Michael Bozhilov

Senior Project I

Fall 23/24

Last page: 51

# Nulascript

## Introduction:

**Nulascript** is a dynamically typed programming language that allows memory access through pointers. Pointers represent a crucial concept in computer programming, particularly in low-level languages like C. They store memory addresses rather than actual values. This distinction becomes important in terms of efficiency because it allows you to work with data in memory directly without making unnecessary copies. For example, in high-level scripting languages running on virtual machines and in deployment pipelines, avoiding data duplication and having direct access to memory locations can lead to faster 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, it it’s 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**:

// avoid copying large inputs

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. This design choice is made to prevent the unnecessary duplication of exceptionally lengthy code segments into memory. By using a reference to the input, the code remains memory-efficient, even when dealing with large code files.

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 to explaining 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 would start at 0:** The Lexer begins processing at the first character of the input string, which is 'l' in the keyword 'let'.

**readPos would start at 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 would be assigned '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 would 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 often used to indicate the end of parsing.

**ILLEGAL**: This token is typically 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, typically used to assign 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 pointers.

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

**PIPE**: Similarly to how ‘|’ is used in shell scripting, the idea for this token is to directly pipe passed input to stdout.

**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 assigning.  
  
**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**;

**case** ')':

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

**break**;

**case** '{':

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

**break**;

**case** '}':

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

**break**;

**case** '\*':

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

**break**;

**case** '&':

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

**break**;

**case** '|':

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

**break**;

**case** '!':

currentToken = newToken(TokenType::BANG\_OR\_NOT, ch);

**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 language syntax.

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.

## 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, statements, and declarations.

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 {

**public**:

**void** statementNode();

};

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

**public**:

**void** expressionNode();

};

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.

**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.

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 a 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.