

Chris Rosler

Cam Alberghini

CSCI 468

Capstone Portfolio

May 6th 2022

Program:

Repository containing all source code and associated files in a zip:

csci-468-spring2022-private\capstone\portfolio\source.zip

Teamwork:

Team member 1 added functionality to the project. Batches of tests were used to aid in test driven development. The test batches in order were tokenizing, expression parsing, statement parsing, evaluation, and bytecode. After completing these individual components of Catscript, team member 2 developed 3 additional tests that were used to confirm that the separate components of the parser functioned together.

Total Estimated Hours: 160

Member 1:

Development

140 hours – 87.5%

Member 2:

Software Tests, Documentation

Estimated Hours – 20 – 12.5%

Design Pattern Contained:

Catscript implements the Memoization pattern, which optimizes the runtime and memory usage by preventing the creation of redundant objects. To code this function, we use a getListType function which allows only one Catscript type per list type. The memoization pattern can be seen in the

CatScriptType.java class inside of the getListType function. A hash map is used to store the types and list type equivalents.

```
static final 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;
}
```

Technical Writing:

Introduction:

Catscript is a simple scripting language that we developed a compiler for inside of the JVM.

Here is a simple example of catscript:

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

Types:

Catscript contains the following types to represent objects and other data values in the language.

1. int - A 32 bit integer.
2. String - A string of characters that act very similarly to the string in java.
3. Bool - A boolean true false value
4. list - a variable storing a list of values of type 'x'
5. null - A representation of the null type for use in catscript
6. object - A general that can represent any type of value

Catscript Grammar:

The section is all of the declarative statements that are contained within the Catscript grammar. These statements are used to do the data transformation and manipulation that makes scripting possible. Some of these are for arithmetic operations within Catscript. That is not all catscript can do though. It can do boolean operations, otherwise known as true false; function building, representation, and call functionality for building functions that can be called for later use; and other logical operators such as the If and for statement for use in building more complicated and computationally complex operations, and as well as recursive statements.

- `catscript_program = { program_statement };`
- `program_statement = statement |
function_declaration;`
- `statement = for_statement |
if_statement |
print_statement |
variable_statement |
assignment_statement |
function_call_statement;`

For Loops:

This is the definition statement for a for loop. A for loop in catscript is used for looping, doing an operation multiple times or until a specific condition is met.

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

If Statements:

This is the definition statement of an if statement in catscript. The expressions used by the if statement in catscript have to evaluate to a boolean or they must be the comparison operators(greater than, equal to, etc...). It also uses the java format of if, if else, and else.

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

Print statement:

The catscript print statement functions identically to the java print statement where it returns the entered data to the user.

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

Variable statements:

The variable statement in catscript is used to define a variable. They can be used in two different ways. They can either be explicitly typed where the type expression field is filled in, aka you include the type in the variable declaration. Or they type can be inferred by the parser as it evaluates the catscript. Example `var hello = "hello"` has no type definition, however `var hello:string = "hello"` is explicitly stated to be a string. The list definitions show that catscript variables, which list is a type of can be in multiple dimensions. IE a list of lists `var exList = [1,2,3,4, [a,b,c,d]]`.

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

Function Definitions and Calls:

Catscript uses the function declaration statement to declare the function name, and body, however the function body is technically a separate helper function that creates the body of a function linked to the definition. Functions in catscript can be either explicitly typed or have the type inferred. A good example of that is not including a return statement, when including a return statement you are explicitly defining the type of the return. You can choose to leave it blank and catscript will infer the default return type of null. It also shows its type inferred with the input types, they can either be explicitly stated or the compiler can be left to infer what type the input variable is assigned. The return statement grammar is a function that allows for the declaration of a function return.

- `function_call_statement = function_call;`
- `function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
[':' + type_expression], '{', { function_body_statement }, '}'`;
- `return_statement = 'return' [, expression];`
- `function_body_statement = statement |
return_statement;`

Equality Expressions:

Catscript has the functionality to evaluate equality expressions for objects of the same type. They are a specific type of comparison expression in catscript. The notation is listed below != is the not equal operation where == is the equals expression.

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

Comparison Expressions

Catscript has the functionality for comparative expressions such as greater than and greater than or equal to. It contains the same for less than. In the parser they are treated as a specific type of additive expression.

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

Unary Expressions

The unary expression in catscript is used to specify if a number is negative, or if a boolean defaults to false. AKA not true or 8 vs -8

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

Arithmetic Operations:

The language of catscript has the functionality to evaluate basic divisions ,multiplication (*), additive(+) and subtractive(-) mathematical operations.

- `factor_expression = unary_expression { ("/" | "*") unary_expression };`
- `additive_expression = factor_expression { ("+" | "-") factor_expression };`

Primary Expressions:

The primary expression is the default expression that catscript evaluates all other expressions to. For example in the arithmetic operations section factor expression is evaluated as a unary expression and then said unary expression evaluates to a primary expression or another unary expression depending on the expression. This allows for the inferred typing of the expressions and how it handles syntax errors.

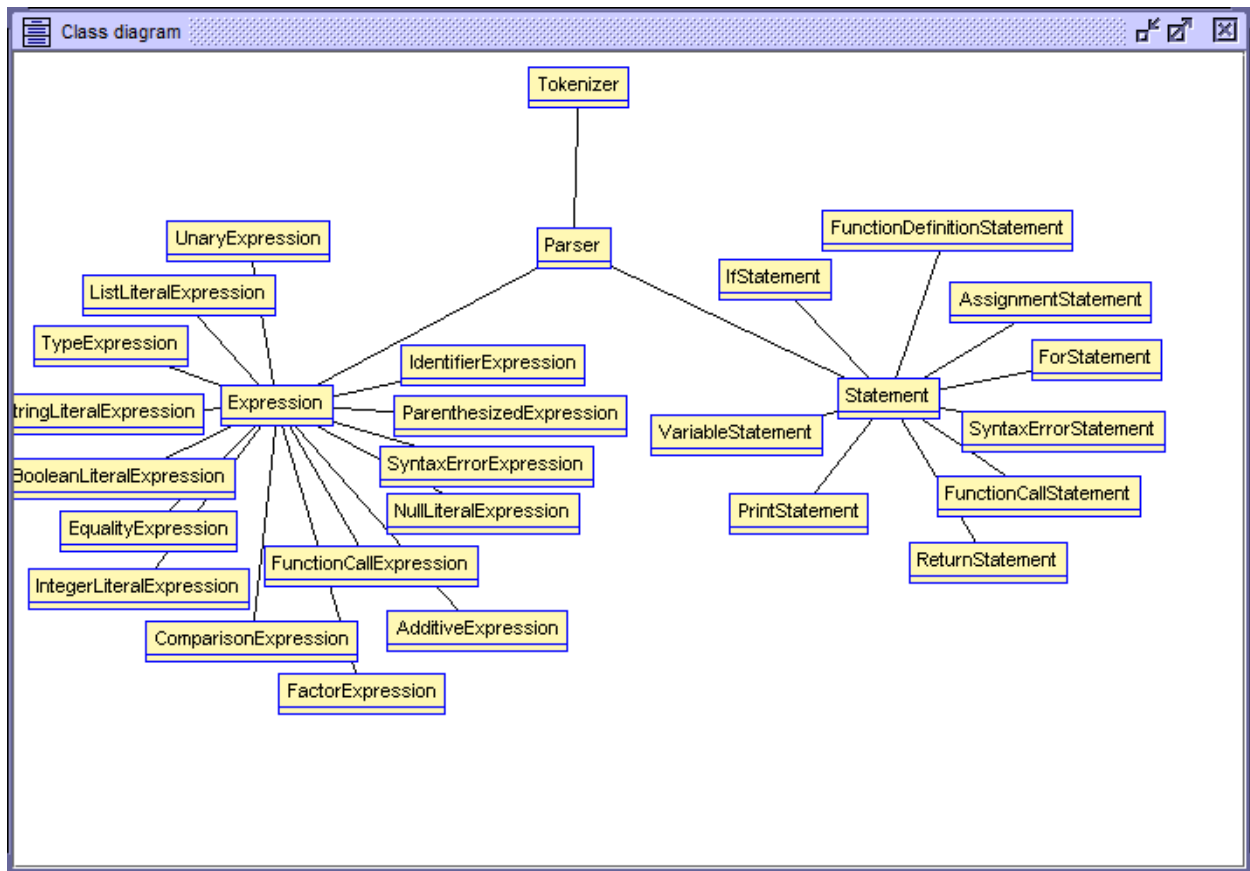
- `primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null" |
list_literal | function_call | "(" , expression , ")"`

UML:

Created using the USE 6.0.0 software

USE and CLT files included in the capstone/uml folder

Catscript Element Parsing UML



Design trade-offs:

The big tradeoff decision that was made for the Catscript programming language project, was to build it using a Recursive Descent Parser, which is a type of top-down parser. Most production parsers are recursive descent parsers. Alternatively, we could have implemented a bottom-up parser where we work backwards to the start symbol tokenizing to a grammar which yields a string's parse tree. As we used a recursive descent parser,

Software development life cycle:

This project uses the Test Driven Development life cycle model, where you start with all the tests that the end product should be able to run and then build functionality to achieve those goals. This development model feels natural to software developers with experience developing based around UML diagrams as you know the functionality of what needs to be built, you just need to develop it. Junit tests insured the functionality and completeness of the software as well. Using Test Driven Development helps to pace the software development into attainable milestones.