

Catscript Capstone

Elliott Pryor

6 May 2022

Contents

1	Teamwork	2
2	Design Pattern	3
3	Documentation	4
4	UML	17
5	Design Trade-Offs	21
6	Software Development Life Cycle Model	22

This project was creating the Catscript programming language as part of CSCI 468 taught Spring 2022. Project members are: Elliott Pryor, Mark Mousel

1 Teamwork

In total there are two members of the group for this capstone project. We worked in a ‘pair programming’ system where member 1 was responsible for primarily coding the project, while member 2 developed interesting tests to ensure the project meets its design goals.

Member 1 was responsible for programming the project. This took the majority of the time to complete at roughly 40 hours. This task included: studying the program architecture, understanding the algorithms, and technical implementation. Member 1 implemented a lexical parser that parses the raw input into meaningful tokens and statements that the program can combine. They also implemented a method for execution of these statements in Java. Member 1 also implemented a method to compile the program into raw JVM Bytecode to produce a compiled version of the program.

Member 2 was responsible for creating tests and for creating documentation. This took relatively little time, roughly 4 hours. These tasks included developing interesting tests to ensure the developed software functions as expected and meets desired specifications. In addition, they developed a language guide for Catscript detailing different language features and syntax. This documentation is important for any future work or for individuals interested in using Catscript so they can understand features of the language and how to use it.

2 Design Pattern

One key design pattern that we used was Memoization, or the flywheel pattern. This pattern can be used to speed up programs by storing the results of a repeated function call in a cache. This prevents duplicate, expensive function calls and can thus speed up execution. Essentially, the program can ‘memorize’ the results of certain computations allowing them to be computed again in constant time (via lookup) at the cost of some space. Note that memorize \neq memoize, but the difference is largely semantic where memoize is the specific computing related term.

Below is an example of us using memoization in the CatscriptType.java file.

```
1 // Use Memoization
2 private static Map<CatscriptType, ListType> cache = new HashMap<>();
3 public static CatscriptType getListType(CatscriptType type) {
4     ListType l = cache.get(type);
5     if (l != null)
6         return l; // if its already in the cache, return it
7
8     l = new ListType(type);
9     cache.put(type, l); // save it in the cache for next time
10    return l;
11 }
```

Essentially the goal is to reduce the number of the small list type objects that are created. The cache (line 2) creates a map object which is where we save the list types. Then we check if we already have an object saved in the cache (line 6) and if so we return it. Otherwise (line 8) we create a new object and save it in the cache before returning.

3 Documentation

Introduction

Catscript is a simple scripting language created by Professor Carson Gross for the purposes of the Compilers course at Montana State University. It is a statically typed language with lexical scoping, along with a syntax and many features that are commonplace amongst C-like programming languages, such as control flow, recursion, and functions.

In Catscript, the valid type literals are integers, booleans, strings, objects, lists, and null, all of which are indicated in code by the `var` keyword.

```
7  var a = 1
8  var b = true
9  var c = "Hello World!"
10 var d = [1,2,3]
11 var e = null
```

Comments are also implemented in Catscript in a way identical to their implementation in Java.

That is, a single line comment is indicated with a double slash (`//`) and a block comment is started with `/*` and ended with `*/`.

```
2  // single line comment
3
4  /*
5  block comment
6  */
```

Expressions

Literal Expressions

In Catscript it is possible to evaluate literals as expressions and the compiler will return the value of said literal after compilation. A literal expression can be any type that is valid within Catscript.

Integer Literal Expressions

Integer literal expressions in Catscript represent a 32-bit integer value.

```
1 17
```

The above program will evaluate to 17. Please note that Catscript does not support floating-point numbers.

Boolean Literal Expressions

Boolean literal expressions in Catscript are indicated either by the keyword true or false.

```
1 true
```

```
1 false
```

The above programs will evaluate as true and false respectively.

String Literal Expressions

String literal expressions in Catscript are strings of any character surrounded by double quotes.

```
1 "Here is a string"
2 this is not a string
```

The string in quotations will evaluate to the phrase contained within. For concatenating strings, see Additive Expressions.

List Literal Expressions

List literal expressions in Catscript are lists of other literal expressions surrounded by square braces, with each element separated by commas.

```
1 [1,2,3,4,5]
```

The above program will evaluate to [1,2,3,4,5].

Additionally, in Catscript not all elements in a list need to be the same element.

```
1 [1, true, "string"]
```

The above is a valid Catscript program and will evaluate to [1, true, "string"].

Null Literal Expression

The null literal expression is an object that has no value.

```
1 null
```

The above program will evaluate to null.

Unary Expressions

As the name implies, unary expressions are made of a single operand. In Catscript, there are two such expressions: negative and not.

Negative

The negative unary expression in Catscript is indicated with the - operand. It will negate the value of the following integer value.

```
1 -42
```

The above program will evaluate to -42.

Interestingly, in Catscript multiple negative unary expressions can be used in succession.

```
1  -|-42
```

The above program will evaluate to 42.

Not

Similar to the Negative unary expression, the Not unary expression is indicated with the not keyword and will negate the value of the following boolean value.

```
1  not true
```

The above program will evaluate as false.

Also like the Negative unary expression, multiple not keywords in succession will continually negate the boolean value.

```
1  not not true
```

The above program will evaluate as true.

Additive Expression

In Catscript, additive expressions are used to either add or subtract integer values, or to concatenate strings. The operators used in arithmetic are left associative.

Add

```
1  34 + 17
```

The above program will evaluate to 51.

Subtract

```
1  34-17
```

The above program will evaluate to 17.

In Catscript, multiple add and subtract additive expressions can be strung together to create a valid program.

```
1 1 + 15 - 6
```

The above program will evaluate to 10.

Concatenation

```
1 "Hello" + " World!"
```

The above program will evaluate to Hello World!.

Similar to the add and subtract, multiple concatenation additive expressions can be strung together.

```
1 "It"+"was"+"the"+"best"+"of"+"times,"+"it"+"was"+"the"+"worst"+"of"+"times"
```

The above program evaluates to Itwasthebestoftimes,itwastheworstoftimes.

Factor Expression

In Catscript, factor expressions are used to either multiply or divide integer values. The operators used in arithmetic are left associative.

Multiply

```
1 3*5
```

The above program will evaluate to 15.

Divide

```
1 16/4
```

The above program will evaluate to 4.

In the event that the result is not an integer, the result will be rounded down since floating point values are not valid within Catscript.

Comparison Expression

Comparison expressions are used to compare two integer values and will return a boolean depending on the result. The valid comparison operands in Catscript are `>`, `>=`, `<`, and `<=`.

```
1 2 > 1
2 2 >= 1
3 2 >= 5
4 2 < 1
5 2 <= 1
6 2 <= 0
```

Each line in the above program will evaluate to true, true, false, false, false, and true respectively.

Equality Expression

Equality expressions compare two expressions and return a boolean value depending on the result. The valid equality expression operands in Catscript are `==` and `!=`.

```
1 1 == 1
2 1 == 2
3 1 != 1
4 1 != 2
```

In the above program, each line will evaluate to true, false, false, and true respectively.

Parenthesized Expression

Parenthesized expressions are simply expressions of any type surrounded by parentheses.

```
1 (1)
```

The above program will evaluate to 1.

In Catscript, parenthesized expressions can have multiple pairs of parentheses.

```
1 (((1)))
```

The above program will also evaluate to 1.

Statements

Print Statements

Print statements print the string value of whatever they are given.

```
1 print(1)
2 print("Here is a string")
3 print(true)
```

The above program will print out 1, Here is a string, and true on separate lines.

Variable Statements

Variable statements create a variable of a given type. If no type is given, the compiler will infer its type.

```
1 var x : int = 100
2 var y = 17
```

Assignment Statements

Assignment statements are used to change the value of a variable during runtime.

```
1 var x = 1000000
2 x = 2
```

Following line 2 of the program, the new value of x is 2. Please note that when assigning the value of a variable, it must remain the type it was, otherwise it will cause a parse error.

For Loops

In Catscript, for loops iterate through lists.

```
1 for(x in [1,2,3,4,5]){
2   print(x)
3 }
```

In the above program, the for loop will iterate through the list of [1,2,3,4,5] and print each one, resulting in 1 2 3 4 5 being printed to the console.

If Statements

In Catscript, if statements are used to branch to a different part of the program depending on the value of a comparison expression.

```
1 var x = 10
2
3 if(x>20){
4   print("The number is bigger than 20")
5 }
6 else if(x>15){
7   print("The number is bigger than 15")
8 }
9 else{
10  print("The number is smaller than 15")
11 }
```

In the above program, the value of the comparison expression in each statement will be evaluated, and when neither the if or the else if evaluates to true, the else statement will be entered the code within will be executed.

Functions

In Catscript, functions are used to enable the repeated usage of a block of code. Functions are defined using the function keyword, along with its name, arguments, and return type. A function can have none, one, or many arguments. If no return type is given, the function will have a return type of void, meaning it doesn't return anything.

```
1 function foo(argumentOne:string){
2   //do stuff
3 }
4
5 foo("hello world")
```

In order to call a function that has already been defined, one must simply type the name and all of the required arguments, as seen on line five of the above program.

Return Statements

In Catscript, return statements are used to return a value from a function.

```
1 function foo(argumentOne:string){  
2   //do stuff  
3   return returnValue|  
4 }  
5  
6 foo("hello world")
```

In the above program, whatever is assigned to returnValue will return with the function call on line five. The program can then use this value later to perform other tasks.

Grammar

```
catscript_program = { program_statement };  
  
program_statement = statement |  
                    function_declaration;  
  
statement = for_statement |  
             if_statement |  
             print_statement |  
             variable_statement |  
             assignment_statement |  
             function_call_statement;  
  
for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',  
               '{', { statement }, '}'  
               );  
  
if_statement = 'if', '(', expression, ')', '{',  
              { statement },  
              '}' [ 'else', ( if_statement | '{', { statement }, '}'  
              ) ];  
  
print_statement = 'print', '(', expression, ')'  
  
variable_statement = 'var', IDENTIFIER,  
                    [ ':', type_expression, ] '=', expression;  
  
function_call_statement = function_call;  
  
assignment_statement = IDENTIFIER, '=', expression;  
  
function_declaration = 'function', IDENTIFIER, '(', parameter_list,  
                       ')' +  
                       [ ':', type_expression ], '{', {  
function_body_statement }, '}'  
  
function_body_statement = statement |  
                        return_statement;
```

```
parameter_list = [ parameter, {',' parameter } ];

parameter = IDENTIFIER [ , ':' , type_expression ];

return_statement = 'return' [ , expression];

expression = equality_expression;

equality_expression = comparison_expression { ("!=" | "==")
comparison_expression };

comparison_expression = additive_expression { (">" | ">=" | "<" |
"<=" ) additive_expression };

additive_expression = factor_expression { ("+" | "-" )
factor_expression };

factor_expression = unary_expression { ("/" | "*" ) unary_expression
};

unary_expression = ( "not" | "-" ) unary_expression |
primary_expression;

primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false"
| "null" |
                    list_literal | function_call | "(", expression,
                    ")"

list_literal = '[', expression, { ',', expression } ']';

function_call = IDENTIFIER, '(', argument_list , ')'

argument_list = [ expression , { ',', expression } ]

type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [,
'<' , type_expression, '>']
```


4 UML

Many parse elements were designed to extend a few abstract classes. Generally there are Statements and Expressions. Statements are elements like ‘if’ and ‘for’ or ‘functions’, while Expressions are more operations like ‘additive’, ‘string literal’, ‘comparison’. Statement Inheritance diagram can be seen in Figure 1 where all the classes extend one abstract class: Statement. The Expression inheritance diagram is similar. It can be seen in Figure 2, where all the classes extend one abstract class: Expression.

Both Statement and Expression extend Parse Element. This is the generic type of object that the parser works with. The high level class diagram of this inheritance can be seen in Figure 3

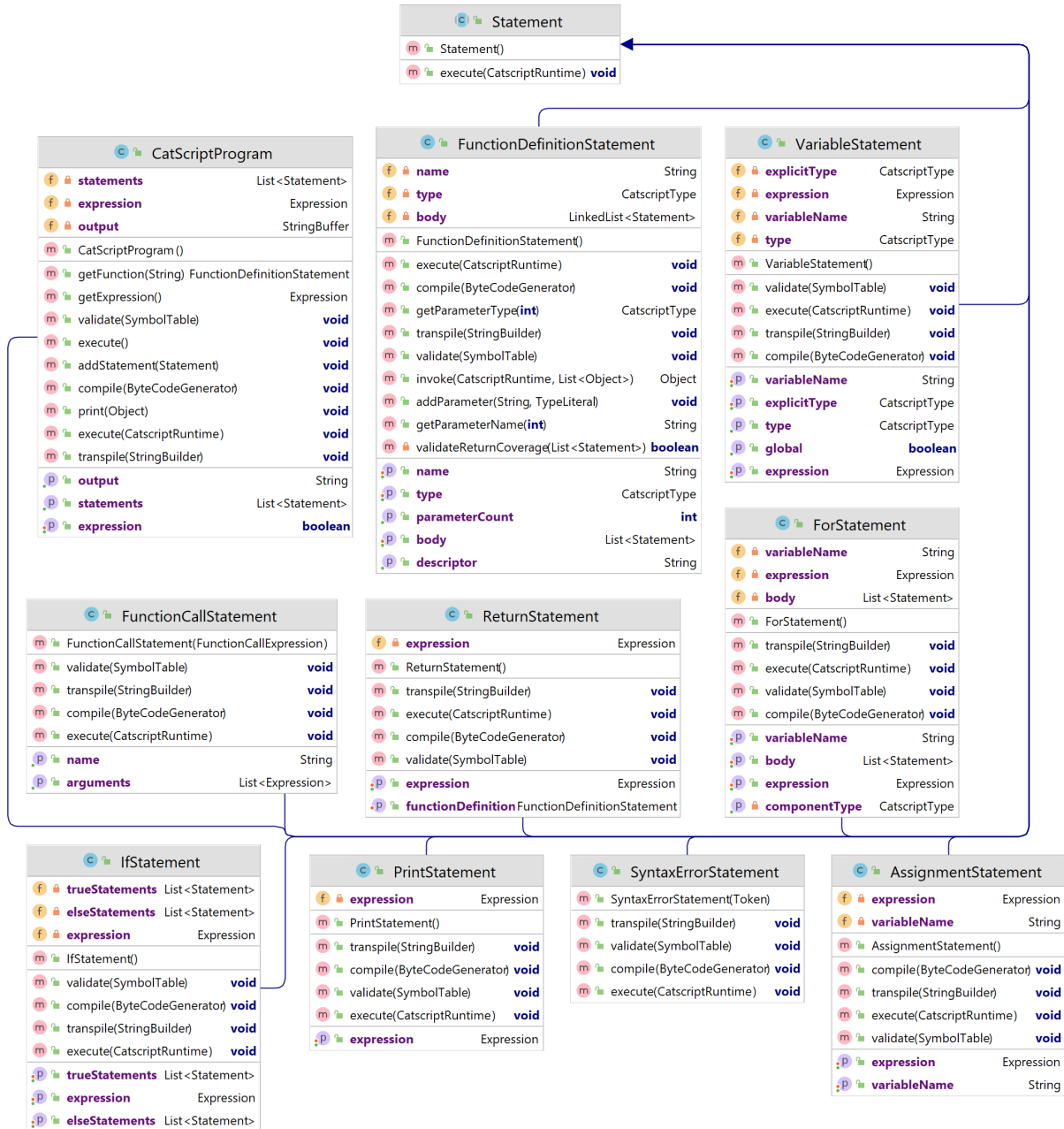


Figure 1: Inheritance diagram of Statements in Catscript

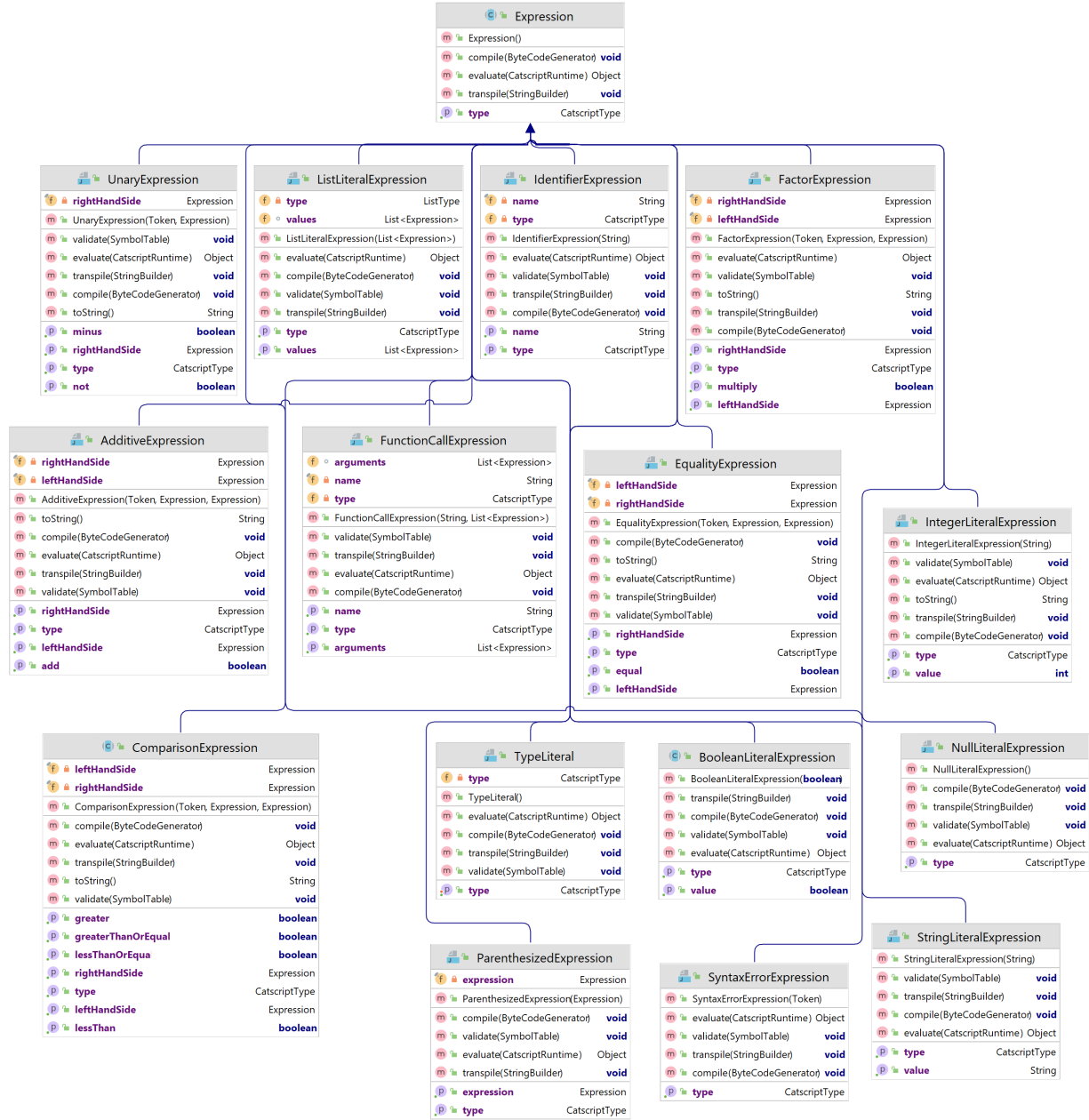


Figure 2: Inheritance diagram of Expressions in Catscript

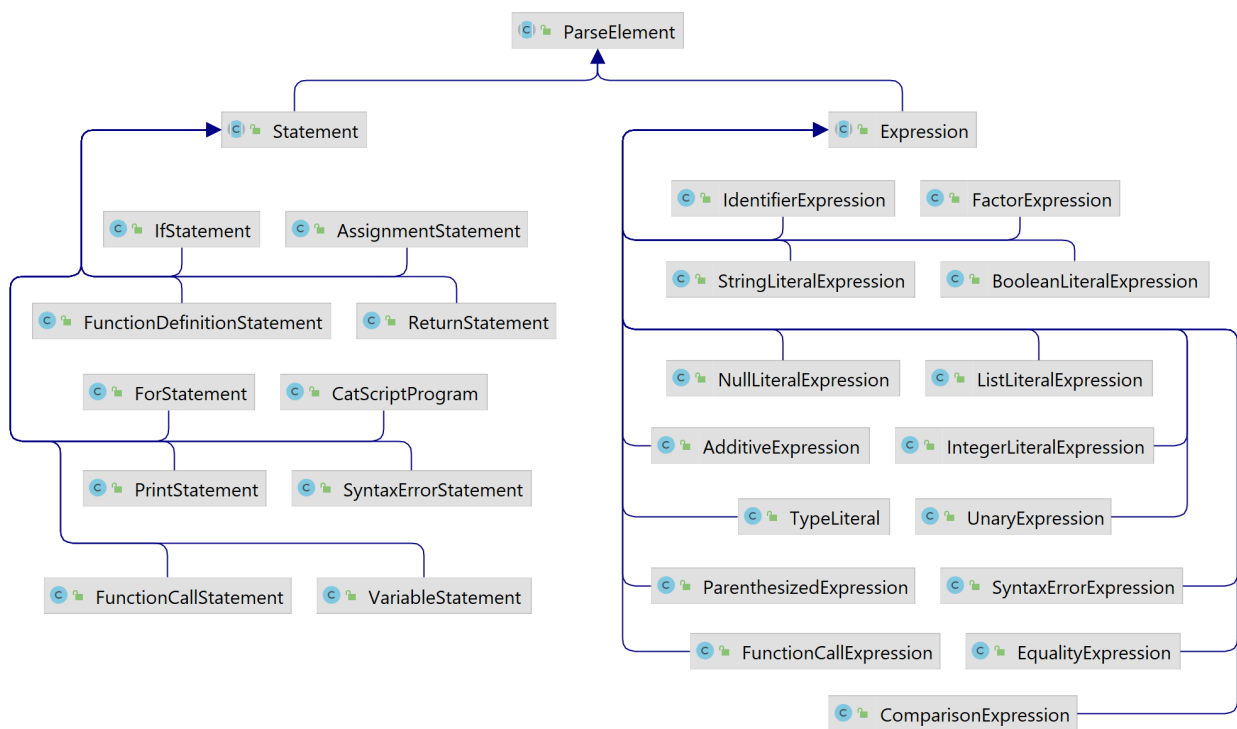


Figure 3: Full inheritance diagram of parse elements. Due to the number of elements, few details are shown. Please see Figures 1, 2 for more details

5 Design Trade-Offs

The made design trade off that was made was to use the recursive descent algorithm instead of parser generators. The recursive descent algorithm is in some ways more simple and easy to understand. It also meshes well with the EBNF definition of the grammar. The EBNF grammar is defined recursively so it fits naturally and intuitively with recursive descent. The primary reason for our choice in the recursive descent algorithm was the ease of understandability of the method.

One drawback of recursive descent is that we have to write more lines of code by hand. This increases the technical complexity of the project as more lines of code have to be written. We thought that this is balanced by the greatly improved comprehensibility of recursive descent. Another drawback of recursive descent is that parser generators are more standard. Parser generators are taught more in universities and are often used in industry (although recursive descent is prevalent in industry). This is another major drawback and is something worth considering, but we still thought it was worth the tradeoffs.

In order to better understand the tradeoffs, we first should understand what parser generators are and how they work. Essentially, a parser generator is something that you can program to generate all the key elements of a compiler (lexer, parser) automatically. The lexer essentially works with defining regular expressions (regex) that match a given lexical element. For example, comments:

```
1 COMMENT
2     :    '/' '/' ~('\'\'|\'\'r\')* \'\'r\'? \'\'n\' {skip();}
3     |    '/' '*' ( options {greedy=false;} : . )* '*' '/' {skip();}
4     ;
```

Make sense? That essentially selects everything up to the new line character for the '/' '/' and everything until the next '/' '*' as the comment. The next is the parser. This makes a parse tree and has some specific headers, but after that there are some components that resemble the EBNF that the grammar is defined by. For example with addition,

```
1 addExpr
2     : ID
3     | INT
4     | STRING
5     ;
```

In general, lots of the syntax is very obscure and it is not an intuitive implementation of the EBNF grammar. For this reason, we chose to use recursive descent algorithm instead.

6 Software Development Life Cycle Model

We used Test Driven Development (TDD) for this project. As the name suggests, this development style focuses heavily on unit tests. Unit tests for program functionality are written before programming starts. This can help make sure the program meets the desired requirements by creating tests for those specific requirements.

The steps of Test Driven Development are:

1. Write a unit test
2. Write minimal amount of code to pass the test
3. Repeat

This can help ensure that the end product does not have errors. Also, after refactoring it can help ensure that a change didn't cause ripple effects that break other parts of the program. It also allows the programmer to isolate individual portions of the code where problems are, and isolate individual features to test.

Overall, I found it very helpful for understanding how the program works. It helped me conceptualize what each individual component of the program was supposed to do and how to link them together. It helped me understand key components and improved my ability to problem solve on the system and think critically about design.