Catscript Compiler

Grace A. Parker

Gianforte School of Computing

CSCI 468 - Compilers

Professor - Carson Gross

Team member - Ryan Dreyer

Spring 2024

Section 1: Program

The source code for my project can be found in the source.zip file.

Section 2: Teamwork

For the capstone project, I worked with one other team member to develop a small compiler and its documentation. My primary responsibility was to design and implement the compiler's core functionality, which includes tokenizing, parsing, evaluation, and code generation. This involved understanding the project requirements, translating the grammar into working code, and implementing algorithms and data structures to achieve the desired functionality. I went through the test suite to ensure my code compiled a given input correctly. Throughout the development process, I ensured that the code was well-structured, and efficiently implemented. This took up 95% of the project time. Meanwhile, team member 1 played a role in documenting the technical aspects of the project. He captured the functionality and use cases of each Catscript expression and statement. Team member 1 provided examples that outlined not only the syntax, but also a potential use case to give any user a full understanding of all the features that the Catscript compiler offers. Each expression and statement was provided a description of what the expression/statement is supposed to be, how it functions, and the end results of the expression/statement functionality. Team member 1's efforts were able to satisfy section 4 of the capstone requirements.

Additionally, team member 1 contributed to the project by writing a set of three test cases to validate the compiler's behavior. The first test he wrote assessed my Catscript compiler's performance and its limits. The test scenario involved creating an extremely large list with integers ranging from 1 to 999, each one separated by commas, and culminating to 999 elements in total. The second test checked how my parser handled two Identifier Expressions that had "identical" names, but different case sensitivities. The example he wrote asserted that identifier expressions "var" and "Var" were considered two different expressions, rather than considered the same expression. Finally, team member 1 wrote a test scenario that checked how my parser handled an integer variable statement with an expression equal to zero. The goal was to ensure my parser did not consider zero and null to be the same thing. Team member 1's tests acknowledged edge cases and scenarios that I did not consider before, so I made sure my Catscript compiler was able to meet the requirements of each test to create a more robust compiler. This, as well as the technical document, took up 5% of the project time.

The test cases discussed are located in src/test/java/edu/montana/csci/csci468/demo/PartnerTest.java.

Section 3: Design pattern

```
34      private static final HashMap<CatscriptType, ListType> listTypeCache = new HashMap<>();
        3 usages  gaparker716 +1
35  @   public static CatscriptType getListType(CatscriptType type) {
36          ListType listType = listTypeCache.get(type);
37          if(listType == null){
38              listType = new ListType(type);
39              listTypeCache.put(type, listType);
40          }
41          return listType;
42      }
```

This section of code here, located in src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java, shows the memoization design pattern. Memoization is a technique used to optimize the performance of functions, particularly in scenarios where the same computation is performed repeatedly with the same inputs. In memoization, you create a data structure to cache the results of the function call. When the function is called with a set of arguments, the function first checks whether it has already computed and stored the result for those arguments in the cache. If it finds a cached result, it returns that result instead of recomputing it. If not, it computes the result and stores it in the cache, then returns it.

If I did not use memoization, I could have not made a new HashMap data structure, and instead had the getListType function reduced to one line, which would have consisted of returning a new ListType(type) instance every time. It would be easier to code, but also less efficient because of how the rest of Catscript deals with parsing type literals, as well as the list type specifically. The list type is different from the rest of the types Catscript offers, in that it has its own subclass to help with giving a list of things a type, if needed (i.e. list<int>, list<object>). However, because of that and the recursive descent parsing design of Catscript, any instance of ‘list’ gets called in both the TypeLiteral class, and the ListType class, which would result in two instances of the same list. Having Catscript “remember” that a list has already been made would mean that there are no unneeded extras, and thus more time and space efficiency.

Section 4: Technical Writing

Catscript Guide

This document is a guide for Catscript to satisfy capstone requirement 4

Introduction

Catscript is a programming language that is designed to be simple and easy to learn, with a syntax that is similar to Javascript. It allows developers to quickly and easily create scripts and applications for a wide range of purposes. As an example, here is some sample code written in Catscript:

```
var x = "foo"
print(x)
```

Features

Expressions

Additive Expression

```
1+4=5
```

The additive expression is a binary expression that checks the left and right hand sides arguments. It begins by calling parsePrimaryExpression() which will parse something. Next, if the tokens list matches either the PLUS or MINUS token type, the token is consumed.

Boolean Literal Expression

```
foo = true
```

The Boolean Literal Expression is an expression that is a subclass of the primary expression. It also dictates whether something is true or false.

Comparison Expression

```
1<5
10>7
```

The Comparison Expression is a binary expression that checks the left and right hand side arguments. it begins by calling parsePrimaryExpression() which will parse something. Next if the token list matches either the GreaterThan, GreaterThanOrEquals, LessThan, or LessThanOrEquals token type, the token is consumed.

Equality Expression

```
x=10
x=9
```

The Equality expression is a binary expression that checks the token operator to see if it is EqualEqual or NotEqual. Then it checks if the right and left hand arguments are equal or not equal, by the operator

Factor Expression

```
4/2 =2

3*3 =9
```

The Factor Expression is a binary expression that checks the left and right hand side arguments. It checks whether the arguments are integers and then multiples or divides the integer arguments.

Function Call Expression

```
foo(true, 1)
```

The Function Call expression has a name and list of expressions in parentheses, called parameters. The function call must include the same amount of parameters, and each parameter has to have a type that must match what was included in the original function definition

Identifier Expression

```
var x
```

The Identifier Expression is a subclass of the parsePrimaryExpression. It checks the type of token and the name to see if it's an identifier.

Integer Literal Expression

```
int x = 3
```

The Integers Literal Expression is a subclass of the parsePrimaryExpression. it checks if the token is of type integer.

List Literal Expression

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

The expression is a subclass of parseTypeExpression. This makes a linked list and adds expressions into the list, then gets the values from the linked list.

Null Literal Expression

```
x = null
```

The expression checks if the type is null. If it's null then Catscript returns null.

Parenthesized Expression

```
(1+2)
```

The expression is a subclass of primary expression that appends parenthesis to the child expression.

String Literal Expression

```
x = "dog"
```

The expression is a subclass of parsePrimaryExpression. This checks if the token type is a string, then adds quotes around the value.

Syntax Error Expression

```
var x ="foo]
```

The expression is a subclass of parsePrimaryExpression. If it can not tell what expression is, it returns an error.

Type Literal

```
int x
var y
```

The expression encompasses all literal type expressions. It can set and return the type for an expression.

Unary Expression

```
if(not 3){...}
var x = -3
```

Unary expressions are expressions with a single “argument” expression. It evaluates the right hand side, then applies the operator to that value.

Statements

Assignment Statement

```
var age = 30

print("The age is: " + age) // Outputs: The age is: 30
```

In Catscript, the Assignment Statement is a fundamental construct that is pivotal for controlling and manipulating data within the program. This capability is essential for any programming task that involves data manipulation, from simple calculations to controlling complex application states. The structure of an assignment in Catscript typically involves a variable name followed by an equal sign (=) and an expression. The expression on the right-hand side of the equal sign is evaluated first; its result is then placed into the variable named on the left-hand side. This process updates the variable's current value or initializes it if the variable was not previously set.

The Assignment Statement in Catscript not only serves the purpose of initializing and updating variables but also facilitates the creation of dynamic and responsive applications. For instance, in a loop or a function, assignment statements can be used to update the values of variables based on user input, calculation results, or responses from external systems, allowing for real-time data processing and interaction. Additionally, it's important to distinguish between assignment and comparison; while assignment uses one equal sign (=), comparisons typically use two (==) or three (===) depending on the language specifics about type coercion and equality. This distinction is crucial to avoid common programming errors, such as inadvertently replacing a variable's value when intending to perform a comparison.

For Statement

```
var numbers = [1, 2, 3, 4, 5]

var sum = 0

for (var number in numbers) {
sum += number // Add each number to the sum
}

print("The sum of the numbers is: " + sum)
```

In Catscript, the For Statement is an essential control flow mechanism that facilitates the iterative execution of code blocks. The For Statement begins with the "for" keyword, followed by a variable declaration which serves as the iterator, the "in" keyword, and an iterable expression that defines the set of values the iterator will take on. This entire setup is enclosed in parentheses. The block of statements to be executed repeatedly is enclosed within curly braces {}. Each iteration adjusts the value of the iterator according to the sequence or
range specified in the iterable expression, executing the enclosed block of statements for each value. This makes the For Statement extremely useful for tasks that require repetitive actions, such as processing items in a list, generating repeated outputs, or performing operations on arrays where each element needs similar manipulation.

The For Statement in Catscript not only simplifies code that would otherwise need explicit and potentially error prone repetition but also enhances readability and maintainability. By containing all details of the iteration— from initialization, condition checking, and final execution within a single statement—it provides a clear and concise overview of what the loop does, which is invaluable in both the debugging and developing stages of a project. In scenarios ranging from simple repetitive tasks to complex data processing workflows. A For Statement can iterate over a range of numbers to perform calculations, traverse arrays or collections to apply functions, or even handle nested loops for multi-dimensional data structures.

Function Call statement

```
bar(x, y)
```

In Catscript, the Function Call Statement is a fundamental feature that facilitates the execution of predefined functions within a program. This statement allows developers to leverage the modularity and reusability aspects of programming by enabling them to call a function anywhere within their code, as needed. When a function call is made, Catscript processes the arguments provided, passes them to the specified function, and executes its body. During this execution, the function can perform a variety of tasks such as calculations, processing data, or manipulating objects based on the arguments it receives. The beauty of this mechanism lies in its simplicity and power—functions. These can return values back to the caller, allowing for the results of complex operations to be used immediately or stored for later use. This capability is essential for maintaining clean and efficient code, reducing redundancy, and enhancing the clarity of the programming logic.

The Function Call Statement in Catscript not only supports calling functions with scalar values like numbers and strings but also allows passing complex data types such as arrays, objects, or even other functions as arguments. This flexibility makes it an invaluable tool for building sophisticated applications that require a high degree of interactivity and functionality. The function call acts as a bridge between different parts of an application, enabling them to communicate and operate cohesively. It exemplifies the concept of abstraction in software development, where details of the code implementation are hidden, simplifying the development process and reducing potential errors.

Function Definition Statement

```
function calculateArea(width, height) {
var area = width * height
return area
}

var rectangleArea = calculateArea(5, 3)
print("Area of the rectangle: " + rectangleArea)
```

In Catscript, the Function Definition Statement is crucial for encapsulating reusable code that performs specific tasks or calculations. It is defined using a clear syntax that typically includes a keyword such as function, followed by a unique identifier (the function name), and a set of parentheses enclosing any parameters. These parameters act as placeholders that accept values passed into the function, enabling it to operate on varying data with each call. The body of the function, enclosed by curly braces, contains a sequence of statements that define the function's behavior. This may include operations such as variable assignments, loops, conditional statements, and calls to other functions.

The return statement within a function definition is pivotal as it specifies the output of the function. This output can be the result of computations performed within the function or it could simply be a value needed as part of a larger operation. The return statement effectively concludes the function's execution and passes control back to the calling context, along with the return value. This mechanism is essential for functions to interact with other parts of the program. For instance, a function might process input data and return a formatted string, a calculated numeric value, or even another function. The use of function definitions in Catscript thus allows for building complex, highly interactive systems where functions rely on each other to perform tasks, process data, and propagate results throughout the application.

If Statement

```
var x = 10
var y = 10

if (x == y) {
	return y
} else {
	return x
}
```

In Catscript, the parseIfStatement function is a pivotal component of the language's parsing mechanism, specifically designed to handle conditional logic expressed through If Statements. This function operates by first evaluating a conditional expression provided between the parentheses of an "if" keyword. If the expression evaluates to true, the function then executes a block of code enclosed within curly braces immediately following the condition. The ability of parseIfStatement to accurately evaluate the expression and subsequently determine the flow of execution is crucial, as it directly influences the decision-making processes within the program.

The robustness of the parseIfStatement function extends beyond just executing a single block of code. In more complex scenarios, it can also handle else and else if clauses, providing multiple pathways for code execution which are dependent on various conditions. This flexibility enhances the functionality of Catscript by allowing it
to support complex conditional structures seamlessly within its syntax. For example, an application can execute one block of code if a user's input is within a certain range, another block if the input is outside of that range, and yet another block if no input is received. Thus, the parseIfStatement not only supports fundamental programming paradigms but also empowers developers to write clean, efficient, and maintainable code that reacts dynamically to the diverse conditions encountered during execution. This function embodies the
essence of control flow in programming, enabling both simple and complex conditional logic to be implemented with precision and ease.

Print Statement

```
print("Hello World")
```

In Catscript, the Print Statement is an essential tool for outputting the results of evaluated expressions directly to the console or another standard output device. This functionality is particularly crucial during the development and debugging phases of software creation, where developers need to inspect values, monitor program flow, or verify correct operation. The Print Statement works by evaluating any expression enclosed within its parentheses and then converting the resulting value into a human-readable format, which is then displayed on the screen. For example, the command print("The sum is " + (a + b)); will evaluate the expression (a + b), concatenate it with the string "The sum is ", and output the complete message. This straightforward mechanism enables rapid feedback on the state of the program's variables and logic at any point in the execution, making it a fundamental feature in the programmer’s toolkit for tracing and diagnostics

The utility of the Print Statement extends beyond simple debug messages. This can be used in Catscript to interact with the user by providing dynamic output based on user input or programmatic conditions. For example, in a user-driven menu system, print statements could be used to display different options based on the user's previous choices. The simplicity of the Print Statement belies its importance in creating interactive applications, where timely and context-specific feedback to the user is crucial. By allowing for immediate visualization of computation results and operational states, the Print Statement not only facilitates debugging but also enhances the interactivity and user-friendliness of applications developed in Catscript. Thus, it plays a pivotal role not just in problem-solving but also in the user interface design and user experience aspects of software development.

Return Statement

```
function sum(a, b) {
	var result = a + b
	return result
}

// Call the sum function and store the result
var total = sum(5, 3)
print(total)
```

In Catscript, the Return Statement is a crucial component of function definitions, serving as the mechanism through which a function outputs a result. Positioned typically at the end of a function,

this statement determines the value or data that gets sent back to the calling context. By including a return statement, developers clearly define what a function is expected to deliver after execution. This could be a simple value like a number or string, a complex data type, or even null or undefined, depending on the function's purpose and the language's capabilities. For instance, a function designed to calculate the sum of two numbers would use a return statement to provide the result of the addition. This explicit declaration of output makes functions in Catscript reusable and modular, allowing them to be called multiple times from different parts of the program or application with consistent behavior and expected outcomes.

Furthermore, the Return Statement in Catscript not only conveys the outcome of a function but also immediately terminates the function's execution once it is reached. This behavior is essential for controlling the flow of execution within a program, particularly in scenarios involving conditional logic or loops within functions. If a return statement is conditional, it can lead to different outcomes based on the input, enhancing the function's flexibility and utility. For example, in a function that checks if a user input is valid, a return statement might terminate the function and send back a boolean indicating validity as soon as a decision is made. This use of the return statement ensures that unnecessary processing is avoided, thereby optimizing performance and resource utilization. Thus, the return statement is not merely a syntax requirement but a fundamental part of defining functions' behavior and interaction in Catscript, shaping how data flows and functions integrate within larger applications.

Syntax Error Statement

```
1. if (x > 10 { print(x) } // Missing closing parenthesis after (x > 10
2. for (var i in x) { // Missing closing curly bracket
```

In programming languages like Catscript, the Syntax Error Statement is not an actual statement defined by the language's syntax. Rather, it is a mechanism used to indicate problems encountered during the parsing or interpreting of code. Syntax errors occur when the code written does not conform to the rules defined by the language's grammar. The Catscript interpreter detects these issues during the compilation or execution phase and raises a syntax error. Syntax errors play a vital role in the development process, as they help developers identify and correct issues promptly. When a syntax error is detected, the environment typically stops the execution and provides an error message that includes details about the nature of the error and its location in the source code. This message is essential for debugging and is often designed to be as informative as possible to assist in a quick resolution of the problem.

For instance, if the error was due to an unexpected token, the error message would include the offending token and a suggestion or reminder about the correct syntax. The proactive approach of handling syntax errors not only aids in maintaining the robustness of the applications developed in

Catscript but also enhances the learning curve for new programmers, who can understand and adhere to the language's syntax more effectively.

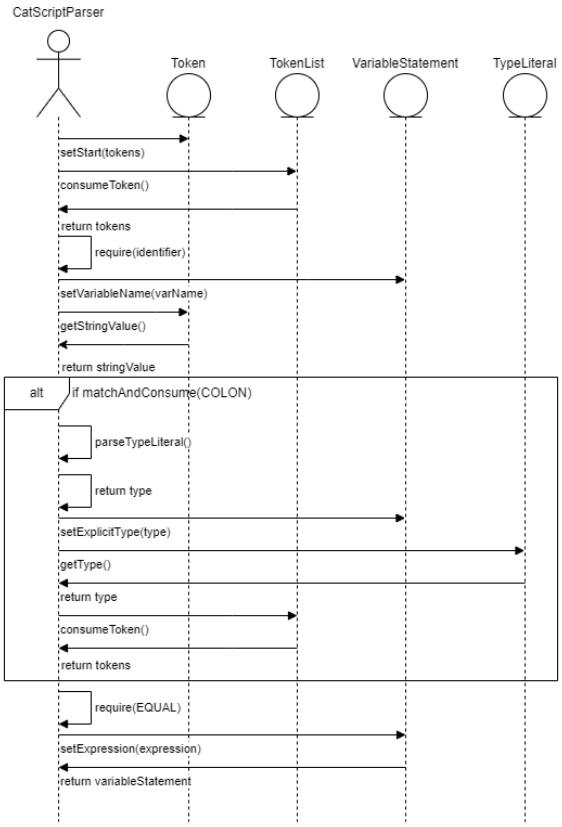
Variable Statement

```
var x = 10
var name = "Ry Guy"
print(x)
print(name)
```

In Catscript, the Variable Statement is central to the creation and initial management of variables within the language. By utilizing the "var" keyword followed by an identifier, and optionally a type, you can declare new variables that store data or references to data. The inclusion of a type is particularly significant in contexts where Catscript supports static typing or where type inference enhances program reliability and readability. This statement effectively reserves space in memory for the variable and establishes its initial state, which might be an explicit initialization or a default setting depending on the language's rules. This dual functionality of declaration and definition within a single statement streamlines code writing and helps maintain clarity.

The Variable Statement in Catscript also plays a crucial role in scope definition. Variables declared within a function or block are typically limited to that context, preventing their values from being accessible or modified outside of that scope. This scoping is fundamental for controlling the lifecycle of data within the program and for avoiding namespace pollution, which can lead to hard-to-track bugs. The use of the Variable Statement, therefore, is not merely a matter of syntax but a core aspect of structuring and securing program behavior. For instance, declaring a variable inside a loop with var means it exists only within the loop, getting re-initialized with each iteration, which can be useful for tasks such as processing collections or managing temporary data.

Section 5: UML



This is a sequence diagram for the parseVarStatement() function, which parses a Variable Statement. The actor represents the main class in which the variable statement parsing takes place, while the rest are other classes whose functions play a role in parsing variable statements. The first function employed, the setStart(tokens) function, sets the parting point of the variable statement. The setStart function has the parameter ‘tokens,’ which is also where the consumeToken() function is being called. The TokenList class is also where the tokens are stored generally, so the arrow pointing from TokenList to CatScriptParser represents the consumeToken() function from the TokenList class returning a value. The parseVarStatement() function then calls the require(identifier) function, with the looping arrow representing the fact that both of these functions are within the same CatScriptParser class. The require function makes an identifier a required condition for the statement being parsed to be considered a variable statement.

The ‘alt’ section represents a sequence of events that only take place when a certain condition is met. The condition in this case is when the next token retrieved from the matchAndConsume() function is a colon (:). If so, the parseVarStatement() function goes through a series of events that consists of the parser getting the next token, parsing the token

that it assumes is the type being assigned to the variable, setting the variable as that type, then moving on to the next token. If the alt section condition is not met, the sequence of events within the section are skipped over entirely. Finally, the require function is called again, except it checks for the next token to be an equal sign, and then getting and setting the expression that comes afterwards. The parseVarStatement() function returns a new instance of the Variable Statement class.

Section 6: Design trade-offs

When it comes to making a parser in compiler design, there's two common approaches: making a parser by hand, or letting a parser generator tool make one for you. When I made the decision, I had to consider various factors such as ease of implementation, performance, flexibility, and maintainability. To give an understanding of my decision to make a recursive descent parser, I will compare it with ANTLR, a well-known parser generator. For starters, recursive descent parsers have the following core idea: each grammar production can be directly mapped to a corresponding function in the code. For example, I have each left hand side of an expression call the grammar function with higher precedence, while the right hand side does a recursive call. This makes the structure of the resulting program closely mirror the defined grammar rules, thus making the parser's structure intuitive and easy to understand and maintain. This also offers high flexibility, where I could customize the parsing process, handle special cases, and integrate additional logic seamlessly.

When I considered using a parser generator like ANTLR, I found that it requires defining the grammar in separate ANTLR-specific files (which end in a .g extension). It starts with defining a lexer, which involves token definitions, regular expressions, character literals and directives. Once the lexer code is generated, you can then define and generate a parser, which has a series of EBNF-like rules. I recognized that using a parser generator can lead to faster development once the grammar is defined, due to the tool generating the parser code automatically. However, this can also lead to more complexity and less flexibility, especially for beginners or those unfamiliar with the ANTLR tool. The lack of direct visibility into the parsing process meant that debugging, maintaining and customizing the generated code requires knowledge of parsing techniques and ANTLR's internals and configuration options. With all of this in mind, I ultimately decided that the learning curve for understanding ANTLR was too steep for a 15-week capstone project and went with recursive descent parsing instead.

Section 7: Software development life cycle model

The software development life cycle model I used for my capstone project is test-driven development (TDD). Test-driven development is a software development approach where tests are written before the actual implementation code. Before writing any production code, you write test cases that specify the desired behavior or functionality of a particular feature. These tests are often written using testing frameworks, which in this case I used JUnit (for Java).
Once the test is written, it is executed to ensure that it fails. This failure confirms that the test is correctly identifying the absence of the desired functionality in the codebase. With the failing test in place, you then proceed to write the minimum amount of code necessary to make the test pass. After writing the implementation code, you rerun the test suite. If the test passes, it indicates that the written code successfully implements the desired functionality. If the test fails, you iterate on the implementation code until the test passes. Once the test passes, you may refactor the code to improve its design, readability, or performance. The process is repeated for each new feature or unit of code.

The main goal of test-driven development is to ensure that the codebase remains correct, reliable, and maintainable throughout the development process. Having tests for various pieces of functionality naturally results in well-documented code, since each test gives clarity for what is the expected behavior of my code. Test-driven development gave me an understanding of what a compiler needs, as well as what the output of my work should look like. I was encouraged to focus on the overall functionality of my code, and I could break down a large project down into small steps, rather than getting intimidated by the risk of fixing bugs I may not understand, and the seemingly large distance of my end goal(s). Seeing the green check after implementing a piece of functionality gave me a sense of accomplishment and an increase in morale. Overall, test-driven development promoted a disciplined approach to software development, with an emphasis on testing, debugging, design, and incremental development to produce the results I needed.

While I encourage the test-driven development model, I will also note the disadvantages and additional problems TDD can bring. A problem I encountered most frequently was dealing with tightly coupled tests that would become fragile and prone to breakage when implementation details of my code changed. For example, when I added implicit list typing for function statement parsing, it resulted in one of my variable statement parsing tests failing, as variable statements had a similar type system, but did not originally account for the implicit list type. While TDD can provide confidence in the correctness of code, passing tests do not

guarantee the absence of bugs or unexpected behavior. There may have been times where I became overly reliant on passing tests and therefore overlooked the bigger picture of my software design, or potential edge cases that are not adequately covered by the test suite. Despite the disadvantages and problems I encountered, the pros of using test-driven development outweighed the cons, and overall led to higher productivity than other software development models I could have used otherwise.