

CSCI 468 - Compilers  
Spring 2021  
Semester Portfolio

Cole Brooks  
Charlie Melnarik - Tester

# 1 Program

Please find the codebase for the semester project as well as the specifications in the form of test cases in the attached "source.zip" file.

# 2 Teamwork

Within this section (and the remainder of the portfolio), Cole Brooks will be referred to as Member 1, and Charlie Melnarik will be referred to as Member 2. Member 1 contributed the codebase, i.e., the implementation of the programming language "Catscript," and Member 2 contributed additional test cases to aid in the Test Driven Development life cycle model. Of the estimated 80 hours spent on the implementation of Catscript, Member 1 contributed an estimated 90 percent of the time, with Member 2 making up the remaining 10 percent.

# 3 Design Pattern

The design pattern used in the implementation of Catscript is the Memoization method, and can be found in the static method `getListType()` in the class `CatscriptType.java`. The Memoization design pattern is used to ensure that expensive methods are not executed more than is necessary, by storing their return values in a static lookup table to be referenced afterward. An argument to the method can be used as the key in the lookup table, with the method's return value being used as the value for the (*key*, *value*) pair. The end result is that rather than running a potentially expensive computation multiple times, it can be run only once, and then substituted with a simple lookup operation.

# 4 Technical Writing

# Catscript Documentation

Cole Brooks

## 1 Introduction

Catscript is a small, statically typed programming language that compiles to JVM Bytecode. It features a recursive descent parser, type inference, objects and primitives, as well as several other interesting features which will be covered throughout the documentation.

## 2 Expressions

Catscript features an abstract *Expression* class from which all other expression types inherit. An expression is some piece of code that eventually evaluates to some value, such as a number, a string, or a boolean. The abstract *Expression* class features an abstract method called *getType()*, which is implemented in the expression classes, and returns the type that an expression will eventually evaluate to.

### 2.1 Primary Expression

The base expression is the *Primary Expression*, which can resolve to any of the following:

- Identifier
- String literal
- Integer literal
- Boolean literal
- Null literal
- List literal
- Function call
- Parenthesized expression

Any expression or statement containing another expression will eventually resolve to some form of a primary expression.

## 2.2 Identifier Expression

The *Identifier Expression* is how Catscript implements variables. The identifier expression holds a string to serve as the variable name, and also holds information on the variable type, as Catscript is a statically typed language. Catscript's type system will be covered in more detail below. The variable name is then used to look up its corresponding value in the symbol table.

## 2.3 Unary Expression

The *Unary Expression* is how Catscript implements the *not* and *negative* operators, which are “not” and “-” respectively. For example, all of the following are valid Catscript unary expressions:

---

```
-1  
not true  
not not foo
```

---

## 2.4 Factor Expression

The *Factor Expression* is how Catscript implements the multiplication and division operations, which are “\*” and “/” respectively. Below are some examples of valid Catscript factor expressions:

---

```
foo * -1  
25 / -5  
(foo - bar) / 2
```

---

## 2.5 Additive Expression

The *Additive Expression* is how Catscript implements the addition and subtraction operations, which are “+” and “-” respectively. Additionally, the “+” operator is overloaded to support string concatenation. Below are some examples of valid Catscript additive expressions:

---

```
x + 5  
5 - 2  
foo + "bar"
```

---

## 2.6 Comparison Expression

The *Comparison Expression* is Catscript's implementation of the relational operations: strictly greater than (“>”), strictly less than (“<”), greater than or equal to (“>=”), and less than or equal to (“<=”). Below are some examples of valid Catscript comparison expressions.

---

```
foo > 5  
foo + 5 > 10  
foo / 2 <= bar * 2
```

---

## 2.7 Equality Expression

The *Equality Expression* is Catscript's implementation of the equal to (“==”) and not equal to (“!=”) operations. Below are several examples of valid Catscript equality expressions:

---

```
foo == bar
foo == 5
x != y
foo - bar == x * y
```

---

## 2.8 Expressions Conclusion

This has been a brief overview of the expressions system Catscript uses to manipulate *values*. The next section documents the statement system that Catscript uses to induce *side effects*.

# 3 Statements

In the same way that all Catscript expressions inherit from an abstract *Expression* class, all Catscript statements inherit from an abstract *Statement* class. As mentioned above, while Catscript expressions evaluate to a value of some kind, Catscript statements instead invoke some sort of side effect in the program, i.e, they change the program's state.

## 3.1 Print Statement

One of the most basic statements is the *Print Statement*. The print statement is called with the “print” keyword, followed by some sort of expression. After evaluating its expression, the print statement then prints that value to the program's standard output stream. The print statement may only have expressions passed to it, as they provide some value to be printed. A statement does not necessarily provide a value, and thus is not a valid input to the print statement. Below are some examples of valid Catscript print statement calls:

---

```
print("foo")
print(foo)
print(foo + "bar")
```

---

## 3.2 Variable Statement

The *Variable Statement* is used to declare and assign new variables. Note that it is not possible in Catscript to declare a new variable without also providing an expression to be assigned as a value. A variable is declared/assigned as follows:

1. A required “var” keyword
2. A required variable name, or identifier

- (a) An optional “:” symbol followed by an explicit type
3. A required “=” symbol
4. A required expression

Should the programmer choose to specify an explicit type, the program verifies that the explicit type is assignable from the type returned by the expression’s *getType()* method. If not, the program throws an *Incompatible Types* error. If the programmer chooses not to specify an explicit type, the variable type is inferred from the expression’s *getType()* method. Below are some examples of valid Catscript variable statements:

---

```
var foo : int = 1
var foo : object = 1
var foo = 1
```

---

### 3.3 Assignment Statement

Related to the *Variable Statement* is the *Assignment Statement*. The *Assignment Statement* is used to change the value of an existing variable. The syntax for the assignment statement is as follows:

1. A required identifier<sup>1</sup>
2. A required “=” symbol
3. A required expression<sup>2</sup>

Below are several examples of valid Catscript assignment statements:

---

```
foo = "bar"
foo = bar
foo = 2 + 2
```

---

### 3.4 If Statement

The *If Statement* in Catscript is very similar to the if statement in other programming languages such as Java or C. It is used to determine whether or not to execute a series of statements. A Catscript if statement is invoked as follows:

1. A required “if” keyword
2. A required “(“ symbol

---

<sup>1</sup>Note that the program verifies that the identifier has already been registered in the symbol table as a variable name. If there is not a variable of the same name already registered, the program throws an *Unknown Name* error.

<sup>2</sup>Note that if the variable type that is registered in the symbol table is not assignable from the type of the assignment expression, the program throws an *Incompatible Types* error.

3. A required expression to evaluate<sup>3</sup>
4. A required “)” symbol
5. A required “{” symbol
6. A required series of statements to execute
7. A required “}” symbol
8. An optional “else” keyword
  - (a) An optional if statement or
  - (b) A required “{” symbol
  - (c) A required series of statements to execute
  - (d) A required “}” symbol

Below is an example of a valid Catscript if statement, featuring the optional else if construct:

---

```
if (foo == "bar") {  
    print("bar")  
} else if (foo == "foo") {  
    print("foo")  
}
```

---

### 3.5 For Statement

The syntax of the *For Statement* is more similar to the for statements of Python, in that it uses an “in” keyword. Currently the for statement only supports iterating through a list, there is no provision for counting up to some value, as a programmer would in Java with

```
int i = 0; i < 5; i++
```

or as they would in Python with

```
i in range 5
```

Catscript’s for statement is constructed as follows:

1. A required “for” keyword
2. A required “(” symbol
3. A required variable name, or identifier
4. A required “in” keyword

---

<sup>3</sup>Note that the program verifies whether the expression evaluates to a boolean. If it does not, an *Incompatible Types* error is thrown.

5. A required expression<sup>4</sup>
6. A required “)” symbol
7. A required “{” symbol
8. A required series of statements to evaluate
9. A required “}” symbol

Below is an example of a valid Catscript for statement:

---

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

---

### 3.6 Function Definition Statement

The *Function Definition Statement* is used to define functions that can be called elsewhere in the code. Functions can have an explicit return type, or if no explicit return type is declared, the function is of type *void*. If an explicit return type is declared, the program verifies return coverage, and that the type of the expression to be returned matches the function’s explicit return type. Additionally, when declaring a function’s parameters, the programmer may choose to give each parameter an explicit type. If they choose not to do so, the parameters default to type *object*. The parameter list is used only for declaring function parameters, and consists of a list of identifiers, each followed by an optional “:” symbol and parameter type. A Catscript function can be constructed as follows:

1. A required “function” keyword
2. A required identifier to serve as the function name
3. A required “(” symbol
4. A parameter list<sup>5</sup>
5. A required “)” symbol
6. An optional “:” symbol followed by the function’s return type
7. A required “{” symbol
8. A series of statement to be executed
9. A required “}” symbol

Below is an example of a valid Catscript function definition:

---

<sup>4</sup>Note that the program verifies whether the expression is a list or not. If not, it throws an *Incompatible Types* error.

<sup>5</sup>Note that the parameter list is not required if the function doesn’t need to take any arguments.



---

```
function foobar (foo : int, bar : int) : int {  
    return foo + bar  
}
```

---

### 3.7 Function Call Statement

Once a function had been defined, a programmer can then call it with the *Function Call Statement*. The program verifies that the number and type of arguments match those defined in the function's definition. Additionally, the program verifies that there is a function with the same name that has been registered in the symbol table. The function call can be constructed as follows:

1. A required function name
2. A required “(” symbol
3. An argument list
4. A required “)” symbol

Below is an example of a valid Catscript function call:

---

```
foobar (foo, bar)
```

---

### 3.8 Return Statement

The final statement is the *Return Statement*, which is used to exit a function, and (if necessary) return a value. Note that the program ensures a return statement can only be executed from within a function. If a return statement is parsed outside of a function, the program throws a *Syntax* error. A return statement is structured as follows:

1. A required “return” keyword
2. An expression to be returned<sup>6</sup>

Please see Section 3.6 for an example of a return statement within a function definition statement.

### 3.9 Statement Conclusion

This has been a comprehensive overview of Catscript's most interesting features, the statements. The next and final section details the Catscript type system.

---

<sup>6</sup>Note that this expression is only valid if the function has an explicit return type.

## 4 Type System

Catscript features a simple type system. The types are as follows:

- `int` - A 32 bit integer
- `string` - a Java-style string
- `bool` - a boolean value
- `list<x>` - a list of values with type “x”
- `null` - the null type
- `object` - any type of value

Note that Catscript types have a function called *isAssignableFrom()*, which works within the type hierarchy. Any type is assignable from `null`, and an `object` is assignable from any type. `String`, `bool`, and `int` are all of equal status.

## 5 UML

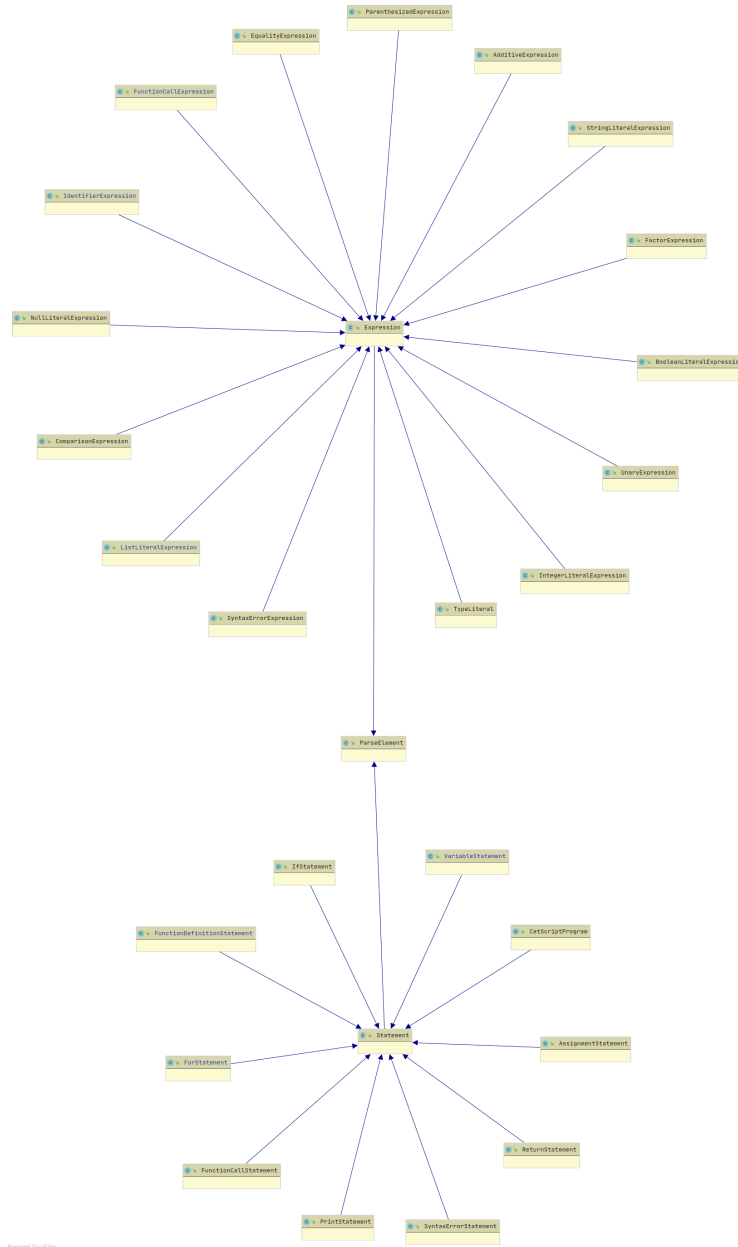


Figure 1: UML diagram of Parse Elements

This UML diagram shows the hierarchy of the Parse Elements in Catscript: Statements and Expressions. The base class is an abstract class called `ParseElement.java`. Both the abstract classes `Expression.java` and `Statement.java` inherit from `ParseElement.java`. Finally, each specific expression and statement, for example `FunctionCallStatement.java` and `BooleanLiteralExpression.java` inherits from their respective parent, either `Statement.java` or `Expression.java`.

## 6 Design Trade-offs

The main design trade-off within the implementation of Catscript was the choice to implement the parser with the recursive descent algorithm, rather than the more typical choice of a parser generator. This decision was made for several reasons:

- Recursive descent is widely used in the industry
- Recursive descent mirrors the recursive nature of grammars
- Recursive descent is easy to implement, modify, and maintain

While many other computer science programs teach compilers using a parser generator, most real world programming languages such as Java and C# are implemented with the recursive descent algorithm. Additionally, many well known parser generators actually generate a recursive descent parser, with one of their drawbacks being that the code they output can be much harder to debug or modify in the future. Because of the ease of translation from the Catscript grammar to the recursive descent parser, any potential gains from the use of a parser generator are offset by the choice to implement the Catscript parser with the recursive descent algorithm.

## 7 Software Development Life Cycle Model

The Software Development Life Cycle Model used in this course during the development of Catscript was Test Driven Development. Test Driven Development (TDD for short) is a software development process in which test cases are written before starting the implementation of required features, giving the programmer a pass/fail case to test their code against. When a new feature is to be added to the project, the first step is to write a test case that passes if and only if the required feature's specifications are met. Before implementation of the feature, the test should fail. Code that passes the test is then written. The preliminary version of passing code is likely not an elegant solution, but the benefit to the

programmer is a vast improvement in their understanding of the problem. With this added understanding, the code can then be refined while maintaining its passing status.