

Compilers

CSCI 468

Spring 2023

Andrew Anselmo, Gabriel Ewsuk

Section 1: Program

See attached .zip file.

Section 2: Teamwork

My team worked on this project by delegating work to each party. I worked on the compiler while my partner, Andrew Anselmo built high-level tests and documentation for the Catscript Programming language. Overall Andrew did 10% as stated above and I completed the compiler accounting for 90% of the project.

Andrew Anselmo's, tests are located in the
/src/test/java/edu/montana/csci/csci468/CapstoneTest directory under
Capstonetest.java.

Description of Tests:

First Test [*ifStatementInsideFunctionExecutes()*] - This test is for ensuring that the if statement inside a function executes.

Second Test - [*functionCallsAnotherFunctionExecutes()*] - This test ensures that embedded function calls are executing properly. It calls one function from another

Third Test - [*globalVariableExecutes()*] - This ensures that global variables work. This is done by creating a variable. Then the variable is called in the function declaration statement.

Description of the Documentation:

Andrew begins things with a brief introduction to the program. Explaining the basics of what script is and what it is capable of doing. From there he goes on to describe expressions and statements within the program to help a user learn to

write their own code. Each description includes helpful examples, descriptions, required parameters, and return values. With this documentation, someone with basic programming knowledge would be able to write their own executable code.

Section 3: Design pattern

Memoization was used in this capstone project and it is located in the file located in the CatscriptType.java file located in this directory:
csci468-private/src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java

From here it can be found in the CatscriptType class and within getListType Method.

This is a screenshot of the method that implements memoization



```
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();
4 usages  ↗ Carson Gross *
public static CatscriptType getListType(CatscriptType type) {
    CatscriptType listType = LIST_TYPES.get(type);
    if(listType == null){
        listType = new ListType(type);
        LIST_TYPES.put(type, listType);
    }
    return listType;
}
```

This design pattern was used so that we don't new up a list type every time we create a list type already created. For instance a list type of int. The way it works is going through the map to check if exists and if not it won't create a new type. This is similar to the idea of caching.

Section 4:

Catscript Documentation

Introduction

Catscript is a simple scripting language. Catscript has mathematical operations, iteration, function declaration, variable declaration, if/else statements, a type system, return statements, and print statements. While it may be simple there is a lot you can do with Catscript. Here is an example of a Catscript program:

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

```
foo("hello")
```

```
//output: hello
```

Features

AdditiveExpression

Example

```
1 + 1;  
3 - 4;  
"Cat" + "script";
```

Description

Lets you add/subtract integers and concatenate strings

Parameters

Left hand side, right hand side, operator

Returns

Product of the additive expression

BooleanLiteralExpression

Example

```
x = true;  
if(x){ *this will execute* }
```

Description

Lets you work with boolean values, which are either true or false

Parameters

NA

Returns

A boolean literal, either true or false

ComparisonExpression**Example**

```
1 < 4;  
4 <= 6;
```

Description

Allows you to make comparisons between two operands

Parameters

Left hand side, comparison operator, right hand side

Returns

True if the comparison is true, false if the comparison is false

EqualityExpression**Example**

```
1 == 1;  
2 == 4;
```

Description

Lets you evaluate if two things are equal

Parameters

Left hand side, equality operator, right hand side

Returns

True if the operands are equal, false if the operands are not equal

FactorExpression**Example**

```
4 * 3;  
10 / 2;
```

Description

Allows you to do multiplication and division

Parameters

Left hand side, multiplication or division operator, right hand side

Returns

The product of the multiplication or division

FunctionCallExpression**Example**

```
f○○(x);
```

Description

Allows you to call functions

Parameters

Function name, argument with compatible type for function

Returns

Calls the function, and then the function will return something if that's what the function does

IdentifierExpression**Example**

```
var x = "";  
function foo(x) { print(x) }
```

Description

Names for things

Parameters

The identifier name

Returns

NA

IntegerLiteralExpression**Example**

```
var x = 1
```

Description

Lets us have integer values

Parameters

NA

Returns

NA

ListLiteralExpression**Example**

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

Description

Lets us have lists in catscript

Parameters

NA

Returns

NA

NullLiteralExpression

Example

```
var x = null;
```

Description

Allows us to have null value

Parameters

NA

Returns

NA

ParenthisizedExpression

Example

```
(1 + 1) * 3
```

Description

Lets us use parentheses to modify order of operations

Parameters

Expression inside parentheses

Returns

Product of expression inside parentheses

StringLiteralExpression

Example

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

Description

Lets us have strings

Parameters

NA

Returns

NA

SyntaxErrorExpression

Example

```
return new SyntaxErrorStatement(tokens.consumeToken());
```

Description

Lets us create errors

Parameters

tokens.consumeToken()

Returns

NA

TypeLiteral

Description

This class lets us set Catscript types and get Catscript types

Parameters

type

Returns

Type if you are using getType()

UnaryExpression

Example

```
!true;  
-1;
```

Description

Lets us have negation and negative values

Parameters

Operator, right hand side

Returns

NA

ForStatement

Example

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

Description

Lets us iterate through code

Parameters

Expression, variable name, body

Returns

NA

FunctionCallStatement

Example

```
foo(x)
```

Description

Lets us call functions

Parameters

Expression

Returns

If the function you are calling returns something then you will see that be returned

IfStatement**Example**

```
if (x < y){ *do some stuff* };
```

Description

Lets us execute code only on certain conditions, if those conditions are not met we can have an else statement after and that will execute

Parameters

Expression

Returns

NA

PrintStatement**Example**

```
print("foo");
```

Description

Lets us print things out

Parameters

Expression

Returns

Prints out what is inside the parentheses

ReturnStatement**Example**

```
return x;
```

Description

Lets us return things. If you have a return statement at the end of a function, when that function gets called that thing will get returned

Parameters

Expression

Returns

Expression

SyntaxErrorStatement**Example**

```
return new SyntaxErrorStatement(tokens.consumeToken());
```

Description

Lets us create errors

Parameters

tokens .consumeToken()

Returns

NA

VariableStatement

Example

```
var x = 2;
```

Description

Allows us to create variables which is just a memory location where a value can be stored

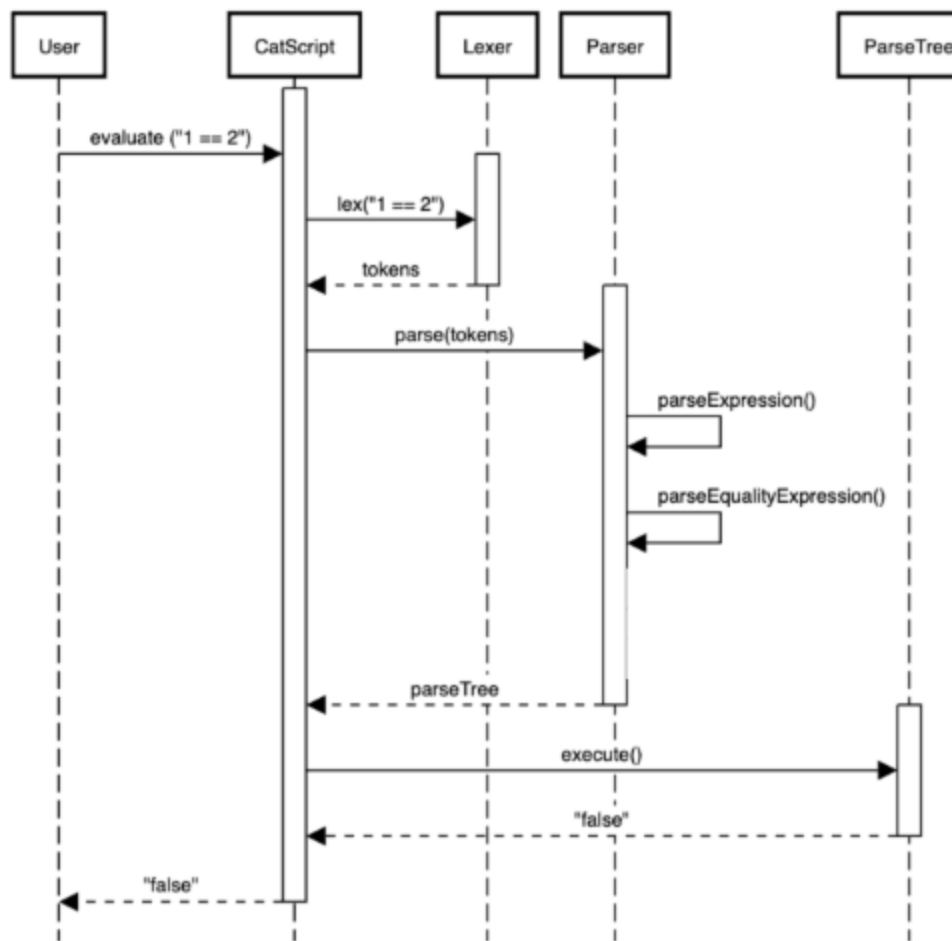
Parameters

The value

Returns

NA

Section 5: UML.



This is a sequence diagram that shows the sequence of events that occur during an equality expression. First, the user sends the “1==2” to the script program to be evaluated. From here the Catscript program sends the “1==2” to be tokenized. Then the tokens are returned to the catscript program. The tokens are then sent to be parsed where it will do a recursive descent by going through `parseExpression`, then `parseEqualityExpression`. From here we return the parse tree. Lastly, we send the `execute` command from the script to `ParseTree` to execute the program. It returns `false` to the script program which then returns `false` to the user.

Section 6: Design trade-offs

For this project was the decision to utilize recursive descent instead of a parser generator. One of the trade-offs is that this required me to actually code a parser myself. While if I had used a parser generator, I would be able to parse automatically. However, the benefit of using recursive descent is that it is simple and easier to debug. Whereas the parser generator is not. This was a huge deal because to get the tests to pass with TDD, we used the debugger in IntelliJ to help solve issues. Overall, recursive decent was the better option for a small lightweight language like catscript.

Section 7: Software development life cycle model

For this project, we used Test Drive Development (TDD). TDD is a lifecycle where the developer identifies a requirement, builds a test for that requirement, implements code to solve the requirement, and tests that code utilizing the relevant Test. Often these requirements are broken up and delegated amongst different parties. In most cases, tests are written by senior engineers. In this case, it was written by our teacher, Carson Gross. This enabled us to understand the requirements needed for the compiler and implement them. These tests are based on the JUnit framework and utilize assert to make sure that outputs match the requirements. For example, the `ifStatementWorksProperly()` test executes the code - `if(true){ print(1) }` - the assert would ensure that the output is 1. With provided tests, we are able to build a solid foundation to build future code on top of. This helped us confidently build a functioning compiler in sequential steps without having to go back and fix previous mistakes.