Ryan Dreyer

05/01/24

Portfolio

Teamwork:

My team member and I played a crucial role in building the compiler that aimed to lay the foundation for a new coding language. This project required careful planning and coordination to ensure its success. In this paper, I will discuss our individual responsibilities and contributions towards the development of this project.

As part of this project, my responsibility was to develop the compiler. Specifically, I was assigned the task of coding the Parser which involved creating expressions, statements, and Cat Script. Additionally, I was responsible for building the byte code. However, I faced challenges while developing the byte code due to the lack of documentation to assist me in translating my byte code into Java. Fortunately, with the help of Professor Carson Gross, I was able to find relevant documentation on building the parser expressions and statements, as well as the Cat Script logic required to construct our compiler.

One of the team members, Team member 1, was assigned to work on the technical documentation and test cases for evaluating the compiler logic. The technical documentation prepared by Team member 1 provides a detailed explanation of every class present in Expressions and Statements, along with code snippets that demonstrate how the logic works in different scenarios.

Design pattern:

```
private static final HashMap<CatscriptType, ListType>
listTypeCache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type)
{
    ListType listType = listTypeCache.get(type);
    if ((listType == null)){
        listType = new ListType(type);
        listTypeCache.put(type, listType);
    }
    return listType;
}
```

For this project, we chose a design partner called memoization. Memoization is a technique that helps speed up the program by storing the results of function calls and returning the cached result when the same input occurs again. This technique can be used in recursive parsing to avoid repeated processing of the same input, which is common in recursive parsers dealing with complex or ambiguous grammars.

Memoization was incorporated into our compiler to address several issues. Specifically, it helped reduce redundancy, improve efficiency, and handle left recursion and ambiguity. Recursive descent parsers often end up re-evaluating the same sub-problems multiple times as they attempt to parse an input according to grammar rules. By caching these results, memoization ensured that each unique sub-problem was computed only once. This greatly improved efficiency by avoiding redundant computations and significantly reducing the time complexity of our parser. Additionally, recursive descent parsers can struggle with left recursion and ambiguous grammar. Without special handling, left recursive rules can lead to infinite recursion and stack overflows, while ambiguous grammars can result in the same structure being parsed multiple times in different contexts. Memoization helped manage these issues by remembering the outcomes of parsing attempts, preventing infinite loops and repeated work.

Technical Writing:

Introduction

Catscript is a statically-typed scripting language. This document covers the expressions and statements that the language offers, in the order they appear in the compiler source code.

Features

Addition Expression:

1 + 4

cat + script

An Additive Expression in Catscript are used to combine multiple operands and/or integer variables of the same type using arithmetic addition or subtraction operations. It is a binary expression, in that it has a left and right hand side that are evaluated. This expression type can be used for tasks such as computing totals, calculating differences, or adjusting numerical values dynamically. The addition operator also supports string concatenation, which appends the left and right hand side together, enhancing flexibility in generating complex strings.

Boolean Literal Expression:

bool foo = true

bool bar = false

A Boolean Literal Expression in Catscript evaluates to either true or false, representing logical values. They are denoted with the keyword "bool" during variable assignment. Boolean literals can be used in conditional statements, loops, and expressing binary logical operations to control program flow based on certain conditions, therefore aiding in decision-making processes.

Comparison Expression:

5 > 3

A Comparison Expression in Catscript is a type of boolean expression that evaluates the relationship between two values/ integer variables, returning a boolean result indicating

whether the comparison is true or false. Operators such as greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=) are supported in Catscript. Comparison Expressions enable users to perform logical comparisons between same-typed data elements. These expressions allow for implementing conditional statements, loops, and other control structures.

Equality Expression

if(x == y){...}

while(x != z){...}

An Equality Expression in Catscript is a type of boolean expression that evaluates whether two values are equal or not. Catscript supports the use of double equals (==) and not equal (!=) to compare the values of two operands. This expression returns a boolean value that is true if the operands are (not) equal, false otherwise. Equality expressions allow for decision-making processes within programs, enabling conditional branching and logical operations based on the equality or inequality of values.

Factor Expression

2 * 4

8 / 4

A Factor Expression in Catscript is an integer expression that enables the performance of a multiplication (*) or division operation (/). These expressions are similar to additive expressions in that they are binary expressions that facilitate arithmetic calculations in programs, allowing users to dynamically combine and manipulate numeric data through the multiplication and division operators.

Function Call Expression

x(true, 4)

A Function Call Expression in Catscript initiates the execution of a specific function defined elsewhere in the code. It serves as a mechanism to invoke predefined routines or procedures to perform specific tasks or operations within the program. By providing

arguments (if required), the function call expression supplies necessary input data to the function, enabling it to execute its logic and potentially return a result.

Identifier Expression

var x

bool y

An Identifier Expression in Catscript refers to the usage of a variable or a function's name to access its stored value or reference. When encountered within code, the Identifier Expression triggers a lookup process via a Symbol Table, retrieving the associated data for further processing. This type of expression enables the manipulation and utilization of variables and functions.

Integer Literal Expression

int y

An Integer Literal Expression in Catscript represents a specific integer within the code. Integer literals can be used in various contexts within a program, including arithmetic operations, variable assignments, and conditional statements, providing a straightforward means to work with numerical data. These expressions are denoted with the keyword "int" during variable assignment.

List Literal Expression

list<int> x = [1, 2, 3]

list y

A List Literal Expression in Catscript allows users to create lists containing multiple elements within their code directly. By specifying the elements within square brackets and separating them with commas, users can initialize lists without the need for separate variable declarations. Lists can be given a type through angle brackets, followed by the name of the type before the variable name, otherwise the type will be inferred. This expression provides a concise and convenient means to represent collections of data. List literals do not support the addition of new values after being assigned elements.

Null Literal Expression

x = null

A Null Literal Expression in Catscript represents the absence of a value. It is commonly used to denote that a variable or object does not currently reference any valid data. Null literals serve as placeholders, allowing users to initialize variables without assigning a specific value. This expression is can be used for handling cases where data may be missing or unavailable, providing a standardized way to represent such scenarios within code.

Parenthesized Expression

(1 + 2)

A Parenthesized Expression in Catscript serves the purpose of altering the order of operations in mathematical or logical expressions. By enclosing certain parts of an expression within parentheses, users can explicitly specify which operations should be performed first, overriding the default precedence rules. This allows for precise control over the evaluation sequence, reducing ambiguity in complex expressions.

String Literal Expression

y = "bar"

A String Literal Expression in Capscript represents a sequence of characters enclosed within quotation marks, denoting plaintext data. It is denoted by the keyword "string". It serves as a method for embedding fixed strings directly into code. String literals enable users to specify text values such as messages, file paths, or database queries.

Syntax Error Expression

var x "foo]

A Syntax Error Expression in Catscript occurs when the syntax rules of Catscript expressions are violated. It indicates that there's an issue with the structure/arrangement of expressions, making it unintelligible to the Catscript compiler. These errors often arise from misspelled keywords, missing punctuation, or incorrect usage of Catscript

expressions. Syntax Error Expressions halt the compilation or execution process, prompting users to review and rectify the code to adhere to Catscript's syntax rules.

Type Literal Expression

var x

function foo(a : object, b : bool, c : int)

A Type Literal Expression in Catscript is a construct used to explicitly represent a data type within the code. It allows users to specify the type of a variable directly within the program, aiding in type inference and enhancing code clarity. Variables and functions can have the following types: integer, string, boolean, object, and null. Functions exclusively can have the type void, while variables can include the keyword "var", where the type will be inferred by Catscript.

Unary Expression

not true

-5

A Unary Expression in Catscript operates on a single operand to perform a specific operation. These expressions allow users to manipulate data in various ways, such as changing its sign, or performing logical negation. Unary expressions are versatile tools for modifying the value or state of variables and data structures. They can be used in conjunction with other operators or within control flow statements to achieve desired program behavior.

Assignment Statement

int x = 5

An assignment statement is a foundational concept in Catscript programming that allows users to assign a value to a variable. It comprises of three or four essential parts, including the variable type (when it's being declared for the first time), variable name, assignment operator (=), and the value that the variable is being assigned. When executed, the assignment statement evaluates the expression on the right-hand side and stores its result in the memory location associated with the variable on the left-hand side.

This process replaces any previous value that might have been stored in the memory location. Assignment statements enable the transfer of values between variables, which is critical in conducting complex computations and logical operations. Such operations range from basic arithmetic calculations to intricate algorithmic procedures. Assignment statements let users perform calculations, store user input, and manage program state by assigning values to variables.

For Statement

for(int x in y){print(x)}

The for statement in Catscript is a control flow structure that allows the repeated execution of a block of code based on a specific condition. It consists of three parts: initialization of a loop variable, an expression, and statements that are looped over. During the initialization phase, variables are usually initialized only for the loop, setting up the initial state for the loop. Between the loop variable and the expression is the keyword "in", which separates the loop variable and the expression.

The for loop then contains an expression that is evaluated and iterated over, determining whether the loop should continue executing or not. If the condition evaluates to true, the code block of statements associated with the for statement is executed; otherwise, the loop terminates. This process continues until the condition becomes false, at which point the loop ends, and the program proceeds to the next statement after the for loop. For loops in Catscript are particularly useful when iterating over a sequence of elements, such as lists.

Function Call Statement

foo("hello", null)

A Function Call Statement in Catscript initiates the execution of a specific function defined elsewhere in the code. It serves as a command to the program to execute the set of instructions encapsulated within that function. When encountered during program execution, the function call statement prompts the program to temporarily pause its current execution flow, transfer control to the designated function, and execute the code block associated with it. This process enables modular programming, as functions can be defined once and called multiple times from different parts of the program, encouraging re-usability.

A Function Call Statement typically includes the name of the function being called along with any required arguments or parameters enclosed within parentheses. These arguments are Function Call Expressions that provide input values necessary for the function to perform its tasks effectively. Once the function execution is completed, the program resumes its operation from the point immediately following the function call statement.

Function Definition Statement

function x(a : bool, b : int){...}

function x(a, b, c) : void {...}

A Function Definition Statement in Catscript is a fundamental tool for creating reusable blocks of code that perform specific tasks. It begins with the keyword "function", followed by its name, and then parameters enclosed in parentheses. These parameters act as placeholders for values to be passed into the function when it's called. Optional types can also be specified for these parameters. The function type can also be optionally included.

The body of the function, encapsulated within curly braces, contains the instructions that dictate what the function does when invoked. These instructions can be used to implement algorithms, manipulate data, or perform any other task that requires a sequence of actions to be encapsulated into a single entity. Functions are incredibly useful in eliminating redundant code segments, as the same functionality can be invoked multiple times without repetition.

If Statement

if(x > y){...}

An "if" statement is a crucial tool for making decisions in Catscript programming. It allows for the execution of specific code blocks based on the result of a condition evaluation. The statement's structure consists of the keyword "if" followed by a condition in parentheses. If the condition is true, the statement(s) associated with the "if" statement is executed; otherwise, it is skipped. In case the condition is false, alternative code blocks specified by "else if" or "else" clauses may be executed.

By using conditional statements, Catscript users can create logical workflows and algorithms within programs. This functionality allows for branching logic within the

program, enabling different paths of execution based on varying conditions. By employing conditional statements, Catscript users can create logical workflows and algorithms within programs. Nested if statements and logical operators can also be combined to create intricate decision-making structures, offering a powerful mechanism for building robust and adaptable software systems. This enhances the program's flexibility and responsiveness, making it more adaptable to different scenarios and inputs.

Print Statement

```
print(1 + 1)
```

A Print Statement in Catscript serves as a fundamental tool for displaying information to users. Print Statements are denoted by the keyword "print", followed by content encapsulated within parenthesis. It allows users to output text, variables, or expressions to the terminal, enabling visibility into the program's state during runtime. This statement can be used for debugging and providing feedback to users, facilitating communication between the program and its users. Users can verify the correctness of their code, track variables' values, and ensure that the program behaves as intended.

Due to Catscript evaluating expressions at runtime, Print Statements are not solely limited to textual output; they can also format data for clearer presentation. For example, users can concatenate strings with variables or format output using placeholders to create structured and readable messages. Additionally, print statements are often used for logging purposes, recording significant events or errors during program execution. Print Statements offer visibility and insight into the inner workings of a software system.

Return Statement

```
return false
```

In Catscript, a return statement is utilized to exit a function, and it can also pass a value back to the caller. This statement indicates the end of a function's execution and can send a result to the part of the program that invoked it. To use a return statement, you must use the keyword "return" followed by an expression, unless the function is of type VOID. The returned value can be any data type supported by Catscript, including integers, booleans, and objects.

Return statements allow for conditional branching and function termination based on specific conditions. When a return statement is encountered, control is transferred back to the point in the code where the function was called, which makes it possible to integrate functions within larger programs seamlessly. The returned values can be utilized in subsequent operations or passed as arguments to other functions.

Syntax Error Statement

A Syntax Error Statement in Catscript is a notification to the user that there is a violation of Catscript's syntax rules. It occurs when the compiler encounters code that does not meet the expected structure or format. This may include incorrectly spelled keywords, misplaced punctuation, missing parentheses, or incorrect use of Catscript's constructs.

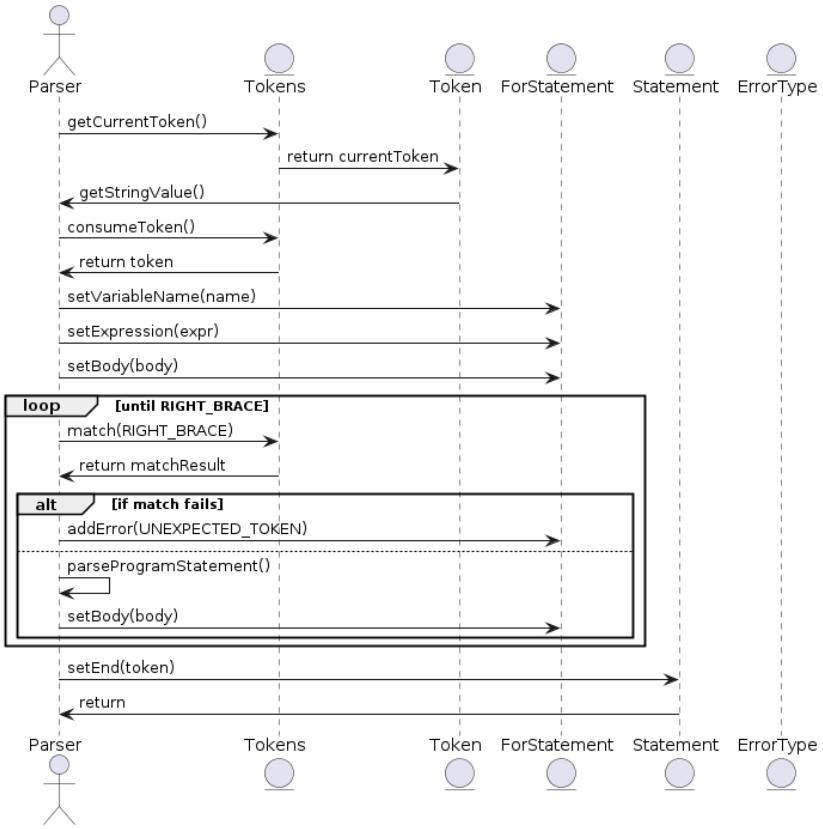
When a Syntax Error Statement is triggered, the compiler stops the program's execution and displays an error message that provides details about the syntax error and its position within the code. This message is a diagnostic tool that helps users identify and correct the error, guiding them towards resolving the syntactical issues in their code. Once the error is identified, users can make the necessary corrections to ensure that the code complies with the syntax requirements for successful compilation.

Variable Statement

In Catscript, Variable Statements are used for declaring and defining variables within a program. This helps in allocating memory space for data storage and associating identifiers with these memory locations. Variable Statements enable users to specify the type and initial value of variables, which make data manipulation and organization more efficient in the code.

They also provide clarity and structure to code by introducing a naming scheme that reflects the purpose and meaning of each variable. Furthermore, variable declarations enforce type constraints, which enable the compiler to perform type checking and ensure data integrity during program execution.

UML:



The code above is a `ParseForStatement` that works by parsing a `for` loop statement. Initially, the parser checks if the current token is the "for" keyword by invoking `Tokens.getCurrentToken()`. If the token returned by `Tokens` corresponds to "for", the parser consumes the "for" keyword using `Tokens.consumeToken()` and proceeds to set up the `ForStatement`. To set up the `ForStatement`, the parser skips over the left parenthesis by consuming tokens with `Tokens.consumeToken()` and sets the variable name in the `ForStatement` using `ForStatement.setVariableName()`. Then, it consumes more tokens for the equal sign and checks it with `Tokens.consumeToken()`. After that, the parser parses the expression by calling another method, `parseExpression()`, and sets it in the `ForStatement` using `ForStatement.setExpression()`. Next, the parser enters a loop to parse all statements within the for loop. It checks for the `RIGHT_BRACE` token using `Tokens.match()`. If the token is not the `RIGHT_BRACE`, it parses a statement using `parseProgramStatement()` and adds it to the `ForStatement` body using `ForStatement.setBody()`. If `Tokens.match()` finds an `EOF` token instead of `RIGHT_BRACE`, `ForStatement` adds an error using `ForStatement.addError()`. Finally, after parsing all statements within the loop, the parser sets the end token of the `ForStatement` using `Statement.setEnd()` after consuming the right brace.

Design trade-offs recursive descent vs. a parser generator?:

We used recursive descent for our compiler because of the many positives with recursive descent. The other way my team member and I could have built this was by using a parser generator. The main reason we chose to use recursive descent in our project was because of the huge amount of control we had over building the project. Another positive was flexibility. The flexibility allowed my team member and I to make tweaks to our program for practical use cases much easier. Finally, transparency was another advantage we saw when building in recursive decent. This allowed my partner and I to understand our code and make the technical writing easier to translate, so people looking at our code for the first time would understand it quickly.

The disadvantages my partner and I noticed were complexity and maintainability. The complexity of recursive descent made writing code very hard to write at times because I would have to write in such a way, that wasn't normal practice for building a compiler in Java. The maintainability of recurse descent also made it difficult to use at times, because if we were to edit or change something in the code. It would break most to all of the code we wrote in a class or a set of functions and tests.

The parser generator was another option we could have used for our project, but chose not to. We found the negatives of the parse-generator would affect the build my partner and I wanted to make. The disadvantages were the lack of control and debugging. having less control over the parsing process and error handling would affect the flexibility of the hand-written parser we wanted to write for the compiler. Finally, debugging the parser generator would be too much for us to handle as debugging generated code can be more challenging, as the source code does not directly correspond to the grammar rules.

Software development life cycle model:

My team and I used the Test-Driven Development technique to build the compiler. This approach helped us to test the code in small increments, which reduced the time we spent on debugging. We were able to isolate parts of the code and test them separately until we achieved the desired output. This method allowed us to build the compiler like solving a puzzle, piece by piece, and expand on it as we progressed. However, we did face a disadvantage to this testing method. When we encountered a scenario for which we had not yet built a necessary test, such as the code for comparison, we could not proceed to the next test until the missing one was developed. We had to follow a checklist to ensure that everything was completed before moving ahead with the next tests.