Montana State University

Capstone Portfolio

CSCI 468 – Compilers

Professor: Carson Gross

Spring 2022

Team Members

Logan Dolechek

Colton Weeding

Section 1: The Program

The overall framework was provided by our Professor Carson Gross. Carson used a test-based system to drive our development in a reverse engineering styled method to complete CSCI 468. Both Team Members used the IntelliJ IDE by JetBrains to develop the compiler using the Java language. The final source code can be found in this directory of my repository in the source.zip file.

Section 2: Teamwork

This semester each Team Member was responsible for writing their own compiler individually. Throughout the entire semester we would communicate to one another on how the project was going and assist one another on certain sections where we would be stuck within the logic. Between our own personal communication and the rigorous use of lab/office hours we were able to complete most of the compiler with some of the bytecode section missing. Team Member 1 (Logan Dolechek) and Team Member 2 (Colton Weeding) individual estimated hours into this project throughout the entire semester is somewhere between 140 to 150 hours. Team Member 1 was responsible for writing the documentation and 3-unit tests for Team Member 2. Team Member 2 was responsible for writing the documentation and 3-unit tests for Team Member 1.

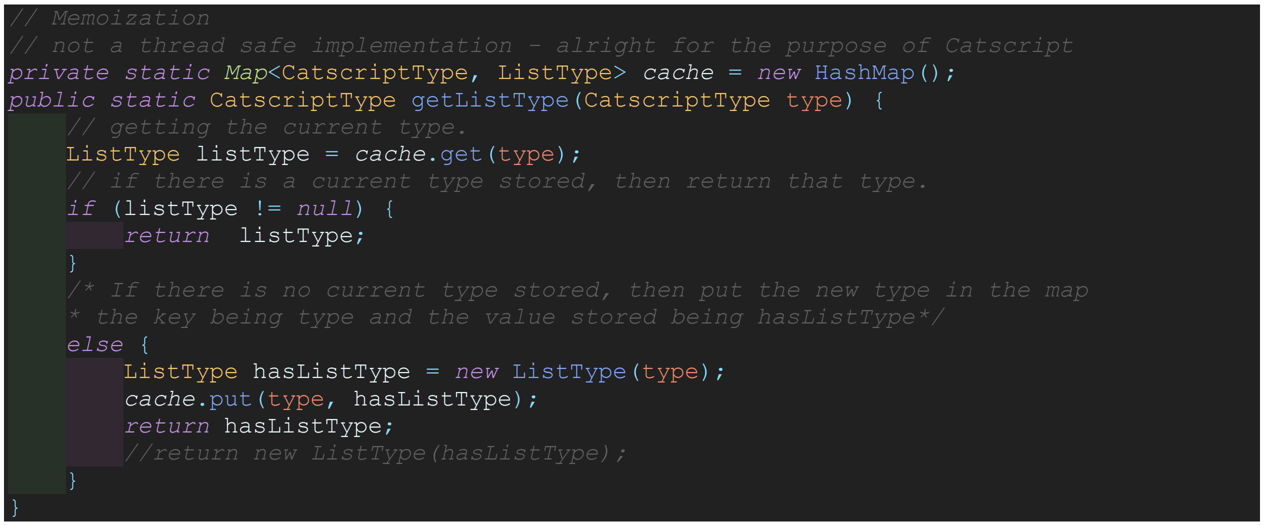
Section 3: Design Pattern

Introduction

In Catscript we use a design pattern referred to as Memoization. Memoization is considered an optimization technique. When a function call happens for the first time the type will be cached and stored into a map. As more function calls occur with different types those will continue to be stored within the same map. Once a function call is repeated the corresponding cached value will be returned. Resulting in saved computational time.

Example in Catscript

The snippet included below can be found within the CatscriptType.java file.



```java
// Memoization
// not a thread safe implementation - alright for the purpose of Catscript
private static Map<CatscriptType, ListType> cache = new HashMap();
public static CatscriptType getListType(CatscriptType type) {
    // getting the current type.
    ListType listType = cache.get(type);
    // if there is a current type stored, then return that type.
    if (listType != null) {
        return  listType;
    }
    /* If there is no current type stored, then put the new type in the map
    * the key being type and the value stored being hasListType*/
    else {
        ListType hasListType = new ListType(type);
        cache.put(type, hasListType);
        return hasListType;
        //return new ListType(hasListType);
    }
}
```

Section 4: Documentation

Introduction

The Catscript language is a relatively simple language that is statically typed. Mathematical operations are supported within the Catscript language. The mathematical operations that are supported are as follows: addition, subtraction, multiplication, and division. Parentheses are also supported to allow for order of operations. The left-hand and right-hand rules are followed within Catscript. The addition token (+) can be overloaded to allow for string concatenation. It should also be noted that Catscript also checks equality using a bang-equal (!=) to check when two values are NOT equal to one another and equal-equal (==) when two values are equal to one another. There are two ways to add comments in Catscript. The first being (/**/) placing the desired comment(s) between the asterisks. The second by using (//) and following the double slashes with the desired comment(s).

Types that are featured within Catscript are as follows: Integers, Strings, Null, Objects, and Boolean expressions. Lists are also featured in Catscript. The two typical types that are normally seen in languages that are not seen in Catscript are the Float and Double types. There are a variety of expressions and statements that are within Catscript and will be explained in greater detail below.

Commenting Examples:

```
// A single line comment
```

Above is a single line comment

```
/*When there
* is
* a lot
* to say*/
```

Above is when you want to have multiple lines of comments

Expressions

Expressions evaluate some value in Catscript. The following expressions will be covered below: String, List, Integer, Boolean, Null, Unary, Additive, Parenthesized, Factoring, Comparison, Equality, Function calls, and Syntax Errors.

String Literal Expression:

Like in many programming languages String literal expressions in Catscript begin and end with quote marks (“,”). From left to right it will construct an array of characters, resulting in a word or sentence(s).

Examples:

```
"C"
```

Above evaluates to C.

```
"This is a sentence."
```

Above evaluates to This is a sentence.

List Literal Expression:

In Catscript List literal expressions begin and end with a square bracket ([,]). List literals are used to generate a list of items or an array of items. All items within the list will be separated by a comma. All items that can be included in a list literal are strings, integers, Booleans, or the null type.

Examples:

Basic List of Integers

```
[0, 1, 2]
```

Above evaluates to [0, 1, 2]

List of multiple types

```
[0, null, "String"]
```

Above evaluates to [0, null, String]

Lists within a List

```
[[0, 1, 2] , ["String", true]]
```

Above evaluates to [[0, 1, 2] , [String, true]]

Integer Literal Expression:

In Catscript you can type any number in, and it will evaluate if it is an integer. Integers in Catscript are 32-bit value.

Examples:

```
21
```

Above evaluates to 21

```
21.5
```

Above will NOT evaluate in Catscript

Boolean Literal Expression:

In Catscript Boolean values evaluate to true or false.

Examples:

```
true
```

Above will evaluate to true

```
false
```

Above will evaluate to false

Null Literal Expression:

In Catscript the Null literal expression has no value assigned to it. It is an object with no actual value.

Example:

```
null
```

Above will evaluate to null

Unary Expression:

In Catscript unary expressions allow for negative values or negation of a Boolean value. The way to create a negative value is to use (-) or the keyword “not”.

Examples:

```
-21
```
Above evaluates to -21

```
not false
```
Above evaluates to true

```
not true
```
Above evaluates to false

Additive Expression:

In Catscript the additive expression may be used for addition or subtraction mathematical equations. Integers may be added to one another, or the addition can be used to concatenate multiple strings together. You cannot subtract an integer from a string or a string from an integer. Subtraction is only allowed when both the left and right hand sides are integers.

Examples:

```
1 + 20
```
Above evaluates to 21.

```
22 - 1
```
Above evaluates to 21.

Addition and Subtraction may be used as many times as needed. Catscript will evaluate left to right until the final evaluation is found.

```
20 + 1 - 21 + 21 - 21
```
Above will evaluate to 0.

As mentioned above the addition operator can be used to concatenate strings.

```
"Some" + "concatenation" + "of" + "strings."
```
Above evaluates to: Some concatenation of strings.

Parenthesized Expression:

In Catscript a parenthesized expression is an expression with a set of parentheses around the expression or multiple parentheses around the expression. As long as the number of left parentheses match the number of parentheses on the right. The parentheses can be used in Catscript to assist with mathematical equations.

Examples:

```
(21)
```

Above evaluates to 21

```
(((21)))
```

Above evaluates to 21

```
((21)
```

Above will not evaluate (missing an ending parentheses)

```
(7 * 4) - 7
```

Above (7*4) evaluates to 28 then that is subtracted by 7 resulting in the final evaluation being 21

Factoring Expression:

In Catscript factor expressions handle multiplication and division mathematics. This is done by following the left-hand and right-hand rules. Both the left and right hand sides must be integers. For multiplication the operand (*) is used and for division the (/) is used.

Examples:

```
7 * 3
```

Above evaluates to 21

```
147 / 7
```

Above evaluates to 21

Just like the additive expressions the factor expression can use as many multiplication or division symbols as needed to find a final evaluation.

```
147 / 7 * 4 / 4 * 2
```

Above evaluates to 42

147 / 7 evaluates to 21 which is then multiplied by 4 which evaluates to 84 then divided by 4 evaluates to 21 which is then multiplied by 2 resulting in the final evaluation of 42. Which happens to be the answer to life and all things.

Comparison Expression:

In Catscript the comparison expression is used to check if an expression is less than, less than or equal to, greater than, greater than or equal to the other expression. This is done by using the operators (<, <=, >, >=). An evaluation will return true or false. Again, this can only be done with the integer type.

Examples:

```
21 < 42
```

Above evaluates to true because 21 is less than 42

```
21 < 7
```

Above evaluates to false because 21 is not less than 7

```
21 > 7
```

Above evaluates to true because 21 is greater than 7

```
21 > 42
```

Above evaluates to false because 21 is not greater than 42

```
42 <= 42
```

Above evaluates to true because 42 is equal to 42

```
21 <= 42
```

Above evaluates to false because 21 is less than 42

```
21 >= 42
```

Above evaluates to false because 21 is not greater than or equal to 42

```
100 >= 42
```

Above evaluates to true because 21 is greater than or equal to 42.

Equality Expression:

In Catscript the equality expression is used to determine if one expression is equivalent or not to another expression. To determine if they are equal to one another we use (==) and to tell if they are not equal to one another we use (!=).

Examples:

```
1 == 1
```
Above would evaluate to true.

```
true != null
```
Above would evaluate to true.

```
true == null
```
Above would evaluate to false.

Function Call Expression:

In Catscript function call expressions can return values of any data type included in Catscript itself. There can also be no return value (null). Depending on the function there will need to be a parameter in the function call.

Examples:

```
func()
```
Above the function func() is called with no parameters.

```
x = func(21)
```
Above the x variable is set to return the value within the function func().

```
func(21, 22)
```
Above is an example of the function func() with multiple parameters being sent into the function.

Syntax Error Expression:

In Catscript unexpected token errors are caught within the expressions themselves.

Example:

```
((21)
```
Above consider the situation in the parenthesized expression where it is missing an end parenthesis. This would return ErrorType.UNEXPECTED_TOKEN instead of crashing the program.

Statements

Assignment Statements:

In Catscript assignment statements are used to assign a new/different value to a variable that has been created previously, changing the value of the variable from what it was originally set as to the new value.

Example:

```
var x = 21;
print(x);
x = 42;
print(x);
```

Above the first print statement will print 21, whereas the second print statement will print 42.

```
var x = 21;
print(x);
x = "String";
print(x);
```

Above would cause an error in Catscript since Catscript is statically typed it does not allow for alterations of a variable.

Function Call Statements:

In Catscript functions are created by using the word (function) followed by whatever name you would like to name the new function. At the end of the new name of your function you need an open and close parenthesis for the parameter section. After the closing parenthesis it would be followed by an open and closed curly brace. Anything within the curly brace would be the contents of the function created.

Example:

```
function legalVotingAge(age : int) {
    if(age < 18){
        print("You are not old enough to vote yet.");
    } else {
        print("You are old enough to vote.")
    }
}
legalVotingAge(18);
```

Above is an example function that receives a parameter of type integer to determine whether or not a person is of legal voting age. At the bottom an integer of 18 is sent into the function. The output would be 'You are old enough to vote.'

For Statements:

In Catscript For Statements allow you to iterate through a list of values.

Example:

```
for(x in [0, 1, 2]) {
      print (x);
   }
```

Above is a basic example of a for loop that will iterate through the list and the output to the console.

Output: 0 1 2

If Statements:

In Catscript if statements are a way to use Boolean logic to determine if this then do this, if not, do something else.

Example:

```
var age = 18;
if (age < 18) {
    print("You are not old enough to vote yet.");
} else {
    print("You are old enough to vote.")
}
```

Above is a basic example of an if statement (as seen before in Function Call Statement example.

Output: You are old enough to vote.

Print Statements:

In Catscript print statements are the most basic statements. Using the word 'print' followed by open and closing parenthesis you are able to print strings or variables that contain a value.

Example:

```
print("This is a print statement.");
```

Above the output will be: This is a print statement.

```
var x = 21;
print(x);
```

Above the output will be: 21

Return Statements:

In Catscript a return statement is a statement that return some value at the end of a function. Any Catscript type can be returned from the function, the return type will be determined within the function that it is returning from.

Example:

```
function returnFalse() : bool {
        return false;
    }
    print(returnFalse);
```

Above the return statement will return false and then print false to the console.

Since Catscript is statically typed it should be noted yet again that if the return type does not match what the return statement is attempting to send back it will cause an error.

Variable Statements:

In Catscript variable statements permit the user to assign a value to a variable. You can assign an integer value to a variable or a string type to a variable. Values may be changed throughout execution of the program if the value is of the same type. Meaning a variable assigned to an integer cannot be changed to a string type.

Examples:

```
var x = 21;
```

Above is an example where x isn't specified as an integer but rather assigned to the integer type by using the right-hand side of the equal sign to determine it as an integer.

```
var x : int = 21;
```

Above is an example where the x is explicitly assigned to the integer type.

Syntax Error Statements:

In Catscript statements also have error handling like the expressions have. Instead of crashing the program an error will be thrown to allow the program to continue to run.

Example:

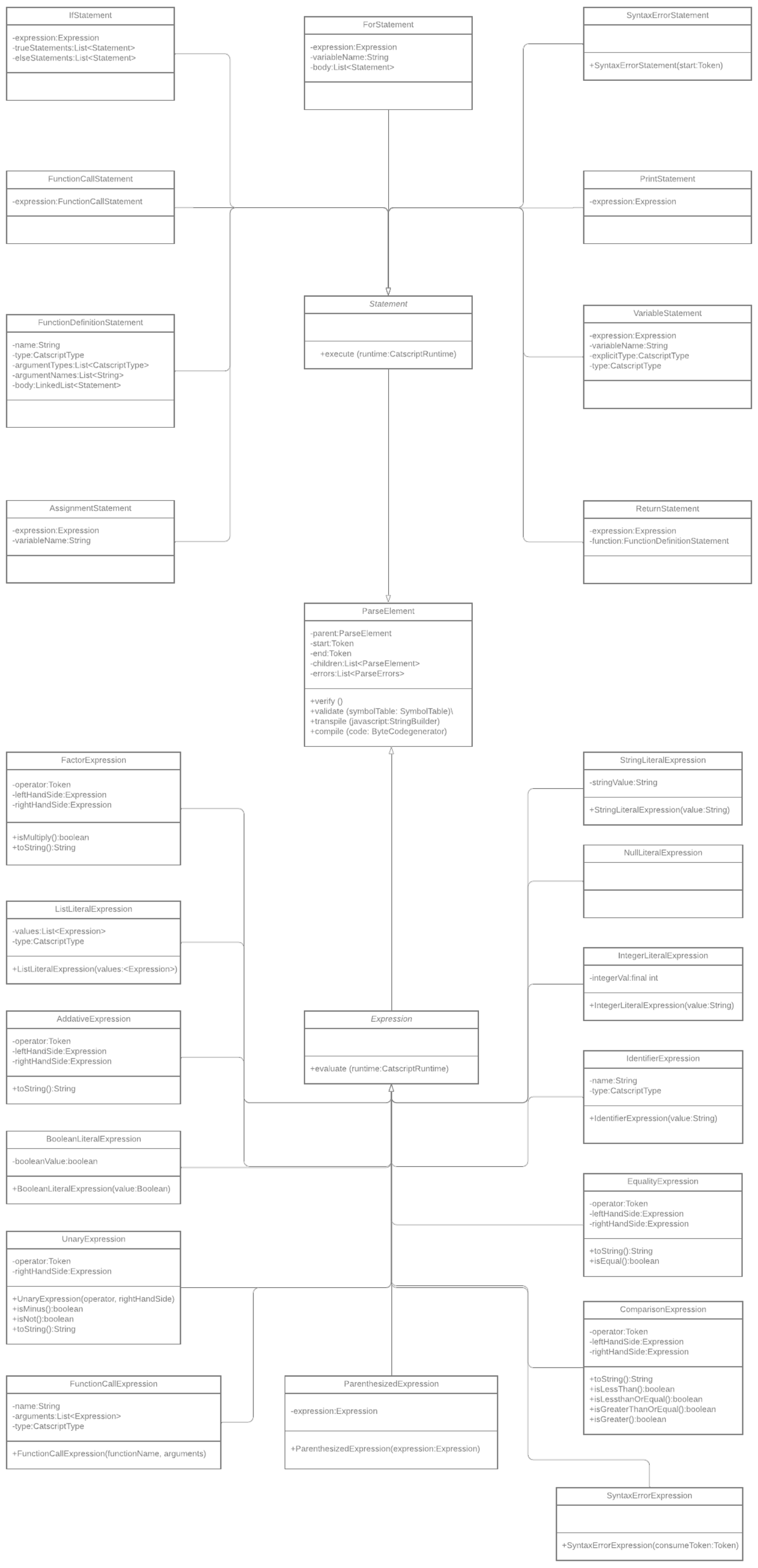
```
print(z);
```

Above is an example where 'z' is not set to any value. The error message will say that the symbol is not defined.

```
var y = 21;
y = "String";
print(y);
```

Above is an example where the variable 'y' was assigned to an integer type. Then 'y' was set to a string type and attempted to print 'y'. This will throw an incompatible type error.

Section 5: UML



Section 6: Design Trade-offs

Introduction

The two ways to create a compiler that were discussed in this course was the use of recursive descent and parser generators. There are positives and negatives to each approach. Within this section I will discuss why our professor Carson Gross chose to use the recursive descent approach over the parser generator.

Parser Generator:

Parser generators take a grammar as the input. The grammar that is taken in will be parsed through by the characters and will create code from that input. This creates a parse tree that shows the grammars production. The root of this parse tree would be the starting node of the grammar and all nodes that are expanded from that root is another production of the inputted grammar.

ANTLR was the parser generator tool that was discussed in class the most. It is a commonly used parser generator tool by many developers. Before Carson Gross took over CSCI-468 ANTLR was used by the previous professor for this class. Carson believes that us as students gain a better understanding of how to create a compiler without using this tool.

Recursive Descent:

Recursive descent is a top-down parser. With this parsing technique the start symbol (non-terminal, the top) is expanded upon to the entire program (to the bottom). This is known as a brute force parser or a backtracking parser. The generated parse tree is created by brute forcing or backtracking through the tree itself.

Comparing Parser Generator & Recursive Descent:

Carson Gross decided that we would use the Recursive Descent method to give his students a more hands on approach when creating the parser for the compiler. In my opinion I am glad that we used the Recursive Descent approach instead of using a tool like ANTLR. By working my way through much of this project I believe that I have learned more than a student would have learned if they were handed a tool like ANTLR. Being able to get down into the thick of it and figure my way out of the issues that arose I feel like I have walked away from this course with a good understanding of how to create a compiler.

Section 7: Software development life cycle model

Introduction

By far the most useful software development tool that was utilized in CSCI-468 was the Test-Driven Development. This was achieved by using the unit-tests that were provided in the project by our Professor Carson Gross. The other tool that was used during this course was the Catscript Server that we could run via web browser allowing us to do some testing outside of the project itself. I personally did not use the Catscript server while working my way through the compiler but it was nice to know that it was there if I needed it.

Test-Driven Development:

The formal definition - Test-driven development refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of unit-tests) and design (in the form of refactoring).

More times than not when starting a big project unit-tests are written first with the goals in mind of what the developer wants the program to accomplish. This assures that quality code is written during the development of the program itself. This prevents developers to write a ton of code that eventually turns out to be useless. Ideally a developer will work their way slowly through the unit-tests by writing small chunks of code and then running individual unit-tests that relate to the code that they are running. If the test continues to fail the developer can debug the code and step through the debugger to find where the errors may be occurring.

In CSCI-468 there were a total of 186 tests to pass. A small portion of those were given to us already passing by our professor. I personally really liked this method of development. This was the first course in my college career that really utilized Test-driven development. I was originally overwhelmed when starting this course in January but very quickly subsided that feeling by being able to slowly step through each unit-test individually.