

CSCI 468 - Spring 2022 Portfolio

Zach Wadhams

Kai Dockens - Tester

Section 1: Program: Codebase zip file: (Will also be attached in submission with this document inside)

https://drive.google.com/file/d/1t5yphEq7e6nuAoQEw6OBGnV-SSYiDp_i/view?usp=sharing

Section 2: Teamwork:

Although this course was primarily individual based, for some final testing documentation ideas I worked with Kai Dockens as my capstone partner. Kai provided three tests that can be found in PartnerTestsKai.java in the above codebase. These three tests examine my Catscript implementation's functionality with return statements within functions, functions that compare greater than statements, and examine scoping. These three tests passed when they were given to me, indicating that this functionality was implemented correctly.

Section 3: Design Pattern:

This Catscript implementation utilizes the Memoization method as its design pattern. This can be found in the class 'CatscriptType'. This pattern is used primarily to ensure that methods are not run more than absolutely necessary. It accomplishes this by storing the values they would return in a lookup table. This lookup table can be searched by providing a key as an argument. This saves time and resources by storing values rather than running methods repeatedly.

Section 4: Technical Writing / Documentation

■ CSCI468Documentation.pdf

Catscript Documentation

Zach Wadhams, Kai Dockens

5/6/2022

1 Introduction

Catscript is a statically typed programming language that compiles to JVM Bytecode. It utilizes a recursive descent parser and can work with type inference, objects and primitives, as well as other features discussed in this documentation.

2 Expressions

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

2.1 Primary Expression

The base expression is the Primary Expression, which may be any of the following:

- Identifier
- String literal
- Integer literal
- Boolean literal

- Null literal
- List literal
- Parenthesized expression
- Function call

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

2.2 Identifier Expression

The Identifier Expression is how Catscript implements variables. The identifier expression holds a string as the variable name, along with the variable's type due to the fact that Catscript is a statically typed language. The variable name is used to look up its corresponding value in the symbol table.

2.3 Unary Expression

The Unary Expression is the way Catscript implements the not and negative operators, which are 'not' and '-'. The following are valid Catscript unary expressions:

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

2.4 Factor Expression

The *Factor Expression* is the way Catscript implements the multiplication and division operations, which are '*' and '/'. The following are valid Catscript factor expressions:

```
foo * -1
10 / -5
(foo - bar) / 5
```

2.5 Additive Expression

The Additive Expression is the way Catscript implements the addition and subtraction operations, which are '+' and '-' respectively. The '+' operator is also overloaded to support string concatenation. The following are valid Catscript additive expressions:

```
x + 6
3 - 2
foo + "bar"
```

2.6 Comparison Expression

The Comparison Expression is the way Catscript implements the relational operations: strictly greater than '>', strictly less than '<', greater than or equal to '>=', and less than or equal to '<='. The following are valid Catscript comparison expressions.

```
foo > 2
foo + 3 > 10
foo / 2 <= bar * 3
```

2.7 Equality Expression

The Equality Expression is the way Catscript implements the equal to '==' and not equal to '!=' operations. The following are valid Catscript equality expressions:

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

2.8 Expressions Conclusion

This was an overview of how Catscript implements and uses expressions. Next, we will discuss Catscript Statements.

3 Statements

In the same way that Catscript expressions inherit from an Expression class, Catscript statements also inherit from a Statement class. While Catscript expressions evaluate to a value, Catscript statements instead change the program's state.

3.1 Print Statement

A very basic statement is the Print Statement. The print statement is called with the 'print' keyword, followed by an expression. After evaluating the expression, the print statement then prints the value to the program's standard output. The print statement must have only expressions passed to it. A statement does not need to give a value, and that makes it not a valid input to the print statement. The following are 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 a new variable. It is not possible in Catscript to declare a new variable without also giving an expression to be assigned as a value. The following demonstrates what is required for a Catscript variable declarations and assignments:

1. A required 'var' keyword
2. A required variable name
3. A required '=' symbol
4. A required expression

If a type is specified, Catscript verifies that the type is assignable from the type meant to be returned by the expression's `getType()` method. If not, the program throws an Incompatible Types error. If a type is not specified,

the variable type is inferred from the expression's `getType()` method. The following are valid Catscript variable statements:

```
var foo : int = 3
var foo : object = 5
var foo = 2
```

3.3 Assignment Statement

The Assignment Statement is used to modify the value of an existing variable. This syntax for a Catscript assignment statement is as follows:

1. A required identifier
2. A required '=' symbol
3. A required expression

The following are valid Catscript assignment statements:

```
foo = "bar"
foo = bar
foo = 1 + 3
```

3.4 If Statement

The Catscript If Statement is nearly identical to the if statement in other programming languages such as Java. It is used to determine whether or not to run the statements below it. Catscript's if statement is created as follows:

1. A required 'if' keyword
2. A required '(' symbol
3. A required expression
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 run
 - (d) A required '}' symbol

The following is a valid Catscript if statement:

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

3.5 For Statement

The syntax of the For Statement in Catscript uses an 'in' keyword. The for statement only supports iterating through a list. It cannot count up to a certain value

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

or as they would in Python with

```
i in range 5
```

Catscript's for statement can be created as follows:

1. A required 'for' keyword
2. A required '(' symbol
3. A required variable name
4. A required 'in' keyword
5. A required expression

6. A required ')' symbol
7. A required '{' symbol
8. A required series of statements to evaluate
9. A required '}' symbol

The follow is a valid Catscript for statement:

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

3.6 Function Definition Statement

The Function Definition Statement is used to define functions that can be called elsewhere. Functions may have an explicit return type, or return void if no type is specified. If a return type is declared, the program checks that it does return that specific type. The parameter list is to declare function parameters as a list of names, each followed by an optional ':' symbol and type. Catscript's function can be created as follows:

1. A required 'function' keyword
2. A required function name
3. A required '(' symbol
4. A parameter list
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 evaluate
9. A required '}' symbol

The following is a valid Catscript function definition:

```
function ab (a : int, b : int) : int {  
    return a + b  
}
```

3.7 Function Call Statement

Once a function had been defined, it can be called with a Function Call Statement. The program verifies that the number and type of arguments match the function's definition. The program also checks that there is a function that exists with the same name that has been added in the symbol table. Catscript's function call can be created as follows:

1. A required function name
2. A required '(' symbol
3. A list of arguments
4. A required ')' symbol

Below is an example of a valid Catscript function call:

```
ab(a, b)
```

3.8 Return Statement

The Return Statement is used to exit a function, and potentially return a value. It can only be ran within a function. If a return statement is found outside of a function, the program throws a Syntax error. Catscript's return statement is created as follows:

1. A required 'return' keyword
2. An expression to be returned

Below is an example of a valid Catscript return statement inside of a function:

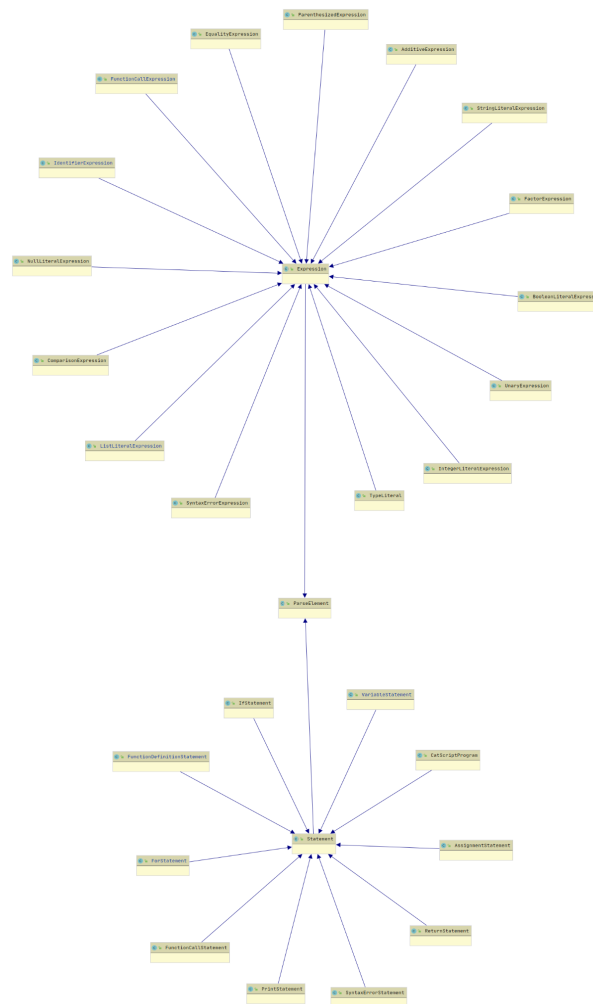
```
function ab (a : int, b : int) : int {  
    return a + b  
}
```

4 Type System

Catscript utilizes a simple type system. The following are valid types:

- int - Any 32 bit integer
- string - a Java-similar string
- bool - a boolean value (True/False)
- list<x>- a list of values with type "x" (must be same type)
- null - the null type
- object - any type of value

Section 5: UML: None needed because design was selected by professor. However, I will Include and discuss one below for reference.



This specific UML diagram shows how the different types of expressions are connected to the different types of statements in Catscript. At the center of the top larger balloon is the general 'Expression' class that has all other types of expression connected to it. Some of these include 'AdditiveExpression', 'FactorExpression', and 'ListLiteralExpression'. The center 'Expression' class is also connected to 'ParseElement' which itself is connected to the lower smaller balloon with 'Statement' in its center. The 'Statement' class is connected to all the different types of statements contained within Catscript. Some of these are 'ForStatement', 'IfStatement', and 'PrintStatement'.

Section 6: Design Trade-offs

The biggest design trade-off was the fact that Catscript implements a parser with a recursive descent algorithm rather than a parser generator. The main reason this was chosen was because recursive descent closely mirrors the recursiveness of grammars. It was also chosen because it is fairly easy to implement and modify quickly. By using recursive descent, I felt that I got a more in depth understanding of how recursion works in a large program such as Catscript. I believe it was also useful because it helped me get a better understanding of how some real world languages work such as C# and Java that implement recursive descent themselves.

Section 7: Software Development Life Cycle

For Catscript, a Test-Drives Development (TDD) model was used. TDD is a software development process in which test cases are written before implementing required features, which gives the programmer a pass or fail case to test their code. When new features are to be added to the project, the first step is to write a test case that passes only if the required feature's specifications are met. This test should always fail before the feature is implemented. The programmer then writes code to pass this test. Although the first solution is most likely not the most efficient, the code can then still be refined while checking to see if the test is still passing.