

Catscript Compiler

CSCI 468

Capstone Project Portfolio

Bryce Leighton

Simeon Shirshov

Montana State University

Spring 2023

Section 1: Program

The source code for this project is included in this directory titled source.zip.

Section 2: Teamwork

This capstone project was completed with a team of 2 people, team member 1 (Bryce Leighton) and team member 2 (Simeon Shirshov). The project was split into 2 different parts; one of which was done independently and the other which was done collaboratively between the two team members. For the first part, the two team members independently developed the CatScript code using test-driven development with a series of test suites curated by Carson. For the other part, the team members created unit tests for each other's Catscript code and traded them with each other. In addition to collaborating on creating the unit tests, teamwork was also used in creating the documentation for the structure of Catscript seen in section 4. Team member 2 wrote all the Catscript documentation for this project, and team member 1 wrote all the Catscript documentation for team member 2's project. The workload for this project between the two members was about 90% for team member 1 and 10% for team member 2. This was the opposite for team member 2's project. For this project team member 1 was the primary engineer and team member 2 was the documentation and test engineer.

Partner tests file path: **src/test/java/edu/montana/csci/csci468/demo/Scratch.java**

Section 3: Design Pattern

An example of a design pattern used within the Catscript parser is the Memoization Pattern. Memoization involves caching computational results so that they can be efficiently retrieved from a cache later, rather than recomputing the results. The motivation of the Memoization Pattern design choice is to minimize memory utilization through the use of caching. The snippet of code pictured below is taken from the Catscript compiler and showcases this design pattern. The code uses a single ListType object to be referenced throughout the compilation process rather than creating multiple identical ListType objects. The result is that there is better space efficiency within the compiler and the performance improved.

```
// Has been memoized :D
2 usages
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();
5 usages  ↗ Bryce Leighton +1
public static CatscriptType getListType(CatscriptType type) {
    CatscriptType listType = LIST_TYPES.get(type);
    if (listType == null) {
        listType = new ListType(type);
        LIST_TYPES.put(type, listType);
    }
    return listType;
}
```

File path: src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java

Section 4: Catscript Documentation

Technical Documentation for Catscript

Introduction:

Catscript is a relatively simple and small programming language that would be suitable for beginners. Catscript is a similar language to Java or Python. The language supports dynamic typing, loops, conditionals, functions, and lists. Catscript is based on a relatively lean grammar (seen below) which dictates the structure of the language and is used to parse and interpret it. An example of what Catscript looks like is shown below with a simple 'hello world' program as well as throughout the rest of the Technical documentation.

'Hello World' Example

```
1 var x = "catscript"
2 print(x)
3
```

Statements:

For Loop Statement:

In Catscript For Loop Statements are used to iterate over a list of elements and execute code for each element of said list taking in an identifier and an expression. Catscript allows for For Loop Statements to execute further statements within the for loop. This allows for all sorts of algorithm structures including nested for loops.

For Loop Example:

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

If Statement:

If Statements in Catscript are used to execute a block of code if a certain condition is true. Catscript also allows for an optional else statement which can execute a different block of code if the condition is false.

If Statement Example:

```
1 if(x > 10){
2     print(x)
3 }
```

Print Statement:

Print Statements in Catscript are used to output the value of an expression to the console. In addition to this the Catscript parser will evaluate any expressions inside the print statement and print the result.

Print Statement Example:

```
1 print("Hi Hunter")
2
3
```

Variable Statement:

Variable Statements in Catscript are used to declare and optionally assign a value to a variable. It can also have an optional type annotation. Catscript also allows for variables to be assigned a type both Implicitly and Explicitly.

Variable Statement/Catscript Types Example:

```
1 //Implicit Type
2 var a = 10
3 //Integer Type
4 var b : int = 10
5 //Boolean Type
6 var c : bool = true
7 //String Type
8 var d : string = ""
9 //Object Type
10 var e : object = ""
11 //List Type
12 var f : list<int> = [1, 2, 3]
```

Assignment Statement:

Assignment Statements in Catscript are used to assign a new value to an existing variable. One important stipulation is that Catscript only allows reassignment of a variable if the new assignment value is of the same type as the previous assignment. As an example an integer can only be assigned an integer.

Assignment Statement Example:

```
1 var a = 10
2 a = 20
```

Function Call Statement:

The Function Call Statement in Catscript is used to call a predefined function with a set of arguments if applicable. A function call statement requires that all required input function variables are given.

Function Declaration/Definition Statement:

Function Declaration/Definition Statement in Catscript is used to define a new function. The Statement requires a name and a block of code (body) which the function executes. Optionally Function Declaration Statements also allow for a list of parameters as input which can be used within the function and can also have return type annotation. Function Definition Statements are also able to have additional statements nested within the body.

Function Definition/Call Example:

```
1 //Function Definition
2 function x(a, b, c) {
3     print(a)
4     print(b)
5     print(c)
6 }
7 //Function Call
8 x(1, 2, 3)
```

Return Statement:

The Return Statement in Catscript is used to return a value from a function. Additionally Return Statements complete/break when hit. Return Statement's can be assigned any expression which will be executed when hit.

Return Statement Example:

```
1 function x() : int {
2     return 10
3 }
```

Expressions:

Equality Expression:

The Equality Expression in Catscript can either have a bang equal or equal equal which split two different expressions. Equality Expression's return a Boolean of True or False. Equal Equal returns True when expressions on either side are equal. Bang Equal returns True when expressions on either side are not equal.

Equality Expression Example:

```
1 // == Equal
2 "Hunter" == "Hunter"
3 // != Does Not Equal
4 "Carson" != "Hunter"
```

Comparison Expression:

The Comparison Expression in Catscript is very similar to the Equality Expressions. In Catscript Comparison Expression's are used to compare two values. They can use the ">", ">=", "<", or "<=" operator. When the operator used is true for the two expressions on either side Catscript returns True. When its false it returns False.

Comparison Expression Example:

```
1 // Less Than
2 0 < 10
3 // Less Than or Equal To
4 var1 <= Var2
5 // Greater Than
6 10 > 0
7 // Greater Than or Equal To
8 Var1 >= Var2
```

Additive Expression:

The Additive Expression in Catscript is used to perform addition or subtraction on two values. It can use the "+" or "-" operator. In Catscript similar to the english language the values are evaluated from left to right. The Additive Expression can also be used to concatenate strings together with the "+" symbol.

Additive Expression Example:

```
1 //Integer Subtraction
2 10 - 5
3 //String Addition
4 "Hunter" + " " + "Montana"
```

Factor Expression:

The Factor Expression in Catscript is used to perform multiplication or division on two values. It can use the "*" or "/" operator. The "*" operator is used for multiplication. The "/" operator is used for division. Factor Expressions similar to Additive expressions are evaluated left to right.

Factor Expression Example:

```
1 // Multiplication
2 10 * 10
3 // Division
4 10 / 10
```

Unary Expression:

The Unary Expression in Catscript is used to apply a unary operator to an expression. It can use the "not" or "-" operator. Generally the "not" operator is used to negate a value such as a boolean, int, or string. The "-" operator is used to symbolize a negative integer. The "-" operator can only be used with integers.

Unary Expression Example:

```
1 // Negative Integer
2 -100
3 // Not Boolean
4 not False
5 not True
6
```

Primary Expression:

The primary expression in Catscript is a basic unit of code that produces a value. It can be an Identifier/variable, a string, an integer, a boolean, a null value, a list literal, or a function call.

Identifier/Variable: An identifier or a variable in Catscript is a string chosen by the user to represent something.

String: A string in Catscript is a java-style string.

Integer: An integer in Catscript is a 32 bit integer.

Boolean: A boolean in Catscript is an expression used to represent True or False Values.

Null Value: A null value in Catscript is an expression used to show an absence of value.

List Literal: A list literal in Catscript is used to create a new list. It takes a series of expressions as inputs.

Function Call: A function call in Catscript is used to call a function with a set of arguments.

Primary Expression Examples:

```
1 //Identifier/Variable
2 var1
3 //String
4 "Hello"
5 //Integer
6 123
7 //Boolean
8 True False
9 //Null Value
10 null
11 //List Literal
12 [1,2,3] ["Hunter","Carson"]
13 //Function Call
14 func("var", 23, null)
15
```

Type Expression:

A type expression in Catscript is used to annotate a variable or a function parameter with a data type. It can be "int", "string", "bool", "object", or "list" with an optional generic type annotation.


```

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 } ]

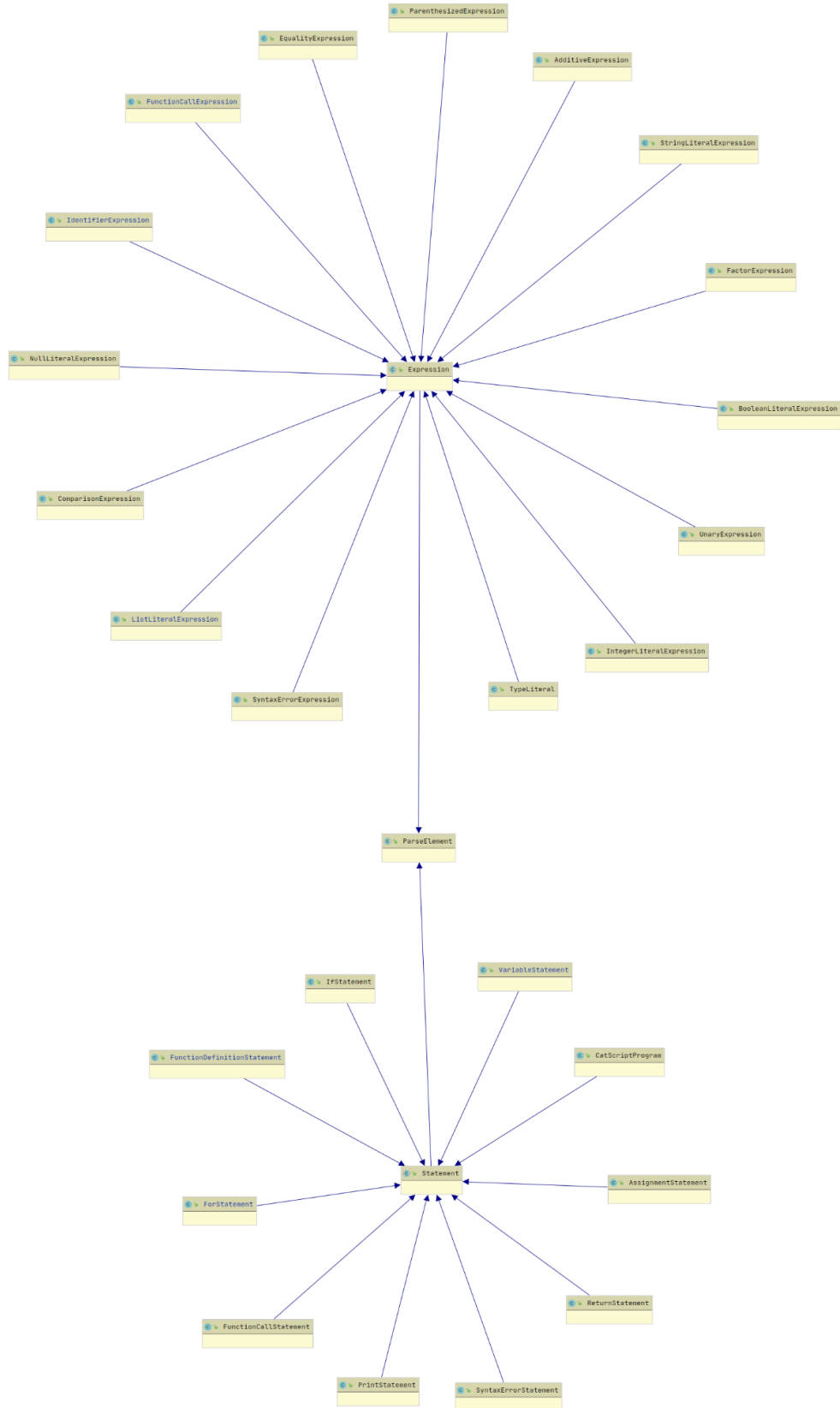
type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [ , '<' , type_expression, '>' ]

```

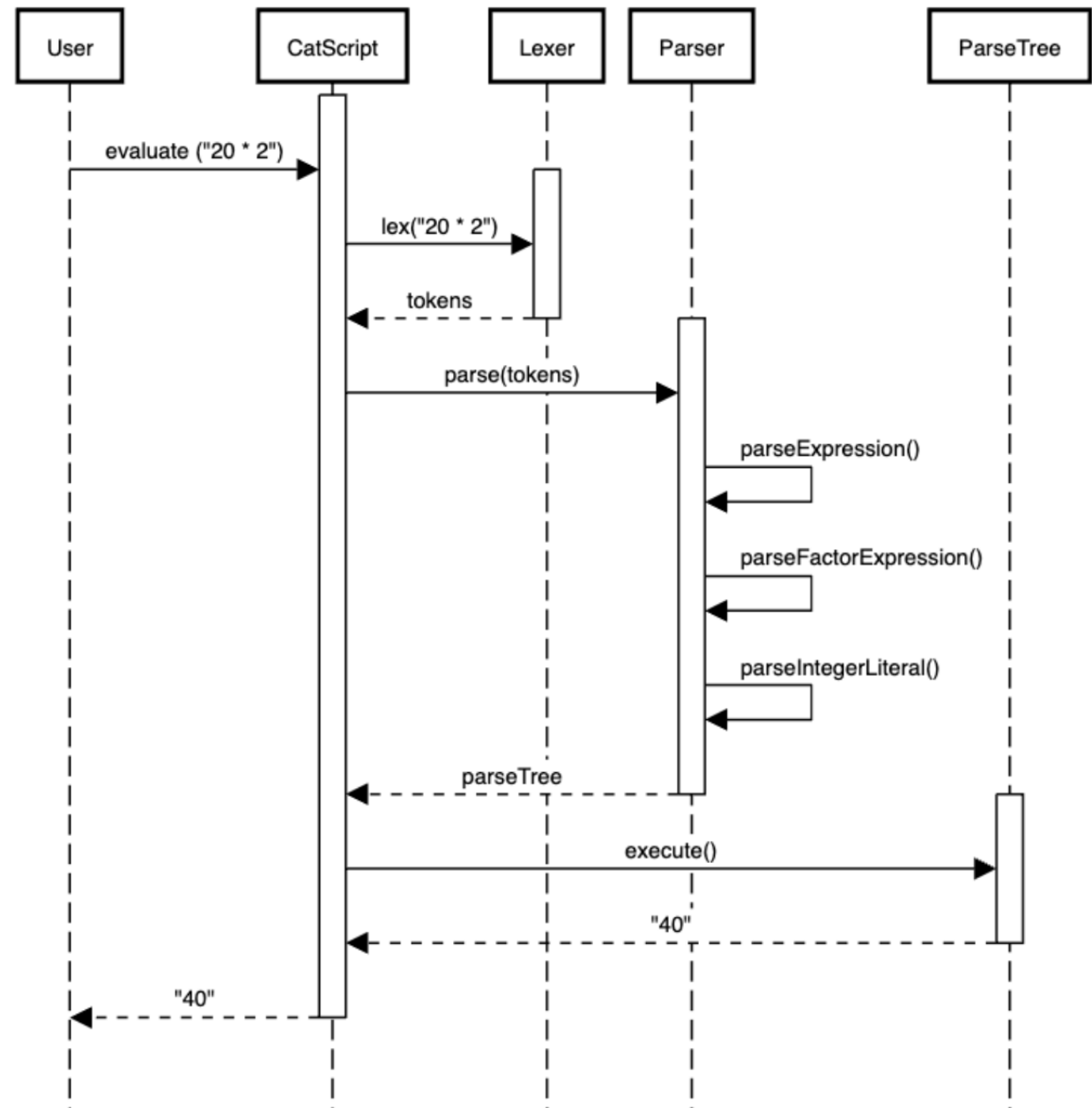
Section 5: UML

The UML Diagram below shows the overall structure of Catscript. As we can see, the abstract parent classes Expression and Statement hold many of the general components/logic that make the entire Catscript compiler work. We see that the Expression and Statement classes are held together by and inherit from the ParseElement class which helps properly set tokens, add errors, and other fundamental methods.

The second diagram shows a sequence diagram for how Catscript deals with multiplication, or a Factored Expression in Catscript. The sequence diagram helps demonstrate the ability of the Catscript compiler and the program sequence that happens in the background when compiling certain code.



Catscript Multiplication Sequence Diagram



Section 6: Design Trade-Offs

The most important design trade-off of this project was developing a compiler for Catscript by hand rather than using a parser generator tool. We used the recursive descent algorithm to create the Catscript parser. Using a parser generator is much easier and faster than hand-writing a compiler. However, due to its artificial generation, a generated parser can become a “black box” where debugging is very difficult due to the developer not knowing the inner workings of the code. Hand-writing the compiler gave us the benefit of knowing how the

entire compiler worked, so debugging and modification of the compiler is more intuitive. Now we understand exactly how the Catscript grammars work from the bottom up. These reasons are why the design trade-off decision is justified.

Section 7: Software Development Life Cycle Model

Test-Driven Development (TDD) was the software development life cycle model used for the development of the Catscript compiler. This method of software development consists of the creation of test suites for each piece of required functionality of the software being developed, and then the tests are what guide the actual coding process. For the Catscript compiler, the tests were split into 4 separate test suite categories: tokenization, expression parsing, statement parsing + eval, and compilation. These test studies were designed to be a comprehensive coverage of Catscript functionality. In addition, these tests were built on top of one another so that one would need to finish tokenization before moving to expression parsing, and so on. This test design aided the development in many ways, the most prevalent being the debugging of code. The subsequent structure of the test suites allowed for easier discovery of code that was necessary to implement, and made it easy to find where code was failing in the case of a failed test. Overall we believe this was the best approach to the development of this project because of the way it helped us maintain efficiency by providing clear goals and a much more clear understanding of the functionality we were building. The sequential manner in which the tests were set up provided a clear path for development.