

Class:  
Compilers (CSCI 468)

Semester:  
Spring 2023

Members:  
Ezra Skoog and Baiden McElroy

# Section 1: Program

---

Please see source.zip in /capstone/portfolio

## Section 2: Teamwork

---

In order for our team to create the CatScript programming language we split the workload into two pieces. With group member 1 being the primary engineer and group member 2 being the testing and documentation engineer. Group member 1 was in charge of writing the methods and algorithms that handled the compilation of the Catscript language and spent approximately 40 hours on this task. This process started by creating a tokenizer that takes some input and gives back a list of the individual tokens. Next, the methods and control flow to parse the tokens were created for all the Catscript expressions and statements. After this, evaluation methods were implemented for each Catscript feature to evaluate the results of a Catscript expression or statement. Finally, compile methods were added to each feature to turn evaluation results into Java byte code. Group member 2's main contributions were building out additional tests in order to ensure the Catscript language works as expected and to help find any potential bugs. In addition, group member 2 created thorough documentation for all the features of Catscript. Group member 2 spent approximately 6 hours working on these tasks.

## Section 3: Design pattern

---

While CatScript may just be a simple scripting language, it remains essential to optimize our code as best as possible. With that said I chose to use the memoization design pattern in order to create a more optimized program. The memoization pattern was used on the `getListType` method in the `CatscriptType` class. Originally this method would always create a new list type for every time it was called even if that same list type had already been created in the past. With the memoization pattern being implemented on this method it now creates each list type only once. It does this by storing previously initialized list types into a hashmap and then reusing the list types in the map if they exist rather than recreating them. This saves our program time and resources by reusing list types rather than reinitializing them repeatedly.

```
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

---

## Statements

### For loop

The for loop statement in catscript is one of the main control flow options. It is used to loop through a list or repeat a set of instructions. Here is an example of the for loop:

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

In this example the for loop is used to iterate through a list and print out its contents.

To use the for loop, use the keyword “for” followed by the control statement then brackets for the code body. In the control statement, put an identifier followed by the keyword “in” then the object you want to iterate through. In the example above, “i” is the identifier and the list “[1,2,3]” is what the loop is iterating through. In this case “i” would evaluate to an integer “1” the first time through the loop, and “2” and “3” respectively until there are no more objects in the list.

The for loop also creates its own scope. “i” will not be accessible outside of the loop. Changing of the scope also does not allow for shadowing. If “i” is declared before the for loop, there will be a compile-time error where “i” is not allowed to be redeclared.

## If Statement

The if statement in catscript is another of the control flow options. It uses the keyword “if” followed by a comparison statement, then a body, and an optional “else” statement. Here is an example of the if statement:

```
var x = 3
if(x>4){
    print("higher than 4")
}else{
    print("lower than 4")
}
```

In this example the if statement is comparing x to 4, and splitting the control flow to change the print statement. If x is greater than 4, then the first print statement is executed, but if x is less than or equal to 4 then the second print statement will execute.

The if statement also changes the scope. After the comparison statement is executed, the body that is used creates its own scope. One thing to note is that at compile time, there will still be an error if there are scoping issues, even if the branch isn't used.

## Function Definition

The function definition is the last control flow statement. It uses the keyword “function” followed by the identifier, then the arguments with optional types, the return type, a body, then a “return” keyword if the function is non-void.

```
function foo(x :int) :int{
    return x
}
```

In this example the function has the identifier “foo”, with one argument that has to be an integer and a return type of integer. The body just returns “x”., but it can contain any number of statements.

The function changes scope, so “x” cannot be used outside of the function.

## Function call

The function call statement is used to invoke a function. It takes a number of arguments equal to the number defined in the function definition, and has an identifier to identify the function it is referring to.

```
foo(1,2)
```

In this example, foo is the identifier, and “1” and “2” are the arguments passed to the function definition.

## Print statement

The print statement takes an expression and prints its value to the console. It can only take 1 argument, it does not support multiple arguments like other languages.

```
print(1)
```

In this example, it prints “1” to the console.

## Variable Statement

The variable statement starts with the “var” keyword, then an identifier, and takes the right-hand side of the equals and assigns it to the identifier.

```
var a = [1, 2, 3]
```

In this example, the list “[1,2,3]” is assigned to a. “Var” automatically boxes the type for the identifier from the type that it is assigned from.

## Expressions

### Equality

The equality expression returns a boolean value based on whether two expressions are equivalent.

```
1 == 1
```

In this example, the result would be true

### Comparison

The comparison expression returns a boolean value based on whether one expression is greater or less than another expression.

```
2 >= 1
```

In this example, 2 is greater than 1, so true is returned.

### Additive

The additive expression takes two expressions and adds or subtracts them.

```
1 + 1
```

### Factor

The factor expression takes two expressions and multiplies or divides them.

```
1*3
```

## Unary

The unary expression negates the value, either with a “not” or “-”

```
-1
```

## Literals

There are five literals in Catscript: int, string, list, bool, and object. Examples of these types are:

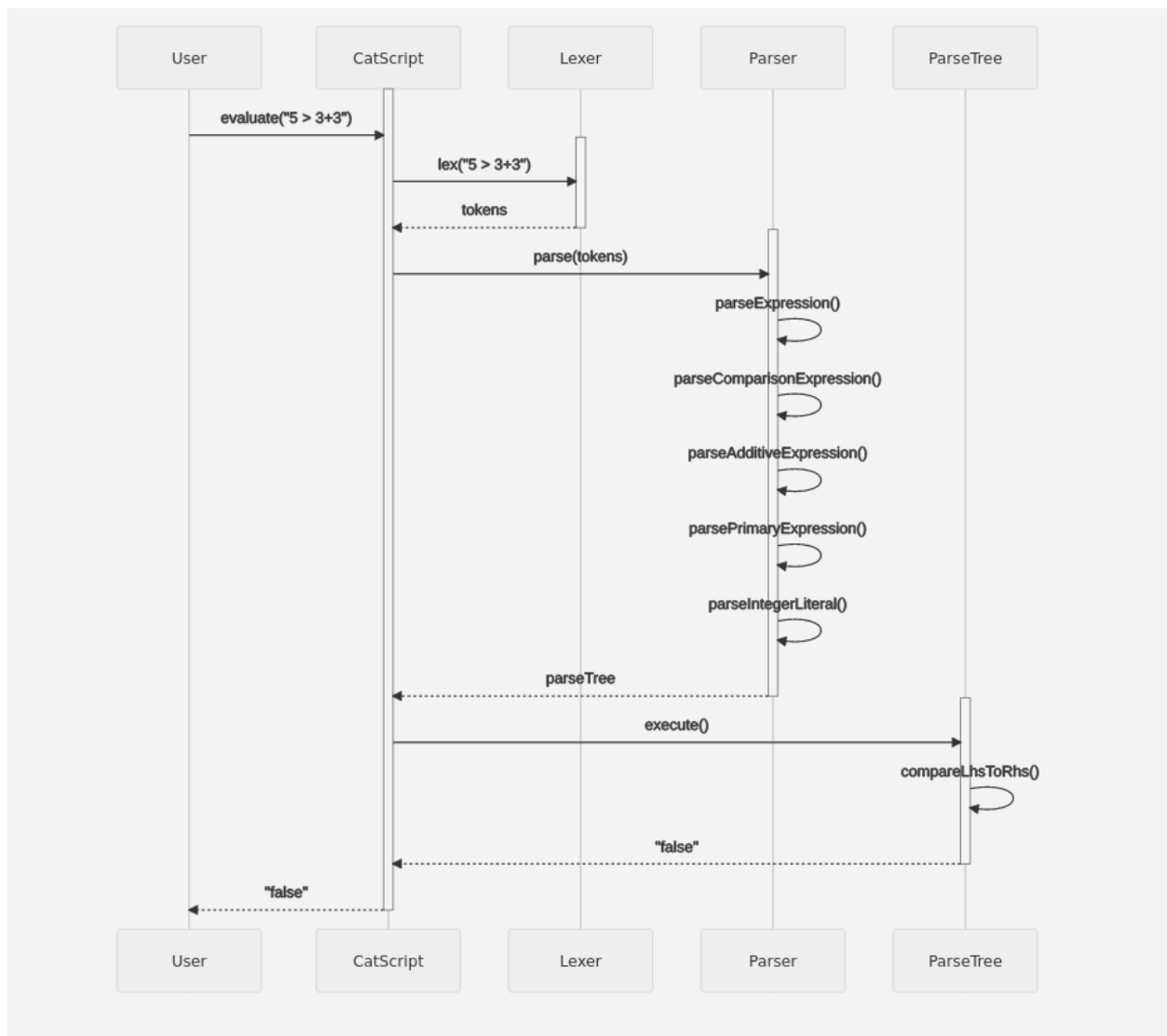
```
1  
"1"  
[1,1]  
true  
object
```



# Section 5: UML

For my UML diagram I choose to use a sequence diagram on the comparison statement. The sequence diagram allows for a good visual representation of some of the main components in the Catscript language such as the lexer and parser. On top of this the sequence diagram also gives a good idea on how the control flow in our compiler works.

## CatScript Comparison Sequence Diagram



## Section 6: Design trade-offs

The main design trade-off in Catscript language is the use of a recursive decent parser rather than using a parser generator. Although both of these methods have their own pros and cons, I decided the recursive decent parser was the more intuitive way to handle our parser. One of the biggest reasons that lead me to make this choice was the amount of control you have over a recursive decent parser versus a generated parser. Although generated parsers are fast and easy to create they give you nowhere near the same amount of control as a recursive descent parser. With a recursive decent parser, you have complete control of all the functionality in the parser and this allows us(the language creator) to implement anything we like such as custom error messages or error handling. On top of the control that recursive decent parsing provides it is also a great way to ensure you properly understand the grammar of your own language. When creating the recursive decent parser, you are forced to build out the control flow for your grammar which in turn will help ensure you understand how your grammar functions.

## Section 7: Software development life cycle model

In order to develop the Catscript programming language we used the method known as test-driven development. In this software development life cycle model, we were given a test suite filled with tests mimicking Catscript behavior and over time fixed these tests by implementing features in Catscript such as the tokenizer, parser, and bytecode generator. I found this method of software development to be extremely useful in helping me better understand and build Catscript. The main benefit I noticed when using test-driven development is that it gave me a much better idea of what I wanted to code before I even started. By having tests written before coding you give yourself a base for what the programming language should look like and act like. While on the other hand, I've noticed if the tests are written after the code then you end up writing the tests to work with the code rather than writing them to mimic how you want your language.