

CSCI 468 – SPRING 2022

PORTFOLIO

ISABELLE BENNETT

ANDREW CILKER

Section 1: Program.

Find the program attachment in the source.zip file

Section 2: Teamwork.

Describe how your team worked on this capstone project. List each team member's primary contributions and estimate the percentage of time that was spent by each team member on the project. Identify team members generically as team member 1, team member 2, etc.

- Team Member 1: Isabelle Bennett
- Team Member 2: Andrew Cillker

Team Member 1 contributed the source code for the implementation of Catscript. Approximately 100 hours were spent coding this project, this contributes 95% of the project. Member 2 contributed 5% of the project creating additional tests and documentation for the language. Team Member 2's contributed unit tests are located in src/test/mytests/MyTests.java

Section 3: Design pattern.

The design pattern used in Catscript was the Memoization pattern. It is located in the getListType() method in the following path:

src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java

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

Implementing this design pattern helps limit the amount of times methods with expensive computations are run. The pattern uses a HashMap to store results in a cache. Now, the act of running many computations can be replaced by a minimal lookup.

Section 4: CATSCRIPT DOCUMENTATION

Include the technical document that accompanied your capstone project.

What is Catscript:

Catscript is a statically typed programming language with a small type system. It was developed with a recursive-descent parser. The language features tokenization, parsing, evaluation, and compilation to bytecode. It is syntactically similar to popular languages like Java, C, and Python with some of its own unique and interesting features.

CONTROL STRUCTURE IN CATSCRIPT

Like most programming languages, Catscript supports decision logic using *for*, *if*, and *else* statements. Below is an example of different ways these statements could be defined and some examples of them in use.

```
if(expression) {  
    statement  
}else{  
    Statement}  
  
for(x in expression){  
    statement  
}
```

CATSCRIPT'S TYPE SYSTEM

These are the primary types/terminal elements that other complex elements of the language evaluate to.

- int – 32-bit integer
- string – java-style string
- bool- Boolean value
- null – null type
- object – any value

PRIMARY EXPRESSIONS IN CATSCRIPT

This expression type can be directly evaluated to any of the literal types as well as:

- identifiers
- list literals

- parenthesized expressions
- and function calls

MATHEMATICAL EXPRESSIONS IN CATSCRIPT

Like most languages, Catscript has the ability to perform mathematical computations. Its implementation is also syntactically similar to most programming languages. Here are some examples of the different mathematical expressions possible in Catscript.

ADDITIVE EXPRESSION

These expressions handle subtraction and addition operations. The '+' and '-' operators are used to separate expressions by their left and right-hand side. The '+' symbol is also overloaded for string concatenation which this expression is also responsible for evaluating.

```
x+y
```

```
4-1
```

```
"Hi" + name
```

EQUALITY EXPRESSION

The equality expression determines the equality relationship of two sides of the expression. The operators in this method include: '==', and '!='.

```
x == 3
```

```
x != value
```

COMPARISON EXPRESSION

The comparison expression handles evaluating the relationship of two sides of an expression. The four operators used in this method include: '>', '<', '<=', '>='.

```
5 < 3
```

```
result+2>4
```

FACTOR EXPRESSION

The factor expression handles division and multiplication using the '*' and '/' operators. The precedence of operations can also be controlled by using parentheses.

```
4 * 4  
4 / 2  
(5 - 3) / 2
```

VARIABLES IN CATSCRIPT

IDENTIFIER EXPRESSION

This expression creates variables and stores additional information about the variable type.

VARIABLE STATEMENT

These statements define and initialize variables. To define a variable, the *var* keyword is needed as well as a variable name to identify it. To initialize the variable's value, follow the declaration by an optional ':' and the variable's type, an equal sign, and the right-hand expression.

```
var name : String = "Bob"
```

FUNCTIONS IN CATSCRIPT

FUNCTIONAL CALL STATEMENT

The function call statement is used to call the function and trigger the function's execution.

FUNCTION BODY STATEMENT

The function body statement holds the statement that the function call will execute. It is optionally followed by a return statement.

FUNCTION DECLARATION

A function in Catscript can execute statements and evaluate expressions. They can be set to return specified values. To create a function in Catscript, begin with the *function* keyword followed by a name to identify the function, then a list of 0 or more parameters enclosed in parentheses— these parameter values can either have their types defined or simply be filled with identifiers. Finally, add a function body statement enclosed in curly

brackets. Below are two examples of how you could implement a function declaration in Catscript.

```
function x(a : object, b : int, c : bool) {}
```

```
function x(a, b, c) {}
```

RETURN STATEMENT

A return statement is used within a function body statement to return the value of the function evaluated. It is implemented using the *return* keyword followed by the expression or value the function returns.

```
return x+y;
```

```
return "Hi" + name;
```

UNARY EXPRESSION

The Unary Expression handles negation and the negative symbol. The two operators it uses are – and not.

```
not(func(x))
```

PRINT STATEMENT

This statement prints desired values to the program's standard output. To create a print statement in Catscript, call the print keyword and follow it by the expression you wish to have evaluated and printed.

```
print("Hello")
```

Section 5: UML. Attach the UML design diagrams for your capstone project that were created **before** you began coding your project.

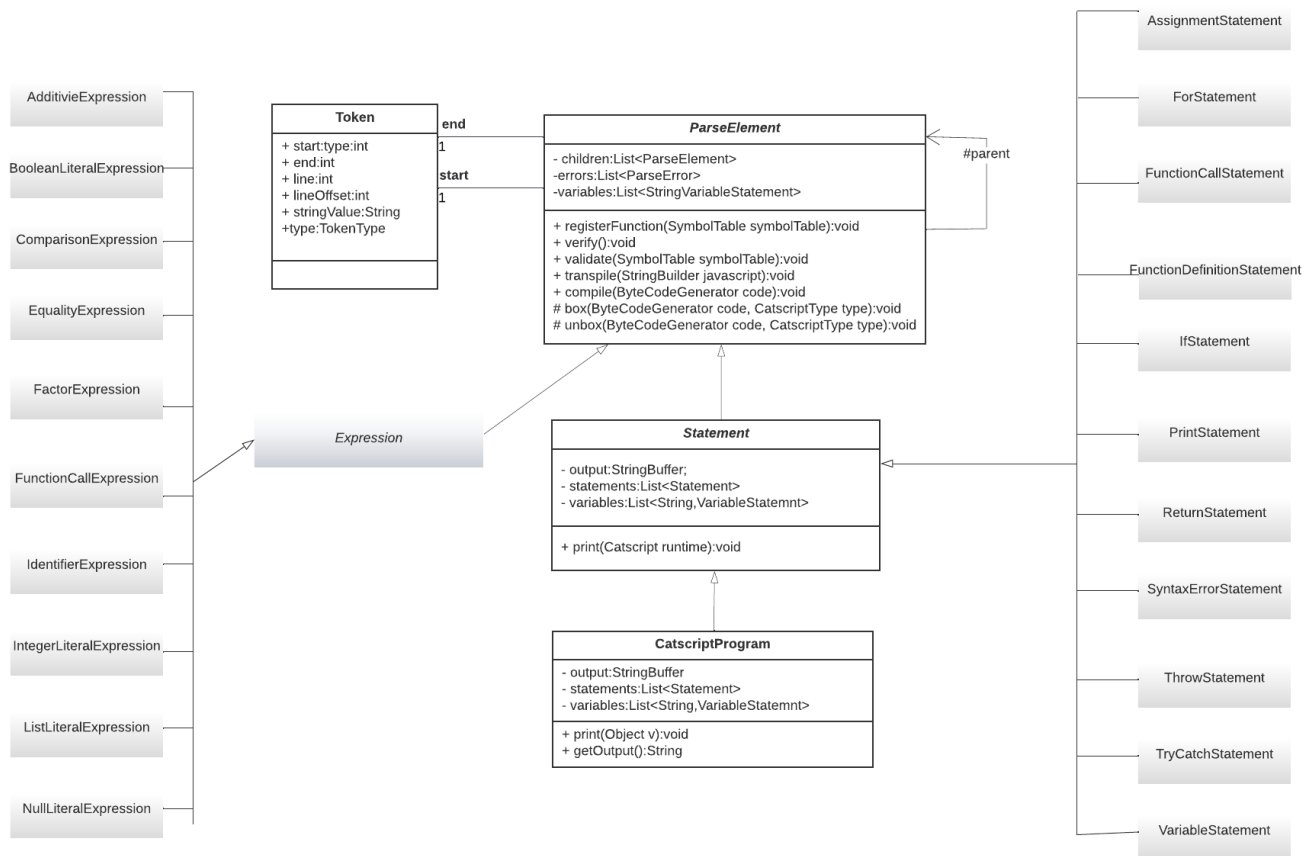


Figure 1: UML Diagram

I created a UML class diagram that demonstrates the general way the compiler is structured. I chose to go into more detail with the CatscriptProgram and left the other statements at a higher level. The CatscriptProgram class is a pinnacle part of the compiler as it is the starting point for any use of the Catscript Language.

Section 6: Design trade-offs. Describe a design trade-off decision (e.g. execution time vs. space requirements or compile-time) in your capstone project and justify the design decisions that you made.

The primary design trade-off in this project was the decision to use Recursive Descent. Catscript implements a Recursive Descent Parser instead of the Parser Generator. The benefit of this implementation is the simplicity. From a student's perspective, recursive descent helped me gauge a better understanding of the recursive nature of grammar. The downside of recursive descent is it requires more handwritten code, and it is not typically the industry standard.

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?

This project followed the Test-Driven Development life cycle model (TDD). Unit tests were provided prior to development by my instructor. These tests verified the completion of different required components of the compiler.

A major benefit of this development model is the reduction of bugs– they are directly addressed throughout the development process. Since the test suite is designed to consider edge cases, TDD produces a higher quality program with a higher percentage of test coverage. It was also helpful to have a test suite for code maintenance. With this structure, it is easier to run tests, later on, to ensure no modifications or bugs mess up anything in the program. As a student with the primary goal of conceptualizing how a compiler works, having the different components of the compiler broken down into chunks of unit tests that I could step through the functionality of was very insightful and encouraged active learning. Breaking functions of the code into unit tests also helped enforce modularity ensuring a more organized and structured project.