CSCI 468

Spring 2021

Jordan Kelly

Garret Turner

Section 1: Program

The source code for this project can be found in this same directory, under the name of source.zip. *Catscript* is the name of the project and is a small statically typed programming language that takes the best principles from other languages and applies them.

Section 2: Team Work

The teamwork in this project was two-fold, team-member 1 wrote tests and documentation for team-member 2's project, and team-member 2 did the same for team-member 1. The workload for each team-member was 70/30, 70% of the work being the project, and 30% writing the tests and the correct documentation for the project. As teammates, we learned how to trouble shoot and find loopholes in the project that neither one of us would have seen solo. By trial and error, the approach that was taken created a bullet-proof program that is better than before by far.

Section 3: Design Pattern

The design pattern that we implemented was the memoization pattern, in essence it increased the speeds of the program greatly, by storing inputs and outputs of like data, thus eliminating the need to do expensive and time-consuming function calls, when that function call has already been done. An example of this being put to use is as follows:

```
static final HashMap<CatscriptType, ListType> CACHE = new HashMap<>();
    public static CatscriptType getListType(CatscriptType type) {
        ListType listType = new CACHE.get(type);
```

```
        if (listType = CatscriptType){
            return listType;
        }
        return new ListType(type);
```

Instead of running every listtype that comes through, it is stored in a hashmap, which is then checked for the same input and output, saving time on computing.

Section 4: Technical Writing

Catscript Guide

Introduction

Catscript is a small statically typed programing language that compiles into JVM bytecode.

It also has some very unique features that were implemented like type - inference and NO MORE SEMI COLONS.

The reason we did this Language was to test our abilities and make a language that was the best of the big languages like Python and Java and make it into a language of its own.

Features

For - Statement

The for statement in catscript is built to mimic the java for statement

- for example a for statement would be set as follows

```
for (identifier in expression){
    "Statement"
    "Statement"
    "Logic"
}
```

If - Statement

The if statement is also much like any other coding language, with the ability to have if else, or continuous if's

```
if(expression){
  statement
}else{
```

```
    statement
}
```

Print Statement

The print statements in Catscript are very similar to Python print statements

```
print("Hello_World")
```

Variable Statements

The variable statements in Catscript are also a breeze to deal with and are written like so, with type being inferred, we just type check to make sure it is non-void!

```
var x = 10
```

```
var x : int = 10
```

Assignment Statements

Assignment statement are much like java

```
x = 10
```

```
var = 'easy'
```

Function Calls

Function calls are again alot like python and are easy to do

```
foo()
```

```
func1(1,2,3,4)
```

Function Declarations

Function Declarations in CatScript are easy with return type declarations

```
function foo(a,b,c) : int {
    for(){
        if(){
        }else{
```

```
        }
    }
    return int
}
```

Type System

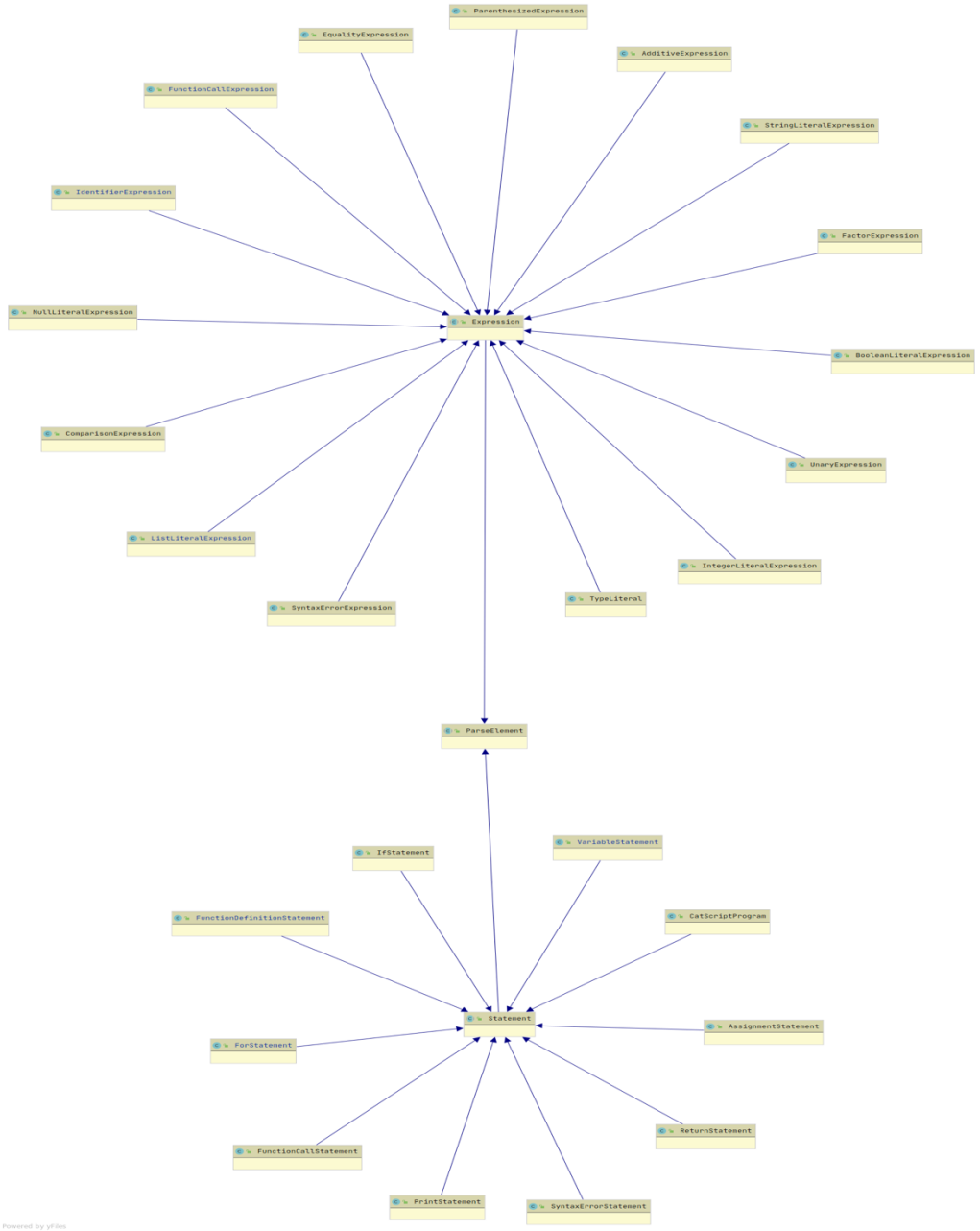
Catscript was also written with a relatively small typesystem for ease of use.

Which include:

- int - a 32 bit integer
- string - a java-style string
- bool - a boolean value
- list - a list of values with the type 'x'
- null - the null type
- object - any type of value

Section 5: UML

This UML explains, in short, expressions and literals are associated with expressions, as statements are associated with statement, and both are connected to the parse element.



Section 6: Design Trade-Offs

Design trade-offs that we did have was between using recursive decent and a parser generator. Recursive decent uses the top-down approach and does works closely with the grammar to make a much faster computational system, with each production a method is created and on each right-hand side of the production the strings are then matched when called. This makes things much easier to follow.

With a parser generator, computations are slowed greatly, and were mostly used in older systems. By using this top-down approach, we were closer to working with real world experience, and what we will be working with in the field of Computer Science.

Section 7: Software Development lifecycle Model

The Development model that we used for this project was Test Driven Development (TDD). Using TDD made the project much simpler when troubleshooting and figuring out what needed to be worked on and what was complete. Using TDD was extremely useful when marching through the code and completing it section by section as tests passed. Encountering something as large as this project can be extremely daunting and knowing where to start is a huge leg up in developing a language like this. When using TDD we had a test that needed to pass and the function that was needed to be adjusted to make it pass, making it seamless writing code to satisfy the requirements. In the future I would love to work on projects with TDD because it takes all of the guess work out of the project and takes it from a birds-eye view, down to a piece of code that can be worked on.