

## CSCI 468 Compilers Portfolio

Montana State University Spring 2023

Jack Brown

## Section 1: Program

<https://github.com/JackBrown/csci-468-spring2023-private/tree/main/capstone/portfolio>

## Section 2: Teamwork

I was the primary engineer for this project. I spent my time programming and writing functions to pass tests. Team member 2 was the documentation and testing engineer. They wrote the tests and documented the programs functionality and progress along the way. I worked on the project roughly 90% of the total project time (~5 hrs/week) and team member 2 spent roughly 10% of the project time on their work.

## Section 3: Design Pattern

The Memoization Pattern is a design pattern which is used to increase efficiency in a program by creating a cache for computations/executables that is later called upon instead of re-computing. An example of this in the Catscript program can be seen in the 'getListType' method of the 'CatscriptType.java' file. Rather than creating a new list each time the method is called, a hashmap search is executed to find an already created list of that type and return it. This pattern was used, rather than coding directly, because its caching method saves both time and memory for the program.

## Section 4: Technical Writing

### *Introduction*

Catscript is a simple scripting language

### *Features*

#### **Statements:**

##### **Assignment Statement:**

Assignment Statements allows catscript to assign a new value to a variable with an existing value.

Variables can only be reassigned the same type that they were originally defined as.

Example:

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

### For Statement:

For Statements in Catscript are used to iterate through lists using the "in" keyword, values are stored until there are no more elements. Unlike other languages, this loop cannot iterate based on boolean conditions meaning it can't count up to a value, but it can perform operations like print-statements and if-statements.

Example:

```
var myList = [1,2,3]  
  
for (i in myList){  
    print (i)  
}
```

### Function Definition Statement:

Function definition statements in Catscript begin with the keyword "function" followed by the function name, argument identifiers, and types, and a block of statements inside the body. Once a return statement is executed, the value is stored, and the function definition completes. Functions can have explicit return types but default to void, and parameters can have explicit typing or default to an object. Return statements can be included and should have a return type defined after the function name and parameter. Functions can be called elsewhere in the program.

Example:

```
function add(x:int, y:int):int{  
    var answer = x + y  
    return answer  
}
```

Alternatively, this add function could just **return x + y**

### Function Call Statement:

Function Call is used to execute a function that has been defined. The call will execute if it matches the number and type of arguments given in the function definition, and the symbol table is checked to ensure the function has been registered.

Example:

```
var answer = add(1,2)
```

```
print answer
```

output:

**3**

### If Statement:

If Statements are used to decide whether to execute the statements inside the block. They work similarly to If statements in other languages and can include conditional statements that limit access to code unless the condition is satisfied.

Example:

```
var x = 3
```

```
if (x > 2){
```

```
print("x is greater than 2)
```

```
}
```

```
else{  
  
    print("x is less than 2")  
  
}
```

### Print Statement:

In Catscript, the print statement is written as "print('expression')" and the expression inside the parentheses is evaluated by the parser and printed to the output stream. This statement can print various contents such as string or integer literals.

Examples:

|                                 |                       |
|---------------------------------|-----------------------|
| <pre>print("hello world")</pre> | <pre>print(1+2)</pre> |
| output:                         | output:               |
| <pre>hello world</pre>          | <pre>3</pre>          |

### Return Statement:

In Catscript, the return statement is used to exit a function and return a value to the caller. When the parser encounters the 'return' keyword, it evaluates the expression, assigns the resulting value to the function definition, and pops all local variables off the stack. Return statements can only be used within a function, and are required if the function has an explicit return type.

Example:

**See Function Definition Statement example**

### Variable Statement:

Variable statements are used to declare and assign variables with the **var** keyword. When defining variables, you can optionally include a type by placing a colon (:) after the variable name and specifying the type. Any primary expressions can be assigned to variables, and variable types can be either explicitly or implicitly defined

Examples:

```
var myString = "Hello"
```

```
var x : int = 2
```

### **Types:**

CatScript is statically typed, with a small type system as follows

int - a 32 bit integer

string - a java-style string

bool - a boolean value

list - a list of value with the type 'x'

null - the null type

object - any type of value

### **Expressions:**

#### **Primary Expressions:**

Identifier Literal:

Integer Literal:

List Literal:

Null Literal:

Parenthesized Expressions:

String Literal:

Type Literal:

Boolean Literal:

### Additive Expression:

Catscript's additive expression uses the (+) and (-) operators for addition and subtraction. It is used to combine two expressions. String Literals can only be added together and not subtracted. The order of evaluation is from left to right, with the operator separating the left-hand side and right-hand side expressions.

Examples:

|                             |                       |
|-----------------------------|-----------------------|
| <pre>var x = 1 + "hi"</pre> | <pre>print(1+2)</pre> |
| <pre>print(x)</pre>         |                       |
| output:                     | output:               |
| <pre>1hi</pre>              | <pre>3</pre>          |

### Comparison and Equality Expressions:

Equality and comparison expressions are both used in Catscript to evaluate two expressions and return a boolean value. While comparison expressions have a higher precedence, both expressions have a left side and a right side. **Equality expressions** use the (==) and (!=) operators to assess whether both sides are equal or not equal, respectively. On the other hand, **comparison expressions** use the (<), (>), (<=), and (>=) operators to compare two Integer Literals.

Examples:

|                     |                  |                      |                  |
|---------------------|------------------|----------------------|------------------|
| <pre>2 == 2</pre>   | <pre>true</pre>  | <pre>2 != 2</pre>    | <pre>false</pre> |
| <pre>2 &gt; 3</pre> | <pre>false</pre> | <pre>2 &lt;= 3</pre> | <pre>true</pre>  |

### Factor Expression:

Catscript's factor expression uses the (\*) and (/) operators to perform multiplication and division. Along with additive expressions, it has a left and right hand side. The left hand side is evaluated first.

Examples:

**2 \* 10**      is 20                      **10/2**              is 5

### Unary Expression:

Catscript has two unary operators, the negative symbol and the not symbol, that can be applied to Integer Literals and Boolean Literals, respectively. These operators allow negation of the value of an expression. The (-) operator is used for integers, while the (not) keyword is used for Boolean values.

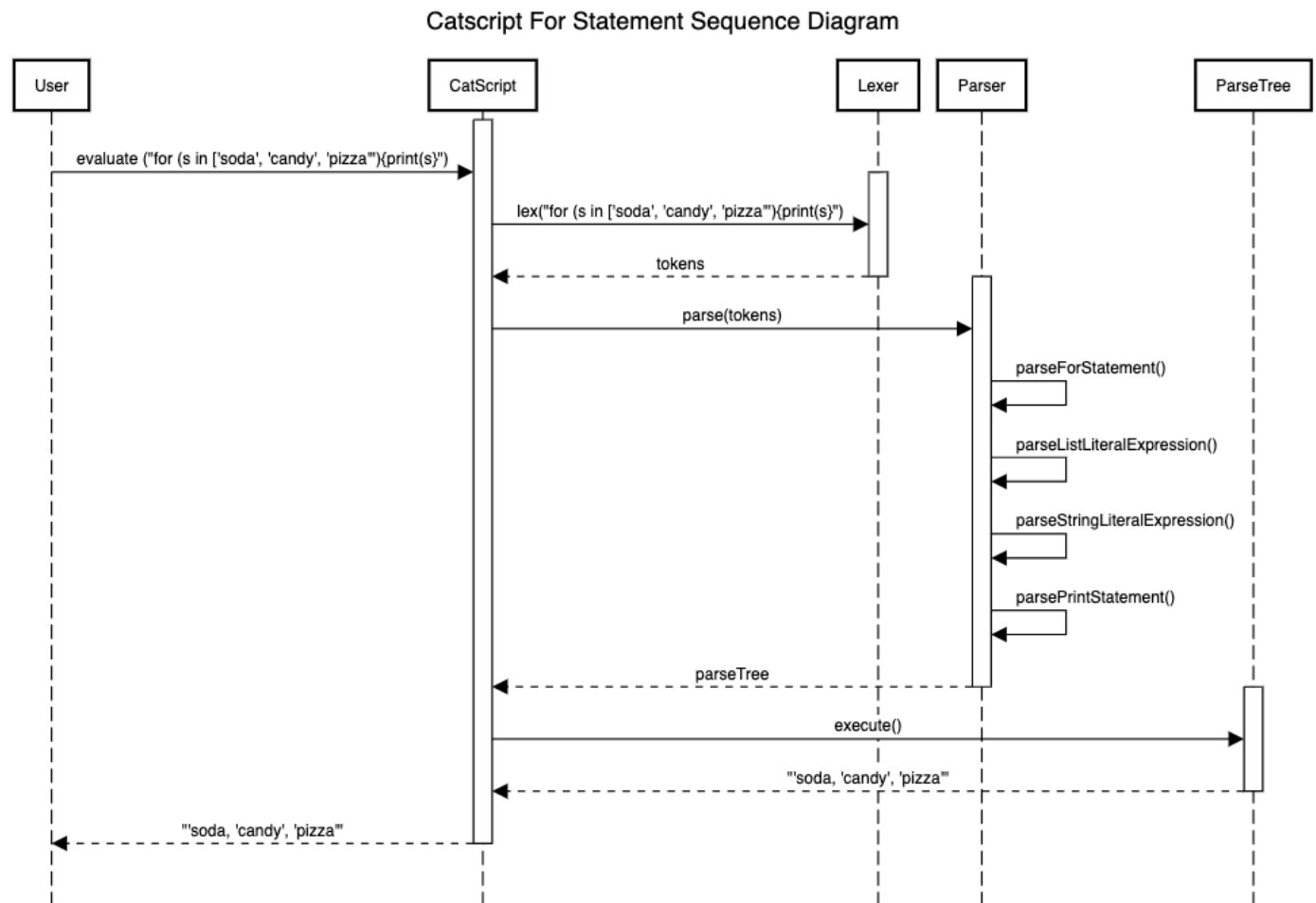
Examples:

**-2**

**not true**



## Section 5: UML



## Section 6: Design Trade-Offs

The main design trade-off for this program is that the parser was created by hand rather than using a parser generator. Our hand written parser uses recursive descent. This method creates a function for each procedure of the grammar, which calls another function recursively in order to accomplish a task. The benefit of this method is that it allows us to get an in-depth understanding of how the grammar works. It also creates readable code that mirrors the structure of the grammar.

The alternative option is to use a parser generator. A generator takes lexical grammar and an EBNF language grammar as input and returns a parser class. The downside of this method is that it creates complex code that is difficult to read and debug. The inputted grammar must also be rather specific which can cause problems in the created parser. Another reason it was not used is because it does not help further the developers understanding of the recursive nature of grammar.

## Section 7: Software Development Life Cycle Model

The life cycle model used for this project was test driven development. I was given a test suite containing a list of tests which specified the language and functionality of the program. As the parser, and other elements of the project, were completed, the tests began to pass. I enjoyed this model for several reasons. It broke a large project into smaller goals which could easily be seen and accomplished. The tests gave a great outline of what was needed before the coding began. This model also allowed for only the necessary code needed, since its purpose was to pass tests.

I prefer this method when compared to an agile behavior driven development model. I have found that behavior driven development can cause developers to get stuck with creativity. There is a lot of work around what the user would want or how something could be developed differently. In the context of this project, I felt a test driven development was best as the objective was to create a functioning compiler.