Final Portfolio

Teamwork:

Our team developing CatScript had two people. My contribution to this project was as the main developer. My teammate took on the role of a quality assurance person. I wrote all the code for the compiler and he developed tests to verify everything was working as intended and documented the process as we went. We also traded some diagrams on how we believed this to work before developing to have a good starting point to create CatScript. Along with this, we developed test cases together to make sure our CatScript was being developed to handle our use cases. Our custom test cases are under, “src/test/java/edu/montana/csci/csci468/eval/NewEvalTests.java”. Estimating the percentages I would probably say 60% of the total workload was on me and 40% of the workload was on them.

Design Pattern:

The design pattern that I used within the project was memoizing the typing system. The location of the code listed is “src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java” at line 37. This small but helpful piece of code allows for when the getListType is being called a bunch it doesn’t have to new up a new type each time, but rather return the one already created and in turn save the computations required to new up one each time. This is especially helpful for a loop that would call this over and over with the same type request. You can also think of this as a sort of caching system for the list types. This pattern implementation will not work with threading.

```
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();
    6 usages  Jadeyn Fincher +1
    public static CatscriptType getListType(CatscriptType type) {
        CatscriptType listType = LIST_TYPES.get(type);
        if(listType ==null){
            listType = new ListType((type));
            LIST_TYPES.put(type,listType);
        }
        return listType;
    }
```

Technical Writing:

Introduction:

Over the course of this final semester at MSU, we in the CSCI 468 Compilers course have been building, by hand, a compiler for the CatScript language that was originally designed by the professor Carson Gross. What follows is the technical documentation for this compiler, breaking down the different components of the language, interspersed with pictures from the underlying Java program that perform these functions. We will separate the functions of the CatScript language into the type system, the control flow system, the assignment system, the functions system, and the operator system. This documentation is not intended to be a totally exhaustive list of functionality, but instead to give the reader documentation of a representative subset of the capabilities of CatScript.

The Type System:

PRIMITIVE DATA TYPES:

There are a total of 6 primitive data types available for use in CatScript, and they are those that are commonly seen in OO programming languages. These are the “int”, “string”, “boolean”, “object”, “null”, and “void” types. Just as a sidenote, those are the exact keywords to use when assigning or comparing values.

```
public static final CatscriptType INT = new CatscriptType( name: "int", Integer.class);
public static final CatscriptType STRING = new CatscriptType( name: "string", String.class);
public static final CatscriptType BOOLEAN = new CatscriptType( name: "bool", Boolean.class);
13 usages
public static final CatscriptType OBJECT = new CatscriptType( name: "object", Object.class);
public static final CatscriptType NULL = new CatscriptType( name: "null", Object.class);
7 usages
public static final CatscriptType VOID = new CatscriptType( name: "void", Object.class);
```

During the tokenizing process, each of these keywords are then translated into a token of that type with their original value (i.e. “11” or “Peanut”) stored on them. Then, during the parsing and eval stage, the compiler looks at the Token type, and runs the rest of the compilation process as necessary.

Complex Data Types:

CatScript has a single complex data type: the “list<>” type commonly seen in most programming languages. A programmer can create lists of any of the primitive data types, except for the “void” type. The resulting variables are then called “list<int/string/…>”. Functionally, the list type is translated into a LinkedList of whatever type is pulled out of the “LIST” token type.

Here is the list type syntax from the CatScript grammar:

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

From the CatScript compiler perspective:

```
9 usages   Carson Gross +1 *
public class ListLiteralExpression extends Expression {
    8 usages
    List<Expression> values;
    3 usages
    private CatscriptType type;

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

The Control Flow System:

IF-STATEMENTS:

The CatScript language has a well-fleshed-out implementation of the “if statement”, that allows the expression being evaluated to be of arbitrary size, and that allows an equally arbitrary number of “then” statements to be added onto the initial “if” statement. Inside of each of the “if” and “else” statements contains a list of high-level expressions, which themselves can contain further statements and expressions.
Here is the if statement syntax from the CatScript grammar:

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

Here is a sample if statement in CatScript:

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

FOR-LOOPS:

CatScript includes the simplified version of the “for” loop in it’s implementation. It would actually be more accurate to call it a “for each” loop. It iterates through each item in the list type that is passed into it, and executes each of the statements that are listed in the loop.

Here is the for-loop statement syntax from the CatScript grammar:

```
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.

```java
@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();
}
```

Here is a sample for loop in CatScript:

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

The Assignment System:

STRONGLY AND WEAKLY-TYPED SYSTEM:

The CatScript language has two flexible options for how to assign types to newly-declared variables. The programmer can explicitly assign a type to the variable by putting the syntax " : CatScript_Type" after the variable name. Alternatively, the programmer can leave that bit off, and the type of the value that the variable is being assigned to, will be automatically assigned to the variable at compile time.

Here is the variable statement syntax from the CatScript grammar:

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

Here is a sample variable statement:

```
var x = 42
```

ASSIGNABILITY RULES:

CatScript contains an Assignment Statement that allows the programmer to assign different variables to the results of expressions on the other side of the "=" symbol.

Here is the assignment statement syntax from the CatScript grammar:

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

The rules for what can be assigned to what are relatively simple. Nothing can be assigned from the “void” type. Everything is assignable from the “null” type. In all other cases, the standard assignability rules for the underlying Java classes (“Integer”, “String”, etc.).

Here is a sample assignment statement in CatScript:

```
"var y = x
```

The Function System:

FUNCTION DEFINITIONS:

Functions in CatScript are declared using the “function” keyword, followed by the name of the function. Functions have the option to include a list of parameters of arbitrary size, as well as a return type of any primitive or complex CatScript type that is specified after the parameter list. All parameters must have their types explicitly set in the declaration. Inside the function body is another list of either regular statements or return statements to be executed, similar to the internals of the “for loop” bodies. You can see all of this is in the CatScript grammar for function declarations:

Here is the function declaration syntax from the CatScript grammar:

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

Here is a sample function declaration:

```
"function foo(x : list<int>) { print(x) }"
```

RETURN STATEMENTS:

CatScript has an interesting return statement grammar. First off, the return statements are not required to return anything, they can simply be called in the middle of a function, similar to the “break” keyword in Java. Also, pointers to functions or other none-CatScript-type values are not allowed (so no arrow functions). However, any type can be returned from functions, even void.

Here is the return statement syntax from the CatScript grammar:

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

Here is a sample return statement instance in a CatScript function:

```
"function foo() : list { return [1, 2, 3] }"
```

The Operator System:

ARITHMETICAL OPERATIONS:

CatScript provides support for the standard arithmetical operations present in most programming languages. These include addition, subtraction, division, multiplication, and negation operations, all with the standard orders of operation. These operations are divided into the “additive expression”, “factor expression”, and “unary expression” categories. It is interesting to not that the additive expressions also allow for basic string concatenation. For this documentation, we will highlight the additive expressions only, as they are representative of the rest of the categories.

Here is the syntax for additive expressions in CatScript:

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

Here is a sample additive expression in CatScript:

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

BOOLEAN OPERATIONS:

CatScript also provides support for the basic Boolean operations seen in nearly all programming and scripting languages. These include “>”, “<”, “>=”, “,<=”, “==”,“!=”, and “!” operations. Below is the syntax and a sample.

Here is the syntax for boolean expressions in CatScript:

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

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

Here is a sample boolean expression in CatScript:

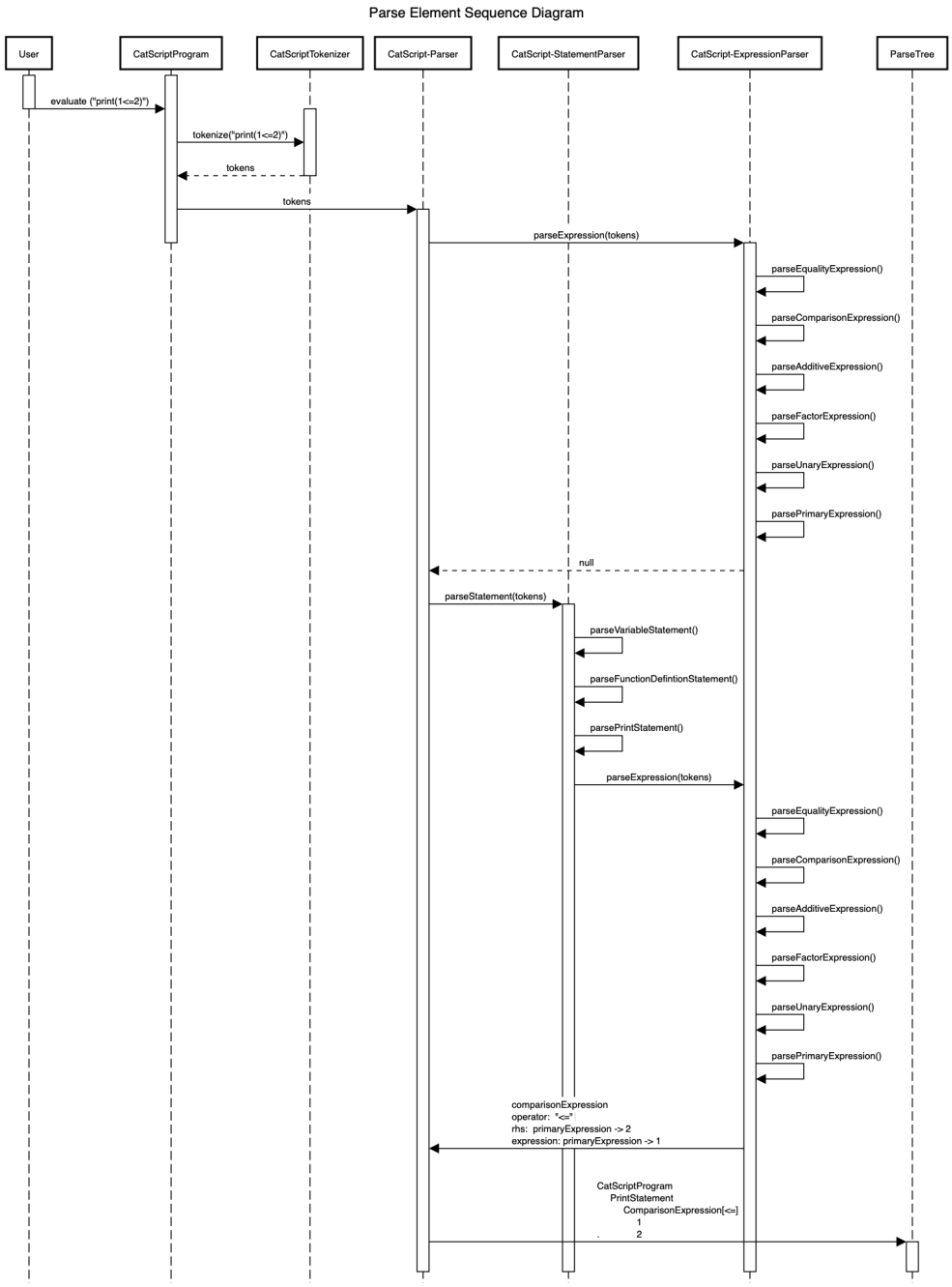
```
"2 <= 1"
```

Conclusion:

This is a middle-depth description of the capabilities, functionalities, and mechanics of the CatScript programming language. Please use the knowledge you have just acquired for good.

UML:

Below shows a sequence diagram of how the string “print(1<=2)” would parse out. We started with the string and end with the elements of the parse tree at the end. The top value of the parse tree represents the left-hand side and the bottom value represents the right-hand side.



Design Trade-Offs:

There were a few tradeoffs that we got to make throughout the development of CatScript. The main trade-off that we utilized during this class was using a recursive decent parser rather than a parser generator and by doing this we also generated our tokens by hand. The tradeoff for using our recursive decent parser was that we defined our tokens manually and parsed them out manually. In a normal parser generator, you would use a grammar file that would take care of extracting the tokens if you define them with some regular expressions. Being able to define it by hand we had much more control over how everything worked together. Along with this manual control we have a much more debug-friendly tokenizer. An example that we went over in class showed the generated Antler tokenizer generated over 1000 lines of code, and the tokenizer we wrote in class was merely 130 lines of code. I believe the trade-off for readability over writability was worth it as finding myself debugging what went wrong was much easier working with my own proprietary software. The only downside to doing our generator the way we did is that a parser generator is much easier to write, and you don't need to worry about a lot of the technical details since they are handled within generation.

By using the recursive decent parser we also didn't need to use the visitor pattern. This is because we had complete control throughout the evaluation lifecycle so no extra visitation was needed. This is unlike a normal parser generator where you don't have control of what happens after the generation is completed, requiring a visitation pattern to accommodate.

Software development life cycle model:

The model that we utilized during the creation of this capstone project was the Test Driven Development model. In this model, the goal is to know what the end result should be before any code is actually 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 wasn'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.