

CSCI 468 – Compilers Spring 2023 Capstone Portfolio

Jacob Rivers

Kate Stallbaumer

May 5, 2023

CS 468 Capstone for Jacob Rivers

Section 1: Programs

Over the course of this semester, we created a programming language called Catscript as a way to explore the underlying software which implements a programming language as well as the structure and design of compilers.

Catscript is a simple scripting language which we designed, implemented, validated, and documented. We ended the semester by writing a compiler for the Catscript language. Our compiler takes in Catscript programs and input and compiles them to Java bytecode. These bytecode programs can then be run on the Java Virtual Machine (JVM).

The file included in this directory named source.zip contains the full programs for the Catscript language implementation, Catscript Server, Catscript testing and validation suite, and the Catscript compiler.

source.zip can also be found in my GitHub repository at the link below:

<https://github.com/SejongKadonk/cs468-spring2023/blob/main/capstone/portfolio/source.zip>

Section 2: Teamwork

There were two major parts of creating the Catscript programs where we collaborated as a team.

- Creation of thorough Technical Documentation of the Catscript Language
- Creation of new and unique tests for the Catscript Testing Suite beyond those provided by Professor Gross.

Each team member wrote a technical document which gave an outline of the major features of the Catscript language. In that technical document they provided brief descriptions of each feature along with examples.

The team members then exchanged the document they personally wrote with their partner. So, Team Member 1 (Jacob Rivers) wrote a technical document for Team Member 2 (Kate Stallbaumer) to include in their capstone portfolio and vice-versa. Section 4 of this document contains the documentation written by Kate Stallbaumer.

In addition to writing technical documentation each team member contributed extra tests to the Catscript testing suite and then exchanged those tests with the other partner.

Team Member 1 wrote three tests which checked that the following features worked properly:

- identifier reassignment
- true and false branches of if statements
- for statements embedded in functions

Team Member 2 wrote three tests which checked that the following features worked properly:

- variable assignment
- scoping of functions
- if statements embedded in functions

Both team members shared the work of creating tests and documentation equally, with each member devoting roughly equal time to each task.

Section 3: Design Pattern

Memoization is a useful design pattern for optimizing code quickly and without making major structural changes to the class structure of a OOP program.

Memoizing a function call is a way of making a function the result of something that can be remembered by the program. The main idea behind it is to execute a function only once. Follow up calls should not run the logic but rather just look up and return the cached result.

In our CatscriptType class we memoized type access for list types. Each time we construct a new list object we store the list's component type in a hashmap. So rather than reinitializing the list types for list objects every time the getListType() method is called it will first look up the listType in the hashmap and return anything that it finds.

Because the Catscript type system is static we don't need to invalidate the cache after we are done with it. If we new up a list of integers a thousand times, we only need to look up the integer listType once for each of those constructors but never need to erase or overwrite the cache.

While this particular implementation of memoization was an instance that is relatively cheap in a computational sense, we implemented it as a learning exercise. It was a good opportunity to practice using the memoization design pattern and also to gain a better sense of the parts of a compiler (or other programs we write in general) where optimization and caching patterns such as memoization would be useful for reducing the amount of redundant computation in a program.

Section 4: Technical Writing

1. A Brief Guide to CatScript

1.1 Introduction

Catscript is a simple scripting language that is comparable to languages like Java and Python. The Catscript Compiler program is a program that takes in, processes, and executes code formatted in the Catscript language. The execution of that code is based on its Java interpretation and the compiler translates between them. An example of the Catscript language can be seen here with more examples shown in the features section:

```
var x = "Hello World"  
print(x)
```

1.2 Features

1.2.1 Statements

For loops

For loops in Catscript are able to iterate over defined lists and perform operations per iteration. Operations are able to contain more types of statements such as print- statements and if-statements statements with no issue.

Example:

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

Output:

```
1  
2  
3
```

If Statements

If statements in Catscript are able to have conditional statements that restrict access to a portion of code unless the conditional statement is met.

Example with conditional met:

```
if (1 < 2) {  
    print("One is less than two")  
}
```

Example with conditional not met:

```
if (1 > 2) {  
    print("One is greater than two")  
}
```

Print Statements

Print statements in Catscript are able to take and print its contents such as String or Integer Literals.

Example:

```
print("Hello World")
```

Output:

Hello World

Variable Statements

Variable Statements in Catscript are able to assign values to variable names. Any of the primary expressions are assignable to variable

names. Types can be explicitly or implicitly defined when creating the variables.

Examples:

```
var x = 1
```

Or:

```
var x : int = 1
```

Assignment Statements

Assignment statements in Catscript are used when a variable is being assigned to a different value. Variables can only be re-assigned to the same type that they were implicitly or explicitly defined as, an Integer variable cannot be assigned to a String value.

Example:

```
var x = 1
```

```
x = 3
```

Function Definition Statements

Function Definition Statements in Catscript are statements that define a function that is callable elsewhere in the program. The Function can be defined with parameters and must be defined with a body. The parameters can be defined explicitly or implicitly just like variable statements. The body can contain any number and types of statements. Functions are also able to contain return statements which should end the function.

Functions with return statements should have a return type defined after the name and parameters of the function.

Examples:

```
foo(a, b, c) {
```

```
print(a)
print(b)
print(c)
}
Or:
foo(a: string, b: string, c: string) {
    print(a + b + c)
}
Or:
foo(a: int, b: int, c: int) : int {
    return a * b * c
}
```

Return Statements

Return Statements in Catscript are statements that pass values from functions to where they were called in the greater scope. Return statements will end the function they are contained within. Return statements can have expressions contained within them. Examples:

```
return x
```

Or:

```
return 1+1
```

1.2.2 Expressions

Primary Expressions

There are eight types of expressions defined under primary expressions: Identifier, Integer Literal, String Literal, Boolean Literal, List Literal, Null

Literal, Function Call, and Parenthesized Expressions. Below, you will find short descriptions and examples of each.

Identifier Expressions are expressions that represent a keyword defined by the user:

x

y

z

Integer Literal Expressions are expressions that represent integer numbers:

42

144

String Literal Expressions are expressions that represent strings of characters:

"Hello World"

"I am alive"

Boolean Literal Expressions are expressions that represent the True and False symbols:

True

False

List Literal Expressions are expressions that represent a set of Integer, String, Boolean, and List Literal Expressions:

[1, 2, 3]

["Hello", "World"]

Null Literal Expressions are expressions that represent the null symbol. Null Literal Expressions are used when there is no value represented for a variable:

null

Function Call Expressions are expressions that contain information about what information to send to an existing functions parameters. Function Call Expressions are used to signal the execution of a function during runtime:

foo(1, 2, 3)

Parenthesized Expressions are expressions that contain any type of expression inside two parentheses. The parentheses do not affect the contained expressions in any way:

("Hello" + "World")

(12 < 24)

Unary Expressions

Unary Expressions are expressions that are applied to only one expression. The two symbols that a unary expression can have, are the negative symbol and not symbol which can only be applied to Integer Literals and Boolean Literals respectively.

Examples:

-1

not True

Equality Expressions

Equality Expressions are expressions that have a double equal or bang equal symbol separating two expressions, with the double equal symbol asserting both sides are the same and the bang equal symbol asserting both sides are different. The separated expressions may be any type of expression.

Examples:

`true == true`

`true != false`

Comparison Expressions

Comparison Expressions are expressions that have a less than, greater than, less than or equal to, or greater than or equal to symbol separating two expressions. The separated expressions may only be Integer Literals.

Examples:

`1 < 2`

`2 > 1`

`x <= y`

`y >= x`

Additive Expressions

Additive Expressions are expressions that have an addition or subtraction symbol, a plus or a minus respectively, separating two expressions. The separated expressions may be Integer Literals, String Literals, or Parenthesized expressions containing either Integer or String Literals. String Literals can only be added together and not subtracted.

Examples:

"a" + "b"

2 - 1

Factor Expressions

Factor Expressions are expressions that have a multiplication or division symbol, an asterisk or a slash respectively, separating two expressions. The separated expressions may only be Integer Literals or Parenthesized Expressions containing Integer Literals.

Examples:

5 * 6

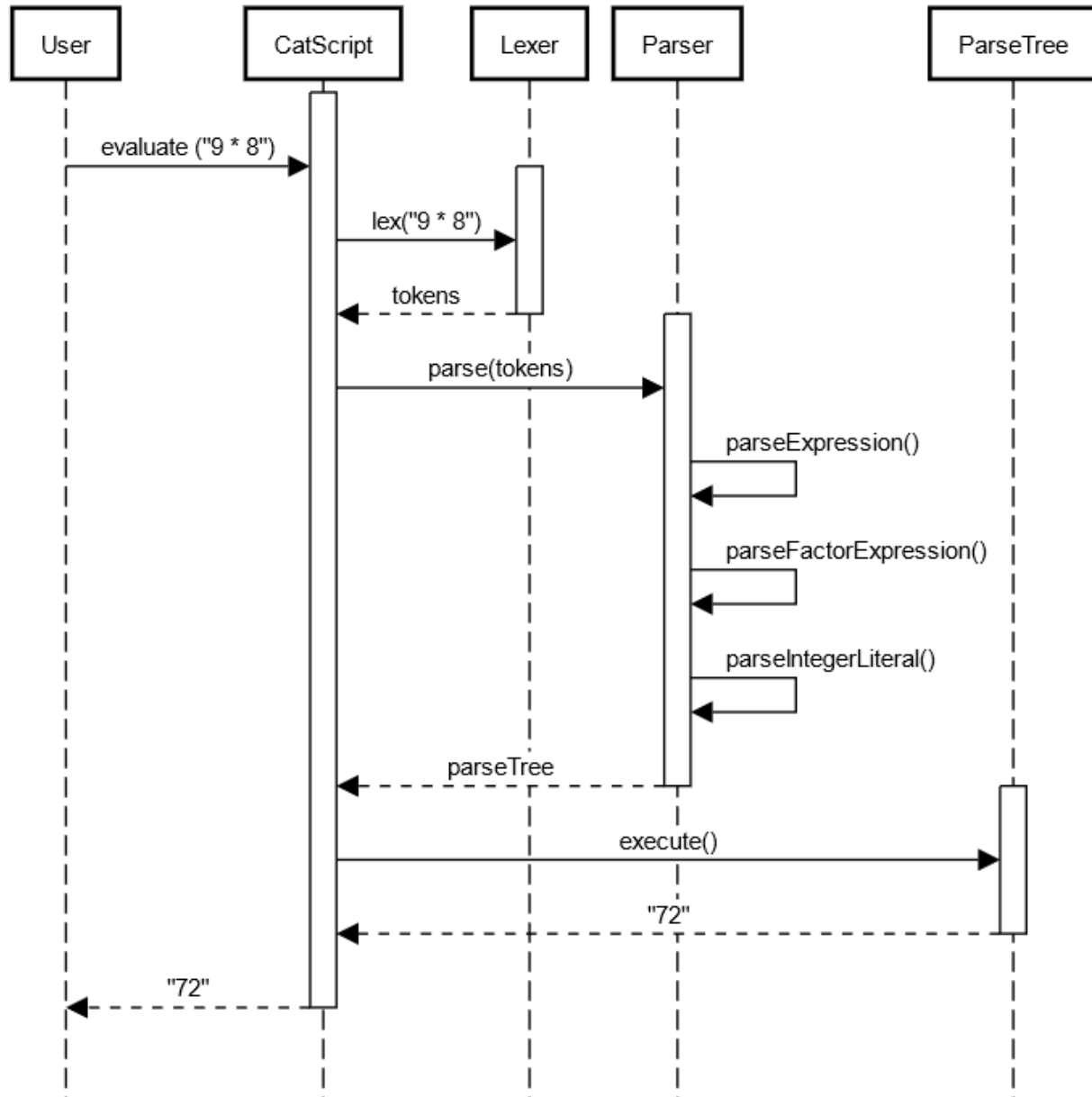
30 / 5

1.3 Conclusion

This brief overview of the major features of the Catscript language shows some concrete examples. For a formal generative grammar see the bottom of the page at the following link: <https://github.com/SejongKadonk/cs468-spring2023>

Section 5: UML

Catscript Factor Expression Sequence Diagram



The above UML diagram shows a typical sequence of operations to evaluate a factor expression in a Catscript program, covering each step from user input to program output.

To begin, a user inputs the expression “9 * 8” into the Catscript server and clicks the “evaluate” button.

From there the Catscript program calls the `lex()` method on the input 9 * 8. This method call is passed to the Lexer which breaks the string into tokens, namely the tokens 9, *, 8.

The tokens are then returned to the top level of the Catscript program which then begins parsing. A `parse()` method call with the returned tokens as input is passed to the Parser.

The Parser works down the recursive descent hierarchy for the expression calling each successive level and returning the values as nodes on a parse tree object.

After parsing is complete and a full parse tree has been assembled for the Catscript program the `execute()` method is called on the Parse Tree Object. This method returns a value of “72”.

Finally, the fully evaluated program output is displayed to the user.

It is interesting to note that this UML diagram can be generalized to essentially any Catscript program with only slight modifications being made. The Parser step would have to descend the recursive descent hierarchy as many times as necessary to fully parse each expression and statement in the program. Beyond this change, the basic structure and flow of the sequence diagram would remain the same, starting with input, proceeding to lexing, parsing, execution, and ending with output.

Noting this deep coherence in the structured sequence needed for the evaluation of any Catscript program makes sense. Executing a program requires the same series of algorithms and processes be executed, largely without concern for the gritty details of the specific program itself.

Section 6: Design trade-offs

Early in the creation of Catscript we had to decide how to implement the Catscript parser. There were two possibilities available: we could implement the recursive descent algorithm ourselves or we could use a parser generator.

We decided that the better option for the Catscript parser was to create our own hand-made implementation of the recursive descent algorithm. There were several reasons for his decision.

First, most compilers have their own native implementations of recursive descent. Therefore, gaining experience with what is the industry standard was a point in its favor. Second, by implementing the algorithm ourselves we would gain direct experience with the details of the algorithm and gain a practical understanding of how it works at the level of code. Finally, at a more conceptual level, we would also gain a more concrete understanding of the recursive nature of grammars. This conceptual understanding is likely to serve us in the future and remain clear in our minds much longer than if we had simply used an API that hid the details of the algorithm from us. Using a parser generator would have not allowed us to gain as deep and intuitive understanding of the grammar.

There were however still some trade-offs in using our own recursive descent implementation. Using a parser generator would have given us more experience with regular expressions and Extended Backus-Naur Form as both are necessary to create input for most parser generators. We briefly experimented with Regex and EBNF earlier in the semester and gained a basic fluency in them. More experience with these would also have been useful. But, while parser generators would have provided more practice in these areas and could have been less work, we felt that it was not worth it if it would rob us of the opportunity to gain a deeper understanding of how formal language grammars work and their inherently recursive nature.

Ultimately, I feel that the decision to implement our own recursive descent algorithm was the right one, despite being less convenient. I finished the course with a confidence that I could alter and refine our implementation to other coding languages, as well as feeling at ease with the ideas behind it.

Section 7: Software development life cycle model

The software development model used to develop Catscript was a Test-Driven Development model (TDD). In TDD, developers write automated tests that capture the intended behavior of the code, and then write code to pass those tests. This approach helps ensure that the code is thoroughly tested and meets the desired specifications.

In short, TDD is a development model which aims to specify and validate that a program's implemented features function correctly and are bug free before release.

The majority of the Catscript testing suite was written by Professor Gross and it was our job to code, refactor, and run tests until we had created a build of our program that passed the entire testing suite.

In addition to Professor Gross' tests, which were used by the entire class, each member of our team also developed an additional set of tests to cover a few of the corner cases and other aspects of the Catscript language that were not fully tested.

I truly enjoyed learning about compilers through a TDD approach and I felt that it really helped me develop a solid understanding of the concepts and ideas involved in the course as well as simply giving me a chance to improve my skills as a programmer.

Testing provides a very concrete pass/fail goal to work toward when writing code. This helped me to break down the very large and at times nearly overwhelming task of creating a programming language and functioning compiler, and to divide problems into more manageable tasks. TDD also acts as an extremely useful compliment to the debugger in the IntelliJ IDE. Over the course of this semester, I became much more capable at navigating the debugger, using it to understand where code was going wrong, and understanding the state of my code at various points in execution.

TDD is also a common industry standard for software development and using it in this course makes me feel as though I have gained practical experience which will be transferable to working as a junior developer in the future.