Montana State University

Compilers Capstone Project

Philip Gehde

Moiyad Alfawwar

Compilers – CSCI 468

Spring 2022

Section 1: Program

Please include a zip file of the final repository in this directory.

See following URL: https://github.com/codephilip/csci-468-spring2022-private/tree/master/src

Section 2: Teamwork

Students worked on their capstone projects independently. The objective was to work effectively with the existing codebase and pass the test suite.

Our teamwork was effective because we had an ongoing discourse about the concepts involved, while working on code independently. The objective for both team members was to understand the principles behind the application, and the manner of our collaboration, helped us to achieve this goal.

An estimate of hours spent on the capstone project is around 90 hours; approximately 6 hours per week.

Based on discussions with team member 2, I believe that he spent a similar amount of time.

Team member 2 provided me with several tests, all of which were passed by my program. Team member 2, also provided documentation for Catscript which is helpful to anyone interested in using Catscript.

```
public class PartnerTests extends CatscriptTestBase{


    @Test
    public void functionForLoopTest() {
```

```
    assertEquals("2\n3\n4\n5\n6\n", executeProgram("var a : int = 1\n" +
            "function func(var1 : int) {\n" +
            "for(x in [1,2,3,4,5]){\n" +
            "print(var1+x)}\n" +
            "}\n"+
            "func(1)"));

    assertEquals("8\n9\n10\n11\n12\n", executeProgram("var a : int = 1\n" +
            "function func(var1 : int) {\n" +
            "for(x in [1,2,3,4,5]){\n" +
            "print(var1+x+2)}\n" +
            "}\n"+
            "func(5)"));
}

@Test
public void ifStatementAndPrintingTest() {
    assertEquals("11\n", executeProgram("var x = 1\n" +
            "if(x != 2){\n" +
            "print(x+\"1\")\n" +
            "}\n"));

    assertEquals("x is smaller than 2\n", executeProgram("var x = 1\n" +
            "if(x > 2){\n" +
            "print(\" x variable is larger than 2\")\n" +
            "} else {\n" +
            "print(\"x is smaller than 2\")" +
                "}"));
    }

    @Test
    public void arithmeticWithParenthesisTest() {
        assertEquals("12\n", executeProgram("2*(3+3)"));
        assertEquals("6\n", executeProgram("(2*(3+3))/2"));
        assertEquals("7\n", executeProgram("(2*2)+3"));
        // negative values
        assertEquals("-100\n", executeProgram("100*-1"));
    }
}
```

Section 3: Design pattern

I chose to discuss the Memoization Design Pattern because I enjoy optimization problems. Memoization is an optimization technique used to speed up computation by caching expensive function calls and returning the cached result when the same inputs occur. In our compilers class this design pattern concerned algorithmic optimization, which ends up saving us processing power. More specifically, our program cached data types so they could be used efficiently throughout the program. The drawback is that we require additional memory space while computational time is improved. When it comes to Memoization we want to determine where in our program its application is most relevant. For example, if we can reduce the duration of an operation by some small amount, but this operation is executed many times over, this may significantly improve performance. Caching variable types is a good example of this.

Here is an example of Memoization demonstrated in my Capstone project:

```
// TODO memoize this call
static final HashMap<CatscriptType, ListType> cache = new HashMap<>();

public static CatscriptType getListType(CatscriptType type) {
    ListType potentialMatch = cache.get(type);
    if (potentialMatch == null) {
        ListType listType = new ListType(type);
        cache.put(type, listType);
        return listType;
    } else {
        return potentialMatch;
    }
}
```

This snippet is part of ParserType file located at: src/main/java/edu/montana/csci/csci468/parser

Section 4: Technical writing. Include the technical document that accompanied your capstone project.

In this project we developed a statically typed programming language called Catscript. In a Statically typed programming language, variable types are known at compile time. Oftentimes, variable types will need to be expressly declared, while some languages offer mechanisms for type inference, meaning that the type-system can deduce the type of a variable.

Advantages of Static type checking include the early detection of errors at compile time, rapid execution, and development cycles, and much more.

Our final project includes several features like type inference, mathematical operations, list literals, syntax error handling and more. Of course, the complexity of catscript does not compare to a higher-level programming language, but it was fascinating to see how these features could be implemented, nonetheless.

Catscript uses lexical scoping, a convention used by most modern programming languages, which means that the scope of a variable may only be referenced within the block in which it was defined. The scope is determined at compile time. Our Catscript language uses if statements, and iterative statements like for loops, recursion, and functions.

This Capstone project was effectively divided into 4 parts: Parsing, Tokenization, Evaluation, and Bytecode. Our efforts in studying each of these sections culminated in passing the test suite in our project directory associated with each topic.

The following Documentation was provided by Team Member 2:

Here we see examples of implementations for various expressions and statements used in Catscript as well as descriptions thereof.

~~~~markdown
# Introduction

Catscript is a simple, and easy to learn high level statically typed programming language. Catscript is made to be readable and has a small type system. Catscript compiles and run in Java Virtual Machine.

# Features

Arithmetic Operators
Plus symbol is used for the Addition operator `+`
Minus symbol is used Substraction operator `-`
Astrik symbol is used for Multiplication operator `*`
Forward Slash is used for the Division operator `/`
```
print(10+10)
print(10-10)
print(10*10)
print(10/10)
```
Output:
```
20
0
~~~~

~~~
100

1
```


## Type System


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





## Variables


NOTE: ```javascript is for syntax highlighting.
Declaring and assigning a string variable, to specify that it is a string type add `:`
after the variable name. As follows:
```javascript
var x : string = "Hello, Cats!"
var x = "Hello, Cats!" // this should also work
```
~~~

Declaring and assigning an integer variable as follows:
```javascript
var x : int = 1
var x = 1 // this also works.
```

Lists
Declaring and assigning lists.

Assigning a list without declaring a type. The type of the list would default to the type of its content. For example, a list containing only integers would be an integer list. If there are different types the list would default to an object type list.
```javascript
var intLst = [1,2,3] // list of integers
var lst = [1,"apple", 3] // list of objects
```

Integer lists
```javascript
var lst : list<int> = [1,2,3] // identical to the one above
```

String lists
```javascript
var stringLst : list<string> = ["apple", "orange", "huckleberry"]

```

Object list

```javascript
var objLst : list<object> = ["apple", true, 1, "object"]
```

For loops

```javascript
for(x in [1,2,3,4]) {
  print(x)
}
```

the for loop should output

```
1 2 3 4
```

Decision Making / Comparison

Catscripts supports Equality expressions, Comparison expressions

Equality operators:

Equal `==` and Not Equal `!=`

```javascript
```

~~~
var myBool1 : bool = (10==10)
var myBool2 : bool = (10!=10)
print(myBool1)
print(myBool2)
```

Outputs:
```javascript
true
false
```
Comparison operators:
Greater than `>`, Greater than or Equal `>=`, Less than `<`, and Less than or Equal
`<=`

```javascript
var myBool0 : bool = (10>9)
print(myBool0)
var myBool1 : bool = (10>10)
print(myBool0)
var myBool2 : bool = (10>=10)

var myBool3 : bool = (9<10)
print(myBool0)
var myBool4 : bool = (10<10)
print(myBool0)
var myBool5 : bool = (10<=10)
~~~

~~~~
print(myBool0)
```
Output:

```
true
false
true

true
false
true
```
### If statements

```javascript
var x : int = 42
var y : int = 42
if(x == y){
    print("x and y are the same")
} else if(x != y) {
    print("x and y are not the same")
} else {
    print("else")
}
```
Output:
~~~~

```
x and y are the same
```

Printing

Catscript uses a simple syntax for the print function similar to python's print function

```javascript
print("Hello, Cats!")
```

You can concatenate strings with integers to output a string
```javascript
var strX : string = ("Hello, Cats! ")
var intY : int = 42

print(strX + intY)
```
this would output:
```
Hello, Cats! 42
```

Printing boolean expressions:
```javascript
```

~~~markdown
print(10==10)
print(20>20)
print()
```
This would output:
```
true
false
```


## Functions
There are two ways to declare a function definition with parameters


First method without specifying types of parameters:
*Not Recommended: it could make the user input incorrect values
```javascript
var x = "Hello, Cats!"

function foo(str) {
   print(str)
}
```


Second Method declaring the types of the parameters:
*Recommended : It is easier to read and understand the types of the parameter.
```javascript
var x = "Hello, Cats!"
~~~

function foo(str : string) {
    print(str)
}

Function call:
```javascript
foo(x)
```

Outputs:

```
Hello, Cats!
```

Function without any parameters:

```javascript
function foo(){
    print("A function without parameters")
}

foo()
```

~~~
Output:

```
A function without parameters
```


## Comments
Catscript supports comments. To comment a line or write a comment use `//`
before the line or start typing.

```
// Catscript Comment
```
~~~

Let us go into more detail concerning some of the Features of Catscript, which are graphically represented in the UML diagram in Part 5:

Type Literals

Our Catscript language implements a variety of data types including bool, null, int, string, and object. These are statically declared when used with functions. If no return type is declared, our function will return type void. As previously mentioned, we use static typing when declaring our variables.

Syntax Error Handling

Catscript provides us with error handling, letting us know, where syntax errors are located. For example, we may receive typecasting errors, or errors for using undefined variables, just like we would in any higher level language.

Expressions

Expressions in programming are the product of combining functions and values, that are combined and interpreted by the compiler to create new values. We can evaluate expressions to either literal or object values, including Boolean, int, or string values.

In Catscript, we deal with the following types of expressions:

Integer Literal Expressions

These expressions represent 32 bit integer value. Catscript does not support floats or double.

String Literal Expressions

Defined as arrays of characters, string literal expressions can be represented by as few or many characters as desired. We use double quotes (“example”) to define string literal expressions.

Boolean Literal Expressions

Evaluates to either True or False. Declared as true, or false.

List Literal Expressions

This is how we represent a list value in our program, similar to Arrays in Java. Lists in Catscript can contain as many elements as desired. We represent our list in the following syntax:

[1,2,3]

In Java, Arrays are mostly read from, and behave covariantly, but may present problems when we try to write to them.

Unlike Java, the type-system of Catscript is considered sound, and therefore allows us to use covariant list types, like java, but unlike java our list is immutable meaning that we cannot write to it.

Parenthesized Expressions

We can use as many or as few parenthesis as we want, as long as they are logically valid.

For example: (((1))) evaluates to 1.

Additive Expressions

This includes addition and subtraction of integers and string concatenations. For example: Value1 + Value2 is valid just as 1+1 is valid. Parenthesized additive expressions are valid and follow basic order of operations. For example: Value

Unary Expressions

In our Catscript we can utilize these expressions to convert values from positive to negative. For example, we might convert -7 to 7 by writing - - 7. Similarly we may perform this operation with Booleans or variables.

Comparisson Expressions

Just like in Algebra class, comparison expressions allow variable values to be compared with variables, constants, or regular expressions. In Catscript, we may use to following operators to do this: <, >, <=,>=.

Ex. 9>10 equates to false.

Many of these expressions are self-explanatory and so I will list them here without an explanation for the sake of brevity. These include:

Equality Expressions

```
public Object evaluate(CatscriptRuntime runtime) {
    if (operator.equals("==")) {
        return getLeftHandSide() == getRightHandSide();
    } else {
        return getLeftHandSide() != getRightHandSide();
    }
}
```

Factor Expressions

```
@Override
public Object evaluate(CatscriptRuntime runtime) {
    Object rhsValue = rightHandSide.evaluate(runtime);
    Object lhsValue = leftHandSide.evaluate(runtime);
    if (operator.getType().equals(TokenType.STAR)) {
        return  (Integer) lhsValue * (Integer) rhsValue;
    }
    else {
        return (Integer) lhsValue / (Integer) rhsValue;
    }
}
```

Function Call Expressions

```
@Override
public Object evaluate(CatscriptRuntime runtime) {
        FunctionDefinitionStatement function = getProgram().getFunction(name);
        List<Object> argList = new ArrayList<>();
        for (Expression argument : arguments) {
            argList.add(argument.evaluate(runtime));
        }
        return function.invoke(runtime, argList);
    }
```

Null Literal Expressions:

The null literal evaluates to the null value,

Identifier Expression:

```
return runtime.getValue(name);
```

Syntax Error Expression:

Ex. "Bad Token"

Let's Have a look at some the Statements in Catscript:

Statements

According to Wikipedia, In computer programming, a statement is a syntactic unit of an imperative programming language that expresses some action to be carried out.

In Catscript, We can use the following Statements:

Assignment Statement:

Changes the value of a variable. Variables need to be declared before they can be assigned. Given that Catscript is a statically typed language, variables can not be changed dynamically and will produce a parse error if we attempt to turn a variable declared as a string into an integer.

Print Statement

Just like in OOP, we can use our print statement in Catscript to take an input of any type and output this to the console.

Print(exampleVar) or print(1)

Other Statements used in Catscript include:

- For Statement:
- Function Call Statement:
- Function Definition Statement
- If Statement
- Return Statements
- Syntax Error Statements
- Variable Statements

We can see some of these statements implemented in the documentation of Member 2.

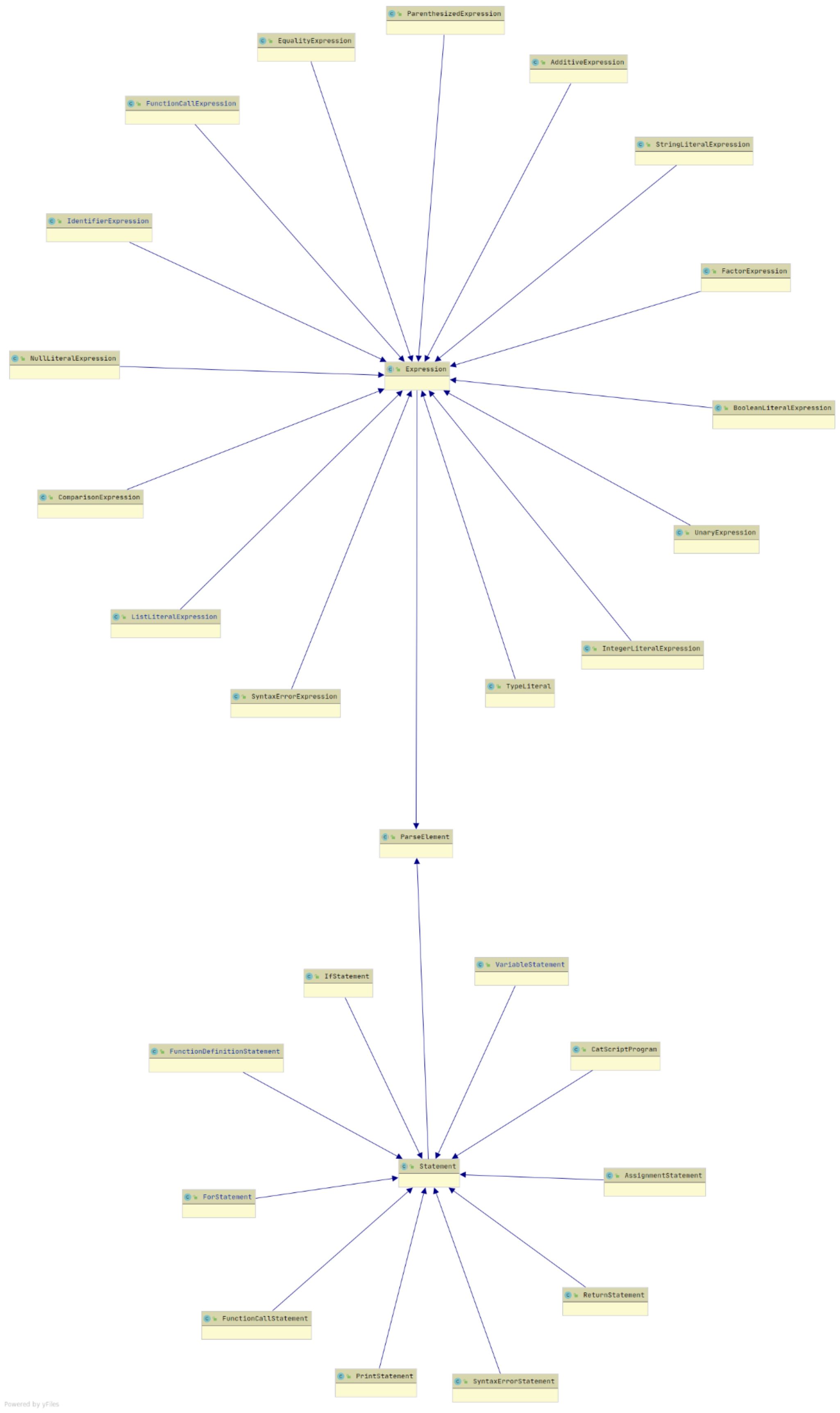
Section 5: UML.

Design was handled by professor, who consequently provided the UML for this project. ParseElement is a public abstract class that is extended by both expressions and Statements. All other classes are public.

I know that there has been a heavy emphasis on UML this year, and I would just like to say that I have found it invaluable. In my experience, UML diagrams have allowed me to quickly adopt to the workflow and organization of production level applications, especially when it comes to databases.

I believe that an emphasize on architecture and design is a core concept of a comprehensive computer science education and distinguishes our university curriculum from a being a technical vocation.

The UML below, clearly illustrates how our ParseElement Class relates to both expressions and statements which are extended by public classes in turn. ParseElement is an example of recursive descent.


ParenthesizedExpression
EqualityExpression
AdditiveExpression
FunctionCallExpression
StringLiteralExpression
IdentifierExpression
FactorExpression
NullLiteralExpression
Expression
BooleanLiteralExpression
ComparisonExpression
UnaryExpression
ListLiteralExpression
IntegerLiteralExpression
SyntaxErrorExpression
TypeLiteral
ParseElement
IfStatement
VariableStatement
FunctionDefinitionStatement
CatScriptProgram
Statement
AssignmentStatement
ForStatement
ReturnStatement
FunctionCallStatement
PrintStatement
SyntaxErrorStatement
Powered by yFiles

Section 6: Design trade-offs

In this class we decided to use recursive descent parsing over a parser generator, which represent two of the most popular ways of writing compilers.

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right [Tutorialspoint.com]. Recursive descent parsing might not be as popular in academia, but it is highly performant. Perhaps the straightforward, simplistic manner of recursive descent parsing is a reason for its absence in academia. It is easy to debug, in fact, far easier, and this was a primary factor in terms of design choice. Furthermore, recursive descent is how most commercial parsers are built. A metaphor provided by Bob Nystrom, author of https://craftinginterpreters.com/, describes recursive descent as follows:

1. We begin at a low level of individual source code characters
2. We ascend up the mountain to the point that we have a high-level understanding of the program
3. We then descend down to bytecode or machine code. [Carson, Lecture Notes].

This method makes it very obvious as to how the grammars work.

A parser generator takes a grammar as input and automatically generates source code that can parse streams of characters using the grammar [https://web.mit.edu/6.005/www/fa15/classes/18-parser-generators/].

I believe that for an introductory class, recursive descent allowed us to implement features more easily and was therefore more fun to use. Parser generators can be tricky to work with when it comes to implementing new features and less flexible when working with different grammar.

Therefore, the recursive descent parser was the better choice for an introduction to compilers.

Section 7: Software development life cycle model

Describe the model that you used to develop your capstone project. How did this model help and/or hinder your team?

We are using Test Driven Development (TDD) for this project. The objective for our project was to satisfy our test suite. This was an excellent tool to work with, as we could quickly test whether our program was meeting the requirements. It was a pleasant experience because the tests were already written, so we knew exactly what was expected of our program, and did not have to define these ‘success parameters’. Oftentimes, there is ambiguity in the expectation of how a program needs to perform, and writing appropriate tests can be a challenging task, as one needs to clearly identify all test criterias. This in itself has made me reflect more carefully on how testing criteria is to be determined in order to meet objectives, and define success.

The testing suite also empowered us to learn more about the debugging tools in our IDE. This was incredibly useful and has made me a much better programmer as a result. In combination with my software architecture class, this experience has led me to better understand software design, debugging, and testing.

I wish that more of our classes were based around coding projects like this, because the expectations are clear.