

CSCI Capstone Portfolio

Partner 1 - (Hayden Xavier)

Partner 2 - (Cole Orelup)

- Section 1: Program -

- The zip file containing the source code for the project is named (located at) `capstone/portfolio/source.zip`

- Section 2: Teamwork -

- Partner 1 (Hayden Xavier) was responsible for implementing the tokenizer, parser, eval, and compilation for the Catscript programming language. Partner 2 (Cole Orelup) was responsible for contributing 3 tests to analyze the functionality of partner 1's implementation of Catscript features. Both partners were successful in completing the work necessary to implement their Catscript programming language and test the functionality of the other's. Additionally, partners exchanged documentation documents for section 4 of this capstone portfolio.

Partner Number	Rating of Contributions on a Scale of 1-10
1	9
2	9

- Section 3: Design pattern –

- The design pattern that was implemented for Catscript was the memoization pattern. The memorization pattern involves saving the solutions of commonly dealt-with problems into a list, array, etc. such that if that problem is faced again, the runtime of retrieving the answer to that problem is minimized because it can be simply retrieved from the data structure of saved solutions. In the case of this project, the memorization pattern was implemented on the static getListType() method within CatscriptType.java so that instead of returning a new list type every time, applicable list types that have already been assigned can be retrieved from a HashMap of saved list types. Below is a screenshot of the implementation.

```
|  
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. Include the technical document that accompanied your capstone project.

Introduction

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

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

Features

For loops

For loops allow for repeated execution of the body block (statements between { and }) for a set number of iterations. Abstractly, this is defined as:

```
for (IDENTIFIER in EXPRESSION) {  
    // iterate over statements  
}
```

As a practical example:

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

OUTPUTS: 1, 2, 3

If/Else Statements

If/Else statements are the main logical control flow for Catscript. Depending on whether a condition is true, the if statement's body is executed or the else statement's body is executed. Abstractly, the If/Else statement looks like:

```
if (EXPRESSION) {  
    // execute if statement's body  
} else {  
    // execute else statement's body  
}
```

Practically an If/Else statement looks like this:

```
if (x == 1) {  
    return 1  
} else {  
    return 0  
}
```

Print statements

Print statements are a form of outputting a value or string to the console.

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

Variables

Variables are a way of storing values in Catscript. Since Catscript is statically typed, there must be an associated type with the variable. The type can be inferred using the syntax:

```
var x = "foo"
```

Alternatively, the type can be explicitly assigned using `: TYPE` where `TYPE` is an of

Catscript's types. `var x: string = "bar"`

Variables are also assignable from each other depending on what their types are. Here is an example of a variable `y` taking on the value from a variable `x`:

```
var x = 10
var y = 0
y = x
```

Functions

Functions enable code to be repeatably called without typing out its body every time you want to use it. They are denoted by `function` and are abstractly defined as:

```
function IDENTIFIER(FUNCTION_PARAMETERS) {
    // statements to be executed here
}
```

Like variables, functions also have implicit and explicit typing. A practical function with implicit type is shown by:

```
function multiply(x, y) {
    return x * y
}
```

Explicit typing is shown again by `: TYPE`

```
function divide(x, y): int {
    return x / y
}
```

Functions can be called by using its `IDENTIFIER` along with its `FUNCTION_PARAMETERS`. Using the `divide` function as an example, `divide` can be called like:

```
divide(10, 2)
```

Which should evaluate to the value 5

Functions also have `return` statements that return an expression value from a function.

Lists

Lists are a form of storing a series of data. They are denoted by any number of expressions wrapped in `[]` . Following the variable's typing, lists can be inferred or explicitly typed out.

Implicit Type

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

Explicit Type

```
var list: list<string> = ["hello", "there"]
```

Types

CatScript is statically typed, with a small type system as follows:

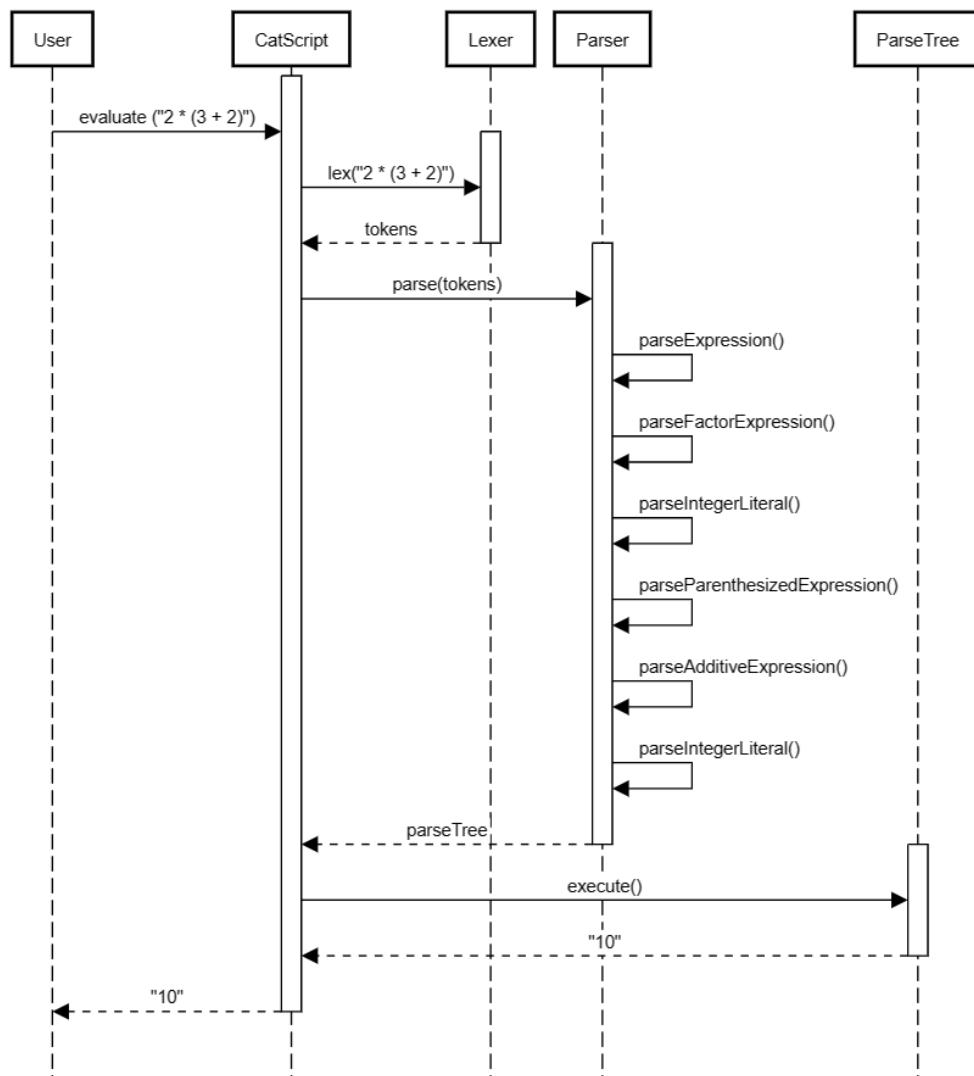
- int - a 32 bit integer
- string - a java-style string
- bool - a boolean value
- list - a list of value with the type 'x'
- null - the null type
- object - any type of value

Operators

- +
-
- *
- /
- ==
- !=
- <=
- >=
- >
- <
- not

- Section 5: UML –

Catscript Multiplication, Parenthization, Addition Sequence Diagram



- Section 6: Design trade-offs –

- The main design trade-offs when developing the Catscript programming language was that we decided to use the recursive descent model to create our parser instead of utilizing a parser generator. The main advantage of recursive descent parsing is its simplicity and ease of implementation. Recursive descent parsers can be easily hand-coded using a programming language's standard control structures such as if-else statements and loops. Additionally, recursive descent parsers can be efficient for parsing small grammars and simple languages, like we did for Catscript. However, recursive descent parsing can become inefficient and complex for larger or ambiguous grammars, and it may require significant manual effort to handle conflicts and errors in the grammar. Parser generators on the other

hand provide a more automated approach to parsing. Parser generators take a formal grammar specification and generate a parser automatically, which has several advantages, including the ability to handle complex and ambiguous grammars efficiently, and the ability to generate parsers for multiple programming languages. Additionally, parser generators often provide error recovery mechanisms and support for generating error messages. However, parser generators can be less flexible and customizable than hand-coded recursive descent parsers, and may require significant manual effort to integrate with other components of a software system.

- Section 7: Software development life cycle model –
 - For the development of the Catscript programming language, the development cycle used was one of test-driven development (TDD). (TDD) is a software development approach where tests are written before the code is written and has several benefits, including improved code quality, faster development cycles, better maintainability, and improved communication between developers and testers. By defining test cases up front, everyone involved in the development process has a clear understanding of what the software is supposed to do and how it should behave. However, TDD requires a new way of thinking about software development and can be time-consuming, especially for large and complex projects. Additionally, TDD may lead to an over-reliance on tests and may cause developers to ignore other important aspects of software development, such as design, architecture, and usability. In this case, using test driven development was extremely helpful as nearly every feature of Catscript could be mapped to a test to ensure full functionality.