Montana State University

Catscript Capstone Portfolio documentation

Carson Whitfield

Cooper Wollschlager

Compilers - CSCI_468

Spring 2025

Professor- Carson Gross

Section 1: Program

Location:

The location of the source code for the program is found in the folder called program.zip which is included in this directory. The Catscript compiler was written in Java and was created in an IDE called IntelliJ by JetBrains. Carson Gross supplied the development environment.

Section 2: Teamwork

Our team consisted of two members:

- **Carson Whitfield**
- **Cooper Wollschlager**

Estimated contribution:

Both team members did equal work on this project, most of which was done with eachother

Section 3: Design pattern

Introduction: Composite Pattern

Composite Pattern is a structural design pattern that allows individual objects and groups of objects to be treated uniformly. For example, in the CatScriptParser, it is set to represent complex expressions as a tree of simpler expressions. Each node in the tree can be a leaf node, like a number or variable, or a composite node, which is a binary operation such as addition or a function call. For example, in the parseAdditiveExpression() method, which is a left-hand side and right-hand side expression that are combined into a new additive Expression object, which itself is an expression. Because all expressions implement the same interface or superclass, the parser and interpreter can process both simple and nested expressions using the same logic. This recursive, tree-like structure makes it easy to build, evaluate, and manipulate expressions of arbitrary complexity, all while keeping the codebase clean and extensible.

Where does this pattern occur in the Catscript program

This pattern occurs in **Catscriptparser** Java class. The file is located in the source src/main/java/edu/montana/csci/csci468/parser/CatScriptParser.java. Here is an example of a composite pattern, which is a function called parseAdditiveExpression() as seen in the figure below.

```
private Expression parseAdditiveExpression() {  2 usages   Carson Gross +1
    Expression expression = parseMultiplyExpression();
    while (tokens.match(PLUS, MINUS)) {
        Token operator = tokens.consumeToken();
        final Expression rightHandSide = parseMultiplyExpression();
        AdditiveExpression additiveExpression = new AdditiveExpression(operator, expression, rightHandSide);
        additiveExpression.setStart(expression.getStart());
        additiveExpression.setEnd(rightHandSide.getEnd());
        expression = additiveExpression;
    }
    return expression;
}
```

Section 4: Technical Documentation

Parser:

The CatScriptParser is a recursive descent parser designed to convert CatScript source code into an abstract syntax tree (AST) composed of Statement and Expression objects. It begins by tokenizing the input using the CatScriptTokenizer, then attempts to parse either a single expression or a sequence of program statements. The parser supports a range of statement types, including function definitions, variable declarations, print, return, assignment, conditionals like if and for loops. Expressions are parsed concerning operator precedence, from equality and comparison operations down to additive, multiplicative, unary, and primary expressions such as literals, identifiers, and function calls. The parser includes helper methods like require() for enforcing syntax expectations and reporting errors, and it features specific handling for type annotations and list literals.

Syntax:

The syntax of CatScript is similar to python while using the JVM compiler. The language is meant to be easy to read while not being super complicated in the back end.

Expressions

Catscript allows a wide range of expressions, allowing equality, comparison, additive, factor, unary, and primary expressions. These expressions can be used in a multitude of spots and has the standard syntax that most languages nowadays use.

Syntax	***Meaning***	***Returned Value***
A != B	*Not Equal*	*True or False*

A == B	*Equal*	*True or False*
A > B	*Greater Than*	*True or False*
A >= B	*Greater Than or Equal*	*True or False*
A < B	*Less Than*	*True or False*
A <= B	*Less Than or Equal*	*True or False*
A + B	*Add*	*Sum of the 2 values*
A – B	*Subtract*	*Difference of the 2 values*
A / B	*Divide*	*Quotient of the 2 values*
*A * B*	*Multiply*	*Product of the 2 values*
Not A	*Inverse*	*True or False*
-A	*Inverse*	*positive to negative or True to False*

Statements

To print out something you can just call "print(x)" with x being what you want to print, whether it's a variable, string, or anything else.

To define variables in CatScript you must declare the creation of a variable with "var" and then you can assign a name and value pair. The whole line would look like var x = "hello". This has implicit declaration so there is no need to assign a type to the variable.

With any variables in Catscript, when you instantiate them; whether it is in a method header or just a declared variable you can choose to follow up the name of the variable with a colon and then declare the variables type, such as "var x : int = 1" declares x as an int and assigns the value 1 to it.

Function Declaration

To declare a function in CatScript you start the line with function to tell the compiler that this will be a function declaration. Then the syntax is as follows:

```
function 'name'(parameter list) {
  statement(s)
}
```

One thing to note is that the parameter list can be empty or have a lot of variables. And you may also declare the variables type if needed.

Function Call

After declaring a function, one might want to call on that function to use it. In order to call functions in CatScript you simply just use the function's name and parameters, an example of a function call would be like: "name(parameters)"

Return

CatScript does allow Return statements to get information from functions and their syntax is simply to say 'return' then the information you would like to be returned

While CatScript does not have while loops, it does have if statements and for loops. The syntax of an if statement also allows for else to be tacked onto the end. And example of an if else statement is as follows

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

And for loops have the syntax as follows:

```
for (variable in expression) {
        statement(s)
}
```

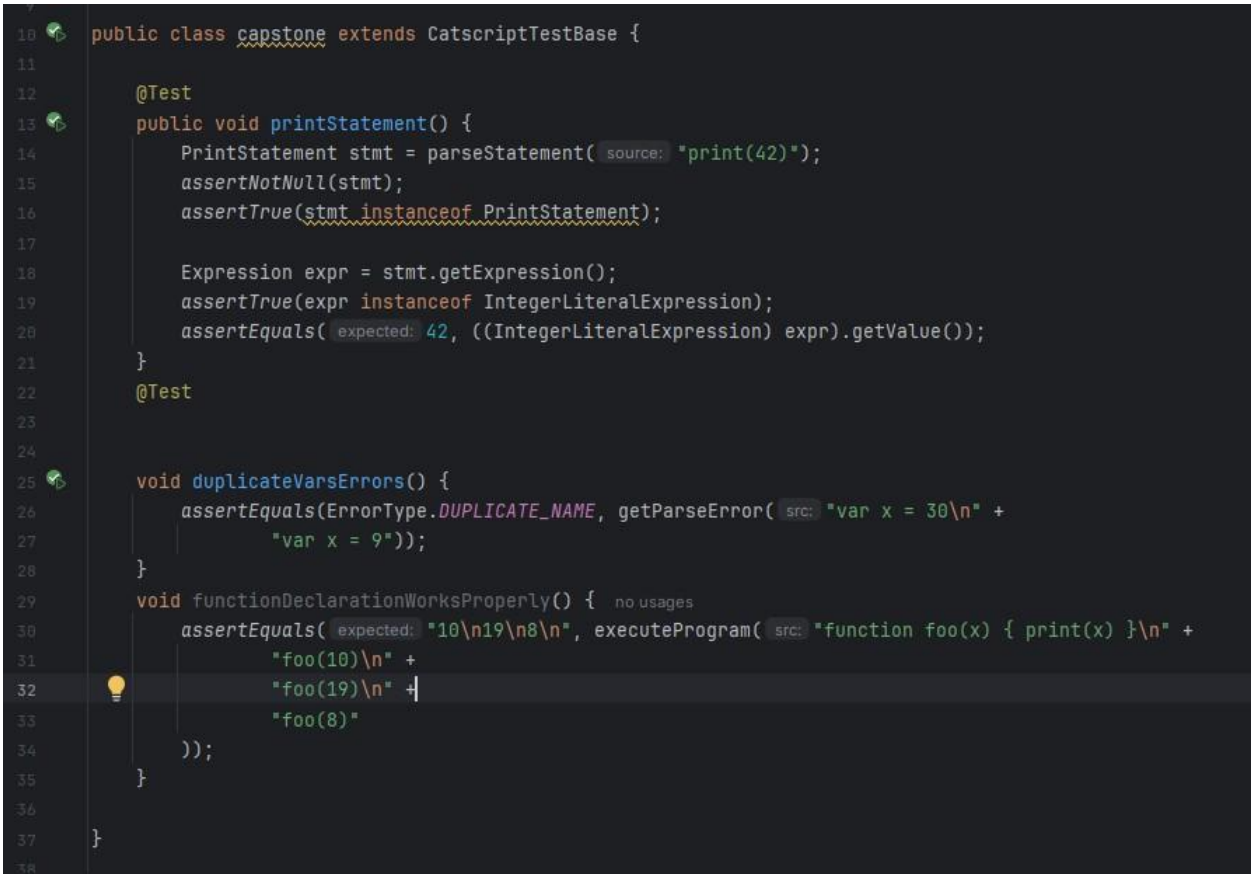
Together with if-else statements and for loops, a majority of needed programs could be written in CatScript even without having a while loop.

TEST CASES:

Cooper Wollschlager's Test Cases:

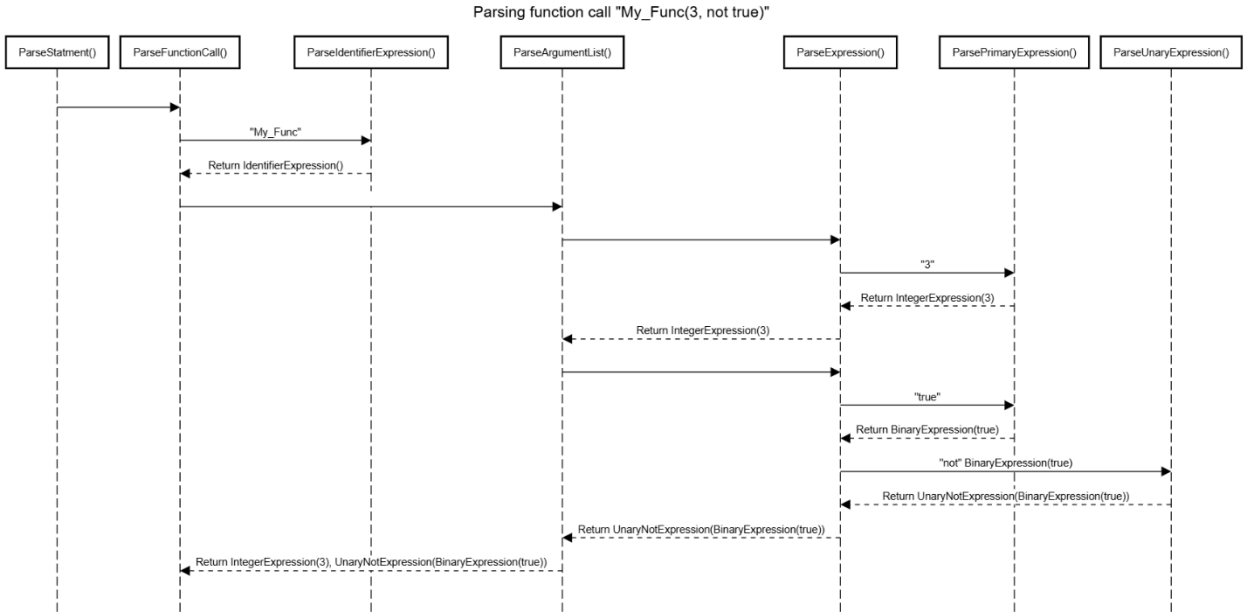
```
@Test
void ifWorks() {
    assertEquals( expected: "1\n", executeProgram(
            src: "var x = 5" +
                "if (x == 5) {" +
                    "print(1)" +
                "}"));
}

@Test
void ForWorks() {
    assertEquals( expected: "1\n2\n3\n", executeProgram(
            src: "var a = [1, 2, 3]" +
            "for( x in a ) {\n" +
                "  print(x)\n" +
            "}\n"));
}

@Test
void mathWorks() {
    assertEquals( expected: 2, evaluateExpression( src: "1 * 2"));
    assertEquals( expected: 4, evaluateExpression( src: "8 / 2"));
    assertEquals( expected: 4, evaluateExpression( src: "2 * 4 / 2"));
    assertEquals( expected: 6, evaluateExpression( src: "2 + 4"));
    assertEquals( expected: 6, evaluateExpression( src: "8 - 2"));
    assertEquals( expected: 6, evaluateExpression( src: "8 - 4 + 2"));
}
```

Carson Whitfield's Test Cases:



```
public class capstone extends CatscriptTestBase {

    @Test
    public void printStatement() {
        PrintStatement stmt = parseStatement( source: "print(42)");
        assertNotNull(stmt);
        assertTrue(stmt instanceof PrintStatement);

        Expression expr = stmt.getExpression();
        assertTrue(expr instanceof IntegerLiteralExpression);
        assertEquals( expected: 42, ((IntegerLiteralExpression) expr).getValue());
    }
    @Test


    void duplicateVarsErrors() {
        assertEquals(ErrorType.DUPLICATE_NAME, getParseError( src: "var x = 30\n" +
                "var x = 9"));
    }
    void functionDeclarationWorksProperly() {  no usages
        assertEquals( expected: "10\n19\n8\n", executeProgram( src: "function foo(x) { print(x) }\n" +
                "foo(10)\n" +
                "foo(19)\n" +
                "foo(8)"
        ));
    }

}
```

Section 5: UML



Section 6: Design Tradeoffs

We decided in this class to use recursive descent because it uses an easier to understand language and can help teach us more about the innerworkings of a compiler compared to using a generative one which does a lot of the backend for us. The other big reason we used recursive descent is because that backed the generative compiler creates is a mess that is hard to read and understand, or in other words a Blackbox. However, for general use that is the big positive of generative models, as the creator has to write less code, and therefore has less they can get incorrect when creating their compiler.

In addition to the above recursive descent is widely used and is even used in some generative models, so knowing how they work and creating our own recursive descent parser and compiler lets us learn way more about how they work and get us more familiar with recursive descent.

Section 7: Software Development Life Cycle Model

We used TDD (Test Driven Development) for our model. The TDD model is a model that focuses on passing test cases while maintaining all the older test cases so they are still working. Using this method, the person goes test by test, slowly building up their project and guaranteeing that the necessary parts work as needed.

This model was easy for both of us easy to work with and led the project along the correct track to being done. The checkpoints of tests to pass guaranteed that we were ready to move onto the next step with confidence that the program was working as expected. One thing that we found annoying with TDD was that sometimes, while coding the current test case, we would break an old test case and not realize until much later if we forgot to run old tests as well.