

Capstone Documentation

Ethan Skelton

Partner – Caitlynn Koback

Note: I am in 482 and 483 for my actual capstone not sure if that matters, but just a heads up. I was told to continue with this documentation either way.

Section 1: Program

Link – [sourceCode](#)

Section 2: Teamwork

Me as partner 1 and my partner as partner 2. Partner 1 was the technical engineer and focused on making a functional catscript language while Partner 2 worked on testing and documentation of all the work that partner 1 did. Partner 2 gave me the testing statement so that I could make sure that my code was working properly.



```
no usages  ⬆ Ethan10Real
@Test
void varAssignsListImplicitly(){
    VariableStatement expr = parseStatement( source: "var x = [1, 2, 3]");
    assertNotNull(expr);
    assertEquals( expected: "x", expr.getVariableName());
    assertTrue(expr.getExpression() instanceof ListLiteralExpression);
}

no usages  ⬆ Ethan10Real
@Test
void testLongMathEquation() { assertEquals( expected: "12\n", compile( src: "7+2*3+3-8/2")); }

no usages  ⬆ Ethan10Real
@Test
void parenthesisMathCompiles() { assertEquals( expected: "12\n", compile( src: "(7+2*3+3-8/2)")); }
}
```

Section 3: Design Pattern

The memoization pattern was used in the construction of this project in order to make sure that the program never gets slowed down by storing results that the program gets. This pattern can be seen in the file 'CatscriptType.java' where we can see that if a CatscriptType has been initialized already then, the program will see that and not go through the effort of making it again.

Memoization is a design pattern in computer science that involves caching the results of function calls so that they can be reused when the same inputs are used in the future. This can significantly improve the performance of a program, especially when the function being called is computationally expensive or time-consuming. By storing the results of previous function calls, the program can avoid redundant calculations and instead return the cached result immediately. This not only saves time but also reduces the workload on the system and can improve the overall efficiency of the program. Memoization is a powerful design pattern that can be applied in a wide range of programming languages and environments, and it is especially useful in situations where the same function is called repeatedly with the same inputs. By implementing memoization, developers can create more efficient and effective programs that are better suited to the needs of their users.

```
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();
5 usages  ▲ Carson Gross *
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

Catscript Guide

Introduction

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

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

Features

Typing

int - 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

Additive Expression

$x + y$ $x - y$ #### Description

Adds or subtracts two expressions from each other. If used to add a string to another value, it concatenates.

Parameters

x - expression
y - expression

Returned

This function returns either a sum, difference, or a new string.

Assignment Statement

`x = e####` Description
Assigns an expression to an Identifier

Parameters

`x` - Identifier
`e` - expression

Comparison Expression

`e > x e >= x e < x e <= x####` Description
Compares two expressions in order to assess if one is greater,
greater or equal,
lesser,
or lesser or equal.

Parameters

`e` - expression
`x` - expression

Returned

The expression returns a boolean

Equality Expression

`e == x e != x####` Description
An equality expression compares two expressions to determine if they are equal or not equal

Parameters

`e` - expression
`x` - expression

Returned

The expression returns a boolean

Factor Expression

$x * y$ x / y #### Description
Multiplies or divides two expressions

Parameters

x - int
 y - int

Returned

This expression returns a product or quotient.

for loops

for (identifier x in expression e){}#### Description
Iterates through elements of the expression

Parameters

x - identifier
 e - expression

Function Call

func() func(Object x , Object y)#### Description
Can take any number of comma separated values and send it to the function called

Parameters

func - identifier name of the function called

object x - an object sent as a parameter to the function

Function Declaration

function x() {} function x() : type {} function x(a, b, c) : type {}#### Description
Declares a function with any amount of comma separated arguments

Parameters

x - Identifier name

a, b, c - expressions acting as arguments for the Function, any typing
type

Returned

The function may return nothing or an object of the explicitly declared type

if statement

if (expression e){} if (expression e | expression x){}#### Description
Assesses expression that returns a boolean, if true it executes the statements within the body

Parameters

e - expression

x - expression

Print Statement

print(expression e)#### Description
Prints out the expression to the console

Parameters

e - expression

Return Statement

return e ##### Description

Used within functions

May return nothing or an expression with the function's declared typing

Parameters

e - expression

Unary Expression

not e -e ##### Description

Processes the opposite of the expression

Parameters

e - expression

Returns

A unary statement an opposite of the expression, i.e not True is False or -1.

Variable Statement

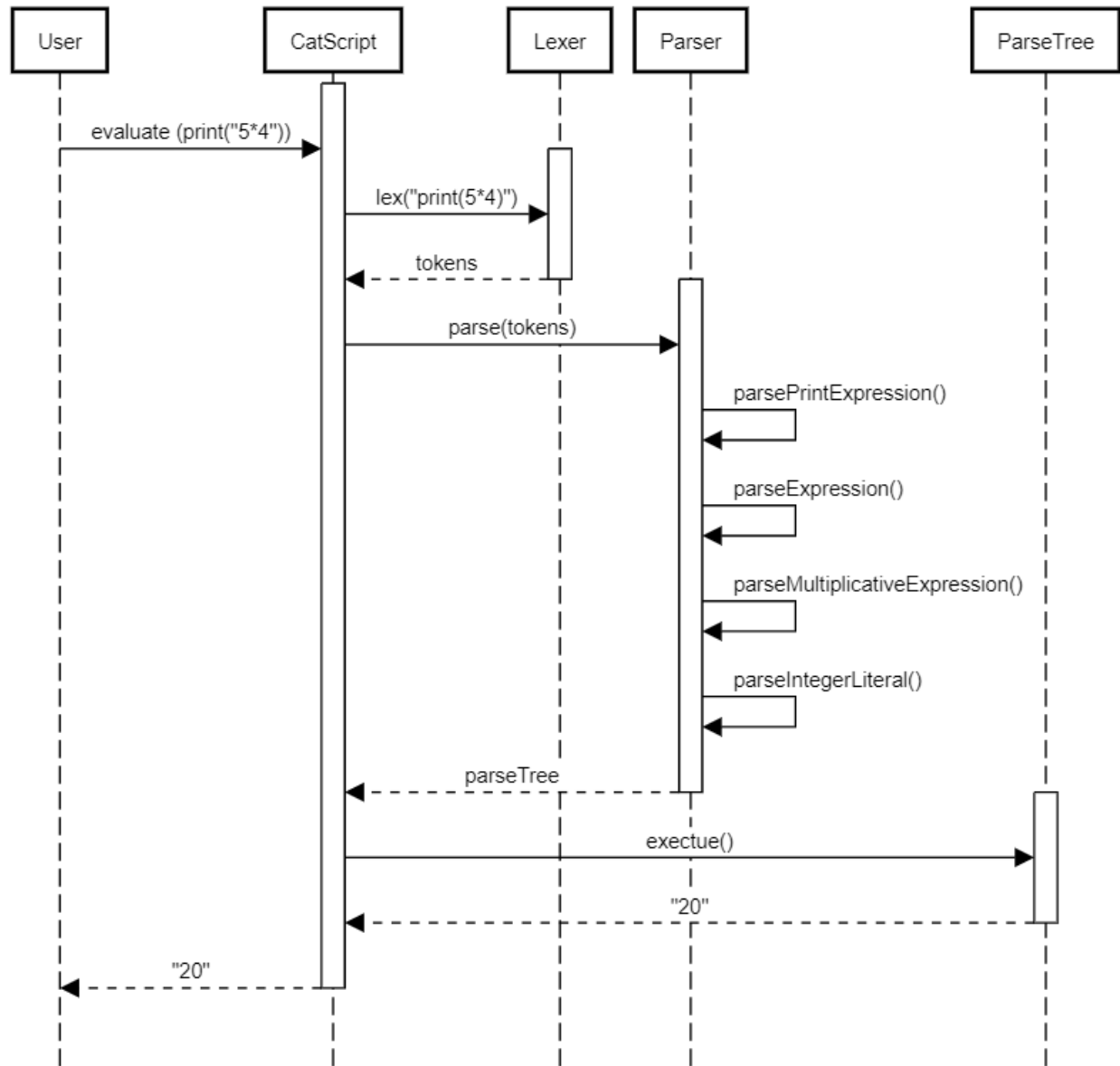
var x = expression e var x : type = expression e ##### Parameters

x - an Identifier string that acts as the name

e - an expression

type

Catscript Sequence Diagram



Section 6: Design trade-offs

We chose to create a by hand parser instead of using one that was already created for us. This caused for a large time loss that we could have used working on other parts of the project, but we felt that it was an important step that we had to get right so we chose to do it by hand. It was also more intuitive to use a recursive decent style of parser and we would learn a lot in the process of doing the parser by hand.

Section 7: Software life cycle

We used a Test Driven Development life cycle. Our life cycle was designed around tests that we were trying get passing in a certain amount of time. I believe that this was a benefit to our team as we could have some direction about where we were going before starting programming. It likely allowed us to work very fast while making sure that our project was meeting requirements that we had set. The benefits of using TDD include increased confidence in code quality, better documentation of the code, and faster development cycles. By writing tests first, developers are forced to think critically about the functionality of the code they are writing, resulting in more robust and reliable software. The automated tests also serve as a form of documentation, providing clear examples of how the code should be used and what it is expected to do. Finally, the TDD process can actually speed up development cycles by reducing the need for manual testing and debugging. While the TDD lifecycle was challenging at first, with practice it allowed us to work extremely fast in completing this assignment. By following a consistent process that emphasizes testing and quality assurance, developers can produce more reliable and maintainable software that is better suited to the needs of their users. Whether working on a small project or a large-scale application, TDD can help developers stay organized, efficient, and focused on delivering high-quality code.