Compilers; CSCI 468

Semester: Fall 2024

Cameron Oberg,

Jon Neumann

Section 1: Program

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

Section 2: Teamwork

My team had one other member. I implemented the CatScript tokenizer and parser, while the team member helped develop documentation and tests. See CapstoneTests.java for the tests.

Section 3: Design pattern

I used the memoization pattern while designing this compiler. This was to help runtime by caching a type expression and by consequence, simplifying the amount of times the compiler had to read the same type on the same objects. The pattern's code can be found in the CatscriptType.java file.

```
    static HashMap<CatscriptType, ListType> cache = new HashMap<>();
    public static CatscriptType getListType(CatscriptType type) {
        //Look in the map if it's there, return it, if not, stick it in there
        if(cache.containsKey(type)){
            return cache.get(type);
        } else {
            ListType l = new ListType(type);
            cache.put(type,l);
            return l;
        }
    }
```

Section 4: Technical writing. Include the technical document that accompanied your capstone project.

See the Catscript.md file in this project's directory for a preview of the programming language. Additionally, the contents have been pasted:

Catscript Guide

This document should be used to create a guide for catscript, to satisfy capstone requirement 4

Introduction

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

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

Features

Type System

Catscript is a statically typed programming language. The supported types in Catscript are as follows:

- Integers
- Strings
- Booleans
- Objects
- Lists
- Null

Catscript will check that types are compatible and will throw type errors if there are any type conflicts.

Expressions

Literal Expressions

Catscript types can be expressed as Literal Expressions.

Expressions

Literal Expressions

Catscript types can be expressed as Literal Expressions.

```
int x = 3
string y = "Hello"
boolean z = true
object a = null
list b = [1,2,3]
```

Additive & Factor Expressions

Catscript supports basic arithmetic expressions.

```
var a = 1 + 1
var b = 3 - 2
var x = 5 * 5
var y = 100 / 25
```

Catscript also supports string concatenation with the "+" operator.

```
var first = "John"
var last = "Smith"
var fullName = first + " " +  last
```

Equality & Comparison Expressions

Catscript supports both comparison and equality expressions. These expressions include: greater than (>), Less than (<), greater or equal to (>=), less than or equal to (<=), equals (==), and not equals (!=).

```
var x = 10
var y = 9
if(x > y){
...
}
```

Unary Expression

Catscript supports the negative sign (-) for numbers and the logical negation sign (!) for booleans.

```
var x = -3
var y = true
var z = !y
```

Statements

Variable Declaration & Assignment

Variables can be declared in Catscript using the following syntax.

```
var x = 10
var y = "bar"
var z = true
```

Variables declared with the 'var' keyword feature type inference.

Lists can be declared either with or without an explicit type declaration. The type of lists without a type declaration is also inferred and is assigned based on the component type of the list.

```
var listA list<string> = ["apple", "orange", "bannana"]
var listB list = [1,2,3]
```

Variables can be assigned and reassigned in Catscript as long as the types are compatible.

```
var x = 10
var y = 25
x = y
```

If Statements

Catscript supports If Statements, If-Else Statements, and Else-If Statements. The syntax for If Statements is as follows:

```
if(expression){
...
}
```

Where the expression is any boolean expression.

If-Else Statements have very similar syntax.

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

Else-If Statements require a boolean expression.

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

Catscript also supports any number of Else-If Statements following an If Statement.

Print Statement

Catscript allows printing output to the console using Print Statements. Print Statement syntax is as follows:

```
var x = "Hello World!"
print(x)
```

For Loops

Catscript supports For Loops for iterating over lists. In Catscript, For Loops use the more modern For Loop syntax.

```
var x = ["Alice", "Bob", "Jill"]
for(name in x){
...
}
```

For Loops will iterate over each item in the given list. Each item is accessible with the identifier declared in the For Loop declaration.

Functions

Functions in Catscript are defined and called in very similarly to other programming languages.

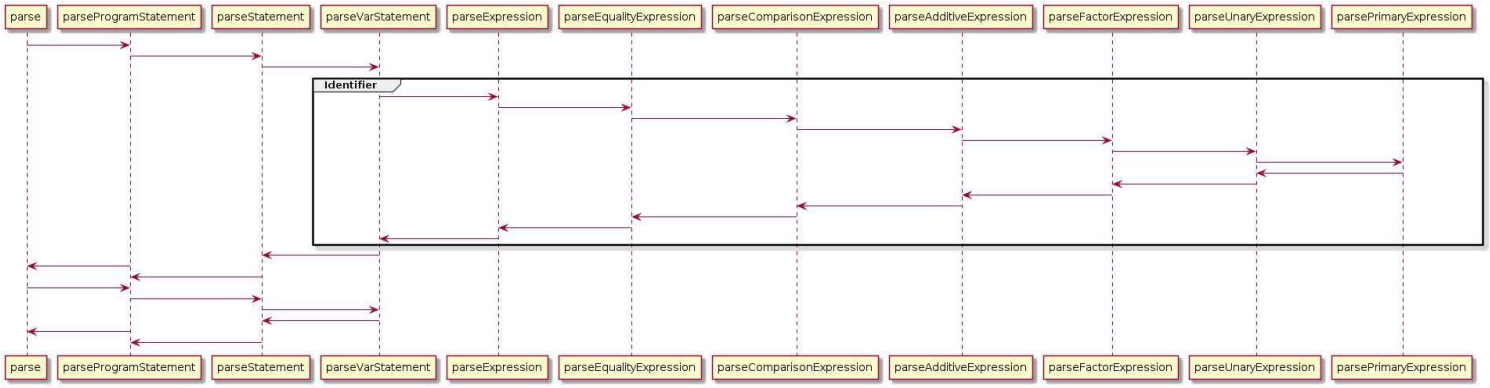
```
function foo(){
    print("bar")
}
foo()
```

Functions can also have a return type or typed input parameters.

```
function bar(a:int, b:int) :boolean{
    if(a > b){
        return true
    }else{
        return false
    }
}
bar(10,9)
```

Section 5: UML.

This document showcases the recursive descent algorithm on a sample variable statement. The diagram is also located in capstone/uml/UML.png. The recursive descent algorithm utilizes a precedence of function calls representing all of the different types in the as described by the grammar. The diagram shows all of functions used to produce a variable statement such as var x = “Hello World”.



Section 6: Design trade-offs

Recursive Descent is a highly effective algorithm for understanding the workings of a parser. The recursive nature of this algorithm makes understanding the parsing process and debugging much easier. However, the tradeoff is the lack of automation that a parser generator can provide, as the actual parsing needs to be done manually. Additionally, though recursive descent is easier to understand and much nicer than the code created via parser generators, it is also not optimized. For learning a parser's inner workings, recursive descent appears to be the better choice. However, it becomes more difficult to implement this type of parser on projects with more complex syntax grammars.

Section 7: Software development life cycle model

We used Test Driven Development (TDD) for this project. This was helpful for rapid changes and understanding the code's expectations. However, TDD did make correctness harder to achieve, as a passing test in one instance may have ended up failing in another. Developing the tests needed to check code correctness would often be a difficult process thanks to this setback.