Capstone Final Document

CSCI-468 | Spring 2025

Kyler Smith

Teamwork

The teamwork involved for my capstone consisted of a team of two. I developed the compiler for Catscript and served as the only developer. My teammate took the job of testing and quality assurance. Since I wrote all of the code for the compiler, my teammate wrote tests to ensure that the code I wrote was of high quality and worked for tests that were not yet presented. We collaborated, testing code and coming up with a plan on how we would help each other develop our compilers for CatScript. The test cases are in:

```
src/test/java/edu/montana/csci/csci468/tokenizer/CatScriptTokenizerTest.java
src/test/java/edu/montana/csci/csci468/parser/CatScriptParserStatementsTest.java
```

on GitHub. The work distribution consisted of me doing 70% of the work while my teammate did about 30% for the tests.

Design Pattern

The design pattern I implemented in the project is memoization within the typing system. The code shown below can be found at this location in my GitHub, `src/test/main/edu/montana/csci/csci468/parser/CatscriptType.java`, starting at line 37. This code optimizes repeated calls to getListType by caching previously created list types instead of instantiating a new object each time. As a result, when the same kind is requested multiple times, especially within loops, the system can return the cached instance, reducing the overhead of object creation and improving performance. This approach essentially acts as a simple caching mechanism for list types. However, it is important to note that this implementation is not thread-safe.

```
static Map<CatscriptType, CatscriptType> MEMOIZATION_CACHE = new HashedMap();
public static CatscriptType getListType(CatscriptType type) {
   CatscriptType existingType = MEMOIZATION_CACHE.get(type);
   if (existingType != null) {
       return existingType;
   } else {
       ListType listType = new ListType(type);
       MEMOIZATION_CACHE.put(type, listType);
       return listType;
   }
}
```

Technical Writing

Throughout the semester, I have been building a compiler for CatScript, a language created for the class. Below is a technical breakdown and the documentation for the compiler, including each component of the language, and each section broken down into different sections. The sections include the assignment section, the function system, the operator system, the type system, and the control system. This will be a minimal breakdown and will not go into depth on how each system functions completely, but rather what they are capable of, to give you an understanding of how the compiler works.

Assignments:

Strongly and Weakly-typed section:

CatScript was created to be flexible. There are two options for assigning to newly declared variables. The first is when the programmer assigns a type to a variable by using CatScript_Type after the variable name. The second is to automatically assign the variable type and leave the assignment type to a variable off.

Here is an example of the syntax for a variable statement:

```
variable_statement = 'var', IDENTIFIER,
    [':', type_expression, ] '=', expression;
```

And here is an example of a variable statement:

```
Var x = 1
```

Assignment Rules:

CatScript allows for the assignment of different variables to the results of expressions through an Assignment Statement, using the ‘=’ symbol.

An example of this in action:

```
assignment_statement = IDENTIFIER, '=', expression;
```

Similar to most programming languages, the rules that must be followed when assigning variables are simple. If there is a ‘void’ type, nothing can be assigned from it, but conversely, everything can be assigned from the ‘null’ type. Since the CatScript compiler is made in Java, we follow all the assignment rules for the rest of the Java classes (int, string, boolean, etc.)

An example of a variable being assigned to a variable:

```
Var x = y
```

Functions:

Function Definitions:

In CatScript, functions are defined using the function keyword followed by the function name. Like in Java, functions can include a parameter list of arbitrary length, with each parameter explicitly typed. After the parameter list, the return type, any valid CatScript primitive or complex type, is specified. The function body contains a sequence of statements, including optional return statements, similar to the body of a for loop. Below are examples of function declarations:

Declaration syntax for CatScript:

```
function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
                       [ ':' + type_expression ], '{',  { statement },  '}';
```

Function Declaration:

```
“Function foo(x : list<int>) { print(x) }”
```

Return Statements:

CatScript has a unique approach to return statements. A return statement doesn't need to return a value—it can be used simply to exit a function early, much like the break keyword in Java. However, returning function pointers or non-CatScript types, such as arrow functions, is not allowed. Functions can return values of any type, including void.

Return statement syntax for CatScript:

```
return_statement = 'return' [, expression];
```

Return statement instance within a function:

```
“Function foo(): list { return[1,2,3]}”
```

Operators:

Arithmetical operations:

CatScript supports standard arithmetic operations found in most programming languages, including addition, subtraction, multiplication, division, and negation. These follow the conventional order of operations and are categorized as "additive expressions," "factor expressions," and "unary expressions." Notably, additive expressions also support basic string concatenation. For this documentation, we will focus on additive expressions, as they are representative of the other categories.

Syntax for additive expressions in CatScript:

```
additive_expression = factor_expression { ("+" | "-" ) factor_expression };
```

Additive expression in CatScript:

```
"1 - 2 - 1"
```

Boolean Operations:

CatScript includes all of the Boolean operations that one would come to expect in all programming languages. The operators are listed below, as well as some syntax examples.

Syntax for boolean expressions in CatScript:

```
equality_expression = comparison_expression { ("!=" | "==")
comparison_expression };

comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" )
additive_expression };
```

Boolean expression in CatScript:

```
"4 <= 2"
```

Types:

Primitive data types:

CatScript provides six primitive data types commonly found in object-oriented programming languages: int, string, boolean, object, null, and void. These are the exact keywords used when declaring, assigning, or comparing values in the language.

```
public static final CatscriptType INT = new CatscriptType("int",
Integer.class);
```

```
public static final CatscriptType STRING = new CatscriptType("string",
String.class);
public static final CatscriptType BOOLEAN = new CatscriptType("bool",
Boolean.class);
public static final CatscriptType OBJECT = new CatscriptType("object",
Object.class);
public static final CatscriptType NULL = new CatscriptType("null",
Object.class);
public static final CatscriptType VOID = new CatscriptType("void",
Object.class);
```

During the tokenization phase, each keyword or value is converted into a token that retains its original value (e.g., 11 or "Peanut"). In the subsequent parsing and evaluation stages, the compiler examines the token type and proceeds with the appropriate steps in the compilation process.

Complex types:

CatScript includes one complex data type: the list<>, similar to those found in many programming languages. Lists can be created using any primitive data type, except for void, resulting in types like list<int> or list<string>. Internally, the list type is represented as a LinkedList based on the type specified in the LIST token.

List type syntax from CatScript:

```
'list'[, '<' , type_expression, '>']
```

In the CatScript compiler:

```
public class ListLiteralExpression extends Expression {
   List<Expression> values;
   private CatscriptType type;

   public ListLiteralExpression(List<Expression> values) {
       this.values = new LinkedList<>();
       for (Expression value : values) {
           this.values.add(addChild(value));
       }
   }
```

Control:

If-Statements:

CatScript features a robust implementation of the if statement. The condition can be an expression of any size, and the if block can include any number of then statements. Both if and else blocks can contain lists of high-level expressions, which may themselves include additional statements and nested expressions.

If statement syntax for CatScript:

```
if_statement = 'if', '(', expression, ')', '{',
                    { statement },
               '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];
```

If statement in CatScript:

```
if(true){ print(1) } else { print(2) }")
```

For-Loops:

CatScript includes a simplified form of the for loop, which functions more like a for-each loop. It iterates over each element in a list and executes the statements defined within the loop body for each item.

For-loop syntax for CatScript:

```
for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',

               '{', { statement }, '}';
```

In the evaluation stage, the statements in the body of the for loop are stored in a Linked List of CatScript statements, which are then run for each item in the original list being iterated over.

```
@Override
public void execute(CatscriptRuntime runtime) {
   Iterable listToIterateOver = (Iterable) expression.evaluate(runtime);
   runtime.pushScope();
   for(Object currentValue:listToIterateOver){
       runtime.setValue(variableName,currentValue);
       for(Statement statement:body){
           statement.execute(runtime);
       }
   }
   runtime.popScope();
}
```

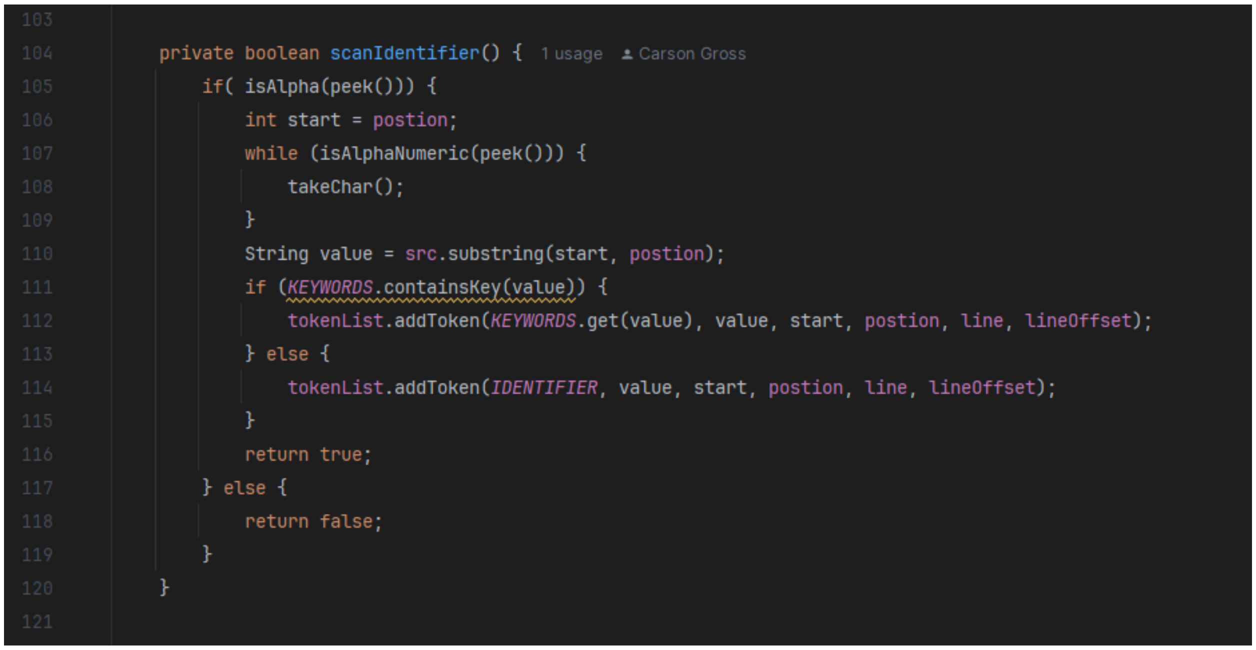
For-loop in CatScript:

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

JW Compiler Code Critique

Tokenizer File

CateScriptTokenizer.java



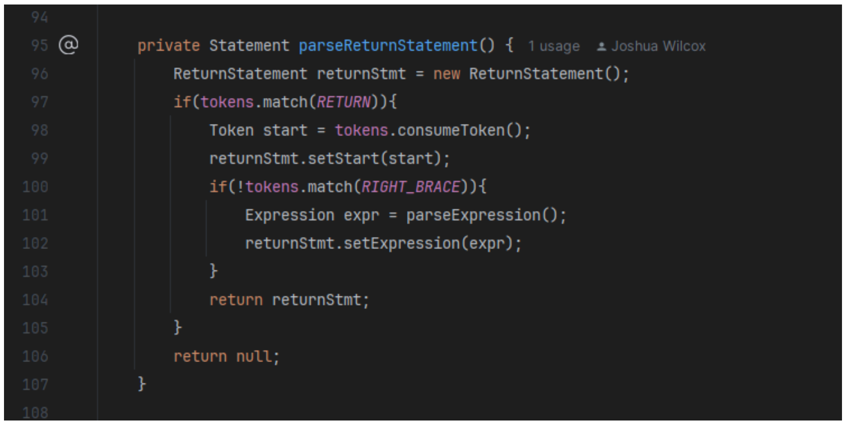
```
103
104       private boolean scanIdentifier() {  1 usage  Carson Gross
105           if( isAlpha(peek())) {
106               int start = postion;
107               while (isAlphaNumeric(peek())) {
108                   takeChar();
109               }
110               String value = src.substring(start, postion);
111               if (KEYWORDS.containsKey(value)) {
112                   tokenList.addToken(KEYWORDS.get(value), value, start, postion, line, lineOffset);
113               } else {
114                   tokenList.addToken(IDENTIFIER, value, start, postion, line, lineOffset);
115               }
116               return true;
117           } else {
118               return false;
119           }
120       }
121
```

- The if-else statement lines 11 - 115 can be simplified using a getOrDefault method call which would save some lines of code and make the program more readable.

Parser File

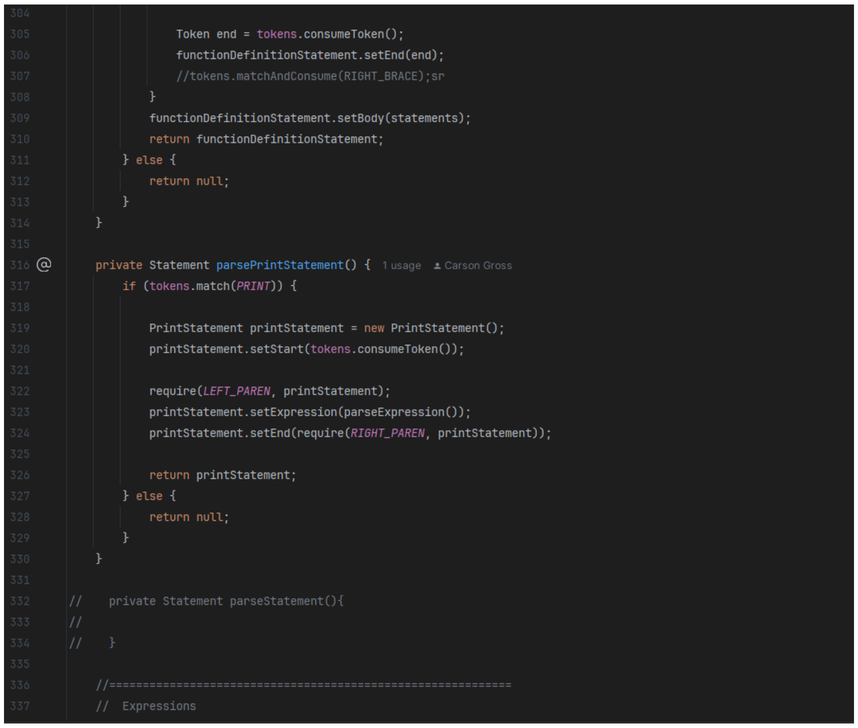
CatScriptParser.java

- Overall, lack of comments and help offered to understand what certain blocks of code mean.



```
    private Statement parseReturnStatement() {  1 usage  Joshua Wilcox
        ReturnStatement returnStmt = new ReturnStatement();
        if(tokens.match(RETURN)){
            Token start = tokens.consumeToken();
            returnStmt.setStart(start);
            if(!tokens.match(RIGHT_BRACE)){
                Expression expr = parseExpression();
                returnStmt.setExpression(expr);
            }
            return returnStmt;
        }
        return null;
    }
```

- Check for the left brace missing.
- Within the second if statement, the right brace check should come at the end.
- Bad practice not to use a require to check for the braces.

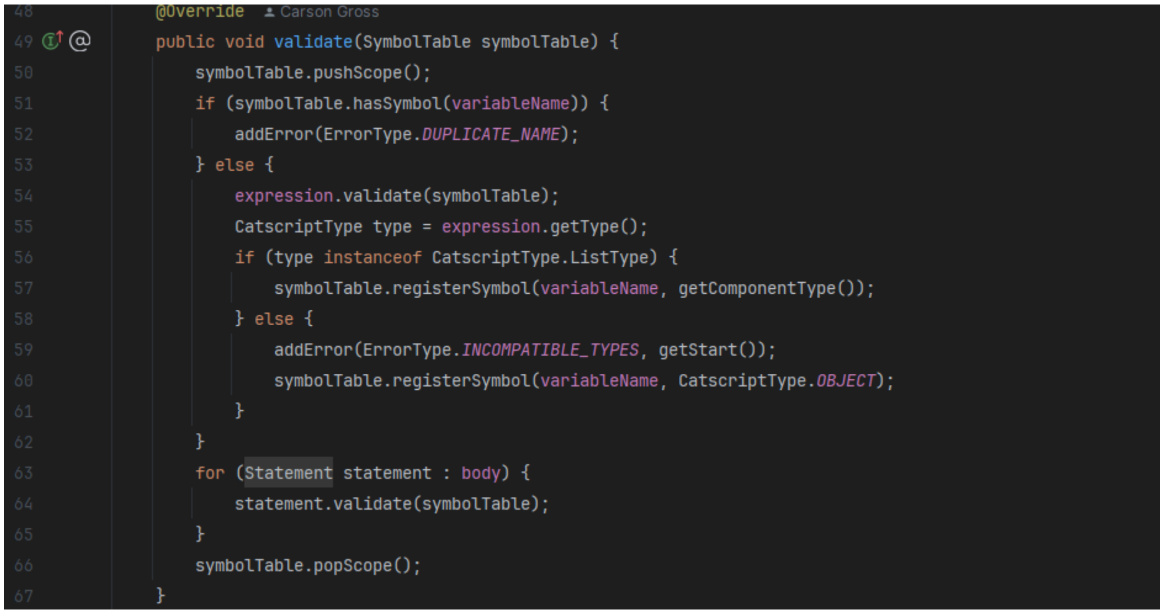


```
304
305                 Token end = tokens.consumeToken();
306                 functionDefinitionStatement.setEnd(end);
307                 //tokens.matchAndConsume(RIGHT_BRACE);sr
308             }
309             functionDefinitionStatement.setBody(statements);
310             return functionDefinitionStatement;
311         } else {
312             return null;
313         }
314     }
315
316     private Statement parsePrintStatement() {  1 usage  Carson Gross
317         if (tokens.match(PRINT)) {
318
319             PrintStatement printStatement = new PrintStatement();
320             printStatement.setStart(tokens.consumeToken());
321
322             require(LEFT_PAREN, printStatement);
323             printStatement.setExpression(parseExpression());
324             printStatement.setEnd(require(RIGHT_PAREN, printStatement));
325
326             return printStatement;
327         } else {
328             return null;
329         }
330     }
331
332 //    private Statement parseStatement(){
333 //
334 //    }
335
336     //==========================================================
337     //  Expressions
```

- Random unnecessary comments on lines 307,332,333,334, this can add unwanted clutter to the program.

Statement Files

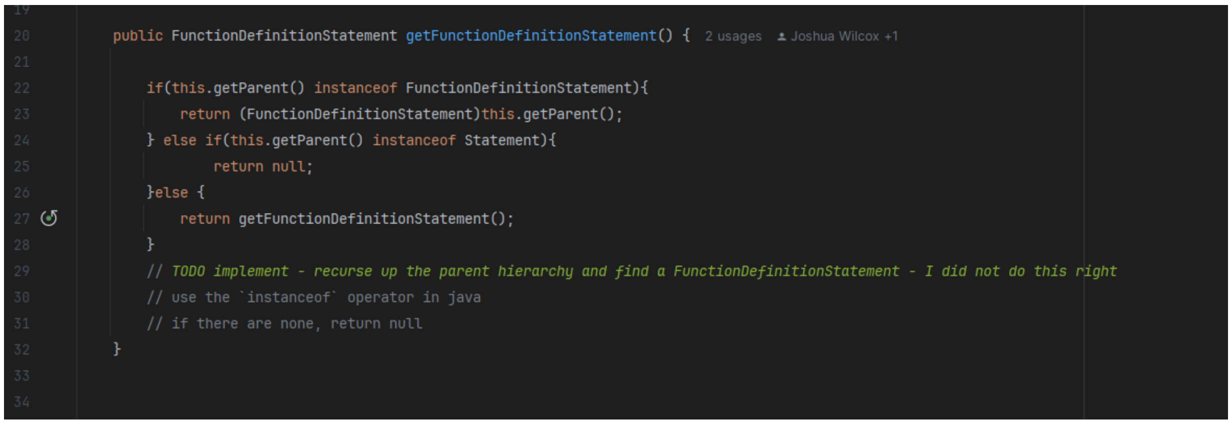
ForStatement.java



```java
    @Override   Carson Gross
    public void validate(SymbolTable symbolTable) {
        symbolTable.pushScope();
        if (symbolTable.hasSymbol(variableName)) {
            addError(ErrorType.DUPLICATE_NAME);
        } else {
            expression.validate(symbolTable);
            CatscriptType type = expression.getType();
            if (type instanceof CatscriptType.ListType) {
                symbolTable.registerSymbol(variableName, getComponentType());
            } else {
                addError(ErrorType.INCOMPATIBLE_TYPES, getStart());
                symbolTable.registerSymbol(variableName, CatscriptType.OBJECT);
            }
        }
        for (Statement statement : body) {
            statement.validate(symbolTable);
        }
        symbolTable.popScope();
    }
```

- Lack of comments makes this code confusing to understand and makes the program have little readability.

ReturnStatement.java



```java
    public FunctionDefinitionStatement getFunctionDefinitionStatement() {  2 usages   Joshua Wilcox +1

        if(this.getParent() instanceof FunctionDefinitionStatement){
            return (FunctionDefinitionStatement)this.getParent();
        } else if(this.getParent() instanceof Statement){
                return null;
        }else {
            return getFunctionDefinitionStatement();
        }
        // TODO implement - recurse up the parent hierarchy and find a FunctionDefinitionStatement - I did not do this right
        // use the `instanceof` operator in java
        // if there are none, return null
    }
```

- FunctionDefinitionStatement does not properly recurse up the parent hierarchy; the base case within the else if is incorrect. The wrong expression was used to find the function statement.

Tests for Kyler

Kyler Smith

3+ Unique Tests for Compiler Code:

INSERT INTO THE CatScriptParserStatementsTest.java `file`

```
@Test
public void printStatementEnsuresOpeningParen() {
    PrintStatement expr = parseStatement("print 1)", false);
    assertNotNull(expr);
    assertTrue(expr.hasErrors());
}
```

INSERT INTO THE CateScriptTokenizer.java `file`

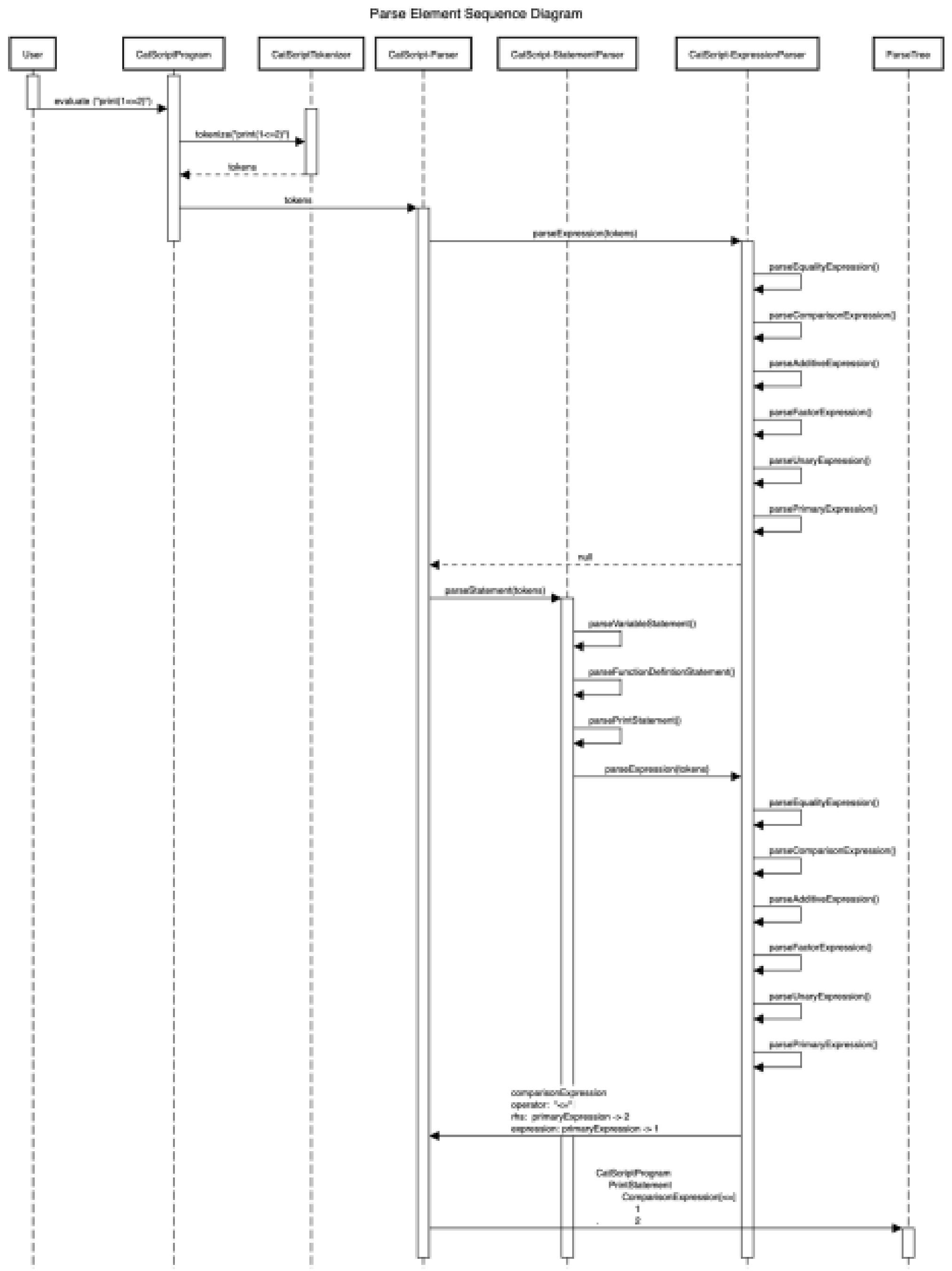
```
@Test
public void basicNumbersAndStrings(){
        assertTokensAre("1 10 234234 \"joosh\" \"Kyler\" true false", INTEGER, INTEGER,
INTEGER, STRING, STRING, TRUE, FALSE, EOF);
}
```

INSERT RED LINE INTO THE CatScriptParserStatementsTest.java functionDefWithParamsStatementTypedAndNot test

```
@Test
public void functionDefWithParamsStatementTypedAndNot() {
    FunctionDefinitionStatement expr = parseStatement("function x(a, b : int, c) {}");
    assertNotNull(expr);
    assertEquals("x", expr.getName());
    assertEquals(3, expr.getParameterCount());
    assertEquals("a", expr.getParameterName(0));
    assertEquals("b", expr.getParameterName(1));
    assertEquals("c", expr.getParameterName(2));
    assertEquals(CatscriptType.OBJECT, expr.getParameterType(0));
    assertEquals(CatscriptType.INT, expr.getParameterType(1));
    assertEquals(CatscriptType.OBJECT, expr.getParameterType(2));
}
```

UML

This is a sequence diagram for the string “print(1<=2).”



First, you start with the string and end with the elements added to the parse tree. The top value of the parse tree indicates the left-hand side, and the bottom value indicates the right-hand side.

Design Trade-offs

During the development of CatScript, several trade-offs were made to balance control, simplicity, and maintainability. One major decision was to implement a recursive descent parser instead of using a parser generator. As a result, we also hand-crafted the tokenizer rather than relying on automated token generation.

While parser generators typically rely on grammar files and regular expressions to automatically handle tokenization and parsing, building these components manually gave us far greater control over the parsing process. This manual approach made the tokenizer significantly more readable and easier to debug. For example, the tokenizer generated by tools like ANTLR can easily exceed 1000 lines of code, whereas our hand-written tokenizer was only about 130 lines, making troubleshooting much more manageable.

Choosing manual implementation over automation improved the system's transparency and debuggability at the cost of additional upfront effort. However, the trade-off favored maintainability and understanding over speed of development. One drawback to this approach is that writing a parser by hand requires managing many technical details that a parser generator would normally abstract away.

Additionally, because we controlled the parsing process directly, we avoided the need for a visitor pattern, which is typically necessary when working with code generated by parser generators. In a generated parser, the evaluation phase is separate from parsing, requiring an additional visitation mechanism to walk the parse tree. Our recursive descent parser combined these steps, simplifying the evaluation lifecycle.

Software Development Life Cycle Model

The directory of the custom tests are:

```
src/test/java/edu/montana/csci/csci468/tokenizer/CatScriptTokenizerTest.java
src/test/java/edu/montana/csci/csci468/parser/CatScriptParserStatementsTest.java
```

The model that we utilized for the making of the compiler was the Driven Development model. In this model, the goal is to know what the result should be before any code is written. So we would write tests that should pass after the completion of the project, and then actually code the code to make these tests pass. An example of this would be making sure the list of tokens from the tokenizer is what you desire in the end, and then coding the tokenizer to make that test pass. This model helped our team strongly as we knew the end goal that we wanted to reach. This especially helped during the evaluation portion of the testing phase. Some of these items would have been hard to complete if it weren't for the tests and knowing what we should get back as a result. Sometimes starting the project is the hardest thing, but if you know where you'll end up, it makes the process easier. I don't believe this model hindered our team whatsoever.