Compilers

CSCI 468

Spring 2024

Lukas Bernard
Nathan Parnell

Section 1: Program

See attached .zip file.

Section 2: Teamwork

Work was split amongst two people within my Capstone team, myself, and my team member (team member 2).

The role of team member 2 in this project was to:

A) Create a thorough documentation for the CatScript Parser and Programming Language

A brief description of the documentation:

Team member 2 wrote a brief introduction of the CatScript parser, providing a basic example of a use case. Team member 2 was also tasked with creating documentation for the expressions and statements that were implemented, intending to provide the user with a comprehensive avenue to making use of each of them. This set of documentation contained the description of the function, an example of its uses, parameters, and the return values. If the user follows this information they will be capable of creating code executable within the CatScript parsing environment.

B) Create high level tests for multiple different functions of the program. *Located at src/test/java/edu.montana.csci.csci468/NathanParnellTests.java*

A brief description of these tests is as follows:

- *DoubleNegativeInFunction()*: This test was intended to test if a function was capable of printing a double negation on a boolean

parameter.
- *NestedFunction()*: This test is intended to test if a function with a parameter can be called from within another defined function.

- *globalBooleanWithDoubleNegation()*: This test was intended to see if a function was able to perform a double negation on a global variable.

My role within the project was to implement multiple different processes in order to allow for the CatScript parser to support a variety of different inputs.

The project workload was an estimated split around 10% of the work done by my partner (team member 2), and 90% done by me.

Section 3: Design pattern

The most prolific design pattern used throughout the program that was implemented was the use of Memoization. The excerpt of code below displays this implementation. A HashMap is created and referenced within the “getListType” function. This design pattern is located within the CatScriptType.java class located at

src/main/java/edu.montana.csci/csci468/parser/CatscriptType.java

```
3 usages
private static final HashMap<CatscriptType, ListType> memo = new HashMap<>();
5 usages   LukasB4 +1
public static CatscriptType getListType(CatscriptType type) {
    if(!memo.containsKey(type)){
        memo.put(type, new ListType(type));
    }
    return memo.get(type);
}
```

The purpose of this design pattern is that memoization allows the program to avoid the creation of unnecessary duplication type/list type

objects.

Section 4: Technical writing.

CatScript Guide

Introduction

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

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

Features

Additive Expression

The additive expression can apply to strings, or numeric types.

```
1+3;
“Str” + “ing” ;
3 – 1;
```

The additive expression takes two strings, or two numeric types. This will return either the concatenated string, or the result of the numeric type.

Boolean Literal

The Boolean literal assigns a literal “true” or “false” value to an

```
object BooleanLiteral = true;

If(X) {“this means it is true”;}
```

This returns a Boolean literal, which will always have the value of true or false. This requires the value as a parameter.

Comparison Expression

1<6;
6 >= 2;
The comparison expression allows comparison between two objects. This takes the operator, and the two operand objects. This returns true or false, depending on the outcome of the comparison

Equality Expression

1==2;

4==4;

The equality expression works much like the comparison expression, except it checks if the two sides of the operator are equal. It will also return true or false, depending on the outcome of the expression.

Factor Expression

1 * 4;

10 / 5;

The Factor Expression enables the usage of multiplication and division. It returns the product or quotient of the left and right side of the operands

Function Call Expression

function(x);

The function call expression allows the usage of functions. It takes the function name, and any arguments the function takes, in the above example, x. If the function returns something, this expression will return that.

Identifier Expression

var variable = 4;

function func(variable) { print(variable);}
This gives the names for variables and functions, and as such, the parameter is the name of the object.

Integer Literal Expression

var integer = 4;

The integer literal expression enables the assignment of integers.

List Literal Expression

[“list”, “of”, “strings”]

The list literal expression enables the usage of lists in catscript. The provided example is a list of strings.

Null Literal Expression

var variable = null;

The Null literal expression enables the usage of null values in catscript.

Parenthesized Expression

(1 + 3) / 4

The parenthesized expression allows the user to modify the order of operations, so that it does not always need to follow operator precedence. The expression takes the expression within the parentheses as the parameter, and returns the result of that expression.

String Literal Expression

var string = “this is an expression”.

String literals allow the usage of strings in catscript. It takes the string value as a

parameter to assign to the variable.

Syntax Error Expression

return new SyntaxErrorStatement(tokens.consumeToken());

This lets users of catscript create errors, and it takes tokens.consumeTokens as a parameter. It returns the syntax error statement.

Type Literal Expression

This expression allows the setting and getting of catscript types. It takes a type as a parameter, and can return the type through the getType() method.

Unary Expression

!false;

-4;

The unary expression allows the negation of Booleans and negative numerical values. It takes the operator and expression.

For Statement

for(x in [“for”, “statement”, “list”]) {print(x);}

The for statement allows the iteration through object and repetition of code. The for statement requires an expression, a variable name, and a body of statements.

Function Call Statement

function(x);

The function call statement allows the calling of functions. If the function takes an argument, that expression is a parameter. If the function returns something, that will also be used as the statement’s return value.

If Statement

If(booleanIsTrue) { *do statements here* };

The if statement allows branching conditions within code. In this example if booleanIsTrue is true, then that condition will be met, and it will execute the code within that block. If it is not, and there is an else statement, those statements will be executed instead. The if statement requires an expression as a parameter.

Print Statement

print(“string to be printed”);

The print statement allows the printing of objects. It takes an expression, in the example given, this is a string literal. It returns what it is printing out.

Return Statement

return var;

The return statement allows values to be passed out of functions, it requires an expression as a parameter, and it also returns this same expression.

Syntax Error Statement

SyntaxErrorStatement(tokens.consumeToken());

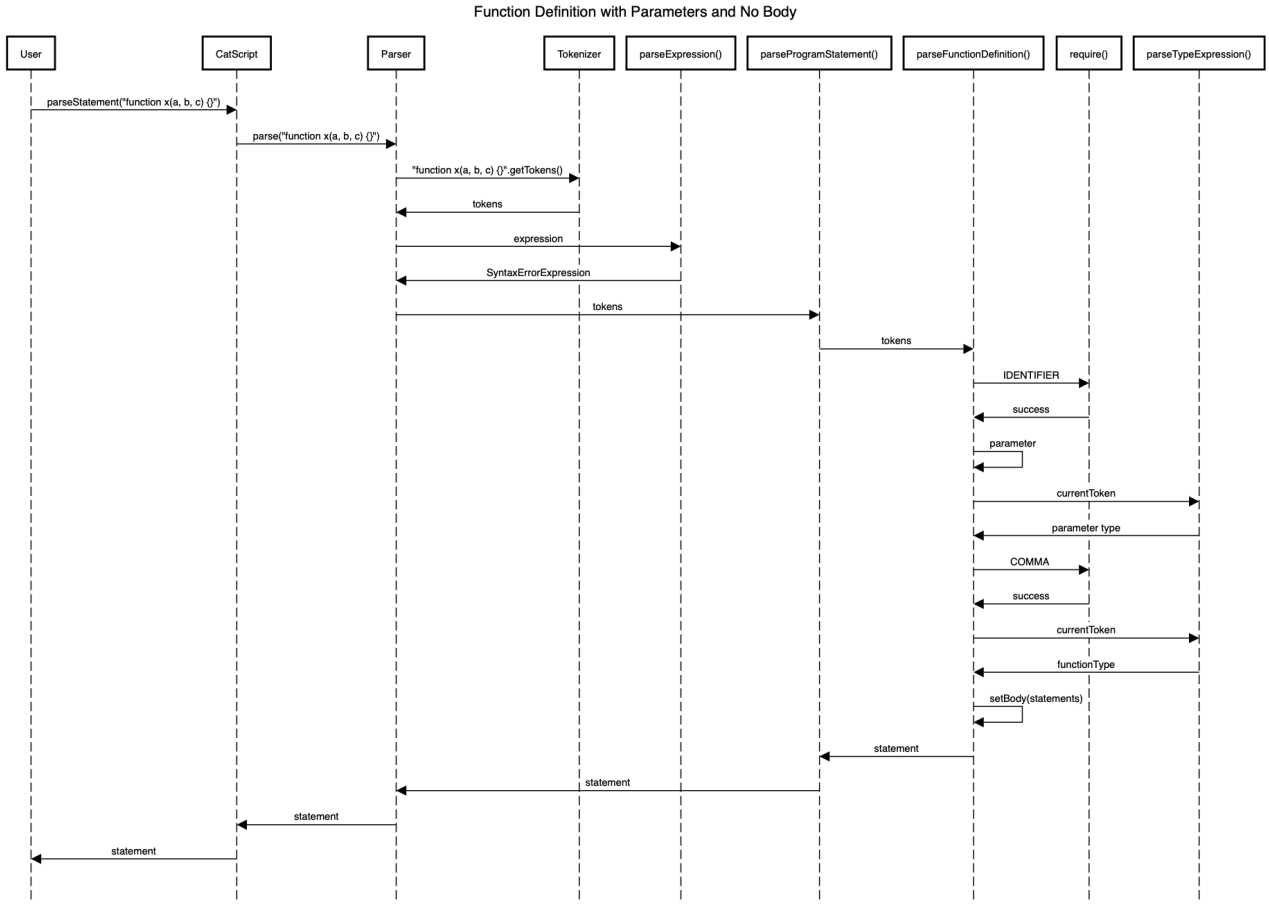
This lets the user create errors if necessary, and takes tokens.consumeToken() as a parameter.

Variable Statement

var x = “variable statement”

The variable statement allows the user to assign values to a memory location under the name of the variable for easy use. It takes a value as a parameter, in this case, “variable statement” string literal.

Section 5: UML



The diagram above represents an example of the user executing a function definition statement “function x(a, b, c) {}” with 3 parameters, no types, and no body. The above sequence diagram traverses from the user input, through CatScript, Parser, Tokenizer, expression parsing, statement parsing, token handling, and type expression handling. The return value of this function is passing a statement from parseFunctionDefinition() back through to the User. The intention of this diagram is to show the flow of data from one area of the program to another, from the source to the destination. CatScript handles creating the parser. Parser handles dealing the function to a tokenizer, then passing this tokenized list into each of the parsers. Firstly it visits

parseExpression to check if it is an expression. Next, since it is not, it visits parseProgramStatement, which sends the token list to each of the statement parsers to be checked, and then parsed if the type of statement matches. In this case it matches parseFunctionDefinition. Internally, most of the work happens within the parseFunctionDefinition() class. It is intended to handle types, identifiers, parameters, and body statements. The first identifier is the title of the function. Portions such as the parameters are handled within the parseFunctionDefinition in a loop between parenthesis where the identifier and then the type are added. Next the type of the function is checked, if it is included, if not it is set to void. Next the body is iterated through with all of the internal statements, if any, being collected. This object that is built is then returned up the sequence for implementation and or further usages.

Section 6: Design trade-offs

Recursive descent was the design that I went with for the CatScript parser. This was chosen over the alternative parser generator, which has its own set of benefits and drawbacks. The benefits of the parser generator is that it is simple to use and does not require you to code the parser in its entirety. However, the major drawback is the lack of ease when debugging and simplicity. CatScript parser is harder to implement, however is conceptually simpler, with easier debugging. Due to this, a major tool in my arsenal when programming CatScript was the IntelliJ native debugger, which allowed me to trace the stack through a multitude of functions, classes, and alterations. A major draw of the recursive descent parser was its compatibility with test driven development, allowing for easier fix implementations when a test fails.

Section 7: Software development life cycle model

As mentioned in Section 6, Test Driven Development (TDD) is the model that was used to develop the CatScript parser. In simple terms, TDD is a development cycle based around creating tests for perceived requirements, in which you implement your code to run against. When all tests are passing, all of your perceived requirements are met, thus completing the stage of development. This model was especially helpful due to the fact that it allowed for instant feedback for every code implementation, allowing for very little downtime in development. Additionally, the tests were a good way of defining ways in which the code was intended to be implemented, such as syntax, and error handling. Due to these two reasons, and many others, test driven development was a large help to the team. Only a few downsides were ever recognized, with one of them being that there were tests that were not initially created that may be beneficial in developing a more comprehensive and cohesive language. For example, when writing tests, my partner and I both came across the occurrence that the function “Assignment Expression” had not been implemented. Beyond this, there was very little hindrance.