

**Compilers – CSCI 468: 001 202330**

**Spring 2023**

**Team Members:**

Griffin Austin

James Lucas

**Montana State University Computer Science Department**  
**Senior Team Portfolio**

## Section 1: Program

Along with this portfolio there is an attached zip file which contains all of the source code for the associated project.

Below is the Grammar for the language that we built from the ground up: CatScript.

```
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, '>']
```

Below are the static types associated with CatScript.

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

## **Section 2: Teamwork**

Team member 1:

Tokenization: 15%

Expression parsing: 15%

Statements parsing: 20%

Evaluations: 25%

Compilation: 10%

Team member 2:

Test Writing: 5%

CatScript Documentation: 10%

## **Section 3: Design pattern**

Throughout our code we come upon places where we have calculated an output for a given input once before. In order to limit the number of calculations performed by our code we used the **memoization** optimization technique. This involves storing outputs that have high processing time or high power associated with generating that output. This pattern can be seen in the *CatscriptType.java* file on line 37-46. By using this pattern for the function *getListType()* on line 38, we eliminate the need to

create a new *ListType()* object when looking for types in a given list repeatedly. If we were to not use memoization in this function we would create a new *ListType()* object every time we called that class. Rather, by using memoization, we only make the *ListType()* object once and add to it when needed. This is necessary because there can be more than one type within each list; which is described within the CatScript grammar.

## **Section 4: Technical writing.**

### **Catscript Guide**

#### **Introduction**

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

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

Which, when ran, will output:

foo

#### **Features (can be structured with the grammar given above)**

##### **For loop**

For loops are implemented in CatScript so that they function like that of other for loops in other similar languages. The for loop allows you to iterate over elements, example below:

```
1 var myArray : list<string> = ["one", "two", "three"]
2 for(str in myArray)
3 {
4     print(str)
5 }
```

### **Output**

one two three

##### **If Statement**

The if statement is used to execute code based on a condition. If the user would like to specify an *else* branch, that is available to them, shown below:

```
1 var decision = true
2 if(decision)
3 {
4     print("Decision")
5 }
6 else
7 {
8     print("No Decision")
9 }
```

## Output

Decision

### Print Statement

Print statements are used to display information to the user. This is done by calling the print function with a given argument, and that is then pushed to the console for display, shown below:

```
1 print("Hello World!")
```

## Output

Hello World!

### Variable Statement

Variable statements are used to declare new variables of any type. The type can be implicit using a colon and a static type, or the optional colon and type can be left out. If the type is left out, then the variable is of type *object*. Variable with implicit declaration is shown below:

```
1 var myInt : int = 14
```

### Function Call

Function calls are used to call functions with optional arguments. A function call is shown below:

```
1 function add(first:int, second:int): int
2 {
3     return(first+second)
4 }
5
6 var additionValue:int = add(10,20)
7 print(additionValue)
```

## Output

30

### Function Declaration

Function declarations are used to declare a function. By using the keyword *function*, and a name followed by optional parameters and return type, you can declare any function that you would like; shown below:

```
1 function add(first:int, second:int): int
2 {
3     return(first+second)
4 }
5
6 var additionValue:int = add(10,20)
7 print(additionValue)
```

## Output

30

### Assignment Statement

Assignment statements are used to assign a new value to an already declared variable, seen below:

```
1 var myInt : int = 10
2 myInt = 15
3 print(myInt)
```

# Output

15

## Equality Expression

Equality expressions are used to check equality between two objects within the program. There are two ways to check for equality by using either “==”(equal) or “!=”(not equal), shown below:

```
1 var one = 1
2 var two = 2
3 print(one == two)
4 print(one != two)
```

# Output

false true

## Comparison Expression

Comparison Expression are used to compare two objects within the program. There are many ways to compare two objects. There is [ >, <, >=, <= ] to compare objects, which can be seen below:

```
1 var one = 1
2 var two = 2
3 var twoAgain = 2
4
5 print(one > two)
6 print(one < two)
7 print(two <= twoAgain)
8 print(one >= two)
```

# Output

false true true false

## Additive Expression

Additive Expressions are used to add two integers or string together, shown below:

```
1 var one:int = 1
2 var two:int = 2
3
4 var strOne:string = "Hello "
5 var strTwo:string = "World!"
6
7 print(one+two)
8 print(strOne+strTwo)
```

## Output

3 Hello World!

### Factor Expression

Factor expression are used to multiply or divide values within the program, shown below:

```
1 var ten:int = 10
2 var five:int = 5
3
4 print(ten*five)
5 print(ten/five)
```

## Output

50 2

### Unary Expression

Unary expressions are used to negate values. This can be seen below:

```
1 var negOne = -1
2 var myBool = true
3
4 print(negOne)
5 print(not myBool)
```

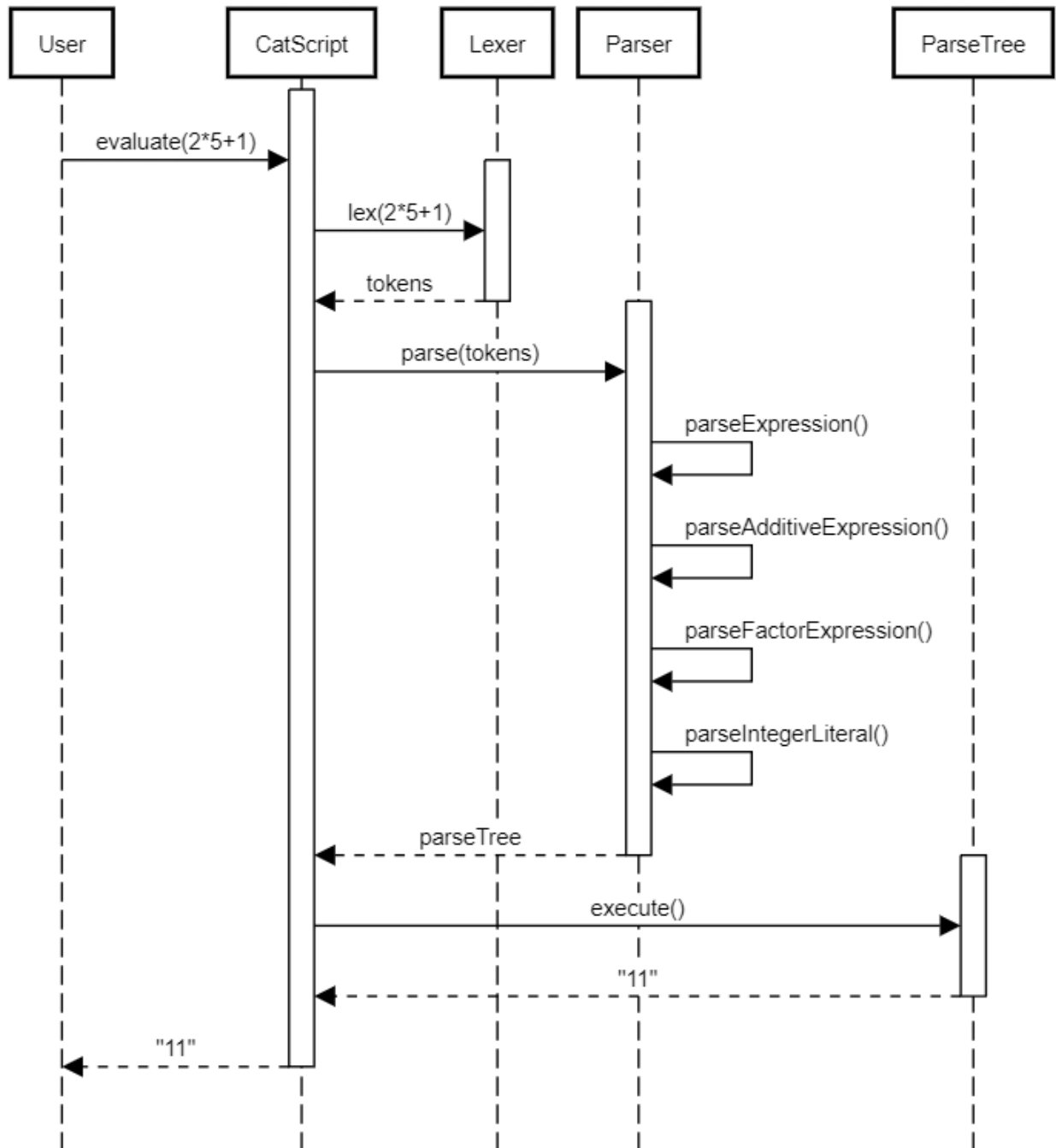
## Output

-1 false



## Section 5: UML.

CatScript Sequence Diagram



## Section 6: Design trade-offs

The most important design trade-off that we did was that we created the parser from scratch rather than just using a tool to create the parser for us. We did this so that we could have a more

intuitive recursive decent algorithm for the parser. By creating the parser from scratch, we not only got to understand parsers on a low level, but we also made the algorithm more understandable to anyone that can read code. However, this took more time and effort than just using a parser tool. Overall, this trade-off was one that we would not take back as making the parser from scratch proved to be useful throughout the rest of the project.

## **Section 7: Software development life cycle model**

Throughout this project we used test driven development. This is where you write tests before you write the code. This allows the team to create inputs and desired outputs which allows you to create the code that makes those tests pass. This model was very helpful for our development because a code language is quite complex. By having the tests, we were able to break the project down into smaller pieces that were easily manageable. Without the tests, the whole project would have felt much more daunting than it already was. By focusing solely on making the tests pass, we could focus on each part of the language separately starting from the ground up. This development model did not hinder us in any way, and I believe that this was one of the only models that would have worked for this project. Overall, test driven development got us thinking about what the code should look like before we even wrote it, which made the whole process much smoother and clear.