

Isaac Schmidt

Capstone Portfolio

CSCI 468

Partner: Michael Clearwater

Section 1: Program

A zip file of the Catscript compiler code can be found at:

<https://github.com/ischmidt707/csci-468-spring2023-private/tree/main/capstone/portfolio/source.zip>

Section 2: Teamwork

This project was done with a team of two engineers. Team member 1 (Isaac Schmidt) acted as the primary engineer in charge of writing the code. Team member 2 (Michael C) was in charge of testing and documentation. Section 4 of this document was produced by team member 2 to document Catscript.

Section 3: Design Pattern

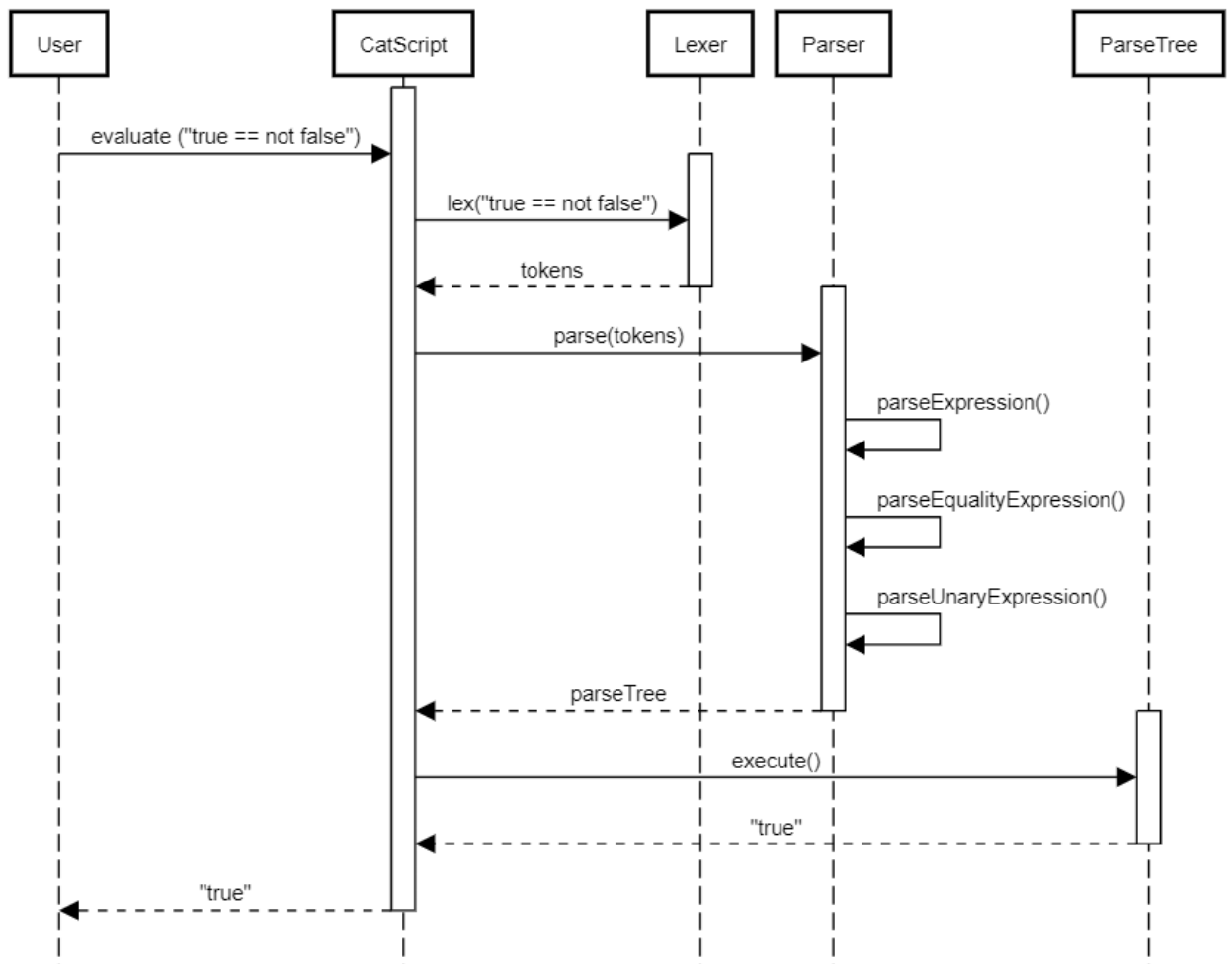
One key design pattern used in Catscript is **memoization**. It can be found in the `getListType` method of the `CatscriptType.java` file in the parser directory. The purpose of utilizing memoization here is to make this method more efficient in computation and memory space. For early calls to the method, memoization is slightly less efficient due to the fact that it is recording new `ListTypes` as they are created, but gains efficiency the more the method is called. Therefore we decided that for a small upfront cost, using the memoization design pattern instead of directly coding the method would be overall a beneficial decision. Large Catscript files have the potential to gain significantly from this, while small Catscript files suffer a negligible overhead cost.

Section 4: Technical Writing

Team member 2 wrote a comprehensive technical documentation of Catscript. This can be found attached at the end of this document as Appendix A.

Section 5: UML

Catscript Comparison Sequence Diagram



Section 6: Design Trade-Offs

In this project, we chose to write the parser using recursive descent, instead of using a parser generator. Parser generators such as `lex`, `yacc`, and `ANTLR`, generate parsers based on a provided language specification in their own specified format. By using recursive descent we were able to gain a much more intuitive sense of how compilers work from the bottom up. Additionally, by writing all code by hand, the designer naturally has much more direct control over the exact execution of the project. On the other hand, parser generators can be quite helpful to quickly cover more features of a language without tediously rewriting similar sections of code.

Recursive descent made the most sense overall because not only are we gaining a deeper understanding of the content, but are also preparing ourselves for better implementations and improvements of this codebase in the future. Where using a parser generator might make more sense is if our only aim was to quickly implement a new language compiler for some specific purpose. In that scenario, maintenance and understanding would be reduced considerations.

Section 7: Software Development Life Cycle Model

We used the Test Driven Development (TDD) model for our software development life cycle. This means that prior to writing code for the compiler, we were provided with a suite of unit tests that defined how the program should run. This methodology comes with many advantages as well as disadvantages.

One key advantage of TDD is that it fits the framework of Agile software development, which is rapidly gaining popularity and adoption in industry, which could be a litmus test for the industry's belief in the overall positivity of TDD. Excess code is also reduced in TDD because the tests define exactly what needs to function. By defining functionality into very tiny slices of unit tests, engineers are also forced to use a better modular design and write code that is very maintainable. Lastly, TDD naturally documents the code through tests while simultaneously providing high test coverage, which is sometimes quoted as an important metric of codebase quality.

Despite all of these advantages, Test Driven Development does come with disadvantages as well. A false sense of security can be created if bugs themselves are incorrect or incomplete, and it is up to the engineer to realize these cases during the course of development. Another possible downside is that tests must be adapted every time requirements change. Luckily for us, this project had stable requirements, thus making TDD an even better choice for this situation by negating this potential disadvantage.

Appendix A - Catscript Guide

This document should be used to create a guide for Catscript, to satisfy capstone requirement 4

Introduction

Catscript is a statically typed functional programming language. Here is an example:

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

Features

Data Types

Catscript supports the use all primitive types, along with lists and objects. The type system is as follows:

- int - a 32 bit integer
- string - a java-style string
- bool - a boolean value (true or false)
- null - the null type
- list - a list of value with the type 'x'
- object - any type of value

Equality Expressions

An equality expression is able to test two different things. One, it can test if two values/objects are equal to each other, or two, it can test if two values/objects are not equal to each other. The syntax for the equality expression is the same as most programming languages, `==` tests for equality, and `!=` tests for inequality. The equality expression will return `true` or `false`, depending on the outcome of the expression's evaluation. The equality expression is most often paired with an if statement.

Equality Expression Examples:

```
1 == 1
```

This expression will evaluate to true, since the integer 1 does equal the integer 1.

```
1 == 2
```

This expression will evaluate to false, since the integer 1 does not equal the integer 2.

```
1 != 1
```

This expression will evaluate to false, since the integer 1 does equal the integer 1.

`1 != 2`

This expression will evaluate to true, since the integer 1 does not equal the integer 2.

Comparison Expressions

A comparison expression is similar to an equality expression, but instead it takes two values/objects and tests to see if one is greater than, greater than or equal, less than, or less than or equal to the other. The syntax for these four expression is as follows:

- greater than - `>`
- greater than or equal - `>=`
- less than - `<`
- less than or equal - `<=`

The expression will evaluate to **true** or **false** depending on the outcome of the comparison. The comparison expression is most often paired with an if statement.

Comparison Expression Example:

`1 > 0`

This expression will evaluate to true, since the integer 1 is greater than the integer 0.

`1 >= 1`

This expression will evaluate to true, since the integer 1 is greater than or equal to the integer 1.

`0 < 1`

This expression will evaluate to true, since the integer 0 is less than the integer 1.

`1 <= 0`

This expression will evaluate to false, since the integer 1 is not less than the integer 0.

Additive Expressions

An additive expression is capable of doing three things. One, it can add two integers together and return the sum, two, it can subtract two integers and return the difference, or three, it can concatenate a string to another string or an integer to a string. The syntax for addition and concatenation is `+`, and the syntax for subtraction is `-`.

Additive Expression Examples:

`1 + 1`

This expression will return the sum of 1 and 1, which is 2.

`1 - 1`

This expression will return the difference of 1 and 1, which is 0.

`"foo" + "bar"`

This expression will return the two strings concatenated together, `"foobar"`.

Factor Expressions

A factor expression is just basic multiplication or division. It takes two integers and returns the product or the quotient based on the provided operand. The syntax for a factor expression is `*` for multiplication, and `/` for division. For division, the first integer provided will be the dividend and the second integer will be the divisor.

Factor Expression Examples:

`1 * 2`

This expression will return 2, the product of 1 and 2.

`4 / 2`

This expression will return 2, the quotient of 4 and 2.

Unary Expressions

A unary expression is capable of negating integers and booleans. The syntax to negate an integer is simply putting a `-` in front of the integer to turn it to a negative integer. The syntax to negate a boolean is to insert `not` in front of the boolean wanting to be negated, which will cause that boolean to act like it's opposite.

Unary Expression Examples:

`-1`

This expression just acts like the integer `-1`. By providing the unary operator `-`, it negates the integer 1 to be a negative integer.

`not true`

This expression will evaluate to false, since `true` is being negated by the `not` operator.

For Loops

Catscript, like many languages, has an implementation of a for loop. The for loop allows you to iterate over every element within a data structure or will repeat for a given amount of times.

For Loop Example:

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

This for loop will print out all the elements of the given list:

```
1  
2  
3
```

If/Else Statements

Catscript has support for if/else statements. These statements allow for the execution of different code based on the boolean output to a passed in expression.

If/Else Statement Example:

```
if (x == y) {  
  print(1)  
} else {  
  print(2)  
}
```

This statement will print 1 if the variables x and y are equal, and will print 2 otherwise.

Variable Statements

Catscript has an implementation of variable statements, similar to those of JavaScript. Catscript supports creating variables with an explicit type, or variables with an implicit type determined by the parser. An explicit type is defined by inserting a colon after the variable name, followed by the type.

Explicit Type Variable Statement Example:

```
var x : int = 1
```

For this statement, the types will be checked by the parser to make sure the value being set to the variable is compatible with the explicit type of the variable.

Implicit Type Variable Statement Example:

```
var x = 1
```

For this variable statement, the parser will determine the implicit type automatically.

Print Statements

The Catscript print statement is similar to the one in Python. The print statement allows the user to print some output during execution of their code.

Print Statement Example:

```
print(1)
```

This statement will print 1 when the code is executed.

Assignment Statements

In Catscript, you can reassign variables to a different value like in many languages. You can reassign a variable to another variable or to an object of any kind.

Assignment Statement Example:

```
var x = 1  
x = 2
```

This block of code creates a variable `x` equal to the integer 1. It then reassigns `x` to the integer 2.

Function Definition Statement

Catscript allows the creation of functions. A function needs to have a valid name (no repeats), and may have a declared return type and arguments. A function without a specified return type will return `void`. The function's arguments may have a defined type, or they can be implicitly defined by the parser. A function with a return type other than `void` must have complete return coverage, meaning that every possible branch has a return statement at the end.

Function Definition Statement Example:

```
function foo(x : int, s : string) : string {  
    var y : string = x + s  
    return y  
}
```

This function definition has a specified return type of `string`, with arguments `x` and `s` with explicit types `int` and `string` respectively. The body of the function creates a new variable, `y`, with an explicitly defined type of `string`, and assigns it to the concatenation of `x` and `s`. Since the function has an explicit return type of `string`, it returns `y`.

Function Call Statement

Since Catscript allows for the creation of functions, it also allows for those functions to be called upon. A function call statement will call upon the function definition statement and execute it. The syntax of the function call is just `functionName(arguments)` like in most programming languages. A function call's arguments must match the number and type of arguments defined within the function definition statement. If the function definition has a return type other than `void`, then the function call can be assigned to a variable matching the type of the function definition return type or a variable whose type is implicitly defined.

Function Call Statement Example:

```
function foo(x : int, s : string) {  
    print(x + s)  
}  
  
foo(1, "bar")
```

This function call statement calls upon the function `foo`, defined above the call itself. The function call passes in the `int` `1`, and the `string` `"bar"`, as `foo`'s definition defines it needs an `int` and a `string` as parameters. Once the code is executed, the function call will execute the function definition using the parameters it passes in, resulting in `1bar` being printed.

Another Function Call Statement Example:

```
function foo(x : int, s : string) : string {  
    var y : string = x + s  
    return y  
}  
  
var newString : string = foo(1, "bar")
```

This example is almost the same as the one above, with a few minor differences. First, `foo`'s function definition specifies it must return a `string`. Within the body of `foo`, a new `string` variable, `y` is created and is assigned to the concatenation of `x` and `s`. The variable `y` is then returned from the function. Second, the `string` variable, `newString`, is being assigned to the function call for `foo`, with the given arguments. Since `foo`'s definition is returning the concatenation of its parameters, the variable `newString` will be assigned to the concatenation of the function call arguments, resulting in `newString` being assigned to `1bar`.