

CSCI 468: Compilers

Senior Team Portfolio

Devon Salveson & Samuel Ertischek

Spring 2023

Capstone Documentation - Devon Salveson

Contents

1. A zip file of the program. [source.zip](#)
2. A PDF of the writeup and documentation. [devon_salveson_portfolio.pdf](#) (this file).

1. Program

The program can be found in the [source.zip](#) file.

2. Teamwork

Team Member 1: Devon Salveson

Team Member 2: Samuel Ertischek

Documentation

The documentation was written by team member 2, Samuel. The documentation can be found in section 4 of this document. This documentation shows the features of CatScript and examples on how to use them. It contains sections on types, statements, expressions, and functions. This documentation is a great resource to get started with writing CatScript programs.

Testing

Testing was done by both team members. We used the test suite located at `/src/test/java` in the [source.zip](#) file. These tests validate the functionality of the CatScript compiler. Additional testing was completed by writing short CatScript programs and evaluating their output. This was done to ensure there weren't any edge cases that may have been missed by the test suite.

Team Member 2 also provided three more complex test cases located in `src/test/java/edu.montana.csci.csci468/partner/TestsFromPartner`. These tests evaluate multiple pieces of the compiler at once, such as multiple math expressions at once, or conditionally printing a string. These test further validate the functionality of the CatScript compiler.

Development

Team Member 1 was responsible for the development of the CatScript compiler. This included the Lexer, Parser, and Bytecode Generator. This work was completed over the course of the semester in the following order:

1. Tokenizing - Turning the input file/string into lexemes, and then into tokens.
2. Parsing - Turning the tokens into a parse tree.
3. Evaluation / Execution - Using the CatScript Runtime to evaluate/execute the expressions/statements in the parse tree.
4. Bytecode Generation - Turning the parse tree into bytecode that can be executed in the JVM.

These 4 steps created a fully functional compiler with the phases necessary to take us from the first step of Lexical Analysis to the final step of Code Generation. These digestible steps allowed the work on the compiler to be steadily worked on throughout the semester without having to jump back and forth between major concepts.

3. Design Pattern

The design pattern I'd like to highlight and talk about is located in

`/src/main/java/edu.montana.csci.csci468/parser/CatscriptType` at line 38.

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

public static CatscriptType getListType(CatscriptType type) {
    CatscriptType listType = LIST_TYPES.get(type);

    if (listType != null) {
        return listType;
    }
    else {
        listType = new ListType(type);
        LIST_TYPES.put(type, listType);
        return listType;
    }
}
```

In this function, we memoized the type access. The point of memoizing this call is to reduce the number of times we need to call the constructor to create a ListType. Whenever we parse a list type, we need to find its type. With other types, we simply pull them from the `CatscriptType` class directly. With a ListType, we need to create that type. In the parser, whenever we're determining the type for a list, we need to call `CatscriptType.getListType()` which originally just constructed a new type and returned it. This meant that we were constructing a new ListType every time there was a list present in the code. By memoizing this call, we're now only creating a new ListType when we haven't yet encountered that ListType. After

creating a `ListType` it's then stored in a `HashMap` so we can easily check if it exists the next time this function is called. This is a great example of memoization and how it can be used to improve performance.

4. Technical Writing

The CatScript Documentation, which serves as the technical writing document, can be found beginning on the next page.

Introduction

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

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

CatScript 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

Features

For Loops

For Loops will go through every variable in a list and use that variable for the function it has.

Here is an example:

```
list<int> x = [1,2,3,4,5]  
for(y in x){print(y)}
```

The code above would output 1 2 3 4 5. This would be because the for statement would go through each variable in the list x and print out that variable.

If Statements

An if statement will run the code it has if the expression it uses is true. If there is an else statement below the if statement it will run the code it has if the if statement is false, otherwise it will continue through the program

Here is an example:

```
var x = 5  
if(x == 5){print("x is 5")}  
Else{print("x is not 5")}
```

The code above would output x is 5 since the variable x is equal to 5. If x was not equal to 5 the code would output x is not 5.

Print Statement

A print statement will print out whatever it contains whether that is a variable, expression or a string

Here is an example:

```
print("Hello World")
var x = 5
print(x)
```

The code above would output Hello World and 5.

Assignment Statements

In catscript you can change the value of a variable using an assignment statement

Here is an example:

```
var x = 5
print(x)
x = 10
print(x)
```

The code above would output 5 and then output 10 since the variable is first 5 and it is then changed with an assignment statement to 10.

Expressions

There are many different things that you can do with variables in catscript. The things that you can do is listed below

Equality:

- **!=**
 - Compare the variable to something else to see if the variable is not equal to what it is being compared to
- **==**
 - Compare the variable to something else to see if the variable is equal to what it is being compared to

Comparison:

- **>**

- Compare the variable to something else to see if the variable is greater than what it is being compared to
- **>=**
 - Compare the variable to something else to see if the variable is greater than or equal to what it is being compared to
- **<**
 - Compare the variable to something else to see if the variable is less than what it is being compared to
- **<=**
 - Compare the variable to something else to see if the variable is less than or equal to what it is being compared to

Additive:

- **+**
 - Add the two objects together
- **-**
 - Subtract the second object from the first object

Factor:

- **/**
 - Divide the first object by the second object
- *****
 - Multiply the two objects together.

Functions

A Function is a chunk of code that you can call to use repeatedly. An example is listed below

```
Function printnum(var x)
{
    print(x)
}
```

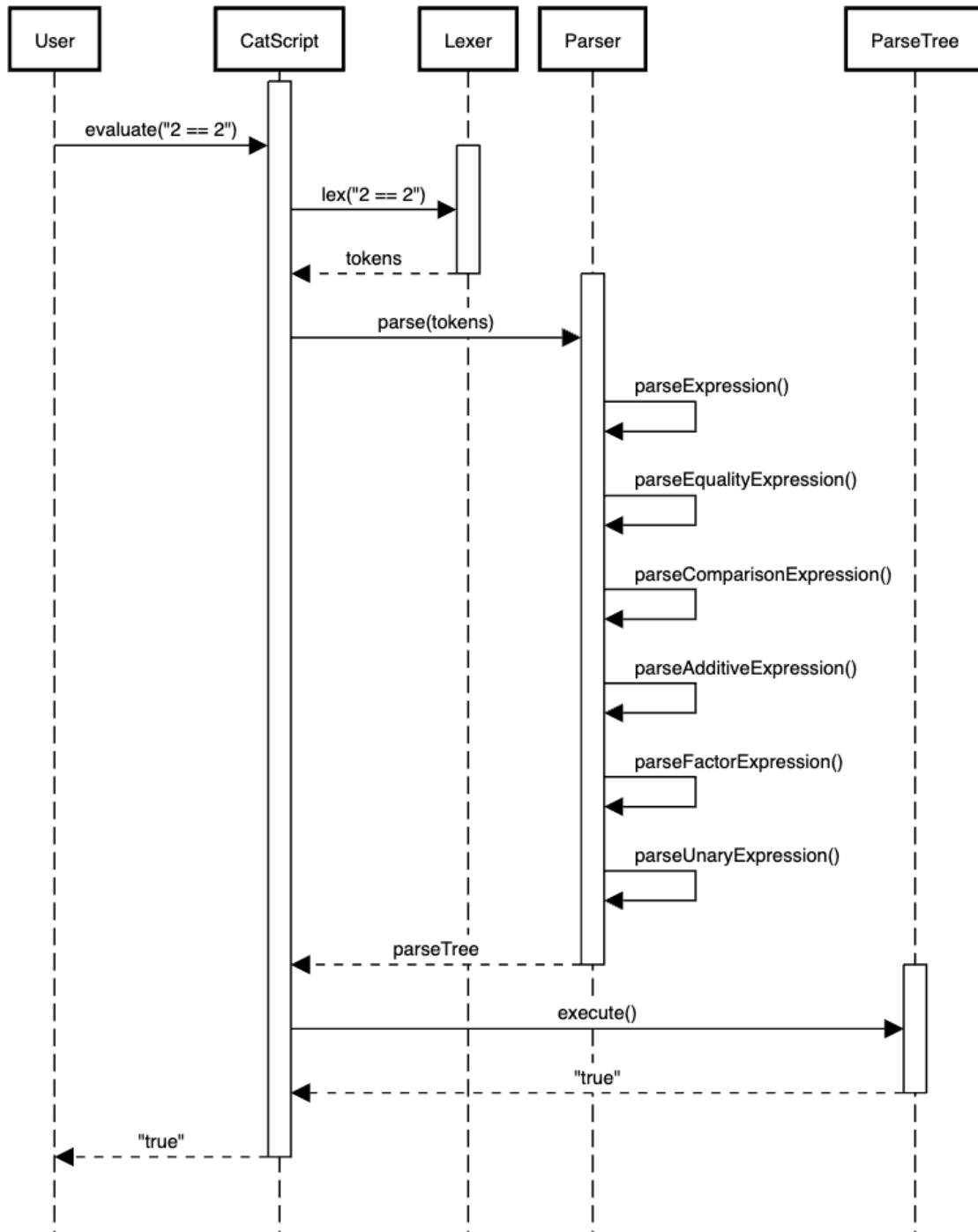
```
printnum(5)
printnum(10)
printnum(15)
```

The code above would call the function printnum and output 5 10 15.

5. UML

Sequence Diagram

Catscript Equality Expression Sequence Diagram



The diagram I want to highlight is a Sequence Diagram. In our grammar, an `if` statement takes an expression as input and that expression must evaluate to either true or false. We can pass an `EqualityExpression` to an `if` statement in CatScript. This Sequence Diagram shows exactly how the major pieces of the compiler come together to evaluate an expression.

First, input is passed to CatScript by the user. CatScript then passes it to the **Lexer** which will tokenize the input. In this example the string input `"2 == 2"` if it were to be represented as a list would be `[INTEGER, EQUAL_EQUAL, INTEGER]`.

Once CatScript has the tokens, it then passes the tokens to the **Parser**. The **Parser** will then call `parseExpression()`. Per CatScript's grammar, it will go down the list of expressions starting with parsing the **EqualityExpression**. The **EqualityExpression** consists of 2 **ComparisonExpressions** with an **EQUAL_EQUAL** or **BANG_EQUAL** token between them. Since we passed in `"2 == 2"`, it will go down the list of expressions until it finally gets to a **UnaryExpression** which will parse `2` to an **IntegerLiteral** which **EqualityExpression** will later use to evaluate.

After CatScript has finished the **Parser** call, it will then have a **parseTree** for the program. We then call `execute()` on that **parseTree** which will evaluate the expression using the values of the **LHS** and **RHS** which is `2`. Now that we've evaluated the expression, we can return the result of the expression which is `true`.

Now that CatScript has successfully taken the input, turned it into Lexemes, turned the Lexemes into Tokens, parsed the Tokens into a **ParseTree**, and evaluated the result of the **ParseTree**, we now have a result that can be finally be passed back to the user.

The result: `true`.

6. Design Trade-offs

One of the design trade-offs I want to talk about is with how we're evaluating code. The **Expression** class has an `evaluate()` method within, which is then inherited by each type of expression within CatScript. **Statements** do something similar with an `execute()` method. This method is used to evaluate the value of a particular expression. Earlier when I was talking about how the **ParseTree** was evaluating the code, this is how. It takes each **ParseElement** and if it is an expression, will call `evaluate()` on an expression, like an equality expression, and then the equality expression will call `evaluate()` on the expressions that make up the **EqualityExpression** on the left and right sides of the operator.

This method has a few pros. It's very easy to think about while writing the code. We only need to consider how this particular element evaluates, and can focus on any subsequent evaluations as entire separate problems. It also makes it very easy to add new expressions and add features to the language, because we only need to define that new expression as a class that plays by the rules of the **Expression** class that it inherits from. It is also very easy to debug, because in the case of an unexpected value, we can examine the values returned by the evaluation of each expression, and find where exactly the bug is.

I do believe that this was the correct design choice for this project. Working on a project like this could get very messy very quickly, but I felt that we were able to keep code very organized and overall, it made the code easy to read and understand, not just to someone familiar with the project, but to those glancing at it for the first time as well.

However, this method might not be considered best practice and there are valid concerns with it. The primary concern would be that we're not separating our algorithm (parse tree evaluation / execution) from our object structure (expression / statement classes). It is not considered best practice for a few different reasons. One of those reasons is that on objects with many operations (such as ours which have `evaluate()`, `transpile()`, `compile()` and `getType()`) adding new operations could become a large task, where they need to be added to not only the abstract `Expression` class, but to every class that inherits from `Expression`.

An alternative design choice would have been to use a Visitor Pattern. The Visitor Pattern would pass a Visitor to the classes, and our implementation of that Visitor would be where any logic is happening. For example, we could have had an abstract class called `ExpressionVisitor` which would implement the methods such as `evaluate()` and then create classes such as `AdditiveExpressionVisitor` which would handle the logic within the `evaluate()` method for an `AdditiveExpression`. This pattern would have been more work to implement and possibly harder to work with considering we've now split each `ParseElement` into different files within the src, but it would have achieved the separation of our algorithm / logic and our object structure.

7. Software Development Life Cycle Model

The development model we used for this project was Test Driven Development. Test Driven Development involves writing tests for the code before writing the code and then evaluating the completion of a task against the passing of the tests. This model was chosen because it allowed us to consider edge cases before beginning development, view a set of tests, and keep in mind the edge cases for a given task while developing the solution. This model was a huge help in developing this compiler because it allowed us to mentally keep track of progress because once the tests pass, that solution was complete except for any additional testing we may want to complete.

There is one drawback I can see with Test Driven Development which is that the tests have to be incredibly thorough for this model to be successful. In a case where edge cases are missed in tests or tests for particular functionality are missed altogether, it could give a false sense of "correctness" or "completeness" to a developer or development team.

Overall, I liked this model for this project. It allowed us to produce good code, and track progress without using an outside tool or having to do much project management. I would absolutely use Test Driven Development again as a model for future projects.