Section 1: Program

- see source.zip

Section 2: Teamwork

Team member 1 contributed to the development of this compiler through documentation (see Section 4), and the creation of unit tests to evaluate proper parsing functionality. Time was spent between myself and team member 1 at 95 - 5%. I was responsible for implementation of the parser, tokenization, and bytecode generation (see section 1).
Below are the unit tests written by Team member 1.

```
    void forLoopInFunctionWithVarReassignment() {
        assertEquals("6\n", executeProgram("function foo() : int {\n" +
                "  var y = 0\n" +
                "  for( x in [1, 2, 3] ) {\n" +
                "    y = x + y\n" +
                "  }\n" +
                "  return y\n" +
                "}\n" +
                "print( foo() )\n"));

    }
    @Test
    public void compareFunctionResultTest() {
        assertEquals("true\n", executeProgram(
                "function foo(x : int) : int { return x + 1 }\n" +
                        "var y = 3\n" +
                        "print(foo(y) > y)"
        ));
    }

    @Test
    public void forLoopAndStringConcatTest() {
        assertEquals("Test A\nTest B\nTest C\n", executeProgram(
                "var x = [\"A\", \"B\", \"C\"]\n" +
                        "var y = \"Test\"\n" +
                        "for(z in x) { \n" +
                        "   print(y + \" \" + z)\n" +
                        "}"
        ));
    }
```

Section 3: Design Pattern

The memoization design pattern was used within the CatScript type system. Memoization is used to optimize function performance by caching the results of computationally expensive or repetitive function calls and returning the cached result when identical inputs are used. This in turn reduces redundant work, efficiency, and in our particular case, wasted memory.

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
   return cache.computeIfAbsent(type, ListType::new);
}
```

The memoization pattern was used here to mitigate wasteful creation of ListType instances. Instead, caching is utilized within a HashMap to search for a pre-existing ListType instance; if it’s not there, a new instance is created.

Section 4: Technical Writing

Catscript Documentation

This document should be used to create a guide for catscript, to satisfy capstone requirement 4

Introduction

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

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

Features

Type Expressions

CatScript includes the following type expressions:

```
int
string
bool
object
list<type_expression>
```

Equality Expressions

Catscript supports equality evaluations which can be used to check for equality or inequality. This can even be used to evaluate the equivalence of string values.

```
1 == 1
```

```
"this" != "that"
```

Comparison Expressions

Integer comparison is also supported in Catscript. These can evaluate whether one integer is greater than or less than another. These comparisons can also include equivalence.

```
1 < 2
```

```
5 >= 4
```

Additive Expressions

Additive expressions are supported for adding and subtracting integer values. These expressions are left associative.

```
3 + 2 - 1
```

In the case of the Catscript addition operator, this implementation also allows for string concatenation where the second string is appended to the first.

```
"One" + "Two"
```

Factor expressions

Factor expressions in Catscript is supported for multiplying and dividing integer values. Similarly to additive expressions, factor expressions are also left associative.

```
4 * 3 / 2
```

Unary Expressions

Unary expressions are supported in Catscript for both integer values and boolean values. For integers, the "-" operator is used to invert the value from positive to negative or vice versa. For boolean values, the "not" operator inverts true to false or false to true. Unlike additive and factor expressions, unary expressions are right associative.

```
-1

not true
```

Primary Expressions

Primary expressions include literals for integers, strings, booleans, nulls, and lists, as well as identifiers and function calls.

```
1
"asdf"
true
null
[1, 2, 3]
foo()
```

Lists

Lists have been implemented as a data type in Catscript. They can be used as the specified data type in a variable statement. Lists can either be set to contain a single explicit data type of values or an abstract object type where they can contain values of different type simultaneously.

```
var a : list<int> = [ 1, 2, 3 ]

var b : list = [ "This", 15, true ]
```

For loops

For loops have been implemented to iterate over a list of values and repeatedly execute a block of code for each of these iterations. The identifier used by the for loop to store each of the values in this list must be unique.

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

If-else statements

If statements have been implemented in Catscript with optional *else-if* and/or *else* statements. These statements are used to instruct a program to execute a block of code only if a certain condition is met.

```
var x = 5

if ( x < 10 ) {
    print("yes")
} else {
    print("no")
}
```

Print Statement

Print statements are implemented to output the result of provided expression.

```
print("This is an expression")
```

Variable Declaration

Catscript allows variable to be declared to store a reference to a specified value. Variable statements begin with the "var" keyword, followed by a name for the new variable, and lastly a value for said variable to be set to. Optionally, an explicit data type can be provided for the variable by adding a colon and the desired data type following the variable name.

```
var x : string = "value"
```

Variable Assignment

Variables in Catscript can not only be declared and allocated, they can also be reassigned. This can be done by calling a previously declared variable name and resetting the value. When resetting a variable, the new value must be of the same type as the value to which the variable was previously set.

```
var x = "Value"
x = "New value"
```

Function Declarations

Function declarations are how functions can be defined in Catscript. These declarations must begin with the "function" keyword followed by an identifier that will serve as the name of the function. Next, an optional list of parameters can be provided for the function to take as input, each of which can optionally be set to take in an explicit type value. An optional return type can also be provided to ensure the function only returns one specific type value. Finally, we have the block of code that the function will be set to execute starting with an open brace and ending with close brace.

```
function foo( x : int, y : int ) : bool {
    return x > y
}
```

Function Calls

After a function has been declared, they are called to be executed by calling the name of the function and providing the necessary parameters.

```
function foo( x : int ) {
    print(x + 1)
}

foo(3)
```

Return Statements

Return statements are an important part of Catscript functions. These statements not only specify the output of a given function but also actively cease the execution of a function if and when they are reached. They are

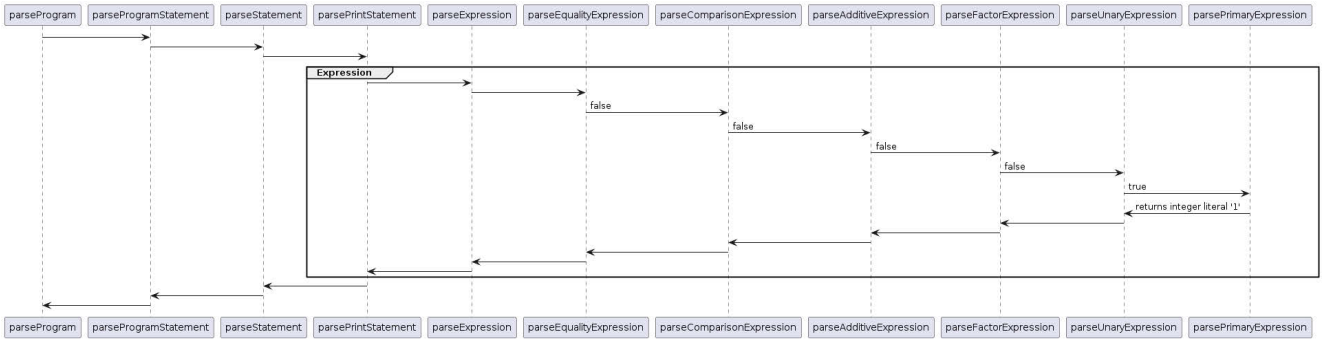
constructed simply by calling the "return" keyword followed by an expression to evaluate. Since functions can return void type, return statements can be used to end function execution at any point.

```
function foo() : string {
    return "Value"
}
```

Grammar

```
catscript_program = { program_statement };

program_statement = statement |
function_declaration;

statement = for_statement |
if_statement |
print_statement |
variable_statement |
assignment_statement |
function_call_statement;

for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
'{', { statement }, '}';

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

print_statement = 'print', '(', expression, ')'

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

function_call_statement = function_call;

assignment_statement = IDENTIFIER, '=', expression;

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

function_body_statement = statement |
return_statement;

parameter_list = [ parameter, {',' parameter } ];

parameter = IDENTIFIER [ , ':', type_expression ];

return_statement = 'return' [, expression];

expression = equality_expression;

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

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

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

factor_expression = unary_expression { ("/" | "*" ) unary_expression };

unary_expression = ( "not" | "-" ) unary_expression | primary_expression;

primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null"|
list_literal | function_call | "(", expression, ")"

list_literal = '[', expression,  { ',', expression } ']';

function_call = IDENTIFIER, '(', argument_list , ')'

argument_list = [ expression , { ',' , expression } ]

type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' , type_expression, '>']
```

Section 5: UML



Sequence Diagram *see printStatementSeqDiagram.png for directly for better view

In the above sequence diagram, the code 'print(1)' is being parsed. After the program is initialized in parseProgram, it is determined that the print statement is the next to be parsed, and parsePrintStatement is called. Once parsePrintStatement has been called, the 'PRINT' token is matched, a new instance of printStatement is created, and the token is consumed as the start of the statement is marked.

Then, a left paren is required before moving into parseExpression within the print statement. Following the parseExpression call, control flow moves down each type of expression until the 'else' condition is met within parseUnaryExpression to then invoke parsePrimaryExpression.

Next, parsePrimary expression will return the integerLiteralExpression '1' back to parsePrintStatement. A right closing paren token is then required as the end of the statement is set, before printStatement is returned to the program. Lastly, the program will finish executing, and '1' is output to the console.

Section 6: Design Trade-offs

We chose to create a recursive descent parser for our project, as they are relatively easy to understand and implement for lightweight grammars. Additionally, directly implementing a recursive descent parser allows us to have explicit control over the parsing process, making it easier to customize and debug. One other benefit is that recursive descent parsers have very high performance for small grammars and input data. A disadvantage that was encountered with our recursive descent parser is that ambiguity in token recognition can arise from left recursion, as some aspects of the grammar begin with the same token. Another disadvantage of writing our own recursive descent parser is that generating a comfortable amount of informative error messages can be challenging.

Many developers in the business of writing a new programming language may choose to use a parser generator when building a program language, as much of the process can be automated with premade parser generator tools. Parser generators excel at handling larger, more complex grammars, and automate the code generation process. A primary disadvantage of using a parser generator is that tools can be unintuitive, and require a steep learning curve to effectively implement a parser. Additionally, developers have less control over the generated code, which can make customization and debugging more difficult. Generated code quality may also be of undesirable quality in terms of readability and complexity.

Section 7: Software Development Life Cycle

The Test Driven Development model was used to complete this project. I believe this model aided in the development of our compiler as it provided clear requirements and concrete feedback on progress. Unit tests clearly define what functionality certain sections of the codebase must have, and their completion facilitated a much faster feedback loop on when progress was made.

TDD also aids in early bug detection, and mitigates regression in intended code functionality due to unintended side effects caused by new changes. Lastly, TDD simplifies collaboration and documentation as tests serve as executable documentation that describe the behavior of the code. Collaboration is facilitated among team members by providing a common understanding of the codebase’s functionality.