

Compilers Capstone

CSCI 468 - Spring 2022

Bryndon Wilkerson (Team member 1)

Abbey Hustis (Team member 2)

Section 1: Program

/capstone/portfolio/source.zip

Section 2: Teamwork

Team member 1 did a lot of the coding to get the tests passing and team member 2 worked on this project as a tester. Since this project utilizes test driven development (TDD), a lot of this code already has tests to ensure that it works properly. Team member 2 was able to find some edge cases that the tests did not already cover and submitted them to me so that team member 1 could refine the code to ensure that this project is robust. Additionally, team member 2 wrote up the documentation describing the Catscript compiler and the many features that it contains.

Section 3: Design pattern

We used memoization in lines 35-43 of CatscriptType.java. We decided to cache the Catscript types to reduce the time that it takes to return a CatscriptType. Since there is a limited selection of Catscript types, the memory cost shouldn't be an issue.

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

Section 4: Technical writing.

Catscript Guide

This document should be used to create a guide for catscript, to satisfy capstone requirement 4

Introduction

Catscript is a simple scripting language. It has many similarities to Python and Java in its methods, expressions and statements. Catscript gets compiled down to ByteCode which is then run on the JVM. Here is an example:

```
var x = "foo"
print(x)

>> foo
```

Features

Statements

For loops

The `for` loop is used to iterate over a section of code a specific number of times. You can iterate over a sequence of numbers, list elements or a string, which updates the value of the identifier each loop through and then will execute the statements that are in the body of the `for` loop.

```
for (i in [4, 6, 2, 9]) {
  print(i)
}
>> 4
>> 6
>> 2
>> 9
```

If Statements

The `if` statement is used to conditionally execute a section of code. The expression is evaluated and will return a True or False which will decide if the code within the body of the `if` statement will be executed or not. You also have the option to have multiple conditionals in the same section of code in the form of `else if` statements. Finally, you can have a default case if the initial expression evaluates to False by using an `else` statement.

```
if (x < 487) {
  print("That's a small number")
} else {
  print(x + "is bigger than 487")
}
```

Print Statements

The `print` statement is used to output any expression given to the method by writing it out.

```
var sun = "cat"
print(sun)
>> cat
```

Variable Statement

A `variable` statement allows you to store some value to a variable name and then be able to reference later in the program. In Catscript, you are given the option to either explicitly or implicitly state the variable type.

```
var moon : string = "dog"
```

Function Declaration Statement

A function declaration statement associates the function identifier that is given to the function body – of executable statements – that is also defined here. The function has the ability to either return a value to where the function was called or to return nothing a simply execute the statements that are in the function body.

```
function foo(tree : string, leaf : string) {  
    var moon : string = tree + leaf  
    return moon  
}
```

Function Call Statement

A `function call` statement calls a reference to a predefined function. The function call will send in the arguments, and they will be used to evaluate the body of the function. A function call statement can be set to a variable if the function that is being called returns a value, otherwise the function calls are just used to evaluate the called function.

```
var x = foo("moon", "dog")
```

Assignment Statement

An assignment statement redefined a preexisting variable. This can be used to update or completely overwrite the existing value that is currently being stored in the variable.

```
moon = "cat"
```

Return Statement

A `return` statement is what allows a function to output a value back to where the function was called. This allows for some computation to occurs and to store the resulting value to be used later.

```
return moon
```

Expressions

Equality Expression

The equality expression will evaluate the arguments that are given determine their equivalence and return True or False based on the equality that is being implemented.

```
6 == 9  
>> False
```

Comparison Expression

The comparison expression will evaluate the arguments that are given and determine whether the stated comparison evaluates to True or False.

```
4 <= 33  
>> True
```

Additive Expression

The additive expression will return the addition, or the subtraction of the arguments that are given to the expression. Additive expression not only compute mathematical addition but also string concatenation.

```
44 + 68  
>> 112  
  
"sun" + "cat"  
>> "suncat"
```

Factor Expression

The factor expression will return the multiplication, or the division of the two expressions that were given. The division that is completed is integer division meaning that it will round down to the closest integer.

```
13 / 4  
>> 3
```

Unary Expression

The unary expression will return the negation of the expression given.

```
var moon = True

not moon

>> False
```

Primary Expression

The primary expression is the base expression that is used to fill in the variables and the values of the expressions higher up in the grammar. These expressions consist of the type literals of Catscript – int, string, bool etc. – as well as variable identifiers and function calls.

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

List Literal

A list literal is one type of object in the Catscript language. It acts like a list that can be made up of the other type literals that Catscript supports including another list.

```
var temp : list <int> = [4, 33, 42]
```

Catscript Types

Catscript has five different types literals int, string, bool, object and a list literal.

- int - a 32 bit integer
- string - a java-style string
- bool - a boolean value
- list - a list of value with the type 'x'
- null - the null type
- object - any type of value

Complete CatScript 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, '>']

```

Section 5: UML.

CatScriptServer is the main driver for this project. Running this gives the UI that the user can interact with. The CatScriptTokenizer scans all of the input and breaks it down into tokens. The CatScriptParser parses through all of the expressions and statements and catches errors. Eventually, all of the ParseElements (both Statements and Expressions) compile down to bytecode and are added to the JVM. CatScriptProgram is an example of a class that extends Statement.

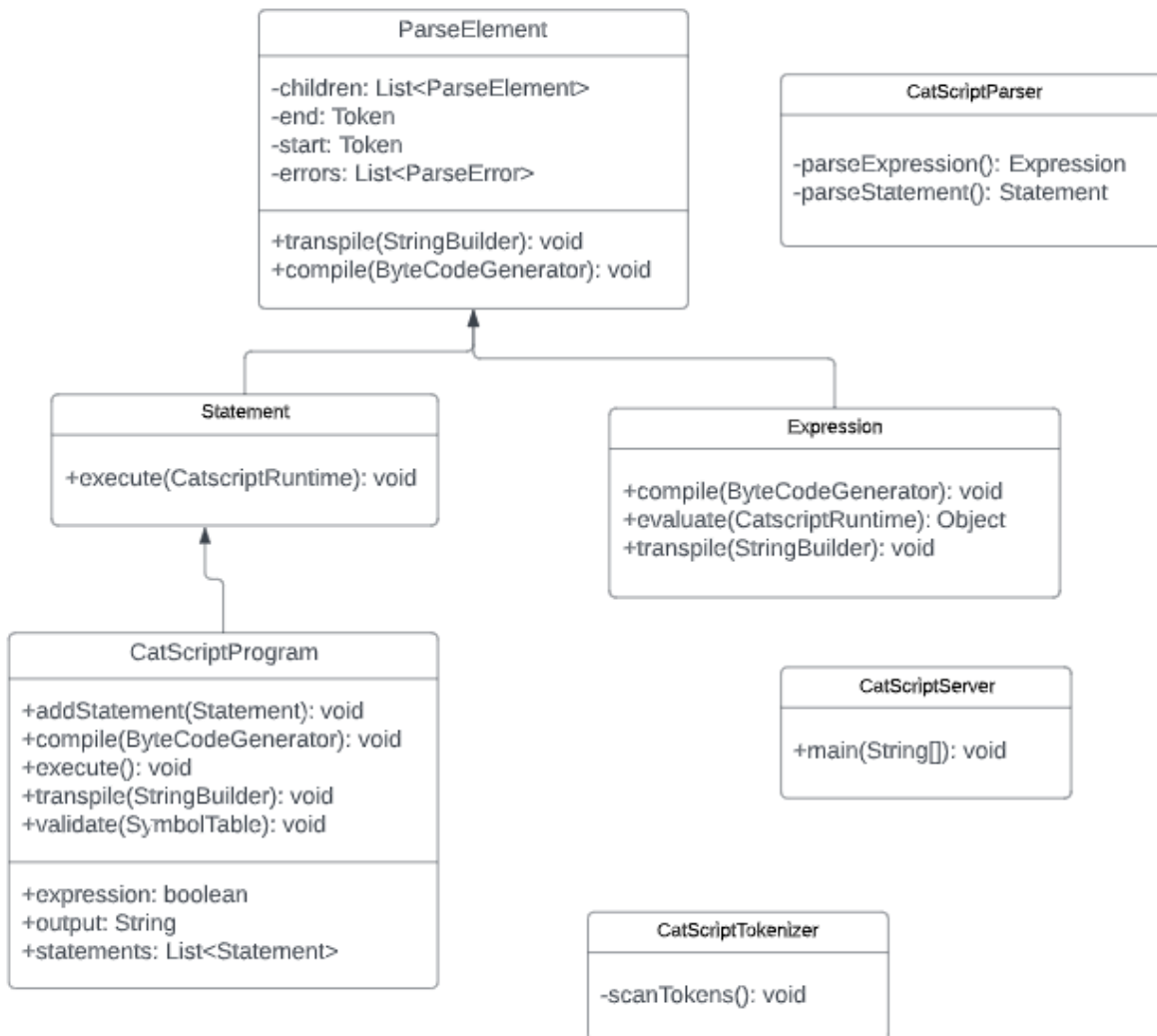


Figure 1

Section 6: Design trade-offs

We use a recursive descent parser rather than using a parser generator. This is because a recursive descent parser is simpler than a parser generator. Also, the recursive descent parser allows for a better understanding of the recursiveness of grammar. One drawback to using a recursive descent parser is that it does require quite a bit more code than the parser generator. Another drawback is that parser generators are more standard than recursive descent parsers.

Section 7: Software development life cycle model

We are using Test Driven Development (TDD) for this project. This model allowed us to set quantifiable goals throughout the semester. Also, more bugs and errors were caught earlier on in the coding process because we were able to test specific parts of the code. With TDD, we were able to visualize the various parts of the code that needed to be done. For example, we knew that in order to get the parsing part of the project done, we needed to have the tokenizing done first.

We did struggle with the TDD process a bit because we assumed that if we got all of the tests to pass, the code would be robust and sound. Occasionally, there were bugs that got through certain phases of the project because we thought we had finished it properly. These bugs would show up in later tests and we would have to go back to code that we thought we were finished with to fix these problems.

However, overall I do believe that TDD is a great model to follow because it at least sets a standard of what the code needs to have, and then further manual testing allows for truly robust code.