

CSCI 468: Compilers

Spring 2022

Joseph Burke

Ali Khaef

Section 1: Program

Here is the link to the source file.

[source code](#)

Section 2: Teamwork

In this project we had a team of two people. I will henceforth refer to these members as **Member 1** and **Member 2**.

Member 1 was primarily responsible for the overall code of the program and making sure 100% of the unit test suite passed. Test suite work included Tokenization, Parsing Expressions, Parsing Statements, and Bytecode generation. **Member 2** was primarily responsible for the documentation of the program (*refer to section 4*), but also included 3 additional unit tests to the overall test suite to provide a more complete testing environment.

Tests from **Member 2** added functionality to the codebase including:

- Return checking working properly

By creating unit tests for this problem, our team was able to identify areas of the code that could be implemented and we were able to discuss solutions together to ensure that the program would work as intended.

Total estimated hours: 200

Member 1

Contributions: Primary programming, ensuring 100% test coverage

Estimated Work Hours: 190, 95% overall contribution.

Member 2

Contributions: Documentation, added unit tests

Estimated Work Hours: 10, 5% overall contribution.

Section 3: Design Pattern

The design pattern used in this project was memoization. Memoization is a technique that allows us to store the results of a function call and return the saved result if the function is called again with the same arguments. This allows us to avoid the cost of re-evaluating the function. It is especially useful when the function is expensive to compute, and called relatively often.

Here is how it was implemented in the code:

```
37 private static Map<CatscriptType, ListType> cache = new HashMap<>();
38
39
40 public static CatscriptType getListType(CatscriptType type) {
41     ListType listType = cache.get(type);
42     if (listType != null) {
43         return listType;
44     } else {
45         ListType newListType = new ListType(type);
46         cache.put(type, newListType);
47         return newListType;
48     }
49 }
```

You can see the function tries to get a return from the cache first and if it is not there (meaning, the result from the cache is null), it then computes a new result and stores that result in the cache to save for later. So, if this function were to be called again with the same argument, it would have the result already saved in the cache hashmap and would be able to directly return the result instead of having to compute a new one again.

Section 4: Technical Writing

Introduction

Catscript is a statically typed language which compiles on java virtual machine. the operators that catscript accepts are as follows: >,<,<=,>=,!=,+,*,/,not. the supported statements are as follows:

ForStatement,IfStatement,Functions,PrintStatement,VariableStatement,ReturnStatement. the types catscript supports are, int,bool,string,object,list<>.

Features

For loops

```
for(x in ["a","b","c","d"]){  
    print(x)  
}
```

the above is an example for loop in catscript, that will iterate through a list. in the for-loop we have our variable x that will become each value inside the list. any number of statements can be inside-of the curly bracers.

If statement

```
if(x < 10){  
    print(x)  
}  
else  
{  
    print("10")  
}
```

the above is an example if statement in catscript, that will evaluate on a boolean expression. if the expression is true, the statements in the if block will be executed, otherwise the statements in the else block will be executed. the else block is not required.

Print statement

```
print("x")  
print(null)  
print(1)
```

print statement is a catscript function that prints values to the console and it only takes one argument. everything that prints in the print statement will end in a new line character.

Variable statement

```
var s = "a"  
var numb = 1
```

variable statements are used to declare variables. variables can be assigned to any data types for example you can have a variables assigned to a number also you can have a variable assigned to a string.

Return statement

```
function f(a: string): string{  
    return a  
}  
function ff(b: string){  
    if(b=="abc"){  
        return  
    }  
    print(b)  
}
```

all functions that have a non-void return type require a return statement for all control paths. void functions can optionally early returns.

Functions

```
function f(a: string): string{  
    return a  
}  
function ff(b: string){  
    print(b)  
}  
// to call a function  
f("abc")
```

all functions require names, to declare a function you must start with a function keyword followed by a functions name and ending with zero or more parameters enclosed in parenthesised. the function body must be enclosed in curley bracers and can have any number of statements. functions without a return type are assumed to be void. to declare a return type add : type before the opening curly brace to specify the return type.

Operators

Additive expressions

```
1+1 // evaluates to 2
2-1 // evaluates to 1
2+"B" // evaluates to 2B
"A"+"B" // evaluates to AB
"A"+null // evaluates to null
```

the additive expression uses the + or - operator. you can concatenate string with the + operator

Comparison expression

```
4>2 // evaluates to true
4<2 // evaluates to false
4 >= 2 // evaluates to true
4 <= 2 // evaluates to false
```

comparison expression compares to integers and returns a boolean value.

Equality expressions

```
4==4 // evaluates to true
4!=4 // evaluates to false
4 == true // evaluates to false
4 == null // evaluates to false
```

the equality expression returns boolean value, the compared values can be any data type, but any values of different data types will not be equal.

Factor expressions

```
4/2 // evaluates to 2
2*2 // evaluates to 4
```

the factor expression require both operands to be integers. they perform numerical division or multiplication

Assignment operator

```
var a = true
a = false // valid assignment
a= 4 // not a valid assignment
```

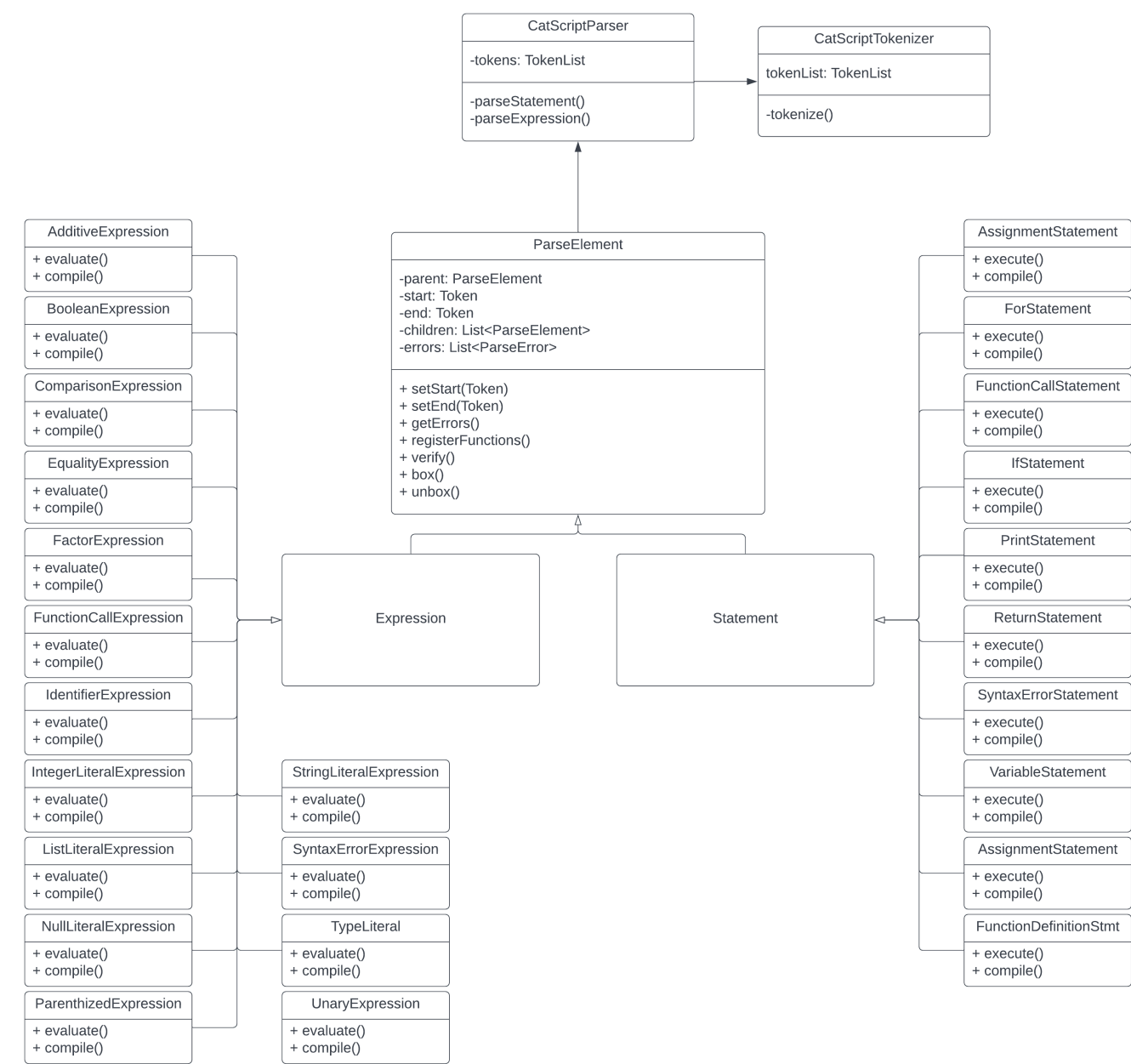
the assignment operator stores a variables value. only values of the variables type can be assigned to it. null be can be assigned to variables at any type. variables of object type can be assigned to any value.

Unary operators

```
-4
not true
```

there two types of unary operators which is minus and not. minus makes numerical values negative, and not makes a boolean value its opposite.

Section 5: UML



Section 6: Design trade-offs

Recursive descent vs. a parser generator

A recursive descent approach is easier to implement and understand. This is beneficial as a learning tool to be able to understand the recursive nature of language grammars more effectively. The downside here is that the recursive descent approach is more verbose and requires more code to implement.

On the other hand, a parser generator is more flexible and can be implemented in a more efficient manner, however it's less obvious to understand as a learning tool. Parser generators are also a more standardized approach to parsing languages.

On this project, we chose to implement a recursive descent approach. This was the most straightforward and easy to implement, and felt it was the most logical choice considering the idea is to deeply learn rather than perhaps making the absolute most efficient compiler.

Not using a visitor pattern

In the statements and expression classes, we have a lot of methods that are repeated. Most notably, `execute()`, `compile()` and `evaluate()` (which can be seen in section 5). Normally, a visitor pattern would be used to implement these methods such that each class doesn't need to repeat the same code. However, it was chosen not use a pattern in this project for simplicity's sake. Everything is in the same place, which makes it easier to understand and implement additional functionality. The downside is that the separation of concerns is non-existent, which can potentially add undue complexity to the code.

Section 7: Software development life cycle model

For this project, we used a test-driven development model. TDD is a process that involves writing unit tests before writing code, such that code you write can either "pass" or "fail" the tests so you know if your code is working as intended.

At the end of the project, there were a total of 160 tests, and we were able to ensure that all of the tests passed. It's worth noting that the majority of tests (all but 3) were written *for* the team and not *by* the team.

This is by-far my favorite development model to use. It gives a clear structure and path to the project, and allows for a more efficient development process. It has an added benefit of showing you roughly how far you are in overall development, and how much work you have left to do, all while getting immediate feedback on your code.

The downside to TDD is that it requires more time to develop and more time to test due to the nature of needing to not only write the tests, but think about all of the different cases you'll need to cover – typically all before you even start writing code. Also, the tests themselves are only as good as their writer, so if you write tests that don't cover as many cases as they should, you can have a false sense of the quality/completeness of the overall codebase. Neither of these downsides are worth abandoning TDD, but it is a good idea to keep them in mind.