

Montana State University
Gianforte School of Computing
CSCI 468 Compilers Capstone Portfolio

Noah Cunningham

Thor Lancaster

Spring Semester 2022

Section 1: Program

Source Code is located at [capstone/portfolio/source.zip](#)

Section 2: Teamwork

Our Group was comprised of two Team Members. Team Member 1 refers to me(Noah Cunningham) and Team Member 2 refers to my partner(Thor Lancaster).

Team Member 1 was responsible for writing all the code to pass the tokenizer, parser, evaluation and bytecode tests.

Team Member 2 was responsible for writing 3 additional tests for the Compiler to pass and The Technical Documentation.

Tests Written By Partner (You may find these tests along with the ones I wrote for my partner in `src/tests/partnertests`)

```
@Test
void myCustomTestsWorkProperly() {
    // Practical Recursion
    assertEquals("1\n2\n6\n24\n", executeProgram("function factorial(x:int):int{\n" +
        "    if(x <= 1){\n" +
        "        return 1\n" +
        "    }\n" +
        "    return x * factorial(x - 1)\n" +
        "}\n" +
        "\n" +
        "var numbers = [1, 2, 3, 4]\n" +
        "\n" +
        "for(i in numbers){\n" +
        "    var f = factorial(i)\n" +
        "    print(f)\n" +
        "}")

    // If inside for loop test
    assertEquals("1\n2\n", executeProgram("var nums = [1, 2, 3]\n" +
        "for(i in nums){\n" +
        "    if(i <= 2){\n" +
        "        print(i)\n" +
        "    }\n" +
        "}")

    // For inside if test
    assertEquals("1\n2\n3\n", executeProgram("var nums = [1, 2, 3]\n" +
        "var test = 2\n" +
        "if(test > 1){\n" +
        "    for(i in nums){\n" +
        "        print(i)\n" +
        "    }\n" +
        "}")
}
```

We Communicated through Discord and met in Person to coordinate and discuss the requirements.

Team Member 1 contributed 120-150 hours to make the compiler while Team Member contributed 2-4 hours to write the tests and documentation.

Section 3: Design pattern

The pattern I used was Memoization/Flywheel. The code is located in `src/main/java/edu.montana.csci.csci468/parser/CatScriptType` on line 36.

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    ListType listType = cache.get(type);
    if(listType == null){
        listType = new ListType(type);
        cache.put(type, listType);
    }
    return new ListType(type);
}
```

In computing, memoization is applied when dealing with expensive recursive function calls and helps to optimize this process. This technique stores the results of expensive function calls and caches the result. So the next time the function is called you can just look it up in the cache instead of repeating the function call.

The benefits of doing this are that it will save time for costly recursive computations. Also If we didn't do this there potentially could be some garbage collection errors depending on the language. Additionally memoizing could help runtime, but in exchange we had to give up some memory space.

Section 4: Technical writing

Introduction

CatScript is a simple programming language designed to solve general-purpose programming problems. The syntax and semantics are optimized for the ease of writing a recursive-descent compiler for it.

Starting CatScript

- Run the `CatScriptServer` class in the "edu.montana.csci.csci468 directory"
- Navigate to `http://localhost:6789`
- Enter code in the program area
- Click the buttons below the program area to perform operations on the code

Features

Variables

Lists

- Specifies a list of numbers. Commonly used in iteration.
- Example: `var numbers = [1, 2, 3, 4]`

Strings

- Specifies a string of text to print. Commonly used in printing.
- Example: `var message = "Hello, World"`

Numbers

- CatScript uses Integer arithmetic and not floating-point.
- Example: `var num = 5`

Booleans

- Specifies a value that may be either `true` or `false`
- Example: `var flag = false`

The null type

- Used internally to represent nothing.

Comparison Operators

Comparison operators are used to compare variables to check if they are equal, not equal, less than or equal to, etc. There are 6 different comparison operators in CatScript. Examples:

```
var a = 1
var b = 2

print((a == b) + "<br/>") // Outputs false
print((a != b) + "<br/>") // Outputs true
print((a < b) + "<br/>") // Outputs true
print((a > b) + "<br/>") // Outputs false
print((a <= b) + "<br/>") // Outputs true
print((a >= b) + "<br/>") // Outputs false
```

For loops

for loops are used to iterate a variable over the contents of a list. The index variable takes on each of the values of the list, in turn, for each iteration of the loop. For example, the following code can be used to manually print a list

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

If Statements

'if' statements take a boolean Expression and execute code based on the result of evaluating that expression. For example, the following code:

```
var daysLeftInSemester = 2

if(daysLeftInSemester <= 0){
    print("Time to party!!!")
} else{
    print("Keep grinding on schoolwork...")
}
```

prints out the text `Keep grinding on schoolwork...` because 2 is greater than 0.

Functions

Functions are supported in CatScript and can be used to reuse blocks of program logic and reduce code complexity. They can take any number of arguments and return one value.

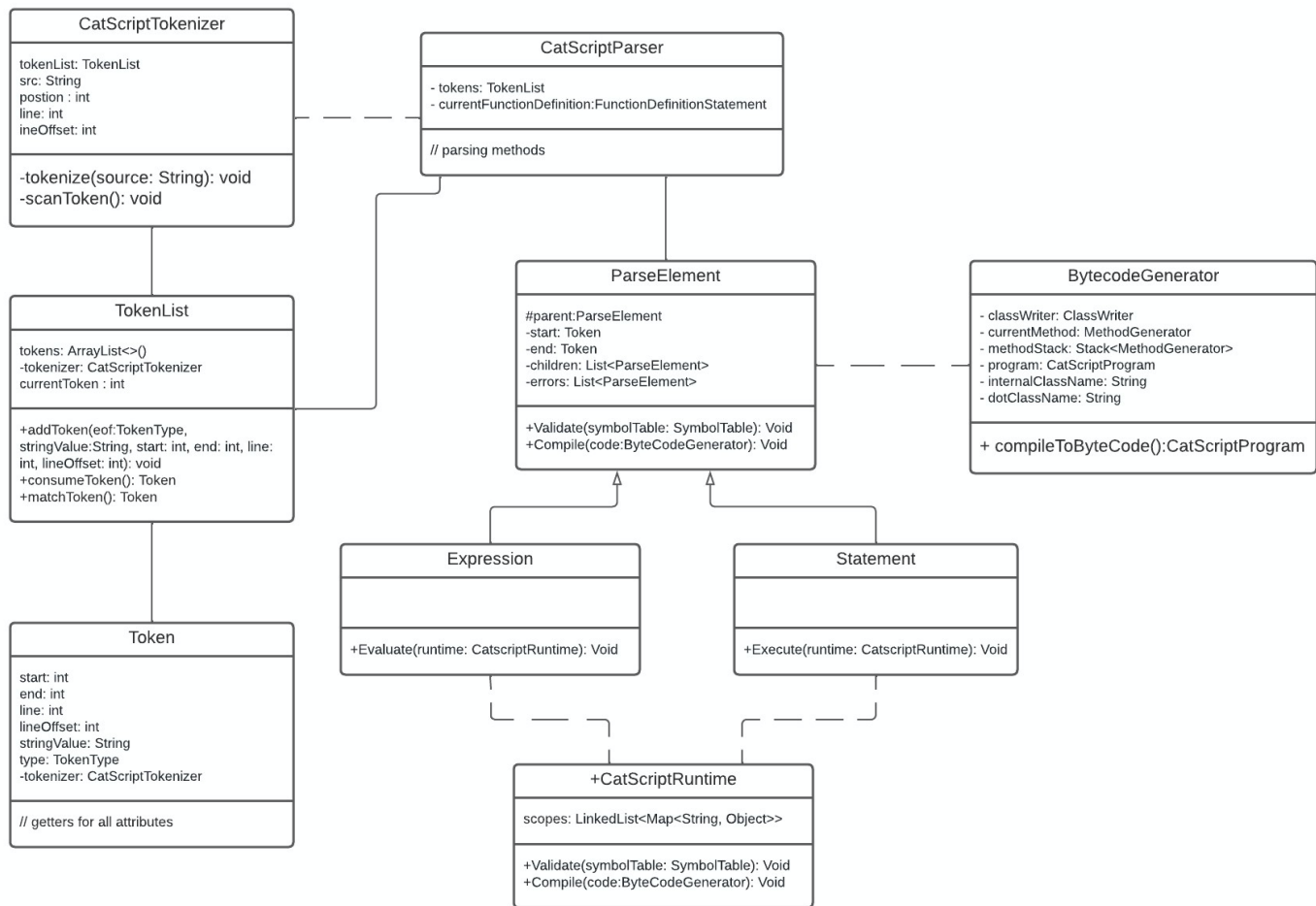
The following code used functions and a for() loop to compute factorials. The example also makes use of recursion.

```
function factorial(x:int):int{
    if(x <= 1){
        return 1
    }
    return x * factorial(x - 1)
}

var numbers = [1, 2, 3, 4]

for(i in numbers){
    var f = factorial(i)
    print("The factorial of " + i)
    print(" is " + f + "<br/>")
}
```

Section 5: UML.



Section 6: Design trade-offs

Decided to use recursive decent parsing(which is a top down parsing algorithm) instead of a parser generator. We chose to do this because recursive decent parsing is much simpler and easier to understand than a parser generator. Also most production parsers are recursive descent, so it will give us a better understanding of how real world compiler work. However recursive decent parsing required me to write much more code than a parser generator.

Parse Tree Nodes Evaluate and Compile directly, Instead of using the visitor pattern or some other way to do that. This is a simpler way to do this because we are not 'separating our concerns'. Everything is in one place, so if errors occur it will be easier to diagnose the problem.

Created a method to check the next token, and used it in `parseAssignmentStatement()` and `parseIfStatement()`. This wasn't necessary but It made it simpler to check the next token without consuming it, if i didn't add this function I would have had to write more code in `parseAssignmentStatement()` and `parseIfStatement()` to get around that problem.

Section 7: Software development life cycle model

We are using Test Driven Development (TDD) for this project. TDD is a software development process which relies on software requirements being converted into test cases before the software is fully developed. This allows programmers to track their progress by testing the software against the test cases.

The positives are that it's extremely straight forward, and you know exactly what to do. I personally have loved using TDD in this class and would like to implement in my professional career if I can. The only negative is that you have to come up with good tests that capture the scope of your project, which can be challenging and time consuming. Me and my partner were able to come up with additional Tests, and it was fun to not only come up with some tests of our own, but also to rewrite some of our program in order to pass them. I would recommend all CS classes be taught with some TDD implemented.