Montana State University

Catscript Capstone Portfolio Report

Jordan Stacy

Anthony Nardiello

Compilers - CSCI 468

Professor - Carson Gross

The Catscript Programming Language

Section 1: Program

Location

The source code for this program can be found in /capstone/portfolio/source.zip included in this directory. The Catscript compiler was written in Java. It was developed on an IDE called Intellij, designed by Jetbrains. The development environment was supplied by Carson Gross.

Section 2: Teamwork

Team Member 1

Team member 1 was responsible for programming and supplied the majority of the code. This included code to get the implementation for tokenizing, parsing and compiling of the Catscript programming language to work correctly. The vast majority of the time spent was put into the programming of Catscript.

Team Member 2

Team member 2 supplied the documentation for the capstone such as the technical writing documentation, and provided multiple unit tests for team member 1 to ensure the program was working correctly. With using tests, this reinforced the test driven software development life cycle model that will be discussed in further detail in section 7.

Time Estimates

Total estimated hours: 120 hours

Member 1:
Contributions - Lead programmer, portfolio writer and editor
Estimated total hours: 110 hours. About 91.7% of hours worked.

Member 2:
Contributions - Technical writing writer and editor, unit tester
Estimated total hours: 10 hours. About 8.3% of hours worked.

Section 3: Design Pattern

Introduction : What is the Memoization Pattern

Memoization is a method used to optimize programs. Instead of having the program make an expensive function call multiple times, the result is stored in a cache using a map. If the function is called again, the cached value is returned instead of the expensive function call. This winds up saving a decent amount of processing power and makes the algorithm more efficient. The one issue with this method however is this in result takes up more space to increase the speed of the computation.

Where This Pattern Occurs in the Catscript Program

This pattern occurs in the CatscriptType.java class. This file is located in the directory src/main/java/edu/montana/csci/csci468/parser in source.zip starting at line 37. Here is the code for easy access with an explanation of how the pattern is being implemented.

```
// Create a map that will store the cached CatscriptType
static Map<CatscriptType, CatscriptType> cacheMap = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    // Grabbing the type if there is one.
    CatscriptType matchingType = cacheMap.get(type);
    // If there already is a type stored, just return that type
    if(matchingType != null){
        return matchingType;
    }
    // Otherwise, put the incoming type into the map with the type being the
    // key and the listType being the value stored.
    else{
        ListType listType = new ListType(type);
        cacheMap.put(type, listType);
        return listType;
    } // NOT A THREAD SAFE IMPLEMENTATION
}
```

Section 4: Technical Writing : Catscript Guide

Introduction to Catscript

Catscript may be a simple toy language, but it includes quite a few features. Catscript includes features such as type inference and list literals that can be any Catscript type. A programmer can use the keyword “var” for their type instead of having to explicitly state the type like in Java or C. String, 32 bit integer, object, null and boolean expressions are all featured in Catscript. Mathematical operations such as addition, subtraction, multiplication and division are included. Logical operators such as not are included as well. Catscript does not support the modulo operator. Strings can be concatenated together by overloading the plus operator.

Catscript is a statically typed language. This means that once a type for a variable has been established, it remains that type throughout the execution of the program. How this works will be expanded upon throughout the documentation.

Catscipt uses lexical scoping like most languages in the modern era do. For example, this means that a variable defined in a for loop will disappear unless it has been defined as a field. This will be covered in greater detail through the documentation.

This language features conditionals such as if statements and iterative statements such as for loops. Recursion is also supported. Functions can be both defined and called upon as well. If the programmer defines a function without a type, it is implicitly set to return void.

As stated before, there are quite a few features to be discussed with this language. The next sections will cover all of these features in more detail with sample code as well.

Features

Type Literals

Catscript uses a variety of data types. Expressions, described in later sections, can be integers, booleans, strings, null or an object. All of these types will be discussed in greater detail later on. The keywords to establish these types are:

int, bool, string, null, object

Note that types in other programming languages such as bytes, shorts, chars, floats, doubles etc. are not featured in Catscript.

Functions, described in better detail later on, can return one of these certain types. These functions return the same types as the expressions above. There is one special case for functions however. If no return type is explicitly stated, the function will return a special type called void. The explanation for void types are found in the function section.

Static typing is used for Catscript, so that means the moment a type of a variable is established, it cannot change throughout the execution of the program. When a variable is established as an integer, it cannot be cast to any other type.

Syntax Error Handling

Catscript has a detailed way of handling errors. Catscript uses line numbers and line offsets to give detailed information on what error occurred. If a parsing error occurs, the output of the program will notify where the error is located. This error below was caused by an integer valued variable trying to be assigned a string value.

```
Parse Errors Occurred:

Line 2:meaningOfLife = "42"
                     ^

Error: Incompatible types
```

If a variable is not defined ahead of time, the error would look similar to this:

```
Parse Errors Occurred:

Line 1: print(x)
             ^

Error: This symbol is not defined
```

The offsets on the error handling are usually one character off, so it is something to consider when debugging a program.

Commenting Code

Commenting code in Catscript is identical to programming languages such as C or Java. Comments are ignored by the compiler, so it will not interrupt the flow of the program. This is the general form of implementing comments in Catscript:

```
// A good and meaningful comment.
```

Expressions

Expressions usually evaluate to some value. They can be evaluated to literal or object values. This includes a string, boolean or integer value for example. Each expression will be discussed below.

Integer Literal Expressions

Integer literal expressions represent a 32-bit integer value. Note that decimal numbers such as floats and doubles are not supported in Catscript. Integers are easily implemented in Catscript. This is a legitimate working program in Catscript.

```
42
```

This program evaluates to 42.

```
42
```

Parenthesized Expressions

Parenthesized expressions are expressions with parentheses around them. Due to the recursive nature of Catscript, expressions can utilize more than one set of parentheses. A general form of parenthesized expressions look like:

```
(expression)
```

A practical example of how this is implemented in Catscript is:

```
(42)
```

This would evaluate to 42.

```
42
```

As stated above, infinite pairs of parentheses can be used. An example below:

```
(((((42)))))
```

This would still evaluate to 42.

```
42
```

String Literal Expressions

String literal expressions are arrays of characters. These can be a single letter, or the entire Lord of the Rings trilogy. The general way of implementing strings in Catscript looks like this. Make sure to include double quotes before and after the desired string value:

```
"String Value"
```

A practical example of using a string in catscript looks like this:

```
"Hello Catscript"
```

This program would evaluate to:

```
Hello Catscript
```

To concatenate multiple strings together, that is covered in the next section.

Additive Expressions

Additive expressions include adding or subtracting integer values and string concatenation. The arithmetic operators are all left associative. Below are some generalized code examples of how this works:

Adding two integer values together:

```
integerValue1 + integerValue2 + ...
```

The ellipsis at the end indicates that the addition operator can be continued an infinite amount of times.

A practical example of a Catscript program using addition looks like this:

```
21 + 21
```

This example would evaluate to 42 just like the basic rules of arithmetic.

```
42
```

Subtracting two integer values together:

```
integerValue1 - integerValue2 - ...
```

As stated above, the ellipsis indicates that this minus operator can be done an infinite amount of times.

A practical example of a Catscript program using subtraction would look like this:

```
62 - 20
```

As expected, this expression would evaluate to 42.

```
42
```

Combining addition and subtracting:

```
integerValue1 + integerValue2 - integerValue3 + ...
```

In Catscript, addition and subtraction can be combined infinitely. Here is a practical example of how this program would work:

```
13 + 14 - 10 - 17 + 42
```

When adding and subtracting everything together, this would evaluate to 42 exactly like one would expect with the laws of arithmetic.

```
42
```

If an expression needs precedence somewhere special, parenthesis can be used. The example above will be used again for an example.

```
13 + 14 - 10 - (17 + 42)
```

This will prioritize the 17 + 42 before anything else in the expression. With these parenthesis, the expression will instead evaluate to -42.

```
-42
```

With string expressions, the addition operator has been overloaded to concatenate string values. The general implementation goes like this:

```
stringValue1 + stringValue2 + ...
```

An infinite amount of strings can be concatenated together. Note that the minus operator will not work with strings and will result in an error. Here is a practical example of string concatenation in Catscript:

```
"Hello " + "Catscript"
```

This would evaluate to:

```
Hello Catscript
```

When using the addition operator, if any value is not an integer, it will always evaluate to a string literal. Take this example for instance:

```
"Hello " + "Catscript " + 1 + null
```

This may look strange and a bit complicated. The null will be discussed in another section, but this expression would evaluate to:

```
Hello Catscript 1null
```

Factor Expressions

Factor expressions are expressions such as multiplication and division. The general implementation for multiplication goes like this:

```
integerValue1 * integerValue2 * ...
```

A practical example of multiplication in Catscript looks like this:

```
3 * 7 * 2
```

This expression will evaluate to 42.

```
42
```

The general implementation for division is shown below:

```
integerValue1 / integerValue2 / ...
```

A practical example of division look like:

```
84 / 2
```

This expression would evaluate to 42.

```
42
```

Multiplication and division can be combined with addition and subtraction along with parenthesis. With Catscript, complicated arithmetic can be done:

```
16 + (8 - 2) * 8 / 2 + 2
```

This expression would evaluate to 42.

```
42
```

Boolean Literal Expressions

Boolean literals evaluate to either a true or a false value. This is how to implement them in Catscript:

```
true
```

This evaluates to true.

```
true
```

A false statement:

```
false
```

This evaluates to false.

```
false
```

Unary Expressions

Unary expressions have only one operand, and typically convert values into their opposites. There are two that are covered in Catscript. The first is used to make an integer negative. This is how these would be implemented in Catscript:

```
-IntegerValue
```

A practical example would be:

```
-42
```

This would evaluate to -42.

```
-42
```

This negation can be done multiple times. For instance this example:

```
--42
```

This would evaluate to 42.

```
42
```

The second type of unary expression is the logical not operator. This can be combined with boolean expressions to modify them. This is a general implementation of the logical not operator:

```
not booleanExpression
```

Here is a practical example:

```
not true
```

This will evaluate to false

```
false
```

The logical not can be done multiple times just like the negation operator.

```
not not false
```

This will be evaluated to false

```
false
```

List Literal Expressions

List literal expressions are similar to arrays. Their values can contain any type that is supported by Catscript. The general form of list literals in Catscript goes like this:

[element1, element2, element3, ...]

Lists in Catscript can have an infinite amount of elements. Here is a practical example of a list in Catscript:

```
[1, 2, 3, 4, 5]
```

This is a list with the elements 1, 2, 3, 4 and 5. This expression evaluates to:

```
[1, 2, 3, 4, 5]
```

A very interesting feature of Catscript is that lists can have elements with varying types. A practical example of this would be:

```
[7, 42, "Carson rocks", true, "Dr. Paxton rules", true]
```

This expression would evaluate to:

```
[7, 42, "Carson rocks", true, "Dr. Paxton rules", true]
```

Note that this list has integers, strings and booleans.

Comparison Expressions

A comparison expression checks a condition, and returns a boolean value based on the condition. A general form of this implementation is:

```
expression comp expression
```

Where comp is the choice of <, >, <=, >=.

A practical example of a comparison expression would be:

```
84 > 42
```

This would return true since 84 is indeed greater than 42.

```
true
```

Another example, but instead evaluating to false:

```
84 <= 42
```

This would evaluate to false.

```
false
```

Null Literal Expressions

A null literal expression is an object that has no value to it. A null is implemented like this:

```
null
```

The expression evaluates to null.

```
null
```

Equality Expressions

Equality expressions compare if two expressions are equivalent or not equivalent and evaluate to a boolean literal. In Catscript, the general form for this is:

```
expression == expression
```

Or

```
expression != expression
```

A practical example of the equality expression would look like this:

```
42 == 42
```

This would evaluate to true.

```
true
```

Boolean and null values can be evaluated as well. For instance:

```
true != null
```

This would evaluate to true since the two are not equivalent.

```
true
```

Statements

Print Statements

Print statements take an input of any type and output the results to the console. The general form for print statements in Catscript is:

```
print(something)
```

A practical example would be:

```
print("Compilers are fun")
```

This would output to the console:

```
Compilers are fun
```

Variable Statements

Variables are values that can change during the execution of a program. A general form to create variables is:

```
var variableName : variableType = expression
```

A practical example for creating variables is:

```
var classNumber : int = 468
```

Note that when attempting to run this program, nothing outputs to the console. To fetch this classNumber variable from the symbol table, a print statement can be used. Here is the example with a print statement included:

```
var classNumber : int = 468
print(classNumber)
```

Now the value will be sent to the console. Here is the output:

```
468
```

There are many keywords that can be used for variableType. These include int, bool, string, null, object or list<>.

Note that the variable type can be left out. With inference, the variable will be assigned to an object type. The general form is:

```
var variableName = expression
```

A practical example of this is:

```
var className = "Compilers"
print(className)
```

Without using the string keyword, the variable is added to the symbol table, and outputted to the console just fine:

```
Compilers
```

Assignment Statements

Assignment statements change the value of a variable. The general form of the assignment statement looks like this:

```
variableName = expression
```

Note that the variable needs to be defined before the assignment statement can be done. Here is a practical example of how this would look:

```
var meaningOfLife = 10
meaningOfLife = 42
print(meaningOfLife)
```

This would output a 42 to the console.

```
42
```

Note that the variable that is named meaningOfLife is being updated from 10 to 42. As described before, this is why the value is variable. It can change through the execution of the Catscript program.

The example below is what not to do in Catscript. Catscript is a statically typed language, so variables do not dynamically change types. Take this program for example:

```
var meaningOfLife = 10
meaningOfLife = "42"
print(meaningOfLife)
```

This program would result in a parse error. This is what shows on the console:

```
Parse Errors Occurred:

Line 2:meaningOfLife = "42"
                       ^

Error: Incompatible types
```

For Statements

For statements use iteration to traverse through lists. A general form of for statements looks like this:

```
for(variableName in list){
     //logic
}
```

A practical example of how a for loop works in Catscript would look like this:

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

The console would output:

```
1 2 3
```

Note that variables can be used to store the list values. Take this other program as an example:

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

This would still output the same results as the last program:

```
1 2 3
```

If Statements

If statements branch to a different part of the program based on a boolean condition. The general form for if statements is:

```
if(condition){
      // do something
}
else{
      // do another thing
}
```

A practical example of how if statements work would look like this:

```
var age = 23
if(age < 21){
      print("You cannot legally purchase alcohol.")
}
else{
      print("You can legally purchase alcohol.")
```

```
}
```

This will output:

```
You can legally purchase alcohol.
```

Functions

Functions are blocks of code that are ideally good for repeated use. For example, if a programmer needs to print out a large section of text, it would be far easier to put the code into a function and call it with a single line of code instead of having to type or copy and paste a large chunk of code over and over again. The general form of defining a function in Catscript is:

```
Function functionName(parameters : parameterType) : returnType {
      // functionality for function
}
```

To call the function that was created, the general form for this is,

```
functionName(parameters)
```

Functions can have zero or many parameters. If the return type for the function is left out, the compiler assumes that the function returns void. A practical example of defining a function would look like:

```
function legalDrinkingAge(age : int){
      if(age < 21){
            print("You cannot legally purchase alcohol.")
      }
      else{
            print("You can legally purchase alcohol.")
      }
}
legalDrinkingAge(18)
```

This program would output:

```
You cannot legally purchase alcohol.
```

Return Statements

Return statements in Catscript return values from functions. The general form of return statements goes like this:

```
function functionName(parameters : parameterType) : returnType {
      // functionality for function
      return returnValue
}
```

A practical example of a return statement looks like this:

```
function returnsTrue() : bool {
      return true
}
print(returnsTrue())
```

This will output:

```
true
```

Note that the return statement must be the last instruction in the function, or it will result in a parse error. This also works for if statements as well.

Recursion

Recursion is a method of computation that a function calls itself over and over again until some condition is met. Catscript is capable of recursion. Here is an example of Catscript doing recursion:

```
function recursionExample(x : int){
      print(x)
      if(x > 0){
```

```
            recursionExample(x - 1)
        }
}
recursionExample(5)
```

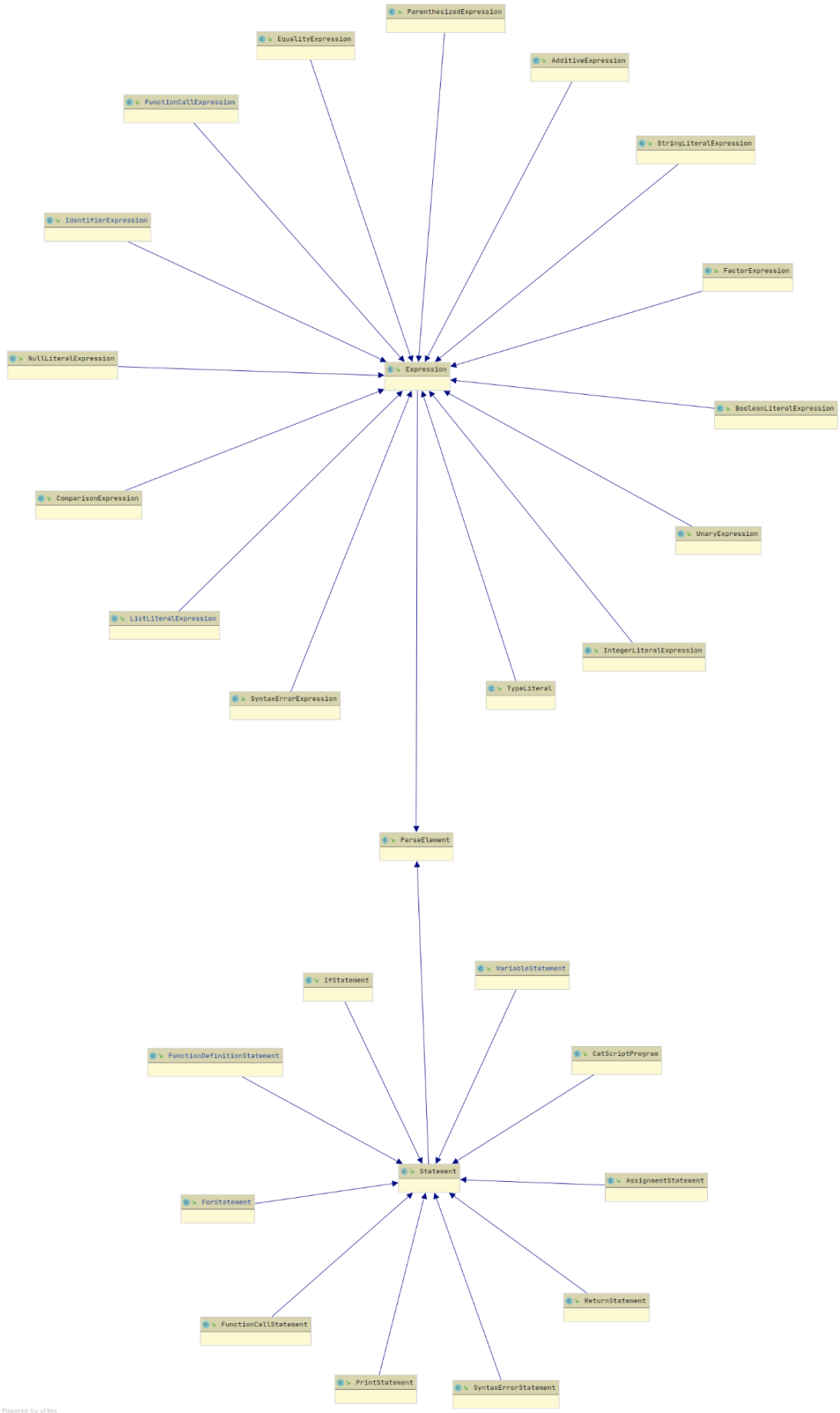
This will output:

```
5 4 3 2 1 0
```

Note how when x = 0, the function stopped executing due to reaching its base case.

Section 5: UML

If the diagram on the next page is hard to see or read, the original image can be found in the UML directory of the zipped source code. The name of the file is expressions_balloon.png.


ParenthesizedExpression
EqualityExpression
AdditiveExpression
FunctionCallExpression
StringLiteralExpression
IdentifierExpression
FactorExpression
NullLiteralExpression
Expression
BooleanLiteralExpression
ComparisonExpression
UnaryExpression
ListLiteralExpression
IntegerLiteralExpression
SyntaxErrorExpression
TypeLiteral
ParseElement
IfStatement
VariableStatement
FunctionDefinitionStatement
CatScriptProgram
Statement
AssignmentStatement
ForStatement
ReturnStatement
FunctionCallStatement
PrintStatement
SyntaxErrorStatement

Section 6: Design Trade-offs

Introduction

Two very popular ways of writing compilers are recursive descent parsing and parser generators. With both of the methods, there are good and bad things to consider for both. This section will cover why the teams, class and instructor made the choice to use recursive descent parsing over a parser generator.

What is Recursive Descent Parsing

Recursive descent parsing is a method of parsing that operates in a top-down fashion. In this sense, every production of a grammar is programmed into a function, and then is called recursively through the grammar. The meaning of the top-down way it works is with the use of trees. The program is worked from the top of the grammar, and keeps working its way down. If there are non-terminals still remaining, the top of the tree is called again. This process continues until every branch is a terminal like a literal value.

Recursive descent is very great and intuitive due to the way that the code to build the parser conforms very well with the EBNF. With the use of while loops, it supports unlimited parameters, and doesn't require too many advanced patterns and is quite efficient. Many of the largest programming languages such as Java and C# utilize recursive descent parsing. As mentioned in section 4, Catscript is among one of these languages that uses recursive descent.

What is a Parser Generator

A parser generator takes two types of input. Usually from regular expressions and a language grammar that is something like BNF or EBNF. With these inputs, the generator outputs or code based on the inputs.

The parser generator that was mentioned in this class is a tool called ANTLR. This generator takes the regular expressions and BNF (EBNF), and outputs (generates) java files that include code for the tokenizer and parser for the grammar.

Comparison Between the Two

When comparing the two, the teams, class and instructor had a preference for recursive descent. In their eyes, it was far more educational typing out the loops of the parser, implementing everything by hand, and being able to write out most of the code individually to help reinforce what was going on under the hood. Parser generators

seemed overly complicated using regular expressions, generating code, and having to go back in and find obscure and difficult to understand errors due to the code generation.

Recursive descent parsing is far easier to debug when the code was written by the programmer. The programmer has control to name their variables whatever they wish, and implement and change the code to their liking.

Section 7: Software Development Life Cycle Model

Introduction

The instructor had teams utilize test driven development to design Catscript. Teams were given a set of unit tests to pass in order to verify that Catscript was functioning as intended. Further testing could be done using the Catscript Server. This could be done by accessing port 6789 on a computer, and that would bring up a Catscript IDE on an internet browser to ensure proper functionality.

What Test Driven Development is

Test driven development is a model that utilizes a system of tests on either no code or incomplete code. The programmer then utilizes whatever programming methods they can to get the tests to pass. As each test passes, the other completed tests are run again to ensure proper functionality of the system. The process continues until all tests are passing. If all parameters have been properly covered in the tests, the system is now gone through again to ensure the code is readable and easy to maintain.

How Test Driven Development Was Implemented When Designing Catscript

Each group was given a program skeleton to start off with. This program included 148 unit tests that they needed to get passed. These tests were split into 4 sections.

The first was the tokenizer section. This consisted of 16 tests that checked to see if each lexeme was properly being tokenized.

The second section was the parser section. This had a series of 69 tests that ensured that each token was properly being parsed out. It also checked to see if proper syntax errors were being thrown when they needed to be, and if the parser was implementing correct type checking.

The third was the eval section. It had 31 tests that made sure that all of the expressions and statements were being evaluated and executed correctly. A chunk of the tests checked both execution of statements and evaluation of functions all in the same test.

The fourth was the bytecode section. 32 tests were given to pass. These tests checked to see if code was being properly compiled into JVM bytecode. The generated bytecode was printed out to the terminal to check and compare to the equivalent JVM bytecode generated by equivalent java code.

What the Team Thought of Test Driven Development

The team held TTD to a very high regard. The model is very straightforward due to having very explicit requirements that need to be passed. It was extremely helpful to be able to debug through each line of code using Intellij, and see any errors in the team's implementation. The j-unit testing framework gave very helpful evaluation details by stating what the parsed code was evaluating to. The answer would be compared with what was expected. It would help when debugging what was going wrong with team member 1's implementation without needing much help from the instructor.