

CAPSTONE PORTFOLIO

COMPILERS 468
SPRING 2022

DARSHAN DIAMOND
JAY FORBES
MARNIE MANNING

Section 1 – Source Listing:

Source Code

All project source code and files are contained within the included in a zip file.

<https://github.com/darshandiamond/csci-468-spring2022-private/blob/master/capstone/portfolio/capstoneportfoliosource.zip>.

Development Environment

Catscript was compiled using a Maven script with JDK 14 and developed within the IntelliJ Idea IDE ecosystem.

Section 2 – Teamwork:

This project was divided into four sections, a tokenizer, parsing, evaluation, and bytecode. The code creation for this was done by each individual person. The teamwork was from each partner two submit tests to validate each other's code. Along with submitting technical documentation for each other's portfolios. In this case, we had to work in a team of three because someone drop the class and failed to inform their team member about it. We each completed three tests and the technical documentation and exchanged it with one other person in the group.

Member 1: Darshan Diamond

Primary software developer, code management.

Estimated hours: 120 hours

Member 2: Jay Forbes

Technical documentation, software test.

Estimated Hours: 1-hour writing tests, 3 hours of technical documentation

Member 3: Marnie Manning

Technical documentation, software test.

Estimated Hours: 1-hour writing tests, 3 hours of technical documentation

Total Estimated Hours: 128 Hours

Section 3 – Design Pattern:

The Memoization pattern was the primary pattern used in the Catscript creation. This pattern was used to increase the efficiency of the “getListType” function and reduce the number of projects being created. Only one ListType is generated for each corresponding Catscript, this improves runtime speed by removing the need for redundant object creation, as well as lowering overall memory usage. To code, this function involves creating and storing a new ListType as well as performing unnecessary comparison operations to determine the proper ListType to create.

The Memoization Pattern can be found at CatScriptType.java located at ~\csci-468-spring2022-private\src\main\java\edu\montana\csci\csci468\parser\CatscriptType.java. This comprises the entire function, control flow, and static final HashMap used to store CatscriptTypes with their ListType equivalents.

```
static final 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 listType;
```

```
}
```

Section 4 – Technical Documentation

Introduction

Catscript is a statically typed scripting language that compiles to JVM bytecode featuring both evaluation and compilation. It takes features from Java and combines them with features from other languages in order to have an improved language to work with.

Features

Comments

Introduction

Catscript is a language very similar to Java. We implemented tokenization, parsing, evaluation, and bytecode using memoization to create a functioning programming language.

Features

Types

Basic data types that allow for arithmetic and data storage. These are:

String - A series of characters

Int – 32-bit number

Boolean - True or False value

Null - Data of type null

Object - Flexible data type

List - Data container

Print

This takes an input from the user and is returned to the output area of the console.

```
print("Hello World") //Will output "Hello World" to console.
```

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

```
//Will output empty string initialized under variable foo to console.
```

Print will concatenate two like variables or two unlike variables.

```
var foo1 = "A"
```

```
var foo2 = "B"
```

```
print(foo1+foo2) //Outputs "AB" to console
```

```
var foo1 = "A"
```

```
var foo2 = 1
```

```
print(foo1+foo2) //Outputs "A1" to console
```

Commenting Code

Commented code is not evaluated by the program. To generate a single line comment use: //

```
//This is a single line comment
```

For multi-line comments use: /* */

```
/*
```

```
This is a multiple
```

```
line comment
```

```
*/
```

Expressions for Evaluation

Equality Expression: Compares two terms, either with == or != and results in a boolean.

```
1 == 1 // Output: True
```

`1 != 2 // Output: True`

Unary: Negation of a term, can be done multiple times

`-1 // Output: -1`

`--1 // Output: 1`

Additive Expression: Adding or subtracting two terms. Terms do not have to be of the same data type.

`1 + 2 // Output: 3`

`1 - 2 // Output: -1` `3 + "Hello World" // Output: 3Hello World`

Factor Expression: Multiplication and division of two terms.

`1 * 2 // Output: 2`

`1 / 2 // Output: 1/2`

Parenthesized Expression

`2(1) // Output: 2`

Comparison Expression: Compares two terms, returns a boolean value. You can compare with greater than or equal to.

`1 > 2 // Output: false`

`1 >= 1 // Output: true`

Null Expression: tests to see if term is equal to null data type

`var testNum = 1;`

`testNum != null // Output: true`

Variable Declaration

Variables that are initialized and then manipulated throughout the program to hold the desired data values. You can initialize a variable of any data type. In fact, you must in order to use the variable in Catscript.

Implicit Declaration: Does not provide a data type during initialization

`var testVar = 1`

Explicit Declaration: Provides data type during initialization

`var testVar: Int = 1`

If Statements

If statements use a comparison to determine how to evaluate a certain attribute. You can use the keywords "if", "else", and "else if". "if" will execute the following code if the parenthesized expression is

true. "else if" will execute the following code if the previous if statement is not true and the following parenthesized expression is true. "else" will execute if the previous "if" and "else if" statements did not result in true.

```
var testOne = 1
var testTwo = 2
// This will result in the "else if" case evaluating to true and executing the print statement.
Output: Case 2
if(testOne > testTwo){
  print("Case 1");
} else if (testOne < testTwo){
  print("Case 2");
} else{
  print("Case 3")
}
```

For Statements

For statements are used for iteration. They can traverse a data type, commonly used for searching.

```
testNums = [1,2,3,4,5]
// For loop traverses testNums. Output // 1\n 2\n 3\n 4\n 5
for(i in testNums){
  print(i); }
```

Functions/Return Statements

Functions have a name, parameters, code to execute, and return statement. Parameters are not required. By default a function will return void, the user must state what they want the return type to be if a specific type is desired. A return statement does not print to the console, rather it is saved to that function call. The output of a function call can be saved to a variable

```
function someFunction(int param1, int param2) : Int {
  if(param1 > param2){
```

```

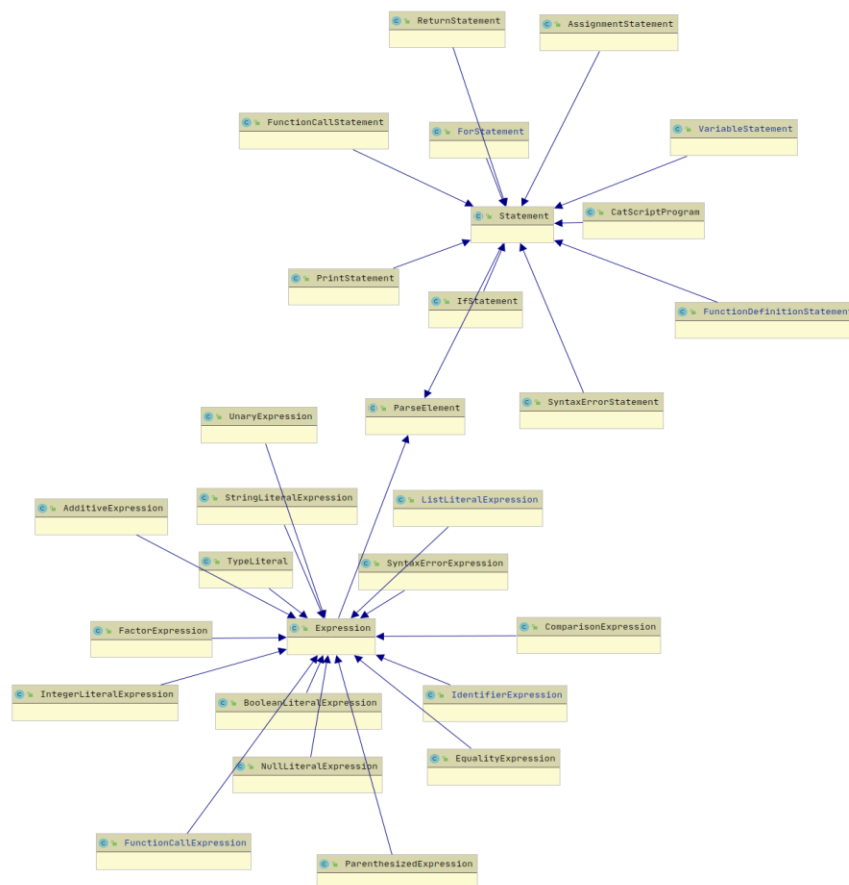
return param1; } else {
return param2; } var biggerNumber = someFunction(3, 4);
print(biggerNumber) // Output: 4

```

Section 5 – UML

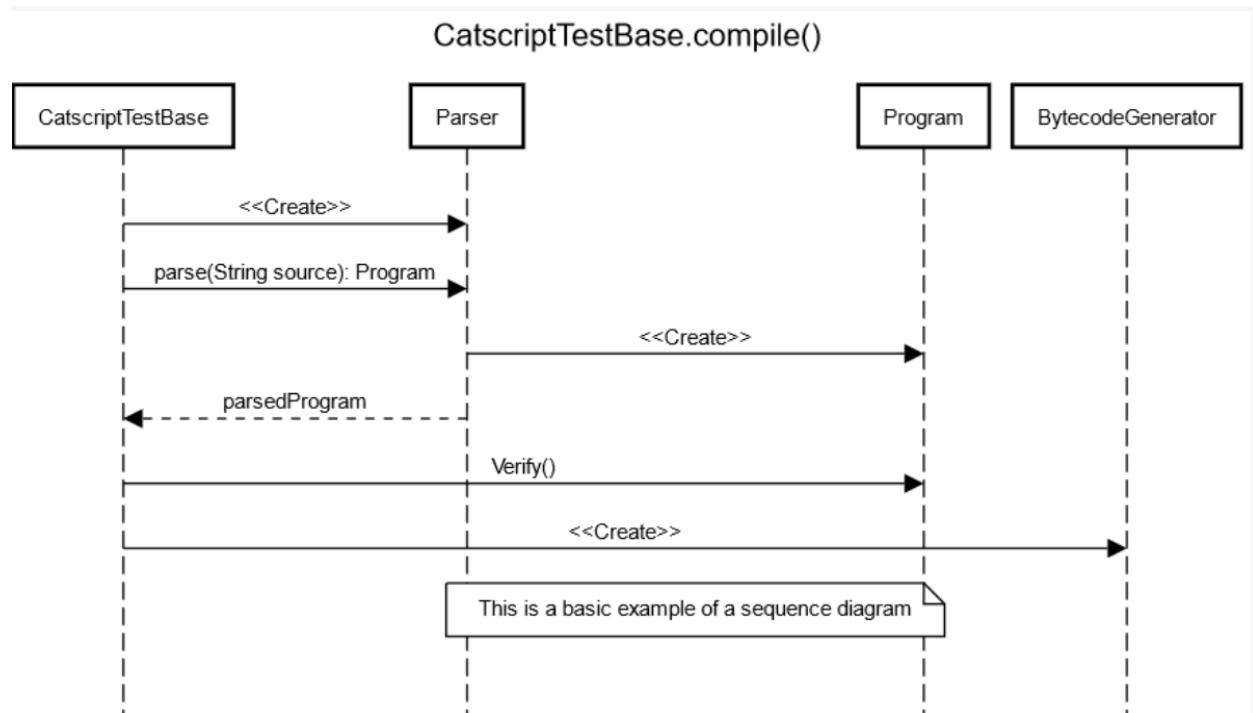
This did not require us to create a UML before starting this project. The entire project was built for us, so all we had to do was fill in the necessary code in order to get all the tests to pass. Below is a UML diagram that was provided in the past. This explains how each component is related to the other.

In this diagram, the Parse Element is divided into two categories: statements, and expressions. All the children fit into one of these categories.



Powered by yFiles

The next diagram is a very simple sequence diagram that shows a little of the process of recursive descent. It is not a fully fleshed-out diagram but shows some of the basic processes of this program.



Section 6 – Design Trade-Offs

The decision to use recursive descent was the primary tradeoff for this project. Recursive descent parsers have several pros and cons compared to other parsing solutions. For one, recursive-descent parsers are not the fastest method available, and they make error message generation difficult. On the other hand, recursive descent parsers have a very simple implementation, and full-featured parsers can be generated quickly. Another pro is it's exceptionally easy to add features within the parser.

Recursive descent parsers also express the natural recursive nature of a programming language's grammar, making that grammar almost parallel with the execution of the parser. This makes it easier for any programmer who can interpret a language's grammar to understand the parser's functions. A recursive descent parser contains a generated parse tree, making it easy to determine the nature of any bugs and how to fix them. This kind of parser has a relatively fast execution while featuring simple implementation and expansion, which makes them an ideal choice for a time-constrained project like this one.

Section 7 – Software Development Life Cycle Model

This project was developed using iterative test development. The development was divided into four main sections: tokenization, parsing, evaluation, and bytecode generation. The completion of each of these was measured by numerous Junit tests intended to determine the integrity of our software solution. Each individual test featured a section of Catscript source code, which was then passed through the appropriate Catscript functions and tested for validity.

The iterative test development model significantly aided this project by allowing the team to proceed at its pace while ensuring that each step of implementation was performed correctly. Having the availability to work on the project when it is most available to each team member, makes it easier. Avoiding cramming to get something done increases productivity and allowed for the completion of other coursework.