Catscript

Colby Roberts, Treyton Grossman

CSCI 468

Compilers

Spring 2024

Section One: Program

Source code can be found at capstone/portfolio/source.zip

Section Two: Teamwork

Team member one was responsible for all implementation tasks related to the Catscript capstone project. Implementation took place in a couple key locations throughout the project including the tokenizer, parser, evaluator, and bytecode generator. During the project, team member one handled writing all of the source code for the Catscript parser, which was built upon basic infrastructure code given to us by our instructor. The first part of the project focused on writing the tokenizer, which was programmed to read in strings that could then be used further in the project. After the tokenizer was written, the next major focus was writing the parser to turn tokens into expressions and statements. Next, team member one wrote the evaluation logic that allowed expressions to evaluate and statements to be executed. The last primary goal of team member one was to write the logic for a bytecode generator that could compile Catscript programs using JVM bytecode. Team member two was responsible for writing tests that were used in debugging and stress-testing the Catscript parser. The first test written by team member two was used to ensure the recursive descent algorithm accurately parsed nested if statements. The second test written evaluated whether or not an if statement condition accurately passed through to print a value. Finally, the third test makes sure the if statement conditional expression is being evaluated for errors. On top of writing tests, team member two handled writing the documentation for the Catscript capstone project included later in this document.

Overall, the work performed by team member one accounts for around 85% of all work for the Catscript compilers project, while the work done by team member two accounted for around 15%.

Section Three: Design Pattern

One design pattern we used in this project was the memoization pattern. This pattern was used in the getListType method of the CatscriptType class. Instead of generating new ListType object instances every time, a HashMap was used to store created ListTypes and retrieve it in the case that it was needed again. If the ListType is not already in the HashMap, it will be stored and then returned. If the ListType was in the HashMap, it was simply returned. The memoization pattern was chosen to reduce the redundant computation of creating new instances when one already exists. This lookup is much more efficient than creating a new instance of a ListType every time, thus optimizing the Catscript compiler project.

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();  3 usages
public static CatscriptType getListType(CatscriptType type) {  5 usages  Trey Grossman +1
    ListType listType = new ListType(type);
    if (cache.containsKey(listType)) {
        return cache.get(listType);
    } else {
        cache.put(listType, listType);
        return listType;
    }
}
```

Section Four: Technical Writing

Catscript Documentation

This document gives a brief outline of Catscript, and its various features.

Introduction

Catscript is a simple scripting language. Here is an example of how it can be used:

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

This code creates a variable x, and sets it to the string "foo". A print statement is then called which prints out x's value. This segment consists of a variable statement, as well as a print statement. These features will be defined in the following section.

Features

Primary Expressions

```
true
"hello"
3
```

Primary Expressions consisting of string, integer, boolean, null, list literal, function call, identifier, or parenthesized expressions. They serve as the building blocks for more complex expressions, providing direct values or references that other parts of the code can manipulate or evaluate. In this case true, "hello", and 3 are all primary expressions, as well as expressions of their more specific type.

Additive Expressions

1 + 2

6 - 3

“Hello “ + “world”

Additive Expressions are used to compute the sum, or compute the difference between two integers. If a string is used in either the left or right slots, the string values of each will be concatenated.

Boolean Literal Expressions

true

false

Boolean Literal Expressions represent "true" and "false" values in Catscript. These literals are fundamental in conditional expressions, directing the flow of execution.

Comparison Expressions

```
1 < 2
1 > 2
1 <= 2
1 >= 2
```

Comparison Expressions evaluate the relationship between two operands. These relationships extend to less than, less than or equal to, greater than, or greater than or equal to. The result of these expressions is a Boolean value that can be used in subsequent program logic.

Factor Expressions

```
8 / 4
8 * 4
```

Factor Expressions perform multiplication and division operations on integers. These expressions are key in math calculations within a program.

Function Call Expressions

```
foo(x)
```

Function Call Expressions invoke functions with specified parameters, executing predefined code. These expressions are crucial for code refraction, allowing complex tasks to be executed repeatedly with different arguments.

Identifier Expressions

```
x
```

```
foo
```

Identifier Expressions are used to reference things such as variables and functions. They act as placeholders for explicit data values, or values resulting from functions or objects. Identifiers are essential for accessing and storing data throughout a program.

Integer Literal Expressions

```
1
```

Integer Literal Expressions evaluate to fixed numerical values. These literals are used for a variety of purposes within programs. They provide a straightforward way to input and manipulate numbers.

List Literal Expressions

```
[1, 2, 3]
```

List Literal Expressions create arrays or lists of any type of value. For this example we have a list of integers, but may include types such as bool, string, or object.

Null Literal Expressions

null

Null Literal Expressions signify the absence of a value or a non-existent reference, often used in object-oriented programming. They are crucial for managing pointers and object references, particularly in error handling and memory management. Using null can help prevent runtime errors by explicitly denoting uninitialized or empty states.

String Literal Expressions

"Hello World!"

String Literal Expressions are sequences of characters used in programs for text manipulation and output. These literals must be enclosed in quotes to be recognized as strings.

Syntax Error Expressions

Unterminated list literal

Syntax Error Expressions result from violations of the language's rules, leading to compilation or interpretation failures. These errors are often due to typos, incorrect use of language constructs, or structural mistakes in the code.

Unary Expressions

```
!true
-6
```

Unary Expressions apply a single operator to one data value. The usable operators in this case are "not" which negates the following logic, or "-" which negates the following number. These expressions are fundamental in manipulating numerical or boolean values.

Type Literals

```
int
```

Type Literals specify the data type of a specified value. They are used in variable statements to explicitly assign the variable data type. In Catscript, the legal types are int, string, bool, object, and list (which can also have a specified enclosed data type).

Assignment Statements

```
x = 10
```

Assignment Statements assign a value to a defined variable. These statements are fundamental in programming for storing values that can be used later in the program. The left side specifies the variable name to be assigned, while the right side provides the value to store.

For Statements

```
for (x in [1, 2, 3]) {
   print(x)
}
```

For Statements are used to iterate over a collection of items, executing a block of code for each element. In this example, x takes on each value in the list [1, 2, 3] sequentially, and the print(x) command executes for each iteration. This loop structure is essential for tasks that require repetitive action, such as processing or transforming each item in a collection.

Function Call Statements

```
foo(x)
```

Function Call Statements invoke an already made function with specified arguments to perform a stored block of code. These statements are crucial for executing code defined in functions. In this example, foo(x) calls the function foo with x as its argument, executing its defined actions.

Function Definition Statements

```
function foo(x) {
   print(x)
}
```

Function Definition Statements declare new functions and specify the code that they execute. The example function foo takes a single parameter x and prints it. The term "function" denotes that the following code is a new function definition to be stored, and this statement can optionally have an explicit return type.

If Statements

```
if(true) {
   print("true")
}
```

If Statements control the flow of execution based on a condition, only executing the enclosed code block if the condition evaluates to true. In this example, print("true") will always execute because the condition true is always satisfied. Booleans as well as anything returning a boolean can be used as a conditional statement.

Print Statements

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

Print Statements output their parameter to a console or another standard output. They are used to display values of variables, messages, or any other information during execution. This example prints the string "Hello World!".

Return Statements

```
return(true)
```

Return Statements terminate the execution of a function and optionally pass back a value to the caller. In this example, return true immediately ends the function and sends back the Boolean value true.

Syntax Error Statements

```
Error: Unexpected Token
```

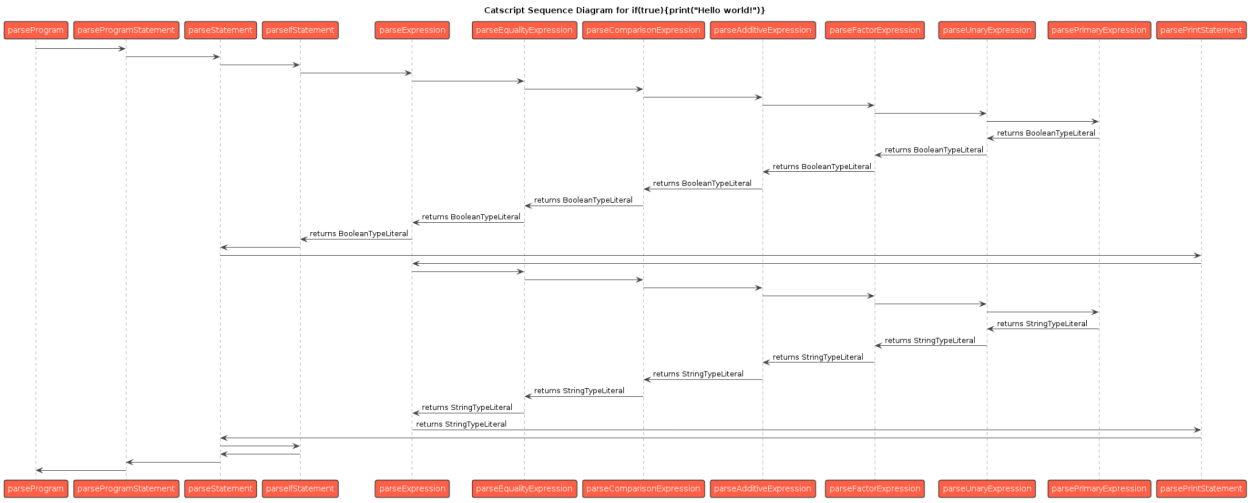
Syntax Error Statements are the statement for a Syntax Error Expression. If the code that breaks Catscript syntax is a statement, a Syntax Error Statement will be passed and the code will fail. Like in Syntax Error Expressions, common sources include missing punctuation, incorrect command usage, or misordered code elements.

Variable Statements

var x = 10

Variable Statements declare new variables and set their type. This statement declares a variable x of type int and initializes it with the value 10.

Section Five: UML



For better readability, please zoom in or see the picture in capstone/uml/ifStatement.png

Section Six: Design Trade-offs

In our project, we opted to create a custom recursive descent parser for Catscript instead of using a parser generator based on the EBNF language diagram. While parser generators have

advantages due to less handwritten code, they introduce additional complexity. The need for translations between EBNF and generator-specific syntax can be cumbersome. Despite requiring more handwritten code, we chose to implement our recursive descent parser which allowed us to quickly start and gain a better understanding of grammars, since recursive descent parsers express the natural recursive nature of grammars more obviously.

Section Seven: Software Development Lifecycle Model

During our project, we used a Test Driven Development (TDD) approach. We started programming our Catscript parser using a suite of tests given to us by our professor, covering all aspects of the language implementation. TDD allowed us to break down the project into manageable pieces. Instead of being overwhelmed by the entire project all at once, we were able to focus on individual tests, or groups of tests, in a controlled manner. TDD served as an effective progress monitoring tool. Working our way through completing pre-defined tests from the bottom up allowed us to have a better sense of where we were in the implementation process, which made setting goals very easy. Overall, I found working with TDD enjoyable because of how easy it was to visualize our progress and ensure every part of out project was functioning correctly at regular intervals.