

Section 1: Program

This project covers writing a compiler for a new scripting language Catscript. The specifications of Catscript follow this 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 2: Teamwork

Team member 1: 50%

The main engineer for the project. Designed the main design patterns and implemented the code. Spent most of the time on the src directory writing code.

Team member 2: 50%

Documentation and testing engineer. Wrote the documentation for the language and design pattern, as well as designing the test suite of unit tests.

An example of the tests are:

@Test

```
void listLiteralExpressionsEvaluatesProperly() {  
  
    assertEquals(Arrays.asList(4, 10, 15), evaluateExpression("[(2+2), (20-10),  
(2+10+3)]"));  
  
    assertEquals(Arrays.asList(25, 10, 30), evaluateExpression("[(5*5),  
(100/10), (50/5*3)]"));  
  
}
```

@Test

```
void ifStatementWorksInforStatementProperly() {
```

```
    assertEquals("not3\nnot3\nfound3\nnot3\n", executeProgram("for(x in [1, 2, 3, 4]) {"
```

```
        "if(x==3) {"
```

```
            "print(\"found3\")"
```

```
        "} else {"
```

```
            "print(\"not3\")"
```

```
        "}"
```

```
    "});
```

```
}
```

```
@Test
```

```
void incrementGlobalVarInForLoopWorkProperly() {
```

```
    assertEquals("2\n6\n14\n", executeProgram(
```

```
        "var count = 0"
```

```
        "for( x in [2, 4, 8] ) {\n"
```

```
        "  count = (count+x)\n"
```

```
        "  print(count)\n"
```

```
        "}\n");
```

```
}
```

Section 3: Design pattern

One design pattern we used was memoization. Memoization is an optimization pattern that is used to prevent similar objects from being constructed many times. Instead, the first time an object is instantiated, it is placed in a list so it can be referenced later without the need of making a new one. It is located on line 36 in CatscriptType.java. The reason you would use this pattern is to save runtime resources. Without memoization, everytime a list type is needed, a new type object would have to be constructed. Construction of objects in java can be very expensive, as it takes up a lot of memory, compute time, and needs to be garbage collected. With memoization however, only one object of each type at most will be instantiated. This saves a lot of time if the user decides to create a bunch of lists with different types.

Highlight the design pattern in yellow. Explain why you used the pattern and didn't just code directly.

Section 4: Technical writing. Include the technical document that accompanied your capstone project.

Catscript Guide

Introduction

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

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

Features

Print

The Catscript print statement is used for outputting and displaying a catscript value onto the screen. Note that this method will append a newline character(\n) onto the end of whatever value you are printing to the screen.

Ex)

```
print("Welcome to Catscript!")  
print(10)  
print(false)  
print([5, 15, 25])
```

Output)

```
Welcome to Catscript!  
10  
false  
[5, 15, 25]
```

Var

The Catscript var statement is short for variable statement and can be used to declare a new variable with a value assigned to it. Catscript variables can contain any of the valid Catscript types as values but once a variable has been assigned a value then that variable is bound to the type of its value permanently. Note that

there are two ways in which we can initialize a new catscript variable. The first is without giving an implicit type for the variable and letting the compiler determine which type the variable should be based on its value.

Ex1)

```
var str = "This is a string variable"  
var x = 10  
var y = x + 5
```

The second way we can initialize a catscript variable is by giving it an implicit type in its initialization rather than letting the compiler determine it for us. This method of initialization can be very helpful when you're expecting a certain type and want to ensure that the variable is of that type.

Ex2)

```
var x : int = 50  
var isTrue : bool = true  
var myList : list<int> = [1, 2, 3]
```

Assignment

CatScript variable assignment is used to change the value of a variable that has been previously initialized using the var keyword. It's important to remember that variables can not change types. This means that the new value must have the same type as that of the original variable value.

Ex)

```
var x = "Cat"  
print(x)  
x = "Script"  
print(x)
```

Output)

Cat Script ```

If Statement

The Catscript If statement uses booleans and logical expressions that evaluate to booleans to decide between different paths of control flow in your CatScript program. If the expression within the 'if' evaluates to true then the code in the brackets directly under the 'if' will be executed but if the expression is false then it will skip the first brackets and execute the 'else' statement if it exists.

__Ex)__

```
var x = 10
if(x > 5){
  print("X is greater than 5")} else {
  print("X is less than 5")}
```

__Output)__

X is greater than 5 ```

For loops

The Catscript For loop is used for iterating over a catscript list. Note that you are able to iterate over a list of any of the catscript types.

Ex)

```
for(x in ["str", 1, true, null]) {
  print(x)}
```

Output)

str 1 true null ```

Function Definitions

Functions in the CatScript programming language are used to group together chunks of code that can be reused throughout your program. A

function in CatScript must be declared and given a function definition before it can be called. We use the "function" keyword to declare that we are creating a new function definition.

```
--Ex)--
```

```
function foo(x) {  
x = x + "Script" print(x) }
```

Function Calls

Function calls in CatScript are used to execute the code that was previously defined in a corresponding function definition. A function can be called any number of times and can pass in different values each time through what are known as function parameters.

```
--Ex)--
```

```
function foo(x) {  
x = x + "Script" print(x) }  
foo("Cat")  
foo(5)
```

```
--Output)--
```

```
CatScript  
5Script ``
```

Returns

The Catscript return statement is a way to pass data from within the function to the place where the function was called from. To declare the function will have a return statement we add a colon followed by a CatScript type after the function parameters. The "return" keyword followed by a value is the way to return a value and end the body of the function call.

Ex)

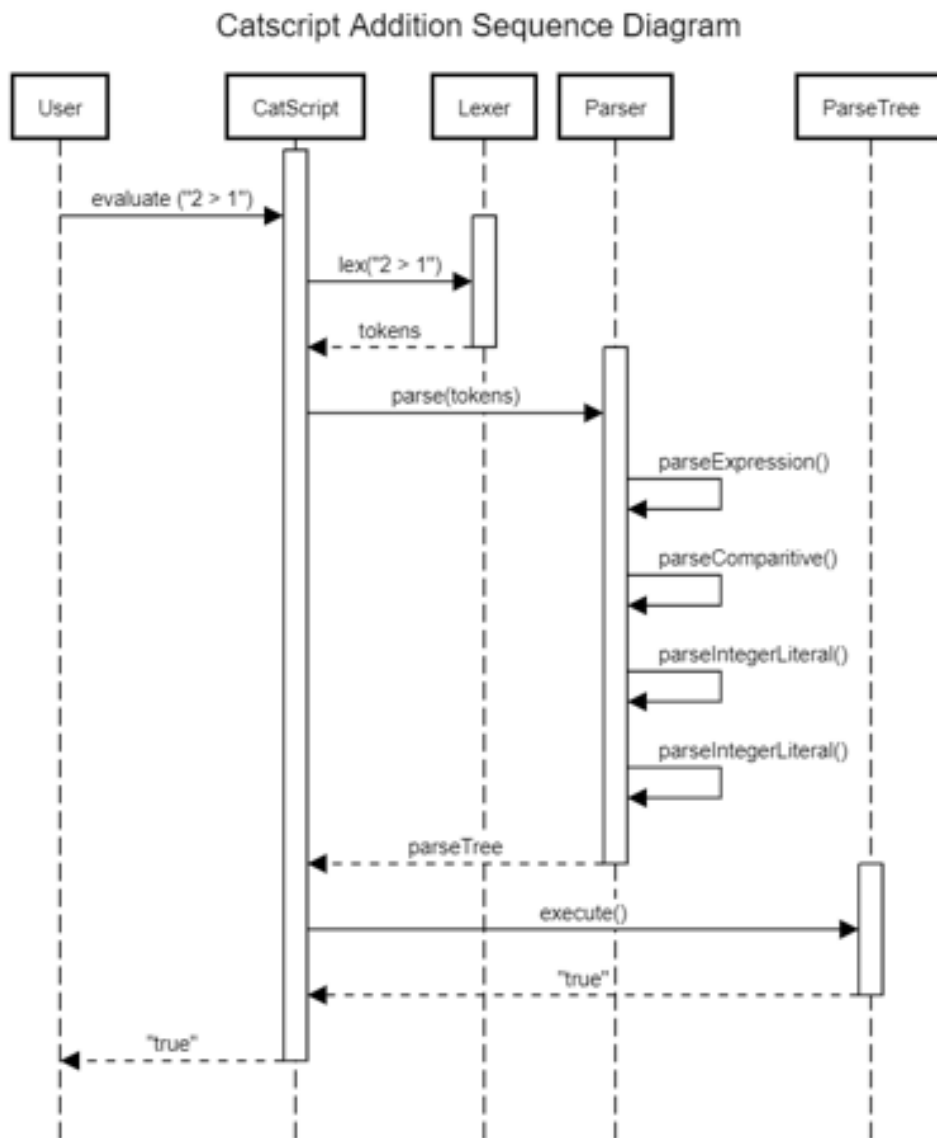
```
function addFive(x : int) : int {  
    return x + 5}  
print(addFive(10))  
print(addFive(-15))
```

Output)

15

-10 ```

Section 5: UML.



Section 6: Design trade-offs

The main design tradeoff in the catscript parser is using recursive descent rather than a parser generator. Most other compiler classes use a parser generator. A parser generator takes the grammar of the language in the form of regex strings, then creates a parser for you. The downsides of this approach is that you need to create the regex strings, which can be error prone and hard to modify at a later date. It also generates the parser for you, so you do not understand the underlying workings of the compiler you are making. What we ended up using was the recursive descent approach to the parser. The recursive descent approach requires you to code the parser by hand, which can be more difficult than passing in the grammar to a parser generator. However, recursive descent is a powerful design pattern that can be used outside of parser generation. For example at my job we used it to parse bytes as a communication protocol. It also allows you to understand how a parser works in greater detail. Recursive descent is an intuitive pattern to follow, and it removes a lot of confusion with how a parser generator works.

Section 7: Software development life cycle model

The software model we used to design our program was Test Driven Design (TDD). TDD is when you write your tests first, then write code to pass the tests. This design is great for adding features, as it allows you to code for specific things you need in your code. The downsides of TDD is that you need to write an extensive test suite before you can start programming. This can have similar pitfalls to the waterfall model, as it is not as agile as writing features first, then adding tests later. Another downside to TDD is that it does not help with designing sustainable code. If you just write code to pass the tests, you can run into extensibility issues that could have been solved by decoupling your features. The upside to TDD is that it requires a thorough test suite. Unit tests are a great way to test your code, as they can be run at any time for little cost, and it can test parts of your code that may not be visible to the outside. This makes regression testing a lot easier when unit tests already account for many of the features.