

# Compilers Portfolio

Marnie Manning

May 2022

## 1 Program

A zip file has been included in the directory of the final repository.

## 2 Teamwork

The design of this project has each student build out a full compiler in the CatScript language developed for the CSCI 468 Compilers class at MSU, then exchange three tests and our documentation. For this I worked with Darshan Diamond and Jay Forbes for the final part of the project.

Over the course of the semester I completed the three primary deliverables for the CatScript compiler, the tokenizer, the parser, and the byte code generator. The CatScript compiler is a recursive decent parser that compiles into Assembly in the Java Runtime Environment.

Once the compiler was completed my team mates and I exchanged our tests; I gave mine to Jay, Jay gave theirs to Darshan, and Darshan gave theirs to me. Darshan's tests looked for; proper tokenization to ensure the tokens are correctly identified with an emphasis on the start and end locations of a given token, basic variable assignment, and If-Else Statement Controls. Theses tests covered a nice mix of situations and were suitably complex. The compiler passed all the tests I was given. Over all I spend around 3 to 4 hours writing my documentation and tests. Darshan and Jay spend around 2 to 3 hours completing each of their documentation and tests.

## 3 Design pattern

For this project I used the memoization design pattern. This was a good design pattern to use as compilers run into many similar smaller elements and this design pattern helps prevent creating all multiples of the same element by storing them in a concurrent hashmap.

While it is not relevant to CatScript as it only runs on one thread, it is important to note that this implementation is not set up to work for a multi threaded program. The implementation of the memoization pattern can be found in the

CatscriptType.java file (src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java) on line 37.

## **4 Technical writing**

This is the documentation provided by my group partner Darshan Diamond:

## 4.1 Introduction

## 4.2 Features

### Comments

Comments are used using `//` for single comments and `/* */` for multiple line comments

```
/*
 * This is the way to do a block comment
 * */
var x = "foo"
// this is a single line comment
print(x)
```

## 4.3 Types

Catscript has basic types for variable. These types can be both explicitly declared and inferred. Catscript also has a shared namespace, variable names from all fields and sub-scopes are stored in the same location.

A list of Catscript' type system:

- `Int` { a 32 bit integer
- `string` { string object
- `bool` { boolean value
- `null` { a null type
- `object` { any value

```
var x = 5
var y = true
var str = "Catscript!"
```

### 4.3.1 Function Definitions

Functions are defined by including the keyword `function` before your identifier and parameters. The return type is void, unless otherwise stated.

### 4.3.2 For loops

The `in` keyword is used in Catscript to iterate through contents in an array as shown:

```
var list = ["a", "b", "c", "d"]
for(i in list){
```

```
print(i)
}
```

The `i` gets assigned to each value as it iterates through the array. This will evaluate to `abcd`.

### 4.3.3 Mathematical Operations

#### Addition & Subtraction

Catscript supports basic addition and subtraction between integer types. In addition, string concatenation may be performed if either side of an additive expression is of type string.

```
print(5+6) //evaluates to 11
print(5-6) //evaluates to -1
print(5+"foo") //evaluates to "5foo"
```

#### Division and Multiplication

Catscript supports division and multiplication between integer types.

```
var x = 5
var y = 6
print(x*y) // evaluates to 30
print(x/6) // evaluates to 0
```

### 4.3.4 Unary Operators

Catscript supports two primary unary operators, integer negation and the boolean "not" operator.

```
var x = 5
print(-x) //print's "-5"
not false // evaluates to true
```

### 4.3.5 Comparison

Catscript uses the basic comparison operators less than, less than or equal, greater, greater than or equal, as shown below:

```
1 < 0 // false
1 <= 0 // false
1 > 0 // true
1 >= 0 // true
```

#### 4.3.6 If statements

If statements in Catscript identical to Java. Using the if, else if, and else format, the user uses boolean literals or expressions that evaluate to a boolean as well as comparison operators to evaluate if statements.

```
var x = "yes"
if(x == "yes"){
  print("accepted")
}
else if(x == "no"){
  print("denied")
} else {
  print("who are you")
}
```

The above returns the string "accepted"

#### 4.3.7 Function Definitions

A function may be defined by using the keyword "function" followed by an identifier and parameters. Function declaration statements also feature type inference. The return type is assumed to be void, unless defined otherwise following the parameters

##### Type inferred

The following function declaration creates a function named "foo", makes use of type inference for its arguments, and features a default void return type.

```
function foo(myint, mystring, mylist){
//statements
}
```

##### Explicitly Typed

The following function declaration creates the same function, but uses explicit types for its parameters, and defined a return type of string.

```
function foo(myint: int, mystring: string, mylist: list<object>) : string {
//statements
  return mystring
}
```

#### 4.3.8 Function Call Expressions

A function may be called by invoking the functions name, and passing in any necessary parameters. The function call statement will then return any object which is returned from the function. In the following example, the function foo

is invoked with the integer value 5. The function then returns the value 5, which is assigned to the variable `y`.

```
function foo(x: int) : int{  
    return x  
}  
var y = foo(5)
```

## 5 UML

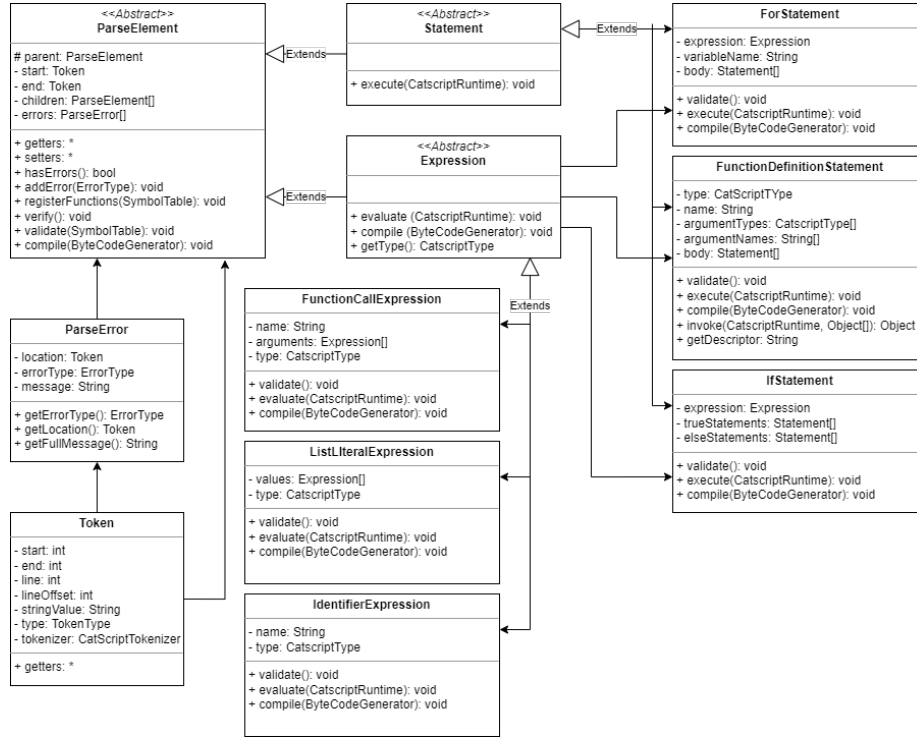


Figure 1: A UML diagram showcasing the relationship of the various parse elements, specifically highlighting prominent aspects of the architecture for clarity.

## 6 Design trade-offs

Due to the nature of the project most if not all major design decisions were decided before students were given the project. While I did not have a direct hand in the decision making process these design choices had a major effect on how the project was completed. There were two important decisions made about the design of the project; the use of a recursive decent parser over a parse generator and the choice not to use a visitor pattern to separate the different sections of the code.

Recursive decent was chosen because it is simpler. Over all it imparts a better understanding of the recursive nature of grammars, both due to its easier to grasp structure and the more hands on nature of coding it. However, because coding a recursive decent parser is more hands on there is more code, and more code that has to be written instead of generated. Additionally by using a recursive decent parser we are not exposed to the far more common parse generators which are standard in many universities.

While usually one would use a visitor pattern to separate the different parts of the code, it was forgone for this project. By embedding everything directly into the parse tree nodes the code base was easier to implement and navigate as a student. However, by having the code all in one place it goes against the popular idea of keeping different sections of code separate, separation of concerns. For a project like this I believe it was a smart choice and promoted the hands on learning suggested by the use of a recursive decent parser.

## **7 Software development life cycle model**

For this project we used Test Driven Development (TDD). TDD involves being given tests and coding with a focus on passing the tests you are given. At the start of the project we were given a repository with over 100 tests to get passing by the end of the semester. For me, this was a fantastic model. It gave a clear and achievable direction to work that provided immediate feedback in the form of passed or failed tests.