken() and returns true. However, if the peeked token does not match the specified tokenType, the function appends an error message using the appendPeekError() function and returns false.

The logic of these functions is used to ensure that the parser is processing the correct tokens and is in sync with the expected structure of the source code. They are particularly useful for enforcing syntax rules and generating appropriate error messages when the source code does not conform to expected patterns.

Here's a brief explanation of how they might be used:

isEqualToCurrentTokenType() and isEqualToPeekedTokenType() are used to check if the current or peeked token has a specific type, respectively.

peekAndLoadExpectedToken() is used to check if the peeked token has a specific type, and if it does, it updates the currentToken to the peeked token, effectively advancing the parser to the next token. If the types don't match, an error message is appended, indicating an expected token type that was not found.

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();

**…**

This function 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 functions, such as parseLetStatement() and parseReturnStatement(). Here's the breakdown:

switch (currentToken.type): The function starts by examining the type of the currentToken. The type of the token is one of the possible values defined in the TokenType enumeration. The switch statement is used to evaluate the token's type and select the appropriate parsing procedure.

If the currentToken is of type LET, it indicates the beginning of a 'let' statement. In this case, the parseLetStatement() function is called to parse and construct the AST node representing the 'let' statement.

If the currentToken is of type RETURN, it signifies the start of a 'return' statement. The parseReturnStatement() function is invoked to parse and generate the AST node for the 'return' statement.

In summary, the parseStatement() function is a central part of the parser that identifies the type of statement based on the current token and delegates the actual parsing of the statement to dedicated functions. This approach helps organize the parsing process and ensures that the appropriate syntax rules and structure for each type of statement are followed.

There's one more crucial scenario we haven't discussed yet—the default case. We skipped it because it utilizes a slightly different model compared to the ReturnStatement and LetStatement.

default:

**return** parseExpressionStatement();

This function performs the magic related to parsing expressions. However, since expressions must be parsed in the context of a statement, we consistently return an object of type ExpressionStatement, encapsulating a node that points to the actual Expression. Before delving into the implementation of one of the parser's most pivotal methods, let's first understand how we initially handle the program's code.

The parseProgram() function is a critical part of the parser, responsible for parsing an entire program, which is essentially the top-level structure of a source code file. Let's explain this code and discuss why it is needed:

Program\* Parser::parseProgram() {

Program\* program = **new** Program();

**while** (currentToken.type != TokenType::EOF\_TYPE) {

Statement\* statement = **this**->parseStatement();

**if** (statement) {

program->statements.push\_back(statement);

}

getNextToken();

}

**return** program;

}

Program\* program = **new** Program();

This line creates a new instance of a Program object. A Program represents the top-level structure of the source code and typically contains a collection of statements.

**while** (currentToken.type != TokenType::EOF\_TYPE)

The parser enters a loop that continues until it encounters the end of the file (EOF), which is signaled by the TokenType::EOF\_TYPE. This loop is responsible for parsing all the statements within the program.

Statement\* statement = **this**->parseStatement();

Inside the loop, the parseStatement() function is called to parse the next statement in the source code. The result is stored in the statement variable.

**if** (statement)

This conditional check ensures that the statement is not null. If the parser successfully parsed a statement (it's not an error or an empty statement), the following code is executed.

program->statements.push\_back(statement)

The parsed statement is added to the statements vector within the Program object. This vector is used to accumulate all the statements in the program.

getNextToken();

After parsing a statement, the parser advances to the next token in the source code using the getNextToken() function.

Once the parser has processed all the statements in the program and reaches the end of the file (EOF), it returns the Program object. This Program object now contains a structured representation of the entire source code.

Why do we need it:

The parseProgram() function is necessary because it orchestrates the parsing of an entire source code file or program. A program can consist of multiple statements, expressions, and declarations, and it's essential to organize these components into a structured format that can be later used.

By parsing the program into a structured representation (in this case, the Program object), the parser allows subsequent stages of the compiler or interpreter to work with the code more easily. For example, it enables semantic analysis, optimization and code generation, as well as providing a basis for executing the program.

This function also ensures that the parser processes statements until the end of the file (EOF\_TYPE) is reached, preventing it from prematurely terminating the parsing process. It handles statements of various types and accumulates them within the Program object for later use or analysis.

### Parsing within the parseStatement method:

Let’s go back to discussing how the parsing within the **parseStatement** method works.

As you recall, there are three cases:

* parseLetStatement
* parseReturnStatement
* parseExpressionStatement

#### Parsing Let Statements

LetStatement\* Parser::parseLetStatement() {

LetStatement\* letStatement = **new** LetStatement(currentToken);

**if** (!peekAndLoadExpectedToken(TokenType::IDENT)) {

**return** **nullptr**;

}

letStatement->name = **new** Identifier(currentToken);

**if** (!peekAndLoadExpectedToken(TokenType::ASSIGN)) {

**return** **nullptr**;

}

getNextToken();

letStatement->value = parseExpression(Precedence::LOWEST);

**if** (isEqualToPeekedTokenType(TokenType::SEMICOLON)) {

getNextToken();

}

**return** letStatement;

}

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.

#### Parsing Return Statements

ReturnStatement\* Parser::parseReturnStatement() {

ReturnStatement\* returnStatement = **new** ReturnStatement(currentToken);

getNextToken();

returnStatement->returnValue = parseExpression(Precedence::LOWEST);

**if** (isEqualToPeekedTokenType(TokenType::SEMICOLON)) {

getNextToken();

}

**return** returnStatement;

}

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.   
  
Important is to not that, we do not return a nullptr if the statements don’t end with a semicolon due to the reason that we allow expressions to be unterminated. One of the few cases where we would explicitly want to have a terminating semicolon is when assigning identifiers as values to other variables on the same line and not separating them with a whitespace.

**def** c = a;**def** b = a; # this works as expected

**def** c = a**def** b = a; # this breaks

* **Parsing Expression Statements**

ExpressionStatement\* Parser::parseExpressionStatement() {

auto statement = new ExpressionStatement(currentToken);

statement->expression = parseExpression(Precedence::LOWEST);

**if** (isEqualToPeekedTokenType(TokenType::SEMICOLON)) {

getNextToken();

}

**return** statement;

}

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.

Top of Form

Expression\* Parser::parseExpression(Precedence p) {

auto it = prefixParsingFunctions.find(currentToken.type);

**if** (it == prefixParsingFunctions.end()) {

std::string err =

"[ERROR] No parsing function was found for prefix of type: " +

std::to\_string(currentToken.type);

appendError(err);

**return** nullptr;

}

auto leftExpression = it->second();

**while** (!isEqualToPeekedTokenType(TokenType::SEMICOLON) &&

p < checkPeekPrecedence()) {

auto infix\_it = infixParsingFunctions.find(peekToken.type);

**if** (infix\_it == infixParsingFunctions.end()) {

**return** leftExpression;

}

getNextToken();

leftExpression = infix\_it->second(leftExpression);

}

**return** leftExpression;

}

Bottom of Form

The function **parseExpression** takes a precedence level (**p**) as an argument and is responsible for parsing expressions in the specified precedence level. We will touch up on why precedence is important a bit later.

It starts by looking up the parsing function 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** variable.

The function then enters a loop that continues as long as the current token type is not **SEMICOLON** (end of expression) and the precedence of the current token is greater than the specified precedence level (**p**).

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 **leftExpression**, indicating that the parsing is complete.

If an infix parsing function is found, it advances to the next token (**getNextToken()**) and calls the infix parsing function with the **leftExpression** 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.

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

auto it = prefixParsingFunctions.find(currentToken.type);

The current token is of type Integer. We find the following parsing function:

Integer\* Parser::parseInteger() {

**try** {

int64\_t literal = stoi(currentToken.literal);

auto lit = new Integer(currentToken);

lit->value = literal;

**return** lit;

} catch (...) {

appendError("Couldn't parse literal to integer");

**return** nullptr;

}

}

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. This happens in this code block:  
  
**while** (!isEqualToPeekedTokenType(TokenType::SEMICOLON) &&

p < checkPeekPrecedence()) {

auto infix\_it = infixParsingFunctions.find(peekToken.type);

**if** (infix\_it == infixParsingFunctions.end()) {

**return** leftExpression;

}

getNextToken();

leftExpression = infix\_it->second(leftExpression);

}

Within the loop's execution, the function 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(5) expression to the function as an argument.

registerInfixFunction(

TokenType::PLUS,

[&](Expression\* left) -> Expression\* { **return** parseInfix(left); });

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 = '+';

### Registering parsing functions:

To give the parser access to the parsing functions for tokens **Nulascript** supports, we have to register them upon initialization. We do this by invoking the **registerPrefixFunction** and **registerInfixFunction**:

registerPrefixFunction(TokenType::IDENT,

[&]() -> Expression\* { **return** parseIdentifier(); });

registerPrefixFunction(TokenType::REF,

[&]() -> Expression\* { **return** parseReference(); });

registerInfixFunction(

TokenType::MINUS,

[&](Expression\* left) -> Expression\* { **return** parseInfix(left); });

registerInfixFunction(

TokenType::PLUS,

[&](Expression\* left) -> Expression\* { **return** parseInfix(left); });

Here, we register the parseIdentifier function for the IDENT token and the parseReference for the token identifying a reference.

The register function methods are really simple In what they do:

They receive as arguments the type of the token and the parsing function and append them to the respective map holding the token type as key and function as the value.

### Parsing functions:

#### parseInfix:

The **Parser::parseInfix** function constructs and returns an **Infix** expression, which represents an infix operation in the programming language. It takes a pointer to the left operand (**Expression\* left**) and uses the current token (**currentToken**) as well.

It initializes an **Infix** object with the current token and the left operand. The **checkCurrentPrecedence** function determines the precedence level of the current token, and this information is used to parse the right operand with the **parseExpression** function.

The function advances to the next token using **getNextToken**. The resulting **Infix** expression has its **right** field set to the parsed right operand. The constructed **Infix** expression is then returned.

In summary, the function encapsulates the logic for parsing infix expressions by handling the left and right operands and operator precedence.

#### parsePrefix:

The **Parser::parsePrefix** function is responsible for parsing expressions with prefix operators.

The function begins by creating a new **Prefix** expression object, initializing it with the current token (**currentToken**). This **Prefix** object will represent the structure of a prefix expression.

It then advances to the next token using **getNextToken()**, assuming the current token has been successfully processed as part of the prefix operation.

The function proceeds to parse the right operand of the prefix expression by calling **parseExpression** with the specified precedence level (**Precedence::PREFIX**). The result is assigned to the **right** field of the **expression** object.

Finally, the function returns a pointer to the constructed **Prefix** expression, encapsulating the operator and its right operand.

In summary, the **parsePrefix** function handles the parsing of prefix expressions, ensuring that the structure is appropriately represented by creating a **Prefix** object and setting its right operand based on the next token.

#### parseBlock:

The **Parser::parseBlock** function is responsible for parsing a block statement.

It starts by creating a new **BlockStatement** object named **currentBlock**, initialized with the current token. The **statements** field of the **currentBlock** object is initialized as an empty vector of **Statement** pointers. This vector will store the statements within the block. The parser then advances to the next token using **getNextToken()**. The function enters a loop that continues until it encounters the end of the file (**EOF\_TYPE**) or the closing brace (**RBRACE**). This loop is designed to parse and collect statements within the block.

Within the loop, it calls the **parseStatement** function to parse the next statement within the block. The parsed statement is then checked for validity (non-null).

If a valid statement is parsed, it is added to the **statements** vector of the **currentBlock** object.

The parser then advances to the next token using **getNextToken()** to prepare for the next iteration of the loop.

Once the loop completes, the function returns a pointer to the fully parsed **BlockStatement** object, including its list of statements.

In summary, the **parseBlock** function iteratively parses statements within a block, populating a vector with the parsed statements, and returning a pointer to the resulting **BlockStatement** object.

#### parseConditional:

The **Parser::parseConditional** is designed for parsing conditional statements, representing an "if-else” construct.

It starts by creating a new **Conditional** object named **conditional**, initialized with the current token. The function checks if the next expected token is a left parenthesis (**LPAR**) by using the **peekAndLoadExpectedToken** function. If not, it returns **nullptr**, indicating a parsing error. It advances to the next token using **getNextToken()** and proceeds to parse the condition expression using the **parseExpression** function with the lowest precedence.

After parsing the condition, it checks if the next expected token is a right parenthesis (**RPAR**). If not, it returns **nullptr**. It then checks if the next expected token is a left brace (**LBRACE**). If not, it returns **nullptr**. The function proceeds to parse the block of statements associated with the "if" condition using the **parseBlock** function and assigns the resulting block to the **currentBlock** field of the **conditional** object. It checks if there is an "else" clause by looking at the next token. If the next token is of type **ELSE**, it enters a conditional block to parse the "else" part. It advances to the next token using **getNextToken()** and checks if the next expected token is a left brace (**LBRACE**). If not, it returns **nullptr**. Inside the "else" block, it parses the statements within the "else" block using the **parseBlock** function and assigns the resulting block to the **elseBlock** field of the **conditional** object. Finally, the function returns a pointer to the fully parsed **Conditional** object, capturing the condition, the "if" block, and optionally the "else" block.

In summary, the **parseConditional** function handles the parsing of conditional statements, including the condition expression, the "if" block, and an optional "else" block.

#### parseFunction:

The **Parser::parseFunction** method is responsible for parsing function declarations.

It starts by creating a new **Function** object named **func**, initialized with the current token.

The function checks if the next expected token is a left parenthesis (**LPAR**) using the **peekAndLoadExpectedToken** function. If the check fails, indicating a parsing error, the function returns **nullptr**. It proceeds to parse the function arguments by calling the **parseFunctionArguments** function. The result is assigned to the **arguments** field of the **func** object. It then checks if the next expected token is a left brace (**LBRACE**). If not, it returns **nullptr**. The function proceeds to parse the block of statements constituting the function's code using the **parseBlock** function. The resulting block is assigned to the **code** field of the **func** object. Finally, the function returns a pointer to the fully parsed **Function** object, encapsulating information about the function declaration, its arguments, and the code block.

In summary, the **parseFunction** function handles the parsing of function declarations, including the function name (captured in the **currentToken**), the list of arguments, and the body of the function as a block of statements. It returns a pointer to the parsed **Function** object.

#### parseInvocation:

The Parser::parseInvocation method is focused on parsing function invocations in Nulascript.

It creates a new Invocation object named invocation and initializes it with the current token. The Invocation class appears to represent a function invocation, and the current token is expected to be the name of the function being invoked.

The function parameter is assumed to be a pointer to an **Expression** representing the function being invoked. It is cast to a **Function\*** since the **Invocation** constructor expects a pointer to a **Function**.

The function proceeds to parse the arguments of the function invocation by calling the **parseInvocationArguments** function. The resulting arguments are assigned to the arguments field of the invocation object.

Finally, the function returns a pointer to the fully parsed **Invocation** object, encapsulating information about the function invocation, including the function being called and its arguments.

In summary, the parseInvocation function handles the parsing of function invocations, creating an Invocation object and populating it with the necessary information such as the invoked function and its arguments. It returns a pointer to the parsed Invocation object.

#### parseString:

In summary, the parseString function handles the parsing of string literals and returns a pointer to a String object encapsulating the information about the parsed string. It creates the String object by providing the current token as an argument since it a new string can be constructed using its literal property.

#### parseReference:

The Parser::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. If the check fails, indicating a parsing error, the function returns nullptr.

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 summary, the parseReference function handles the parsing of identifier references, creating a Reference object and populating it with the referenced identifier's name. It returns a pointer to the parsed Reference object or nullptr if the parsing encounters an error.

#### parseForLoop:

The **Parser::parseForLoop** function is for parsing "for" loop constructs.

It begins by creating a new **ForLoop** object named **fl** and initializes it with the current token. The **ForLoop** class likely represents the structure of a "for" loop in the language. The function checks if the next expected token is a left parenthesis (**LPAR**) using the **peekAndLoadExpectedToken** function. If the check fails, indicating a parsing error, the function returns **nullptr**. The function advances to the next token using **getNextToken()**. It then attempts to parse a **LetStatement** representing the loop variable initialization by using **parseStatement** and casting the result to a **LetStatement**. If the parsing of the loop variable is unsuccessful (i.e., if the cast to **LetStatement** fails), the function returns **nullptr**. The function advances to the next token and creates a new **Identifier** object representing the loop condition variable, initialized with the current token. It then parses the loop condition by calling **parseInfix** with the identifier. If the left or right operands of the resulting **Infix** object are not present, the function returns **nullptr**. The function advances through a few more tokens to parse the incremental statement. It creates an **Identifier** for the incremental variable and then parses the increment expression using **parseInfix**. If the left or right operands of the resulting **Infix** object are not present, the function returns **nullptr**. It checks if the next expected tokens are a right parenthesis (**RPAR**) and a left brace (**LBRACE**). If either check fails, the function returns **nullptr**. The function proceeds to parse the block of statements within the "for" loop using the **parseBlock** function. Finally, the function updates the **ForLoop** object (**fl**) with the parsed information, including the block of code and the loop variable initialization, condition, and increment. In summary, the **parseForLoop** function handles the parsing of "for" loop constructs, including the initialization, condition, increment, and the block of statements within the loop. It returns a pointer to the fully parsed **ForLoop** object or **nullptr** if the parsing encounters an error.

The parsing functions are instrumental in constructing instances of classes defined in the **ast.h** and **ast.cc** files, serving as the various nodes within our abstract syntax tree (AST). These functions play a crucial role in transforming the textual representation of code into a hierarchical structure that reflects the syntax and semantics of the programming language. Each parsing function is tailored to handle specific language constructs, such as expressions, statements, and loops, creating corresponding AST nodes. The resulting AST provides a structured and organized representation of the parsed code, facilitating subsequent analysis, interpretation, or compilation processes. Overall, the parsing functions serve as a bridge between the raw source code and a comprehensible, tree-like representation that captures the program's structure and logic.

### 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 14. 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.

## Evaluation

### Why evaluation?

Scripting languages have become integral components of modern software development, playing a pivotal role in various domains such as web development, automation, system administration, and data analysis. One of the fundamental aspects that differentiate scripting languages from traditional compiled languages is the nature of evaluation. In this extensive exploration, we delve into the profound significance of evaluation in scripting languages, examining its role in code execution, dynamic behavior, rapid prototyping, and the facilitation of high-level abstractions.

1. **Dynamic Nature and Interactivity:**

One of the defining features of scripting languages is their dynamic nature, where code is often executed at runtime rather than being compiled beforehand. This dynamic behavior allows for interactive development, empowering developers to experiment, test, and modify code in real-time. Evaluation plays a central role in this process, enabling developers to execute code snippets, observe results, and iterate quickly. The immediate feedback loop provided by dynamic evaluation enhances productivity and fosters an exploratory programming style.

2. **Rapid Prototyping and Iterative Development:**

Scripting languages are particularly well-suited for rapid prototyping and iterative development due to their interpreted nature and dynamic features. Evaluation facilitates a quick turnaround in the development cycle. Developers can incrementally add, modify, or refine code sections without the need for time-consuming compilation steps. This agility accelerates the development process, allowing teams to experiment with ideas, make adjustments on the fly, and promptly respond to evolving requirements.

3. **Facilitating Code Exploration and Debugging:**

Evaluation is a powerful tool for code exploration and debugging in scripting languages. Developers can interactively inspect variables, execute specific sections of code, and observe intermediate results, providing a deeper understanding of program behavior. This capability aids in identifying and fixing errors more efficiently. Debugging tools in scripting environments often leverage the evaluation feature to provide interactive debugging sessions, allowing developers to step through code, inspect variables, and make corrections in real-time.

4. **Dynamic Typing and Polymorphism:**

Many scripting languages feature dynamic typing, where variable types are determined at runtime. Evaluation is fundamental to handling dynamic types and enabling polymorphic behavior. Code can adapt to different data types dynamically, promoting flexibility and reducing the need for explicit type declarations. This dynamic aspect of scripting languages simplifies code, enhances expressiveness, and supports more natural, high-level abstractions.

5. **Enhancing Expressiveness through Higher-Order Functions:**

Scripting languages often support higher-order functions, where functions can be treated as first-class citizens. Evaluation plays a crucial role in the implementation of higher-order functions, allowing functions to be passed as arguments, returned as results, and assigned to variables. This feature contributes to concise and expressive code, promoting modularization, code reuse, and the development of more abstract and generic algorithms.

6. **Enabling Reflection and Metaprogramming:**

Reflection and metaprogramming are powerful features in scripting languages that enable code to inspect and manipulate itself dynamically. Evaluation is at the core of these capabilities, allowing scripts to examine their own structure, query metadata, and generate code dynamically. Reflection and metaprogramming empower developers to write more flexible and adaptive software, facilitating the creation of frameworks, libraries, and tools that can adapt to changing requirements at runtime.

7. **Supporting Domain-Specific Languages (DSLs):**

Scripting languages are often used as a foundation for developing domain-specific languages (DSLs) tailored to specific application domains. Evaluation is instrumental in the design and implementation of DSLs, enabling the creation of specialized syntax and semantics that align with the problem at hand. This extensibility allows developers to craft solutions that are more intuitive and closely aligned with the requirements of a particular domain.

8. **Interactive Data Analysis and Visualization:**

Scripting languages are widely employed for data analysis, scientific computing, and visualization tasks. Evaluation is essential in these contexts as it supports interactive exploration of data. Scientists and analysts can run data analysis scripts step by step, visualize intermediate results, and iteratively refine their analyses. The interactive nature of evaluation facilitates a more intuitive and exploratory approach to data exploration.

9. **Scripting in System Administration and Automation:**

Scripting languages are extensively used in system administration and automation tasks, where repetitive operations need to be executed efficiently. Evaluation allows administrators to build and execute scripts that automate complex tasks, such as system configuration, file manipulation, and network operations. The ability to evaluate scripts interactively and see the immediate impact on the system streamlines the process of creating and maintaining automation scripts.

10. **Fostering Collaboration and Script Sharing:**

The dynamic and interactive nature of scripting languages contributes to collaborative development practices. Developers can easily share code snippets, functions, or even entire scripts for evaluation. Online platforms and forums often leverage this interactive scripting paradigm to facilitate collaborative problem-solving and knowledge sharing within the developer community.

11. **Facilitating Scripting in Web Development:**

Scripting languages are fundamental in web development for implementing client-side logic, server-side logic, and managing dynamic content. Evaluation is essential in creating responsive and interactive web applications. Technologies like JavaScript, executed in web browsers, leverage evaluation to handle user interactions, update the DOM dynamically, and provide a seamless user experience.

12. **Enabling Scripting in Embedded Systems:**

Scripting languages find applications in embedded systems, where resource constraints may limit the use of traditional compiled languages. The ability to evaluate code dynamically is particularly advantageous in these contexts. Scripting languages allow developers to create flexible and adaptable embedded systems, where scripts can be updated or modified without requiring a full recompilation.

### Storage class:

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:

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.
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.
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. **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.
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.
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.
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.
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.
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.
10. **Extensibility and Modularity:**
    * 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.

### Environment class:

The **Environment** in Nulascript is a pivotal component for handling variable scopes and bindings during code interpretation. It operates through key methods: **get**, **set**, and **setOutsideScope**.

The **get** method is responsible for retrieving the value associated with a given variable key (**k**). It first searches the local environment (**store**) for the key. If the key is found locally, the corresponding **Storage** object is returned. If the key is not found in the local environment and there is an outer scope (**outsideScope** is not **nullptr**), it attempts to find the key in the outer scope's environment. This hierarchical scoping allows for the resolution of variables that may be defined in an outer context. If the key is not found in either the local or outer scope, an **ErrorStorage** object is created, indicating that the variable is undefined.

The **set** method is responsible for associating a value (**v**) with a variable key (**k**) in the local environment (**store**). It sets the key-value pair in the local environment and returns the associated **Storage** object.

The **setOutsideScope** method is crucial for establishing a link between the current environment and an outer environment. This is particularly significant in the context of functions, for-loops, and block statements. When functions or block statements are encountered during code interpretation, they create a new environment to encapsulate their local variables. The **setOutsideScope** method allows the newly created environment to be linked to the outer scope, facilitating the correct resolution of variables in an encompassing context.

In scenarios involving functions, for-loops, or block statements, the hierarchical scoping enabled by the **outsideScope** connection ensures that variables defined within these constructs can access and modify variables from their containing environments. This behavior aligns with the expected lexical scoping rules, where the scope of a variable is determined by its location in the source code. The **outsideScope** mechanism plays a critical role in maintaining the integrity of variable references, contributing to the accurate interpretation of code in Nulascript.

### 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 a 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.

### REPL:

A REPL, which stands for Read-Eval-Print Loop, is a programming environment that facilitates an interactive and iterative approach to software development and code execution. This type of environment allows developers to enter code, have it evaluated or executed, and immediately receive feedback, all within the same interface. The fundamental components of a REPL include a read phase, an evaluation phase, a printing phase, and an optional loop that enables continuous interaction.

In the read phase, the REPL takes user input in the form of code snippets or expressions. This input is then parsed and converted into a data structure that the system can understand. This step involves tokenization and syntactic analysis to transform the raw code into a format that can be processed.

Following the read phase, the evaluation phase comes into play. The parsed code is executed, and the results are computed. This phase involves interpreting or compiling the code, depending on the specific implementation of the REPL. During evaluation, variables are assigned values, functions are executed, and the overall behavior of the code is manifested.

Once the evaluation is complete, the printing phase takes center stage. The results of the executed code, such as output values or error messages, are displayed to the user. This provides immediate feedback on the behavior and correctness of the entered code, facilitating a quick and iterative development process.

The loop aspect of the REPL is what enables continuous interaction. After the printing phase, the REPL typically returns to the read phase, allowing users to enter new code or modify existing code based on the feedback received. This iterative loop is particularly valuable for experimenting with code, testing hypotheses, and refining solutions in a step-by-step manner.

One of the key advantages of a REPL is its interactive nature, which fosters a dynamic and exploratory programming experience. Developers can incrementally build and refine their code, testing small portions at a time and immediately observing the outcomes. This real-time feedback loop enhances productivity, as it reduces the need for time-consuming compilation or execution cycles that are common in more traditional development environments.

REPLs are commonly associated with interpreted languages, where the code can be executed directly without the need for explicit compilation. Languages like Python, Ruby, JavaScript, and Lisp have popular REPL implementations. However, REPLs are not limited to interpreted languages, as some compiled languages also offer REPL environments for rapid prototyping and testing.

In summary, a REPL is a versatile and powerful tool that provides an interactive and iterative environment for coding, testing, and debugging. Its read-eval-print loop structure enables developers to enter code, have it evaluated, receive immediate feedback, and iterate on their solutions in a seamless and efficient manner, ultimately contributing to a more dynamic and exploratory programming experience.

The C++ code found in the repl folder encapsulates the implementation of Nulascript’s Read-Eval-Print Loop (REPL). The REPL is designed to continuously prompt the user with a distinctive "> " symbol, awaiting input for code snippets. Within this interactive environment, the user's input undergoes a series of stages for interpretation. Initially, the input string is processed by a lexer, responsible for breaking it into individual tokens, the building blocks of the programming language. Subsequently, a parser takes charge of creating an abstract syntax tree (AST) from these tokens, capturing the syntactic structure of the input code. This AST is then fed into an evaluator, a key component responsible for interpreting the semantics of the code and producing results. Notably, error handling is integrated into the parsing stage, ensuring that any syntactic errors are promptly identified and reported to the user. The evaluated results, represented as storage objects, are then displayed to the user, providing feedback on the execution of their input. The use of an **Environment** object underscores the management of variable scopes and value storage during the REPL session, enhancing the overall functionality. This code embodies the essence of an interactive and dynamic programming environment, allowing users to iteratively input, evaluate, and observe the outcomes of their code snippets in real-time.

#### How to run the REPL:

Let's break down the process:

1. **CMake Configuration:**
   * The CMake version is specified (minimum required version 3.12), and the project is named "repl."
   * The C++ standard is set to 11, and the requirement for it to be strictly followed is enforced.
2. **Including Source Files and Directories:**
   * The file(GLOB ...) command is utilized to gather all subdirectories within the parent directory (assumed to be the project root) and include them in the build. This is achieved by iterating through the subdirectories and adding them to the include directories.
3. **Source Files Filtering:**
   * Another file(GLOB ...) command captures all source files within the subdirectories, excluding any files with names containing "test" (presumably unit tests). This filtered list of source files is used in building the executable.
4. **Executable Definition:**
   * An executable named "repl" is defined using the add\_executable command. The source files, including those filtered in the previous step and a "main.cc" file, are specified for compilation.
5. **Main Function:**
   * The "main.cc" file includes the "repl.h" header file, and the REPL is initiated by calling the REPL::start() function within the main function. A simple message, "Nulascript," is printed to the console before starting the REPL.
6. **Run Target and CMake Commands:**
   * A PHONY target named "run-repl" is defined to encapsulate the process of building and running the REPL. The associated commands, executed in the terminal, involve changing into the "nulascript/repl/build" directory, running CMake configuration, building the project, and finally executing the built executable.

In essence, this CMake configuration and main script provide a streamlined workflow for building and running the REPL. The use of CMake facilitates platform-independent project configuration, and the "run-repl" target simplifies the process for developers to test and interact with the REPL in a consistent manner. The main.cc file serves as the entry point to the program, initializing the REPL and signaling the beginning of the interactive session with the user.

A screenshot of a computer program

Description automatically generated

The provided sequence of commands and output demonstrates the execution of the Nulascript REPL (Read-Eval-Print Loop) using the **make run-repl** command. Let's break down each step:

1. **Command Execution:**
   * The command **make run-repl** is executed in the terminal.
2. **Change Directory:**
   * The command changes the working directory to **nulascript/repl/build**.
3. **CMake Configuration:**
   * **cmake .** configures the build system using the CMakeLists.txt file in the current directory (**.**). This step generates build files and prepares the environment for compilation.
4. **Build Project:**
   * **cmake --build .** initiates the build process. It compiles the source code, linking necessary libraries and generating the executable named **repl**.
5. **Run REPL:**
   * **./repl** executes the built REPL binary. This action initiates the Nulascript REPL environment, printing the introductory message "Nulascript:"
6. **REPL Input - For Loop:**
   * The user enters a Nulascript code snippet into the REPL: **for (def i = 5; i < 10; i + 1) { log("Hello World!"); }**
   * This code defines a for loop that iterates from 5 to 9 (inclusive) and logs the string "Hello World!" during each iteration.
7. **REPL Output:**
   * The REPL processes the input code, evaluates it, and executes the for loop. The output shows that "Hello World!" is logged five times, corresponding to the five iterations of the loop.
8. **Explanation:**
   * The Nulascript code snippet demonstrates a basic for loop construction, introducing the **def** keyword for variable definition (**i**), a loop condition (**i < 10**), and an increment expression (**i + 1**). Within the loop body, the **log** function is called to print "Hello World!" during each iteration.
   * The successful execution of the for loop illustrates the functionality of the Nulascript interpreter, parsing and executing code interactively within the REPL environment.

In summary, the provided terminal output indicates the successful build and execution of the Nulascript REPL, with the user inputting a simple for loop and observing the expected output in the form of repeated "Hello World!" messages.

### How can we execute an entire file of code in one go rather than utilizing a REPL for interactive code input?Top of Form

The Interpreter::interpret function is the linchpin of the Nulascript interpreter, acting as the gateway for executing entire code files.

1. **File Opening and Reading:**
   * The function takes a filename as a parameter and attempts to open the corresponding file using **std::ifstream**.
   * If the file fails to open, an error message is printed to the standard error stream (**std::cerr**), indicating the issue. The function then returns, preventing further execution.
   * If the file is successfully opened, the function reads its content line by line, appending each line to a **code** string.
2. **Environment Initialization:**
   * An **Environment** object is instantiated to manage variable scopes and store values during the interpretation process.
3. **Lexical and Syntax Analysis:**
   * The **code** string is used to initialize a **Lexer** object (**l**), which tokenizes the code.
   * The tokenized code is then passed to a **Parser** object (**p**), which constructs an abstract syntax tree (AST) representing the syntactic structure of the code. The resulting AST is stored in a **Program** object.
4. **Error Handling during Parsing:**
   * If there are any errors during the parsing process (syntax errors), the function prints each error message to the standard output (**std::cout**) and returns, preventing further execution.
5. **Evaluation:**
   * The parsed **Program** and the initialized **Environment** are passed to the **evaluate** function, which executes the code.
   * The result of the evaluation is stored in the **resolved** variable.
6. **Result Printing:**
   * Depending on the type of the result, the function prints a corresponding message to the standard output (**std::cout**).
   * If the result is of type **NIL**, "undefined" is printed. If the type is **EMPTY**, a newline is printed. If the type is **ERROR**, the error message is displayed.

In summary, the **interpret** method encapsulates the process of opening, reading, parsing, evaluating, and printing results for code stored in a file. It handles potential errors during file opening and parsing, providing feedback to the user and contributing to the robustness of the Nulascript interpreter.

The key difference between the two Interpreter::interpret method and the REPL:start method snippets lies in their purposes and usage contexts within the Nulascript project.

1. **Interpreter method:**
   * This method is responsible for interpreting and executing code stored in a file. It reads the content of the file, performs lexical and syntax analysis, evaluates the parsed program, and prints the results. This method is intended for batch processing of code from files.
2. **REPL (Read-Eval-Print Loop):**
   * The **REPL::start** function initiates a loop, prompting the user for input and processing each line of code on-the-fly. It utilizes the same components (Lexer, Parser, Evaluator) as the **interpret** method but is designed for interactive usage rather than file-based processing.

The first code snippet (**interpret** method) is tailored for batch processing of code stored in files, providing a way to execute complete scripts. The second code snippet (**REPL::start** method) is designed for interactive use, allowing users to input and immediately execute code line by line in a REPL environment. Both snippets share common components for lexical analysis, parsing, and evaluation, but they serve different use cases within the Nulascript project.

Fundamentally, the primary distinction between the two lies in their handling of return storage objects. The REPL (Read-Eval-Print Loop) diligently evaluates each return storage object, presenting its string representation. On the other hand, interpreters follow a different paradigm. They are designed to execute output only in specific scenarios—either upon invoking a particular standard library function or when the program's execution is abruptly halted due to an error. In the latter case, the error is stringified and displayed. This distinction underscores the contrasting behaviors of the REPL, which actively processes and outputs results, and interpreters, which adhere to a more reserved output strategy triggered by specific events in the program's execution.

#### How to run the interpreter:

Similarly to the REPL, a CMake configuration outlines the steps to build the Nulascript interpreter. Here's an overview of the build process:

1. **CMake Version Requirement:** Specifies the minimum version of CMake required for the build.
2. **Project Definition:** Defines the project name as "nulascript."
3. **Setting C++ Standards:** Sets the C++ standard to 11 and specifies that it is required.
4. **Including Subdirectories:** Collects all subdirectories within the project directory and includes them. This allows access to header files and resources in subdirectories.
5. **Filtering Source Files:** Gathers all source files (\*.cc) in the project directory, excluding those with "test" in their filenames. This step helps exclude test files from the build.
6. **Creating Executable:** Generates an executable named "nulascript" from the specified source files, including "main.cc."
7. **Build Commands:** Introduces a custom target, "build-interpreter," which, when invoked, navigates to the "nulascript/interpreter/build" directory, configures the build using CMake, performs the build, and then copies the resulting "nulascript" executable to the "../../../bin" directory.

This build configuration provides an organized and streamlined process to compile the Nulascript interpreter. It ensures that test files are excluded from the build, and a custom target simplifies the build steps and facilitates the deployment of the executable to a specific directory.

A screen shot of a computer program

Description automatically generated

## Nulascript semantics

Nulascript, a programming language with a distinctive syntax and semantics, introduces innovative features that set it apart from traditional programming languages. In this exploration, we delve deeper into Nulascript's semantics, providing an extended understanding of its unique design choices and the historical context that influenced its development.

**1. For Loops with References: A Historical Perspective**

Nulascript's **for** loops break away from conventional syntax by consolidating the initialization, condition, and increment parts into a single header. This approach draws inspiration from early programming languages like C, where loop headers were concise, yet powerful. However, Nulascript takes a step further by introducing an explicit increment part, adding clarity to loop logic.

References in **for** loops showcase a historical evolution in programming languages' handling of variables. Traditionally, loops operated on the value of a variable, but Nulascript introduces the concept of references, allowing variables to be assigned and manipulated by reference. This design choice reflects a broader trend in modern languages towards more expressive and flexible variable management.

**2. Reference Handling: Navigating the Evolution**

References in Nulascript offer a nuanced approach to variable manipulation. Unlike languages that strictly require dereferencing when working with references, Nulascript provides a more flexible model. In the initialization header of a **for** loop, dereferencing is optional, emphasizing readability and reducing verbosity. This echoes the evolution of languages like C++ and Rust, which introduced smart pointers and borrow-checking mechanisms to enhance memory safety.

Nulascript's semantics regarding references further highlight the language's commitment to balancing simplicity with power. By allowing assignments to references without explicit dereferencing, Nulascript streamlines code while maintaining the essential understanding that the reference is being updated, aligning with the principles of languages like Python that aim for code clarity and brevity.

**3. Logging Function: Bridging Clarity and Functionality**

The **log** function in Nulascript serves as a bridge between simplicity and functionality. While logging is a common feature in many programming languages, Nulascript's decision to omit the need for dereferencing when working with references is a deliberate departure from the conventions of languages like C and C++, where explicit pointer dereferencing is required. This approach aligns with the language's overarching goal of minimizing boilerplate code and reducing cognitive load on the developer.

The logging function in Nulascript also draws inspiration from languages like JavaScript, where console logging is a ubiquitous and developer-friendly feature. By allowing multiple values to be logged on the same line, Nulascript follows the footsteps of scripting languages that prioritize concise and expressive syntax for quick debugging and output visualization.

**4. Conditionals: Boolean Expressiveness and Beyond**

Nulascript's choice to support only the **if** statement for conditionals is reminiscent of languages like Python, which favor a simple and readable syntax. The language incorporates boolean operators (**is**, **not**, **==**, **!=**) to enhance expressiveness in conditional statements.

The concept of truthy values, where non-zero numbers are considered truthy, is deeply rooted in the history of programming languages. It traces back to the C programming language, where zero is treated as false, and any non-zero value is considered true. Nulascript's adoption of truthy values aligns with the simplicity-driven design seen in languages that prioritize intuitive and minimalistic syntax.

**5. Closures: A Modern Touch in Language Design**

Closures in Nulascript represent a modern touch in language design, aligning with the paradigm shift towards functional programming concepts. The use of the **func** keyword to define closures draws inspiration from languages like Swift and Kotlin, where concise syntax and first-class functions have become integral to modern development.

The ability of closures to capture variables from their outer scope enhances Nulascript's flexibility and introduces a powerful mechanism for creating reusable and parameterized code blocks. This concept has historical roots in Lisp and Scheme, early pioneers of functional programming, where closures, or lambda expressions, played a pivotal role in defining functions with dynamic behaviors.

**Conclusion: Nulascript's Semantics in the Tapestry of Programming Languages**

Nulascript's semantics weave together historical influences and modern design principles, creating a unique tapestry in the landscape of programming languages. By reimagining traditional constructs like **for** loops and embracing modern concepts like closures and truthy values, Nulascript offers a distinctive programming experience. As the language evolves, its design choices will likely continue to reflect the ongoing dialogue between the simplicity of early languages and the expressiveness demanded by contemporary development practices.