

Compiler Capstone Portfolio

Kolson Mendelsohn

CSCI 468

09 May 2022

The Program

Included below is a link to the source code file:

<https://github.com/KStormM/csci-468-spring2022-private/blob/master/capstone/portfolio/source.zip>

Teamwork

For the purposes of this section, I will be referenced as team member 1, and my partner will be referenced as team member 2.

Team Member 1 Contributions: Code tests, documentation

Team Member 2 Contributions: Invalid code tests

Team Member 1 Estimated Time: 95%

Team Member 2 Estimated Time: 5%

The task of each team member for this project was to provide the other team member with three code tests as well as a detailed documentation of the Catscript programming language. Team member 1 and team member 2 met up to discuss the desired approach to the team components of the project. Following the meeting, team member 1 repeatedly attempted to contact team member 2 in regard to the documentation and the tests. Team member 2 responded with two code tests and no documentation. The code tests were invalid. Team member 1 again repeatedly attempted to contact team member 2 in regard to the invalid tests and documentation with no success. The two tests provided by team member 2 will be included below. After a brief discussion with Professor Gross, team member 1 decided to submit his own work as team member 2 did not contribute the required components.

Tests Provided by Team Member 2:

@Test

```
public void functionDisallowRecursion() {  
    FunctionDefinitionStatement expr = parseStatement("function x() {x()}", false);  
    assertNotNull(expr);  
    assertTrue(expr.hasError(ErrorType.RECURSIVE_FUNCTION));  
}
```

@Test

```
public void functionDefStatementEnsuresClosingBrace() {
```

```

FunctionDefinitionStatement expr = parseStatement("function test() { ", false);

assertNotNull(expr);

assertTrue(expr.hasErrors());

}

```

The first of the two tests included above is an invalid test. The Catscript grammar supports recursive function calls whereas the test's purpose is to output an error when a recursive function call is made. The second test included above tests a functionality that was already tested through the tests provided by Professor Gross. In lieu of the two tests provided by team member 2, team member 1 developed three new tests. The tests are included in the PortfolioTests class file in the included source code as well as in the technical writing section of this portfolio.

Design Pattern

A design pattern that was used in the program is memoization. Memoization is the process of storing the results of a function to reduce processing strains. Typically memoizing a function is used for particularly expensive functions in regard to processing time.



```

36  static HashMap<CatscriptType, ListType> cache = new HashMap<>();
37  @ public static CatscriptType getListType(CatscriptType type) {
38      ListType listType = cache.get(type);
39      if (listType == null) {
40          listType = new ListType(type);
41          cache.put(type, listType);
42      }
43      return listType;
44  }
45

```

In the CatscriptType class, there is a function call that returns the type associated with a list. The function was memoized using a hashmap. If the type has already been instantiated, it is simply pulled from the hashmap. This reduces the processing time for list types that have already been used. In this case, memoization was used because the getListType method is used frequently. Its frequent use would have a small but noticeable impact on the processing time of the program.

Technical Writing

The two tests discussed in the teamwork section were not valid tests. Included below are the three tests provided by myself.

Tests Provided by Myself

```

PortfolioTests.java
1 package edu.montana.csci.csci468;
2
3 import ...
4
5
6
7
8
9 public class PortfolioTests extends CatscriptTestBase {
10
11     @Test
12     void nestedIfStatementWorksProperly() {
13         assertEquals( expected: "1\n", executeProgram( src: "if(true){ if(true) { print(1) } }"));
14         assertEquals( expected: "1\n2\n", executeProgram( src: "if(true){ print(1) if (true) { print(2) } }"));
15         assertEquals( expected: "1\n1\n", executeProgram( src: "if(true){ print(1) if(true){ print(1) } else { print(2) } } else { print(2) }"));
16         assertEquals( expected: "1\n2\n", executeProgram( src: "if(true){ print(1) if(false){ print(1) } else { print(2) } } else { print(2) }"));
17     }
18
19     @Test
20     void functionCallFromAnotherFunctionWorksProperly() {
21         assertEquals( expected: "1\n2\n3\n", compile( src: "function foo(x) { print(x) }\n" + "function foo1(x) { foo(x) }\n" +
22             "foo1(1)\n" +
23             "foo1(2)\n" +
24             "foo1(3)"
25         ));
26     }
27
28     @Test
29     public void booleanEqualToInt() {
30         Object expr = evaluateExpression( src: "1 == true");
31         assertEquals( Boolean.FALSE, expr);
32         Object expr2 = evaluateExpression( src: "1 == false");
33         assertEquals( Boolean.FALSE, expr2);
34     }
35 }
36

```

The required documentation was not provided by team member 2. Included below is the documentation developed by myself.

Catscript Guide

The following is a simple documentation explanation of the Catscript programming language.

Introduction

The Catscript programming language is a simple, statically typed scripting language. It's inclusion of all basic programming structures (for loops, if statements, functions, etc) allows any new user to effectively produce a desired product. Furthermore, the simplistic grammar structure enables anyone to use the Catscript language as a learning tool for developing a compiler.

Here is an example of a short Catscript program:

```
function addUntilEquals(x, y) {  
  if (x == y) {  
    return x  
  }  
  else {  
    return addUntilEquals(x + 1, y)  
  }  
}  
addUntilEquals(4, 14)
```

The above function takes in a starting value x and a goal value y. If x is equal to the goal number, the function returns x. If x is smaller than the goal number, the function increments x and recursively calls itself with the incremented value.

Features

For Loops

Catscript uses a very restricted for loop system through the use of lists. The user may define an iterator variable that is used to iterate through each item in a list. The list may contain any variable type. There is no explicit way to cycle through a for loop for a specified number of times. Instead, the user may iterate through a list containing the intended number of items to achieve the same effect. Catscript also allows for loops to be nested within one another if the user desires. Two examples of a for loop in Catscript and their outputs are as follows:

```
Code:  
for (x in [1,2,3,4,5]) {  
  print(x)  
}  
Output:  
1 2 3 4 5
```

```
Code:  
for (item in ["apple", "orange", "banana"]) {  
  print(item)  
}  
Output:  
apple orange banana
```

The above loops will print each value included in the respective list.

If Statements

If statements in the Catscript language are comparable to if statements in many other languages such as Java. An if statement consists of a conditional statement, a group of statements to execute if the conditional statement is true, and optionally an else block with statements to execute if the conditional statement is false. Similar to for loops in Catscript, if statements can be nested within one another if the user desires. Two examples of an if statement and their outputs are as follows:

```
Code:  
if (false == not true) {  
  print("This statement is true.")  
}  
else {  
  print("This statement is false.")  
}  
Output:  
This statement is true.
```

```
Code:  
if (false != not true) {  
  print("This statement is true.")  
}  
else {  
  print("This statement is false.")  
}  
Output:  
This statement is false.
```

The above if statements will print the specified string corresponding to the outcome of the conditional statement.

Print Statements

Print statements in the Catscript language are very similar to print statements in other languages such as Java. A print statement outputs the specified target. The argument for a print statement in catscript can be an int, bool, or string. All objects are wrapped into a string at compile time for the print statement to successfully output. Two examples of a print statement are as follows:

```
Code:
  print(1)
Output:
  1
```

```
Code:
  print("Hi")
Output:
  Hi
```

The above prints statements output their included argument.

Variable Statements

As a statically typed language, variable statements are a key component in Catscript for ensuring a variable is the desired type. The variable statement consists of an the var keyword, an identifier, an optional type declaration, and the identifiers associated value. If the variable type is not specified, the type is inferred. If the variable type is specified, the parser ensures that the variable value is compatible with the associated type. Two examples of a variable statement are as follows:

```
Code:
  var x = 1
  print(x)
Output:
  1
```

```
Code:
  var x : string = "Hi"
  print(x)
Output:
  Hi
```

The above variable statements assign a value to a variable. A print statement is then called using the variable names.

Assignment Statements

An assignment statement in Catscript is used to change the value of an exisiting variable. If the variable type is defined when the variable is initialized, the type of the argument in the assignment statement must match. If the variable type is not defined when the variable is initialized, the type of the argument in the assignment statement must match the type inferred by the parser. Two examples of an assignment statement are as follows:

```
Code:
  var x = 1
  x = 14
  print(x)
Output:
  14
```

```
Code:
  var x : string = "Hi"
  x = 14
  print(x)
Output:
  Error: Incompatible Types
```

The first assignment statement above changes the value of the specified variable. The second assignment statement above produces an error for incompatible types.

Return Statements

The return statement in Catscript is used to return a value from a function similar to many other languages such as Java. A return statement is not required for a void function, but it may be used if the user desires. In this case, the compiler automatically inserts a return instruction for a void function because Java will report an error without the return instruction. If the function type is not void, a return statement is required and the argument for the return statement must match the required function return type. Two examples of a return statement are as follows:

```
Code:
  function foo() {
    var x = 2
    x = 14
    print(x)
```

```
    return
  }
  foo()
Output:
14
```

```
Code:
function foo() : string {
  var x = "Hi"
  x = "Goodbye"
  return x
}
print(foo())
Output:
Goodbye
```

The first return statement above returns with no argument which indicates that the function successfully executed. The second return statement returns a variable with a value sharing a type with the required return type of the function.

Functions

Functions in Catscript are used in a similar way to methods or functions in other languages such as Java. Functions in catscript are explicitly declared using the function keyword. Function definitions may include a required return type or they may be left as void. Functions may have one or many arguments with type defined or no type defined. Two examples of a function definition and its respective call are as follows:

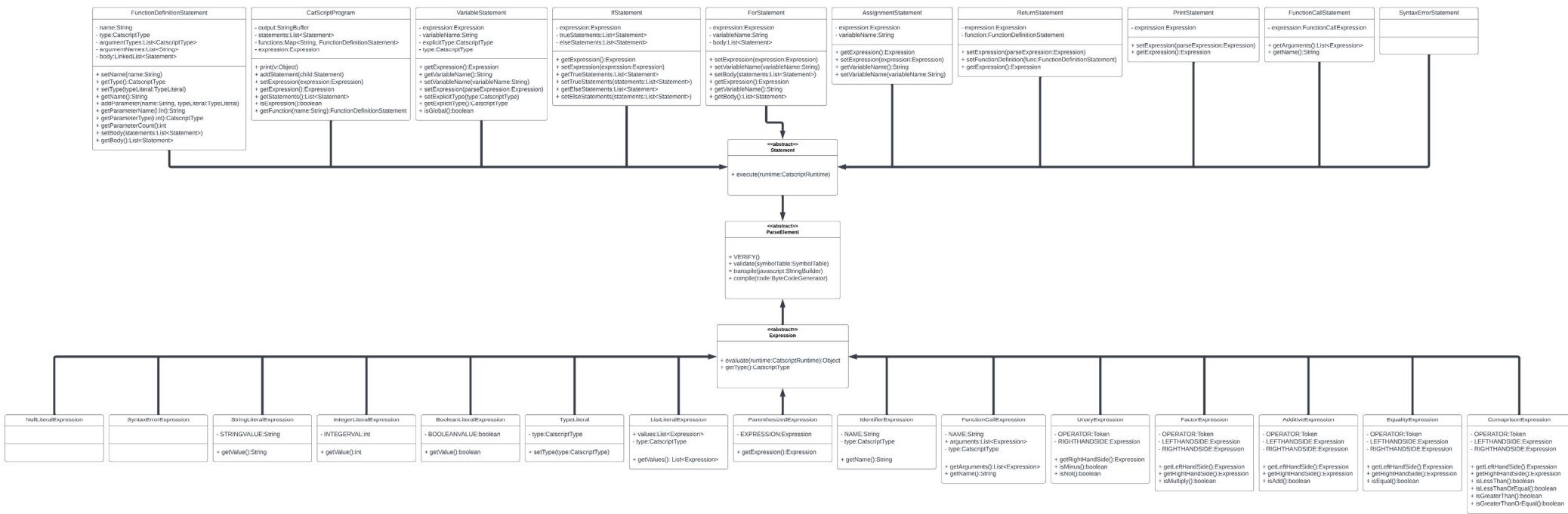
```
Code:
function foo() {
  var x = 2
  x = 14
  print(x)
  return
}
foo()
Output:
14
```

```
Code:
function foo(input) : string {
  return input
}
print(foo("Hello"))
Output:
Hello
```

The above function definitions and calls define a function "foo" with no return type and a string return type. The function is then called after the definition.

UML

Included on the next page is a UML class diagram produced on LucidChart. This UML diagram shows the abstract class ParseElement and its associated classes. In an attempt to make the diagram viewable, the next page has been translated to a landscape orientation.



Design Trade-Offs

One of the major design trade-off decisions that we made was generating our own parser rather than using a parser generator. A parser generator is a tool that produces a parser based on the grammar of the language. In most cases a parser generator is sufficient to complete a compiler. A major downside to a parser generator, however, is that they often do not produce a user-friendly parser. For example, a parser generator does not incorporate error recovery. The error messages that are produced can often be vague or hard to understand. Our decision to produce our own parser allowed us to incorporate error recovery, our own error messages, and gave us some flexibility to improve efficiency.

Another major reason why this decision was made was to understand how a parser works. Using a parser generator would have resulted in the same end program. However, we would have missed out on learning the inner workings of a parser. By producing our own parser, we were given the opportunity to understand exactly how a recursive decent parser works as well as why it is the most popular choice for parser design among many companies.

Software Development Life Cycle

For this project, the software design strategy that we used was test-based development. This means that we have a group of code tests that are developed to test a specific aspect of our compiler. We then develop the compiler to get all tests passing. This design strategy has several benefits. One of the major benefits is that bugs in our code are easily weeded out. Another benefit of test-based designing is that there are clear milestones for completion of the compiler. In this case, there were clear test groups used for the lexing, parsing, and compiling stages of the compiler.

One disadvantage of this design strategy is that there is very little flexibility in the code. There is a clear purpose to the tests and the code must accomplish that purpose. In this case, it is beneficial to understand how a compiler works. In the real world, on the other hand, this design strategy may limit the potential scope of a program.