CatScript

Robert Marsh, Jaclyn Saunders

Gianforte School of Computing, Montana State University

CSCI 468: Compilers

Mr. Carson Gross

Spring 2023

Section 1: Program

Source code is in attached zip folder.

Section 2: Teamwork

For this version, Team member 1 worked as the primary engineer; Team member 2 worked as test engineer. As primary engineer, Team member 1 was responsible for writing the tokenizer, parser, evaluator, and bytecode generator. Throughout the semester, team member 1 developed all of the (new) source code for this project, based on the scaffolding provided by the instructor, Carson Gross. First, they implemented the tokenizer to read in string tokens from CatScript source code. Then they implemented the parser to turn the tokens into a statement/expression tree based on the CatScript EBNF grammar. Then they implemented the evaluation logic to interpret CatScript programs, and finally they implemented the bytecode generator, to be able to compile CatScript programs. Team Member 2, as test engineer, wrote tests for the evaluation portion of the project, acting as integration tests that combined the tokenizer, parser and evaluator. Team Member 2 also created the Catscript Guide and Documentantion that is included in this portfolio. For this version, Team Member 1 spent 80% of the time and Team Member 2 spent 20%. For the project as a whole, Team members 1 and 2 spent equal amounts of time on the project, since they traded roles when working on Team Member 2's version of the project.

Section 3: Design Pattern

One design pattern we used in this project was the **memoization pattern.** It is used in the CatscriptType class, in the method “getListType”. On lines **36-44.** This method is used to lookup Catscript *List Types* that contain a list of a specific Catscript Type. Rather than generate a new ListType object of the correct class each time the method is called, we implemented memoization with a Hashmap (line 36). When a certain list type is created, we check if it is in the map before creating a new one, and if not then we create a new one and store it in the map. We used this pattern to avoid the repeated computation of creating a new instance of the class each time that the method is called with the same input arguments. While this adds a little bit of overhead to maintain the map, a map lookup is faster than creating a new instance of the object each time, so it provides helpful optimization for larger programs.

3.1 Design Pattern Source Code

```
 private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new
ConcurrentHashMap<>();
    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

The technical writing for this portfolio is a guide on how to use Catscript:

Catscript Guide

Introduction

Catscript is a small, scripting programming language. It is very similar to the Java syntax. It uses recursive descent all throughout the language. There are a few major components including types, expressions, statements, and more within this language which are outlined below.

Types

The Catscript type system is statically typed with the following variables: int, string, bool, list, null, and object. The way these types are used are very similar to those in Java, but they vary slightly.

Expressions

Expression

The expression evaluates an equality expression.

Equality Expression

The equality expression evaluates whether or not two comparison expressions equal each other using “==” and “!=”.

```
true != false;
```

Comparison Expression

The comparison expression evaluates whether two additive expressions are greater than, less than, greater than or equal to, or less than or equal to each other using “>”, “<”, “>=”, and “<=” respectively.

```
15 >= 10;
```

Additive Expression

The additive expression evaluates the addition “+” or subtraction “-” between two factor expressions.

```
(15 * 100) - 2;
```

Factor Expression

The factor expression evaluates the multiplication “*” or division “/” between two unary expressions.

```
-1 * 100;
```

Unary Expression
The unary expression evaluates whether or not a primary expression or unary expression has a “-” or “not” label prepended to it.

```
not true;
```

Primary Expression
The primary expression evaluates whether an expression starts with an identifier, string, integer, true, false, null, a list, a function call, or an expression surrounded by parenthesis.

```
["banana", 1, true];
```

Identifier and String, Integer, Boolean, Null, and List Literals Expressions
These are the foundations of all the other expressions. Each of these expressions evaluate to exactly what their names suggest such as string, true, or the name of a function (identifier).

Function Call Expression
An expression starting with an identifier (count) and followed by a parenthesized argument list (numOne, …).

```
count(numOne, numTwo, numThree);
```

Statements

Program Statement
The program statement executes a statement or function definition statement.

Statement
The statement executes the for statement, if statement, print statement, variable statement, assignment statement, or function call statement.

For Statement
The for statement works like a typical Java for loop would including an identifier (i), expression (arguments), and statement (print(i)).

```
for (i : arguments) {
    print(i);
}
```

If Statement

The if statement also works like a typical Java if/else statement including an expression (i == 2), statement (return true), and optional if/else statements with more statements inside of it.

```
if (i == 2) {
   return true;
} else if (i == 1) {
   return false;
}
```

Print Statement

The print statement visually displays or prints an expression ("hello").

```
print("hello");
// output would look like this:
// hello
```

Variable Statement

The variable statement executes assigning an identifier (dogName) to an expression ("Harlee"), optionally assigning a type (string) to that identifier.

```
var dogName : string = "Harlee";
```

Assignment Statement

The assignment statement is very similar to the variable statement, except it does not initialize a new variable. An identifier (dogName) is assigned to an expression ("Chloe").

```
dogName = "Chloe";
```

Function Call Statement

The function call statement executes a function call expression.

```
foo(dogName);
```

Function Definition Statement

The function definition statement declares a new function with an identifier (helloWorld), parameter list (dogName), optional type expression (string), and one or many function body statements, which is either a statement or return statement (print()).

```
function helloWorld(dogName) : string {
     print(dogName + " says hello world!");
}
// prints: Harlee says hello world!
```

Return Statement
The return statement either stands by itself or has an optional expression following it.

```
return;
// or ...
return true;
```

Misc.

Parameter, Parameter List
A parameter is an identifier (height) followed by an optional type expression (int). A singular or list of parameters can be used in a function definition statement.

```
function myFunc(height: int) ...
```

Argument List
An optional list of one to many expressions.

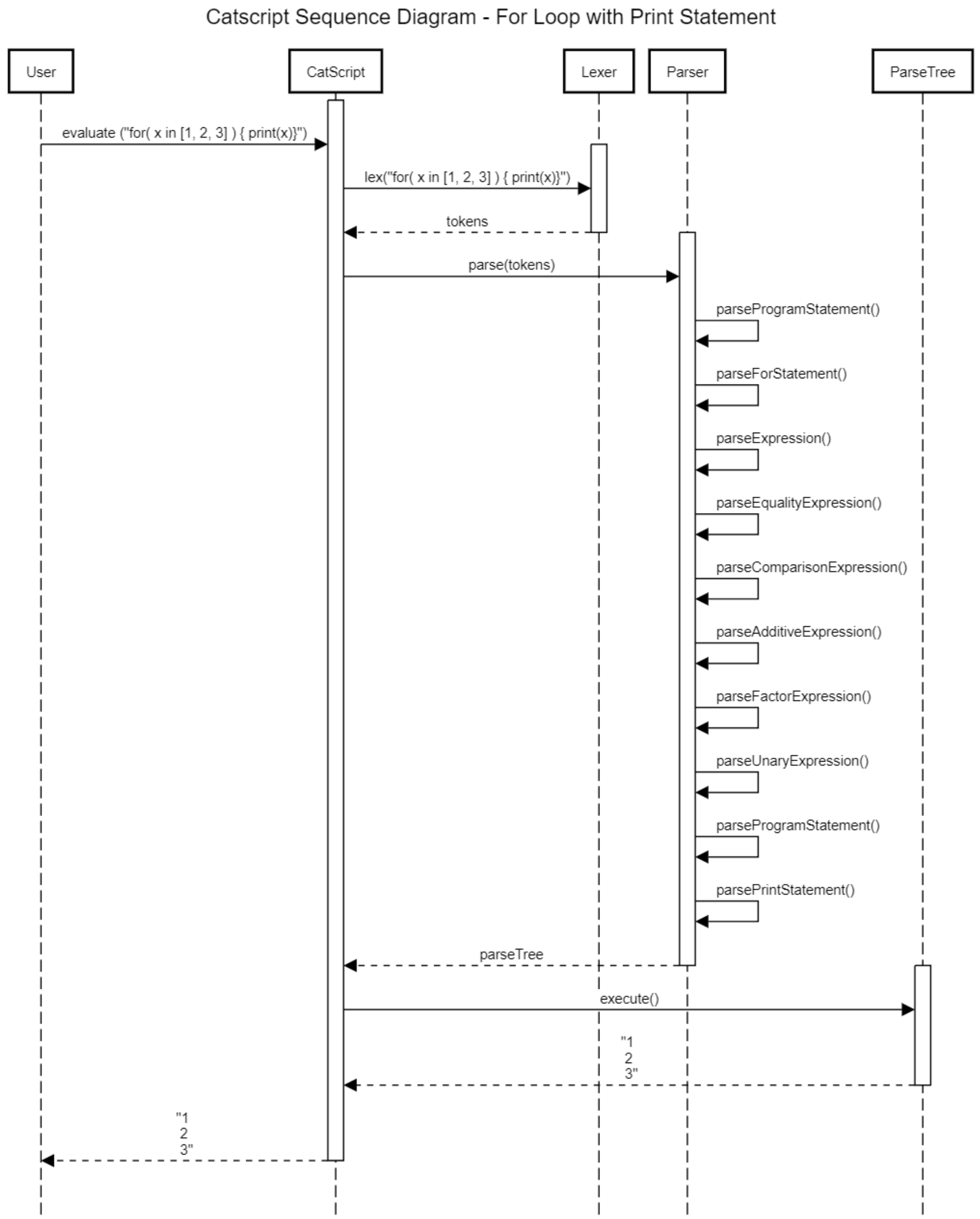
```
(numOne, myFunc, true);
```

Parenthesized Expression
An expression with parentheses around it.

Semantics

The Catscript compiler has four main stages: tokenizing, parsing, evaluating, and bytecode generation. Further details about these stages can be found within the compiler itself in the code. The examples shown above are only just a small portion in how types, expressions, and statements work together, as there are thousands of different combinations in which these components can be used.

Section 5: UML



Section 6: Design trade-offs

The main trade-off in this project was that we created our own bespoke parser for catscript from scratch, rather than using a tool to generate a parser from the EBNF language diagram. This allowed us to create a more intuitive parser using recursive descent, in order to gain an in-depth understanding of how the grammar works and maintain more control over how our parse tree was being generated. While parser generators have a high upside because they require less handwritten code, they create additional complexity. For one, they require one additional set of translations by the author to turn the ENBF grammar into the parser-generator-specific language, before translating it again into Java and actually working with it. It requires generator-specific syntax to do things that would be easier to handwrite, because the author also doesn't have control over each of the classes that is being generated. In turn, this can require additional implementation of a visitor pattern on top of the autogenerated classes in order to affect their functionality. While it took more handwritten code, using a handwritten recursive descent parser required less work to get started with as a student that is just now learning about the field, so it was faster to write in the end, and provided the opportunity to gain a better understanding.

Section 7: Software development life cycle model

For this project, we used a Test Driven Development (TDD) model. We started the course with a large suite of tests built out by the instructor to specify all parts of the language implementation - tokenizing, parsing, evaluation, and execution. When first starting out, TDD offered a helpful way to break down the project into small pieces which could be easily understood. Rather than worrying about the status of the project as a whole, we could focus on just one test (or test suite) at a time, in a self-contained environment. TDD was a helpful communication tool as well - since the specifications were created by someone else, it is important that our team met every single one of the specifications, and had tests to verify our code. Since the test *were* the specifications as part of TDD, upon completion of each test suite we could be sure that we didn't miss anything. TDD also offered the benefit of ensuring that we have 100% test coverage, since everything we wrote was centered around meeting the test specifications! Thanks to this test coverage, it was easy to find errors generated later in the project when making changes to support each new portion of the project (tokenizing, parsing, evaluation, and execution). I enjoyed working with TDD on this project, due to the high number of constraints and importance of meeting all specifications.