

Cameron Watson
Capstone Document

CSCI 468 - Compilers
Spring 2023

Section 1: Program

A zip file of the final repository is included in this directory.

Section 2: Teamwork

This project was created with a test-driven development life cycle. The following tests were created by my partner, Luke Cormier. He was able to provide higher-level tests to check my Catscript program for any flaws. The tests provided were as follows:

Test 1

```
public void returnStatementWorksTest1() {
    assertEquals("11\n", executeProgram(
        "function foo(x : int) : int {\n" +
            "    return x + 2\n" +
            "}\n" +
            "print(foo(9))\n"
    ));
}
```

Test 2

```
public void localAndVarStatementWorksProperlyTest() {
    assertEquals("16\n", executeProgram("var x = 16\n"+
        "var y = x\n" +
        "print(y)"));
    assertEquals("null\n", executeProgram("var x = null\n" +
        "print(x)"));
    assertEquals("5\n18\n93\n", executeProgram("for( x in [5, 18, 93] ) {\n" +
        "    var y = x\n" +
        "    print(y)\n" +
        "}\n"));
}
```

Test 3

```
public void varInsideFunctionWorksProperlyTest() {
    assertEquals("82\n", executeProgram("function foo() : int {\n" +
        "    var x = 82\n" +
        "    return x\n" +
        "}\n" +
        "print( foo() )\n"));
    assertEquals("18\n21\n35\n", executeProgram("for(x in [18, 21, 35]) { print(x) }"));
}
```

The first test ensures that the return statement works when using the additive functionality, and also includes a print statement to parse through. The second test verifies variable initialization works properly while also manipulating the data

through division. It also includes a for loop to ensure you can use declared variables in for loop logic. The final test checks to make sure function declaration and function calls work. The tests are located in /csci-468-sprint2023-private/src/test/parser/ProvidedPartnerTests/PartnerTests.

The documentation provided by my partner Luke goes through the basic functionalities of the Catscript language. It is important to keep in mind that Catscript is statically typed. The documentation starts by reviewing simple comparisons. The document then proceeds to explain included operations like addition, subtraction, multiplication, and division. The guide also mentions the implementation of unary operations in Catscript. The language also includes lists. You are able to initialize lists and manipulate lists in Catscript. A major factor of Catscript is functions. You are able to call functions and also create functions with parameters. The parameters can be any supported type in Catscript. The functions in Catscript can return any type and can also be sent any type. The language also includes return statements which can also compute simple operations on the same line. Catscript also includes if and for statements. It is important to point out that in order to use an else statement it must be before an if statement. Lastly, Catscript supports print and variable statements. Catscript supports all the fundamental parts of a program.

My primarily work on the Catscript program was writing code that passed tests provided by Carson Gross. The project was split into three checkpoints, the first being Tokenization, Expression Parsing, Statement Parsing and Evaluation and Compliation. Each checkpoint consisted of numerous tests that had to be working before moving on to the next checkpoint. For example, tokenization had to be completely functional before working on the parser. This is because the parser takes in a stream of tokens provided by the tokenizer, so it was my responsibility to ensure there was no bugs or failures in methods I wrote that fulfill the test requirements. This pattern would repeat itself for all of the other checkpoints in the project.

Section 3: Design pattern

An important design pattern I used in the parser is memoization. Memoization is the practice of improving efficiency and run time in the architecture of a program. This is done by storing the outputs of any function in a cache or a memory structure. There is no need to re-compute values that have already been computed. This is when memoization comes into play. A great time to use memoization is during an expensive operation. An expensive operation is a function or statement that has a high run time. For example, in a file that contains movies with corresponding genres, you could create memoization to improve efficiency whenever parsing through the collection of objects for horror movies. Whenever you want to call horror movies, you could call the memoization created that stores all of the horror movies instead of redoing the parse through a massive file. This dramatically improves the efficiency of your data collection process.

Sometimes it is not worth it to use memoization. There is no need to use memoization on simple functions. The cache takes up memory, and if you have memoization for every simple function or statement, it might end up making your program less efficient. Another time to not use memoization is when the data changes. This is because the value stored in the cache would be outdated compared to the newly manipulated data. For example, if you were to add a horror movie to the previous example, it would make the cache miss that newly added horror movie.

In the source code, the memoization design pattern used in the Catscript parser is located in `csci-468-spring2023-private/src/main/java/parser/CatscriptType`

The way memoization was implemented in the Catscript parser started with implementing a `HashMap`. This will serve as our cache and will be static. It is static because there is no reason for memoization to be dynamic because of how it fundamentally works. The function then grabs the type of Catscript. It then checks to see if an instance of that type already exists in the previously created `HashMap`. If it is equal to null or doesn't exist, it will add the `listType` to the `HashMap`, but it also checks to see if it is already there to prevent redundancies. This is an efficient way to save on runtime in the Catscript parser.

Section 4: Technical writing

Catscript Guide

This document is a guide for catscript. It will discuss the function, features, operation of Catscript as well as how to code and the expected output after the execution of the code block.

Introduction

Catscript is a simple scripting language. Here is an example:

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

Features

Catscript holds many features such as support for comparison expression. Such comparisons as greater than, less than, greater equal, less equal and equal equal are all supported by catscript. Here is an example:

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

Output:

```
false
true
false
true
```

Catscript also supports the use of equality expressions. Such equalities include the use of equal and not equal expressions which are essential for conditional statements. Here is an example:

```
1 == 1
1 != 2
```

Output:

```
true
false
```

The Catscript language has built in support for additive, and factor expressions for expressions in math. These are essential for calculations in the Catscript language. Here is an example:

```
int w = 1 - 1
print(w)
int x = 1 + 1
print(x)
int y = 2 * 3
print(y)
int z = 6 / 3
print(z)
```

Output:

```
0
2
6
2
```

Unary is also supported through Catscript. This provides the capability to code operands with operators to return a result. Here is an example:

```
not true
-1
```

Output:

```
false
-1
```

Primary expressions are also supported by the Catscript language. Primary expressions are used to identify the results and are the building blocks of more complex expressions. Here is an example:

```
int x = null
bool y = true
str z = "false"
```

Null and integer would be the primary expressions.

A list literal is an expression followed by another expression in a list. Here is an example:

```
function foo() : list {
    return [1, 2, 3]
}
print(foo())
```

Output:

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

Function calls are used in Catscript to create an identifier with an `argument_list`. Function calls allow users to declare a function, create the body of the function then call the function to run the code in the body. Here is an example:

```
function foo() {  
    var x = 10  
}  
foo()  
print(x)
```

Output:

```
10
```

Argument expressions are supported by Catscript which are important in creating argument variables for functions to take in. Here is an example:

```
function foo(x) {  
    print(x)  
}  
foo(1)
```

Output:

```
1
```

Types can also be utilized by the Catscript language to create types for variables such as integers, strings, booleans, object and lists. Here is an example:

```
function foo() : list<int> {  
    return [1, 2, 3]  
}  
print(foo())  
int w = 1  
print(w)  
str x = "Hello"  
print(x)  
bool y = true  
print(y)  
object z = object  
print(z)
```

Output:

```
[1, 2, 3]  
1  
Hello  
true  
object
```

Parameters are also a part of the Catscript language and allow for assigning identifiers to expressions. Here an example:

```
str: "Liam"  
## Or a parameter list  
[str: "Liam", str: "Joe"]
```

For loops

Catscript is also built to support the use and function of for loops. With for loops, blocks of code can be executed a specified number of times. Here is an example:

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

Output:

```
1  
2  
3
```

Return statements

Return statements are crucial in getting a value from a function when it runs a block of code. Here is an example:

```
function foo() {  
  var x = 10  
  return x  
}  
print(foo())
```

Outputs:

```
10
```

If statements

If statements are another functionality that is available in the Catscript language. If statements allow for conditions that need to be met in order for the program to execute. Here is an example:

```
int y = 12  
int x = 11  
if(x > 10){  
  print(x)  
} else {  
  print(y)
```

Output:

```
11
```

Print statements

Print statements are another feature of the Catscript language that allow for printing a desired line of text or a variable. Here is an example:

```
print(1)
```

Output:

```
1
```

Variable statements

Variable statements are used in the Catscript language to identify operands as variable in the language to be called upon or even altered in a later expression. Here is an example:

```
var x = 10  
print(x)  
var y = x + 10  
print(y)
```

Output:

```
10  
20
```

Assignment statements

Assignment statements are used in the Catscript language to give variables or operands a value such as "true", "false", or "null" to variables such as strings, integers, and booleans. Here is an example:

```
var x = null  
print(x)
```

Output:

```
null
```

Function Call statements

Catscript is also built to support the use and function of Function call statements. This allows users to call the function they've created in order to run the code built into the body of the function. Here is an example:

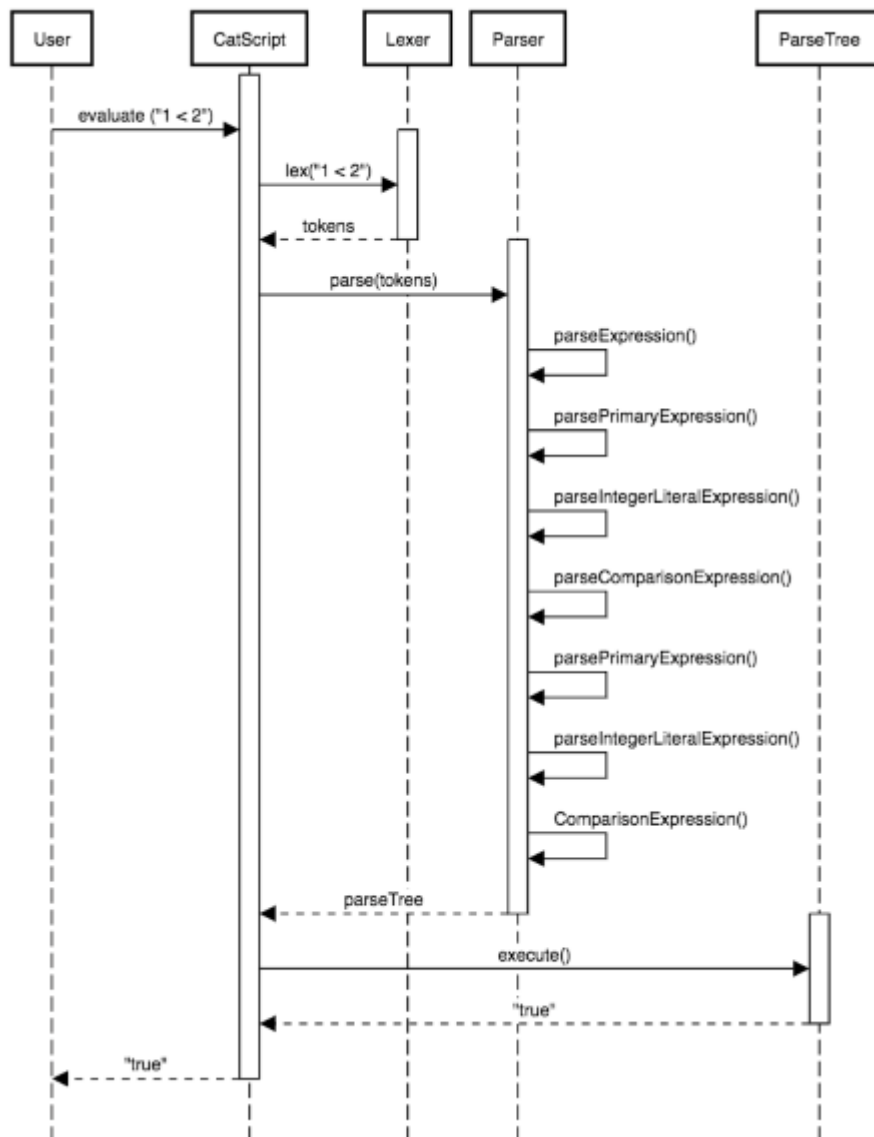
```
function foo() {  
  print(1)  
}  
foo()
```

Output:

```
1
```


Section 5: UML

Catscript Comparison Sequence Diagram



The figure above is a sequence diagram of a simple comparison expression "1 < 2", which should return true. It first starts with the user sending a string to Catscript. The string is read from left to right, so you will be able to see how the Catscript program handles a simple comparison string. Catscript then sends it to the lexer. The reason for this is that the Catscript program uses the lexer to break up the inputted string into a stream of tokens. This is a major component of the Catscript program, because without a tokenizer the parser will not know how to sort through it.

In the next step of this example, Catscript sends the tokens to the parser. The parser then calls `parsePrimaryExpression()` to identify the type of values. For this example, the function identifies '1' as an integer with the help of `IntegerLiteralExpression()`. After identifying the integer, the parser then calls the `parseComparisonExpression()`

to identify the "<"; Catscript knows it as "LESS." The parser then calls `ParsePrimaryExpression()` again to identify the '2' with the help of `IntegerLiteralExpression()`. After the parser reaches the end of the file, it then calls `ComparisonExpression` in the parser to evaluate the expression. The evaluate function grabs both 1 and 2 and assigns them to a left and right-hand side value. It then checks to see if it is true or false, in this case, it is true. The parser then sends the parse tree back to Catscript and then calls `execute()`. After executing the parse tree, the return value is passed back to Catscript which is then passed to the user.

Section 6: Design trade-offs

A fundamental design choice for the capstone project was the decision between a parser generator and a recursive descent parser. Both a parser generator and a recursive descent parser contain lexical grammar and language grammar. In a parser generator, you are injecting files with specific syntax set by the developers of the parser into the generator. They both create a tokenizer and pass a string into it. The tokenizer generates tokens that then get passed to the parser. The parser then converts it into a parse tree.

Recursive descent is the most popular form of parser and for good reason. They are used in the most popular programming languages. The recursive descent parser is built recursively with no way to parse backward. Based on the grammar, you are able to design the parser to recursively call other functions as listed in the grammar.

Parser generators function differently and act more like a tool. They take in grammar files that specify how the generator should work. Parser generators do not develop the machine code or byte code like recursive descent parsers do. So in order to generate a lexer you will need a lexical grammar file. The lexical grammar then gets processed by the generator and then turned into generated code. You are then able to define a language grammar, and just like the lexical grammar, it gets processed and turns into generated code.

If you are in need of a basic parser, it would be wise to take advantage of a parser generator tool like Antlr. However, if you are a company that will use the parser extensively it would be better to create your own recursive descent parser. This is for flexibility reasons.

There are pros and cons to using a parser generator or a recursive descent parser. The big difference between a recursive descent parser and a parser generator is the recursive descent parser is built with recursion in mind. There is a parser generator called Antlr that generates a recursive descent parser, but it is not fully customizable like the Catscript parser that was built with the recursive descent algorithm. Another pro for creating your own recursive descent parser is that it is debuggable. Since the parser generator creates its own code, it is difficult for the programmer to customize the tokenizer because the code is so abstract.

Another reason to create your own recursive descent parser and avoid a parser generator is because of the natural hierarchy. There are so many internal components to a parser generator. You will not have true access to the generated tree without using the visitor pattern. For example, for the Antlr parser generator, the parse tree class is internal to Antlr. In the Catscript program, there are no restrictions to the parse elements, because it was made by hand.

Section 7: Software development life cycle model

The software development life cycle model used to create the compiler was Test Driven Development. This development cycle focuses on writing the tests and expectations for your code. There are five stages to Test Driven Development. The first stage is to write a simple test for your program. The most important part of this life cycle is writing tests that match your expectations. The next stage is to run the test. The test will fail because there is no implementation to match the logic of the test. Next, is to actually write the code that makes the test pass. It is important to only write enough code to make your test pass. Next, you will refactor your code. This ensures there are no duplicates or bugs in your code. Examples of refactoring are moving your code to a more logical spot, or even splitting your functions into smaller sub-functions which makes the program even more versatile. Finally, you would go back to step one and repeat the process until your program meets all expectations. If you maintain this life cycle your development might take longer than other approaches, but you will have close to no bugs or unexpected logic.

There are an enormous amount of good outcomes that come out of this development cycle. I personally prefer it over others because it keeps you on track. Before moving on to the next component of the compiler, like executing expression statements, it was reassuring that the precious parts of the program like the scanner were working perfectly. Another strength of this lifecycle is because of how the cycle is built, you have complete control over your tests.

A disadvantage of this type of development cycle is time constraints. Since you can not move on to the next step of the program until you have passed all the tests you are required to get the foundation completely right before working on other easier parts of the program. Oftentimes we can overlook a simple mistake because we have been staring at it too long.

Another disadvantage of this development cycle is there is little room for flexibility. If a team were to work on this project using Test Driven Development, it would be difficult to do asynchronous development. This is because a test has to be completely passed before moving on to the next stage of development. The team could not work on different aspects of the project concurrently.

Overall, I believe this to be the best possible software development cycle when building a program by yourself. If it was more than one person developing this I would prefer an asynchronous development cycle. This was a great way to keep me on track and also to keep my code from causing problems in the other components of the compiler.