

CSCI 468: Compilers

Spring 2023

Travis Brase
Aurora Duskin

Section 1: Program

Catscript is a statically typed functional programming language that was developed over the course of this project. The source code for the project is located in the 'main/java/edu.montana.csci.csci468' folder within the contained zip file. The testing suite is located in the test/java/edu.montana.csci.csci468' folder within the contained zip file. The tests given from the Documentation and Testing Engineer are in the 'test/java/edu.montana.csci.csci468/tradedTests' folder. All other files not mentioned are part of the resources used to make this project.

For all source code, see the source.zip in this directory.

Section 2: Teamwork

For this project, both team members worked individually and had separate portfolios. The initial starting code for the project was provided by the capstone instructor. During the course of the semester, the Primary Engineer (Team Member 2) was responsible for completing the four sections of code, which included the tokenizer, parser, evaluation, and bytecode.

After the Primary Engineer finished their part, the team switched codes, and the Documentation and Testing Engineer (Team Member 1) of this portfolio took over. They were responsible for completing the documentation for section 4, as well as handling all debugging and checks in the testing suite while adding some extra tests.

The team's approach allowed each member to focus on a specific set of tasks, which helped to maximize their productivity and efficiency. Overall, the team's collaborative effort resulted in a successful project outcome, with each member playing a crucial role in completing the project.

The estimated percentage of time for this project is shown below:

Total Estimated Hours: 60

Code Development 83.33% : (Member 1) 50 Hours split evenly across the 4 sections

Technical Writing 6.67% : (Member 2) 4 Hours

Testing/Debugging 3.33% : (Member 2) 2 Hours

Portfolio Completion 6.67% : (Member 1) 4 Hours

Section 3: Design pattern

In our program, we decided to use the Memoization Pattern. Memoization is an optimization method that is commonly utilized to enhance the speed of computer programs. It involves storing the output of resource-intensive function calls and returning the cached result when the same inputs are encountered again. Besides improving program performance, memoization has also been applied in various other situations, including simple mutually recursive descent parsing.

The Memoization Pattern in our program stores the list type into a hashmap to use for a later call with the same argument. We decided to use this pattern to optimize the runtime of the function call. The pattern is located in the 'main/java/edu.montana.csci.csci468/parser/CatscriptType.java' file lines 38-46 (Shown Below).

```
public 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

Introduction

Catscript is a statically typed functional programming language. Included in this language is a small type system that includes:

- 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 value

With these types, you can create simple to complex code programs using the features in the following section. Here is a simple code example:

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

Features

Print Statement

The 'Print Statement' is a statement that displays output on the console or terminal. It allows you to print text, variables, or expressions to the console. In Catscript, the print statement can be written like this:

```
...  
print(1);  
...  
print("Apple");  
...
```

The output for the above statements would look like this:

```
...  
1  
...
```

Apple
...

Variable Statement

The 'Variable Statement' is a statement that declares a variable and optionally assigns an initial value to it. A variable can hold a number, string, or boolean. Following its declaration, a data type can be applied. In Catscript, the variable statement can be written like this:

...

var x;
...

var x = 10;
...

var x = 'apple';
...

var x : int = 10;
...

var x : string = 'apple';

Return Statement

The 'Return Statement' is a statement that is used to exit a function and return a value to the calling code. When a return statement is executed within a function, the function terminates immediately, and the control is passed back to the calling code along with the returned value. In Catscript, the return statement can be written like this:

...

return (i);
...

return (combine);
...

return(flag == true);
...

Assignment Statement

The 'assignment statement' is a statement that assigns a value to a variable. In Catscript, the assignment statement can be written like this:

```
...  
x = 10;  
...  
y = 'foo';  
...  
z = true;  
...
```

For Statement

The 'for statement' is a type of control structure that allows you to iterate over a sequence of values and perform a set of actions for each value in the sequence. The sequence can be a range of numbers, a list of values, or any other iterable object. In Catscript, the for statement can be written like this:

```
...  
numbers = [1, 2, 3, 4, 5]  
for (num in numbers){  
    print(num);  
};  
...  
words = ['banana', 'apple', 'orange']  
for (word in words){  
    print(word);  
};  
...
```

The output for the above statements would look like this:

```
...  
1  
2  
3  
4  
5  
...  
banana  
apple  
orange  
...
```

If Statement

The 'If Statement' is a type of control statement that allows you to execute a block of code conditionally based on a boolean expression. The boolean expression is evaluated and, if true, the code inside the statement is executed. If the boolean expression is false, the code inside the if statement is skipped. An if statement can also hold else statements. In Catscript, the if statement can be written like this:

```
...
number = 5;
If (number > 0) {
    print("The number is greater than zero");
};
...

word = 'apple';
If (word.equals('banana')) {
    print("This is a banana");
} else {
    print("This is NOT a banana");
};
...
```

The output for the above statements would look like this:

```
...
The number is greater than zero
...

This is NOT a banana
...
```

Function Body Statement

The 'Function Body Statements' are statements that belong to a Function Definition Statement. The function definition statement can have one or more function body statements. The function body statements must end in a return statement. In Catscript, the function body statement(s) can be written like this:

```
...

int i = 1;
i++;
return (i);
...

string combine = x + i;
```

```
return (combine);
...

return(flag == true);
...
```

Function Definition Statement

The 'Function Definition Statement' is a statement that defines a named block of code, function body statements, that can be called later in the program. Following its declaration, a data type can be applied that corresponds to that function's return value. They can optionally accept input arguments and return output values. In Catscript. The function definition statement can be written like this:

```
...

function func1() {
    int i = 1;
    return (i);
}
...

string x = 'The number is ';
int i = 2;
function func2(string x, int i){
    string combine = x + i;
    return (combine);
};
...

flag = true
function func3(bool flag):bool{
    return(flag == true);
}

...
```

The returned value for the above statements would look like this:

```
...

1
...

'The number is 2';
...

true
...
```


Function Call Statement

The 'Function Call Statement' is a statement that invokes a named function previously defined in the program, passing it any required input parameters. In Catscript, the function call statement can be written like this:

```
...
```

```
func1();
```

```
...
```

```
int i = 2;
```

```
string word = 'apple';
```

```
func2(word, i);
```

```
...
```

```
bool flag = true;
```

```
func3(flag);
```

```
...
```

Section 5: UML

In this sequence diagram is the compile function of the forStatement class in the project. It begins with the Actor calling the compile() method, followed by the activation of the forStatement class.

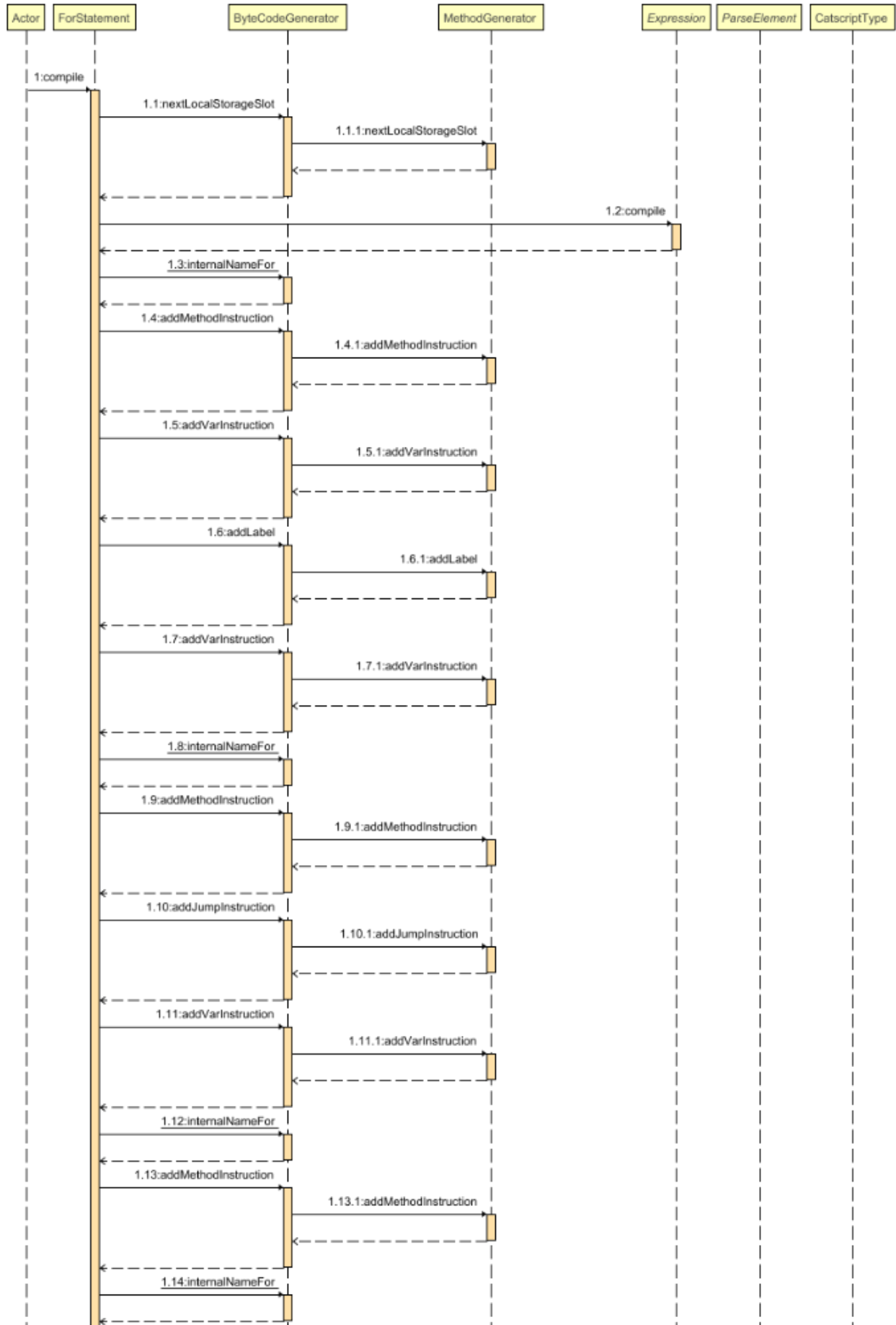
From there, the forStatement class calls the ByteCodeGenerator class then the GenerateMethod class. This is all done to get the next available storage slot on the stack.

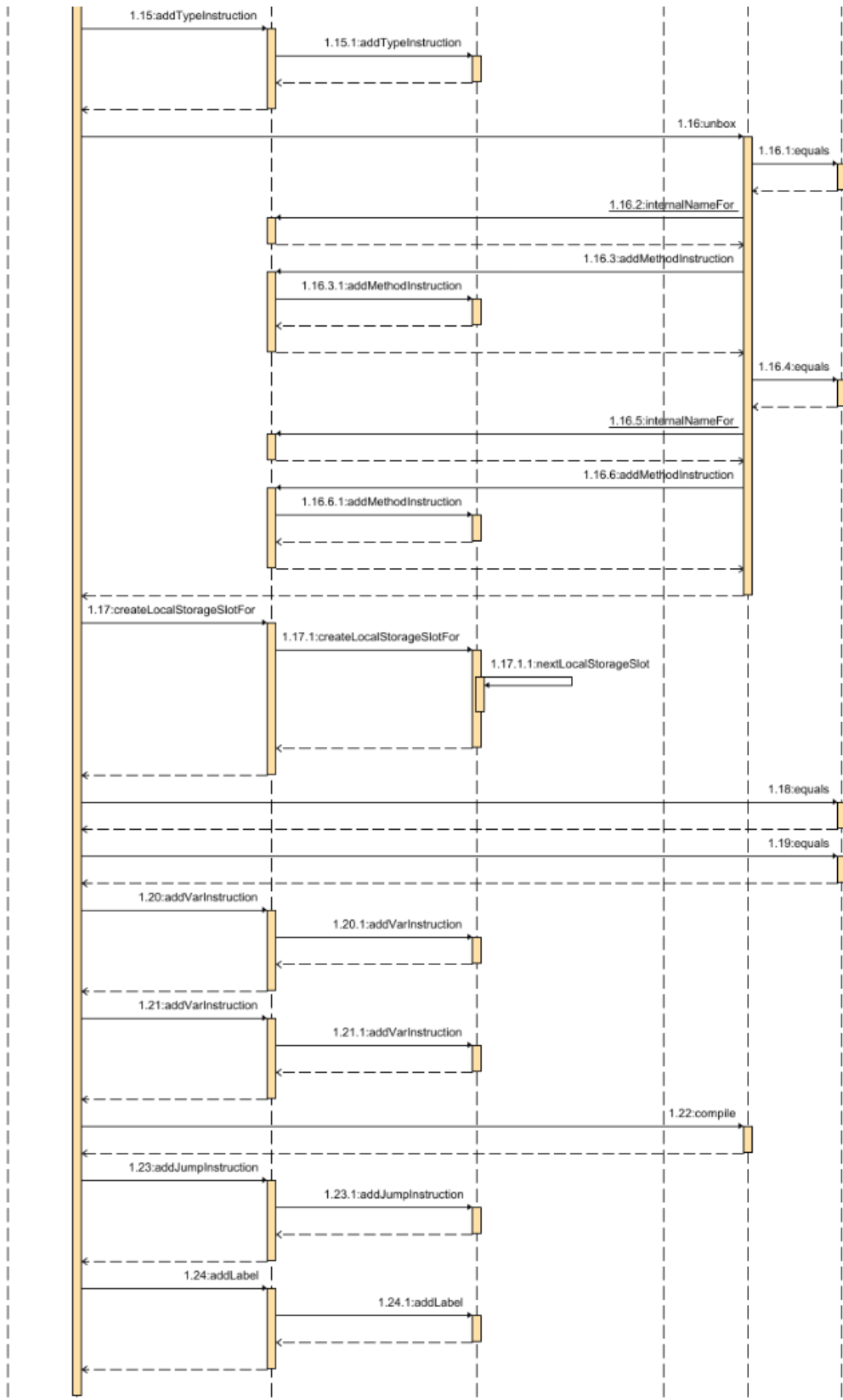
The forStatement class then calls the Expression class to compile the expressions that were given from the Actor.

Multiple calls are then completed from the forStatement and ByteCodeGenerator to complete the bytecode section of the compiled expressions.

Once that is complete, the forStatement class moves into the parsing stage where the expression is split even more and placed into a ParseTree.

After a few more calls to the generators, the given code has been compiled and is ready for evaluation.





Section 6: Design trade-offs

For this project, we made the deliberate decision to construct the parser generator manually rather than relying on a parser generator that uses regular expressions. The motivation behind this decision was to develop a more intuitive recursive descent algorithm, which would lead to a better understanding of the Catscript grammars and their underlying mechanisms.

If we had opted to use a parser generator, the Lexical grammar would have been expressed using a regular expression, and the language grammar would have been written in Extended Backus-Naur Form (EBNF). However, taking this route would have meant that the recursive descent algorithm would not have received the attention it deserved. Consequently, it would have been difficult to provide a complete explanation of how the parser works and generates parser trees from tokens.

By manually constructing the parser generator, we were able to gain a deeper understanding of the Catscript grammar and how it functions. Additionally, we developed a more intuitive recursive descent algorithm that provided a clear explanation of how the parser works and generates parser trees from tokens. Overall, this approach enabled us to build a more effective and efficient parser.

Section 7: Software development life cycle model

Our project team employed the Test Driven Development (TDD) methodology in the development of our project. We were given a test suite to work in from the capstone professor and the Documentation and Testing Enginner wrote some tests to go into it. This approach proved to be advantageous as it enabled us to precisely define the expected functionality of the code, including edge cases, well in advance of completion. TDD is recognized for its ability to improve code quality and facilitate effective collaboration among team members.