

CSCI 468
Spring 2023
Camille Custer
Avery Jacobson

Section 1: Program - include a link to the zip file mentioned above

See source.zip in this directory

Section 2: Teamwork - Discuss your partners documentation and testing contribution, as well as your primary work on the project

My partner Avery Jacobson wrote the Catscript documentation that is in section 4 of this document. He wrote complex tests for me to test my code as well. The workload was shared with about an 80-20 ratio as I did all of the coding on this project.

Section 3: Design pattern - We are going to memoize type access in the review session

A design pattern used in this project was the memoization pattern. It is used when costly functions are called multiple times with the same input instead of taking up excess space. The first time it is called, the result is saved in a hashmap. When a function is called with the same argument, the hashmap is searched and if the result is stored in it, the result will be returned. If it is not in the hashmap, it will be added for future use. This improves the speed and efficiency of the program.

Below is my implemented memoization pattern.

```
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();

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;
}
```

Section 4: Technical Writing - Include the documentation generated by your partner for the catscript programming language

Catscript Guide

Introduction:

Catscript is a statically typed functional programming language that consists of commonly known expressions and statements in programming languages. Documentation for this language is listed below.

#####

Catscript Types

var: The Catscript “var” type initializes a name to a value within the script. The “var:” may be followed by an explicit type, expression, or can be implicitly defined.

Implicit:

```
var message = "hello"    //implicitly takes string type
var number = 3           //implicitly takes int type
var isOpen = true        //implicitly takes bool type
var girlfriend = null    //implicitly takes object type
```

Explicit:

```
var message: string = "hello"
var number: int = 3
var isOpen: bool = true
var girlfriend: object = null
```

object: The Catscript “object” is the root of the class hierarchy. Every Catscript class has object as their superclass. “int”s, “bool”s, and all other types implement methods of the object class.

int: The Catscript “int” type represents a whole number integer value ranging from -2^{31} to $2^{31}-1$.

string: The Catscript “string” type represents a set of characters in a string format. String values are immutable by default.

bool: The Catscript “bool” type represents a Boolean value. The “bool” type can take on only two values, true or false. The bool type is essential in programming logic.

null: The Catscript “null” type represents a null object value. This null object value has no distinct string, integer, or Boolean value. The null object represents an empty value.

list: The Catscript “list” type represents a list literal of any of the above Catscript types. The type must be specified within brackets. The Catscript list can be implemented as an iterator.

```
var enemies: list<string> = [“Frank”, “Venessa”, “Chad”]
```

Expressions

Comments:

Comments in Catscript are snippets of text that are not read in by the Catscript interpreter. You begin a comment with two forward slashes. A comment is terminated at the end of the line.

```
print (“My name ”) //THIS IS A COMMENT. THE NEXT LINE WILL BE INTERPRETED  
print(“jeff”)
```

Primary Expressions:

Primary Expressions in Catscript consist of any literal expressions or identifiers present in the program. Primary expressions can be parenthesized.

The following are all examples of primary expressions:

```
x  
9  
“sup”  
(9 + 4)  
13
```

Unary Expressions:

Unary expressions in Catscript are either “-” or “not”. The “-” expression is only applicable with an integer value, and will flip the value from positive to negative, or negative to positive. The “not” expression is applicable with Boolean values, and will invert the Boolean value.

The following are examples of unary expressions:

```
not true  
-5
```

Factor Expressions:

Factor expressions in Catscript are only applicable to integer types. Factor expressions use only “*” and “/” operators, are used to multiply and divide integers, respectively.

The following are examples of factor expressions:

```
x*y  
y/x
```

Additive Expressions:

Additive Expressions in Catscript are used to add or subtract values. Int type objects can be added or subtracted from one another, while string values can be added (by concatenation), but not subtracted. If an integer is added to a string, the integer value will be treated as a string value and concatenation will occur.

The following are examples of additive expressions:

```
7-3  
x+y  
“My name “ + “jeff”  
“I have “ + 8 + “jelly beans”
```

Comparison Expressions:

Comparison Expressions in Catscript are used to compare two expressions, using either “<”, “>”, “<=”, or “>=”. These operators (respectively) are less than, greater than, less than or equal to, or greater than or equal to. These are used to compare the lefthand side and righthand side. These expressions return a Boolean value of either “true” or “false”.

The following are examples of Comparison expressions:

```
3<4  
5>=7  
9<189
```

Equality Expressions:

Equality expressions in Catscript determine if the lefthand side is equal or not equal to the righthand side. This expression uses the equality operator “==” and the not-equals operator “!=”. This expressions returns a true or false value similar to the comparison expression.

```
3 == 4 //not True  
5 == 5 //true
```

Statements

For Loop Statements: For loops in Catscript are very similar to for loops in other languages. For loops begin with “for”, followed by an identifier, followed by “in”, followed by another iterative identifier. This loop will repeat for every element in the second iterative identifier, referred to by the first identifier name.

```
for(num in [5,6,7]) {  
  print(num+" is the number")  
}
```

OUTPUT:

```
5 is the number  
6 is the number  
7 is the number
```

If Statements:

If statements in Catscript rely on an expression that returns a Boolean value. If the expression within the if statement evaluates to true, the following block of code will execute. A Catscript if statement also has the functionality to implement an “else” clause. This else clause will execute if the expression within the if statement evaluates to false. If statements typically rely on a comparison expression, but can also execute off of an equality or Boolean literal expression.

```
If(true) {  
  print("smelly")  
} else {  
  Print("not smelly")  
}
```

Print Statements:

Print Statements in Catscript operate using the “print” keyword followed by an expression with surrounding parentheses. This expression is typically a string value enclosed in quotations, but can also be a variable value. The print statement then returns the value to a terminal.

```
print("my message")
```

OUTPUT:

```
my message
```

Variable Declaration Statements:

Variable declaration statements in Catscript begin with the “var” keyword and are followed by an identifier. The Catscript Type of the var can be determined implicitly by following the identifier with an equals and assigning it a value. Variable types can be explicitly defined by following the identifier with a colon, and a Catscript type.

```
var x: int = 7
or
var x = “boof”
```

Function Declaration Statements:

Functions in Catscript are initialized using the “function” keyword, followed by an identifier. The function is then later referenced or called using this identifier. Following the identifier is a set of parentheses. If the function requires parameters, you will initialize parameters within the parentheses. This is followed by a block of statements enclosed in curly braces. This block of statements is executed when a function call statement to the given functions identifier is called. Functions will either return an object, integer, string, list, Boolean, or null.

```
function divideThree(x: int) { //This is a function Declaration
  print(x/3)
}

divideThree(9)                //This is a function Call
```

OUTPUT: 3

Function Call Statements:

Function call statements in Catscript begin with an identifier that is linked to a function definition. If the function definition requires input parameters, these are placed within parentheses in the function call. If the function does not require parameters, leave the parentheses blank. When the function is called, the body within the function definition executes, given the parameters you implemented.

Return Statements

Return Statements in Catscript begin with the “return” keyword and are followed by an expression. Return statements are only callable within the function definition body. The expression value following the return statement will be returned by the function.

```
function divideThree(x: int) {
  return x/3
}

print(divideThree(9))
```

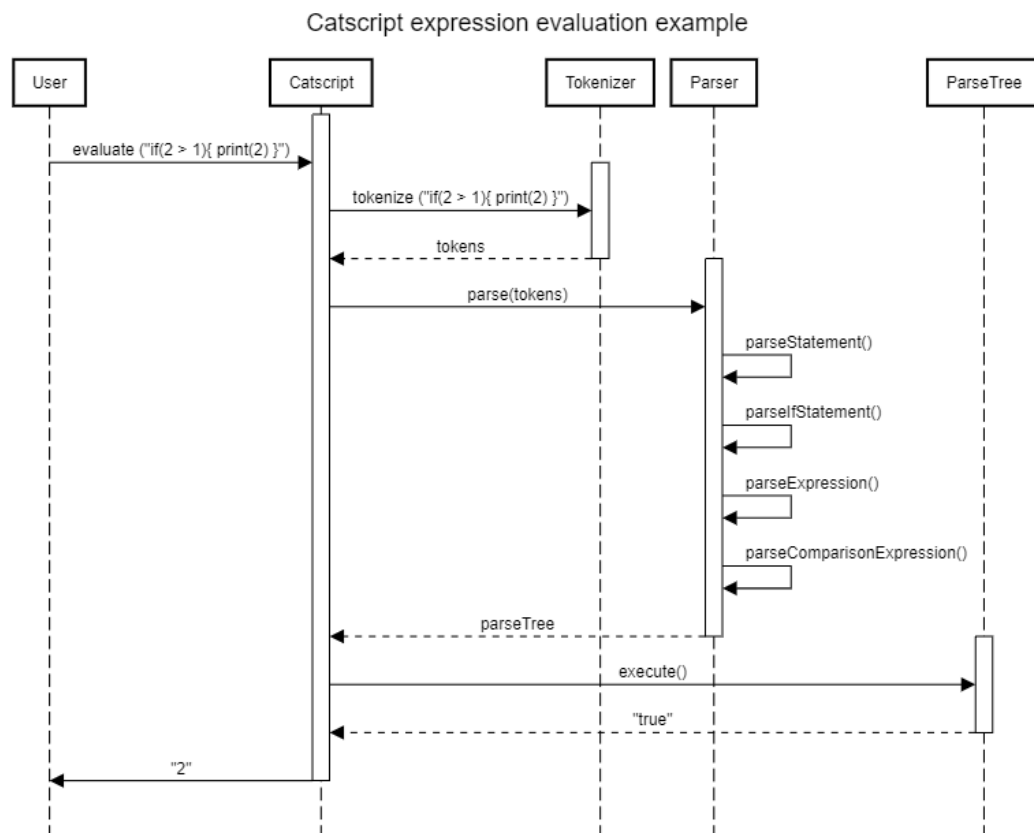
OUTPUT: 3

Section 5: UML - Include and discuss one of the UML diagrams included in this directory

Below is a simple UML sequence diagram of the compiler evaluating an expression.

```
if (2>1){  
    print (2)  
}
```

The compiler starts by tokenizing the expression. The tokenizer then passes those tokens on to the parse to build the parse tree. The recursive descent algorithm begins with `parseStatement()` function and goes through each function until there is a match. In this case, the match is with the `"parseStatement()"` method. After the if statement has been parse, the inside condition (2>1) must be parsed. The inside expression is parsed as a comparison expression. After the comparison expression evaluates to "true", the inside expression of the if statement will be executed. True is returned and "2" is printed.



Section 6: Design trade-offs - You must write this, maybe discuss recursive descent vs. a parser generator?

Using a recursive descent parser instead of a parser generator made understanding the structure of the parser much easier. Recursive descent gives total control over the parsing and it takes much less work. Recursive descent is much more efficient and easier to debug than a parser generator. This project was my first time using recursive descent and I found it to be logical and simple to understand. The recursive descent structure of the parser followed the grammar almost exactly, making a pyramid-like design for the code, starting with the broadest function at the top.

Section 7: Software development life cycle model - We did Test Driven Development, please discuss your experience with it

TDD was the best possible software approach for building the compiler as it is such a large project. With TDD, I was able to start small and take each test on one by one, rather than feeling overwhelmed trying to write the entire thing at once. My experience with TDD really helped me learn to debug my code as well. Prior to using TDD, I manually debugged my code which takes way longer than if I had help from the debugger in VSCode. Most tests began as failed and through simple code writing I was able to get tests to pass. Once the tests passed, I was able to increase the complexity and quality of my code until it met all requirements.