

Capstone Portfolio

CSCI 468 Compilers

Spring 2022

Armand LaPlume

Brian Keith

Section 1: Program

Included in this directory is a zip file of the project titled source.zip

Section 2: Teamwork

Team member 1 was the primary coder. Team member 2 assisted with testing. Team member 1 wrote code for the tokenizer, parser, evaluation, and the bytecode for the project. Team member 2 assisted by writing tests and assisting with debugging. Additionally team member 2 wrote the documentation for Catscript.

Section 3: Design Pattern

The memoization design pattern is used in Catscript. It is used in the CatscriptType class in the getListType method which is located at lines 36 to 44 in CatscriptType.java. Without memoization it would create a new ListType every time and return it which is inefficient. The memoized version utilizes a hashmap to first check if a ListType has been created for a given CatscriptType and if it has not been created then it creates a new ListType, stores it in the hashmap, and returns it. If it has been created then it returns the ListType from the hashmap.

```
//This has been memoized
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;
}
```

(memoized pattern in CatscriptType.java)

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. Here is an example:

...

```
var x = "foo"
```

```
print(x)
```

...

Features

Types

Catscript has 5 types : Int, String, Bool, Obj, and List<>\

grammar:

...

type_expression = 'int' | 'string' | 'bool' | 'object' | 'list'

[, '<' , type_expression, '>']

...

example:

...

var x : int = 5

var y : List<int> = [1,2,3]

...

Expressions

Catscript has many expressions to choose from and uses a recursive descent tree to organise them

####equality and comparison Expressions

the equality and comparison expressions are used to compare two objects of the same type and will always return a boolean

Grammar:

...

equality_expression = comparison_expression { ("!=" | "==") comparison_expression };

comparison_expression = additive_expression { (">" | ">=" | "<" | "<=") additive_expression };

...

Examples:

...

Bool x = (val1 >= val 2)

if (bool1 == bool2)

...

####additive and factor Expressions

the additive and Factor expressions are used to perform basic arithmetic and string concatenation

Grammar:

...

additive_expression = factor_expression { ("+" | "-") factor_expression };

factor_expression = unary_expression { ("/" | "*") unary_expression };

...

Examples:

...

Int x = 5 + 6

Int y = 6 / 3

String foo = "bar" + "foo"

...

####unary and primary Expressions

unary and primary expressions are the lowest level of expressions that generally represent direct values

Grammar:

...

unary_expression = ("not" | "-") unary_expression | primary_expression;

primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null" |

list_literal | function_call | "(" , expression , ")"

...

Examples:

...

-15

true

"foo"

...

###Statements

####Variable Definition

Variable definition is very simple, it always starts with the keyword var and the type is identified by a colon

Grammar:

...

variable_statement = 'var', IDENTIFIER, [':', type_expression,] '=', expression;

...

Examples:

...

var x : int = 5

var y : List<int> = [1,2,3]

...

####Function Declaration and call

Function Declaration and calls are very straight forward. they are preceded by the key word "function" then use parenthesis for parameters and a colon for return type. functions can return any catscript type including void

Grammar:

...

function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
[':' + type_expression], '{', { function_body_statement }, '};

function_body_statement = statement |
return_statement;

function_call = IDENTIFIER, '(', argument_list , ')'

...

Example:

...

```
function add5(x : int) : int {  
  return x + 5  
}
```

...

####for loop

Catscript has a basic for loop that iterates through a set of objects

Grammar:

...

```
for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',  
                '{', { statement }, '}';
```

...

Example:

...

```
for ( i in [1,2,3]) {  
  print (i)  
}
```

...

####if statement

the if statement in catscript is very standard and supports else and else if

Grammar:

...

```
if_statement = 'if', '(', expression, ')', '{',  
               { statement },  
               '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];
```

...

Example:

...

```
if(x > 10){  
  print(x)  
} else {  
  print("Too smol")  
}
```

```
...
```

```
####print statement
```

the print statement can print strings or object values

Grammar:

```
...
```

```
print_statement = 'print', '(', expression, ')'
```

```
...
```

Example:

```
...
```

```
print("x: " + x)
```

```
...
```

```
##Grammar
```

this is the 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 } ']' ;

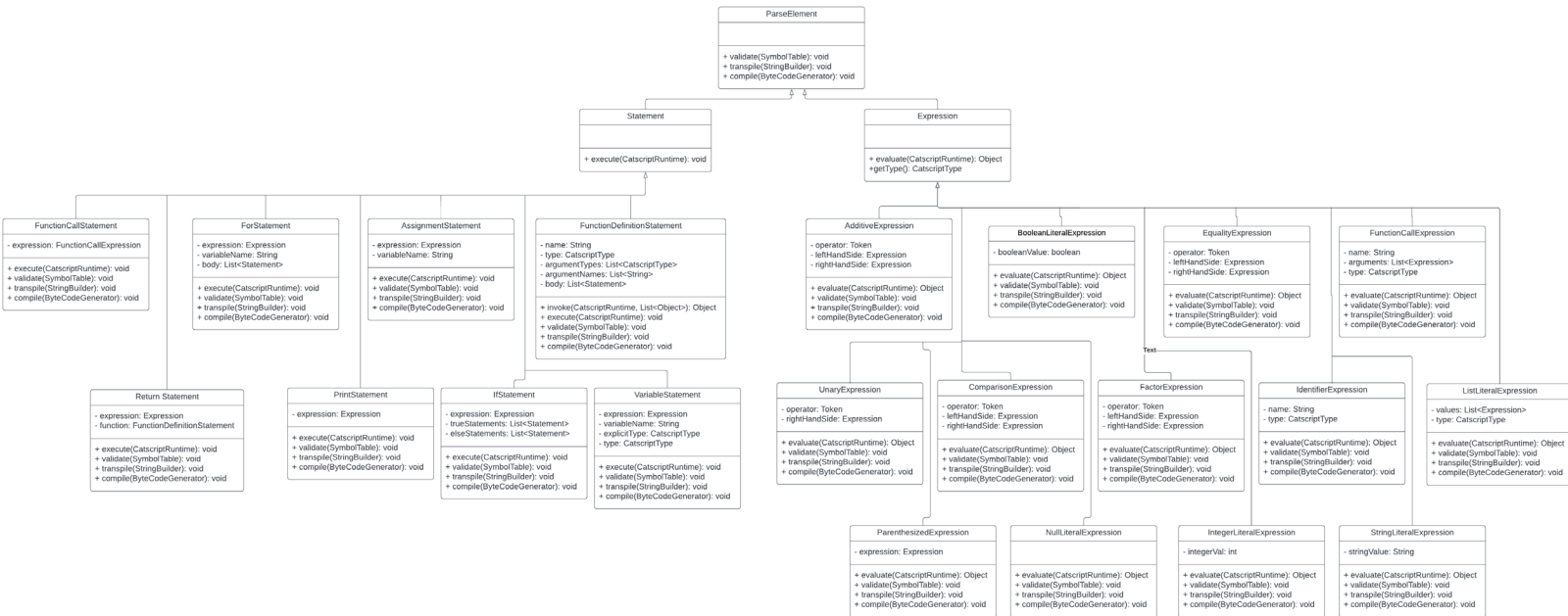
function_call = IDENTIFIER , '(' , argument_list , ')'

argument_list = [expression , { ',' , expression }]

type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' , type_expression , '>']

...

Section 5: UML



Section 6: Design Trade-Offs

A design decision that was made was to write Catscript with recursive descent instead of using a parser generator. The advantages of using recursive descent is that it is simpler compared to parser generators and it provides a better understanding of the recursive nature of grammars. The drawbacks to using it are that parser generators are more standard for compilers courses at universities and recursive descent involves much more code written by hand.

Section 7: Software Development Life Cycle Model

We used Test Driven Development (TDD) for this project and it worked really well. The tests gave a clear end goal to work towards and gave a good outline of how the compiler should end up working. It was also very easy to track the progress of the project using TDD because it is shown by how many tests are currently passing. Overall TDD is a great development model and it worked well for Catscript.