

CSCI 468 - Spring 2023 Portfolio

By Mike Gotta

In association with Layton McCafferty

Section 1: Program

This capstone project should be viewable in the /capstone folder

Section 2: Teamwork

This class had an emphasis on individual work. However, Layton provided three additional unit tests to my project that way I could continue to test the codebase I have been working on. These tests are runnable under `test/java/edu.montana.csci.csci468/parser`

Section 3: Design Pattern

This implementation of the Catscript program uses the Memoization method as its primary design pattern. It is viewable in the 'CatscriptType' class. The pattern is in use to primarily ensure that methods are run only the number of times they need to be. Each method returns a value that is then stored in a lookup table. This table can be searched through if the user provides a key as an argument. This will save time by storing values instead of running methods repeatedly.

Section 4: Technical Writing / Documentation

Catscript Documentation

Mike Gotta, Layton McCafferty

1 Introduction

Catscript is a statically typed programming language that compiles to JVM Bytecode. It uses a recursive descent parser to evaluate expressions, statements, and primitive types.

2 Expressions

An expression is a piece of code that will evaluate to an integer, string, boolean, or some value. Using the recursive descent parser, expressions have a hierarchy of evaluation. This hierarchy will be described below.

Equality Expression:

The equality expression is how the Catscript program can use the '==' and '!=' operations. This can be illustrated like this:

Var1 == Var2

Var1 == 5

Var1 != 5

Comparison Expression:

The comparison expression is how the Catscript program can use the mathematical relational operations. Greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=). These are used in Catscript like this:

Var1 > Var2

Var1 <= 5

Var1 < 5

Foo >= 10

Additive Expression

The additive expression is how the Catscript program can use the seemingly basic 'plus' and 'minus' symbols. One interesting feature is how the '+' can also be used to serve as string concatenation. The following are valid additive expressions:

Var1 + 8

Var1 - 5

foo + "bar"

Factor Expression

The factor expression is how the Catscript program can use the basic operations of multiplication (*) and division (/). They are used in Catscript like so:

```
-----  
Var1 * -1  
10 / 2  
(Var1 * Var2) / Var3  
-----
```

Unary Expression

The unary expression is how the Catscript program can implement the not ('not') and negative ('-') operators. They are used in Catscript like so:

```
-----  
-2  
not false  
not not false  
-----
```

Identifier Expression

The identifier expression is how the Catscript program can implement variables. This is stored as a string. The variable name is used to look up its value in the symbol table of the program. A variable can be named anything that a regular Java string can be named.

Primary Expression

In Catscript the most basic expressions can be any of the following

- Identifier
- String literal
- Boolean literal
- Integer literal
- List literal
- Null literal
- Parenthesized expression
- Function call

Due to the nature of recursive decent, any expression containing another expression will eventually boil down to some basic form of a primary expression.

3 Statements

Just like how Catscript expressions use recursive decent to jump down the hierarchy of expressions, statements act much in the same manner, only they change the programs state.

Print Statement

The print statement is always called with the ‘print’ keyword, followed by any expression. This is how the program can return a value to the standard output of the program. The following are valid print statements in Catscript:

```
-----  
print("hello")  
print(var1)  
-----
```

Variable Statement

The variable statement is how Catscript declares and assigns a new variable, this will also require an expression to be assigned. The variable statement does require a few items:

- Require ‘var’ keyword
- Require some sort of variable name
- Optional ‘:’ to specify type
- Require ‘=’ symbol.
- Require some sort of expression.

The following are valid variable statements in Catscript:

```
-----  
var test : int = 5  
var test2 = 2  
-----
```

Assignment Statement

The assignment statement is how Catscript can alter the value of an already defined variable. The following are acceptable in Catscript:

```
-----  
var1 = "hello"  
foo = 1 * 3  
-----
```

If Statement

The if statement is how Catscript can handle some sort of conditional statement. It is very similar to a Java if statement and requires a few items:

- Requires 'if' keyword
- Requires '(' symbol
- Requires some expression
- Requires ')' symbol
- Requires '{' symbol
- Requires a statement to execute
- Requires '}' symbol
- Optional 'else' keyword (if statements can exist inside here)

The following would work in Catscript:

```
-----  
If (foo == "bar"){  
    Print("buzz")  
}  
Else if (foo == "foo") {  
    Print("bar")  
}  
-----
```

For Statement

The for statement is how Catscript iterates through objects. It is very similar to how most other modern programming languages would iterate. It requires a few items:

- Requires 'for' keyword
- Requires '(' symbol
- Requires some sort of variable name
- Requires 'in' keyword
- Requires some sort of expression
- Requires ')' symbol
- Requires '{' symbol
- Requires a statement to evaluate
- Requires a '}' symbol

The following is acceptable in Catscript:

```
-----  
For (var in [1,2,3]) {  
    Print (var + " found you")  
}
```

Function Definition Statement

The function definition statement is used to define functions in Catscript that can be called elsewhere in the program. They can either return a specified type or return nothing at all. It requires a few items:

- Requires 'function' keyword
- Requires a function name
- Requires '(' symbol
- Requires a parameter list
- Requires ')' symbol
- Optional ':' followed by a defined return type
- Requires '{' symbol
- Requires a statement to evaluate
- Requires '}'

The following would work in Catscript:

```
-----  
function test (a : int) : int {  
    return a + 6  
}  
-----
```

Function Call Statement

The function call statement is used to call an already defined function in Catscript. It requires a few items:

- Requires a function name (that exists and is defined)
- Requires '(' symbol
- Requires a list of arguments
- Requires ')' symbol

The following would be acceptable in Catscript:

```
-----  
test(a)  
-----
```

Return Statement

The return statement in Catscript is used to exist a function prematurely, or at the end of a functions scope. This can only happen inside the scope of a function and should return a value. It requires a few items:

- Requires 'return' keyword
- Requires an expression to be returned

The following would be acceptable in Catscript:

```
-----  
function test (a : int) : int {  
    return a + 6  
}  
-----
```

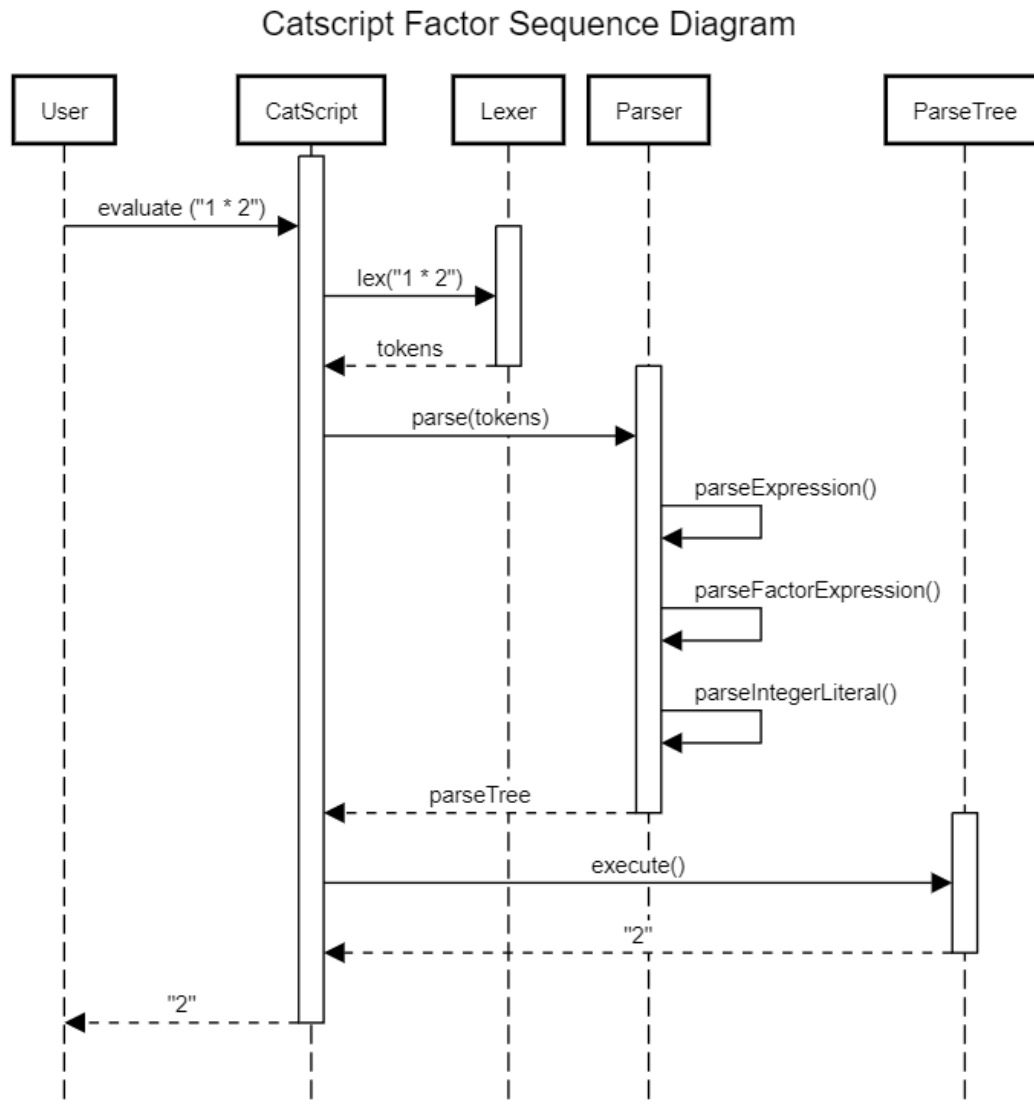

4 Type System

Catscript uses a straightforward type system. The following are acceptable in Catscript:

- int (Any 32-bit integer)
- string (like Java)
- bool (True/False)
- list<x> (A list of values of type 'x')
- null (the null type)
- object (any type of value we want that is define by Catscript)

Section 5: UML

There was no UML needed for this class, the design was already created by the professor. However, a small part of that UML can be illustrated below:



This UML diagram is showing how the simple multiplication of '1 times 2' is run through the Catscript parser. First the user asks the program to evaluate the statement. Then the Catscript is tokenized and run through the parser. The parser recursively finds the appropriate functions to act on the tokens we fed through. This parse tree then returns us the actual multiplication of 1 times 2 and sends back to the user the value of 2.

Section 6: Design Trade-offs

The design trade-off for Catscript was that it implements a parser with a recursive decent algorithm instead of a parser generator. This was chosen because recursive descent roughly mirrors the recursiveness of the grammar. This kind of algorithm is also very straightforward and repeatable if you must do it for a lot of different functions. It also seems to be a more realistic approach to how to create a compiler. It may not be the most efficient, but it seems to be the easiest to understand, and more importantly to modify and debug.

Section 7: Software Development Life Cycle

The Test-Drives Development model was used. This TDD is a process in which large test suites are created and run to see if the codebase is acting correctly in accordance with the parameters passed into the test functions. Anytime a new feature is added to the project, tests are added first to make sure that the new features specifications work correctly. Test will not work until the feature is implemented correctly. This means that software is developed only when tests are passing correctly.