

Montana State University – CSCI 465 (Compilers)

Compilers Capstone

Carson Gross

Mark Braun – Team Member 2, Grant Dunbar – Team Member 1
Spring 2023

Section 1: Program

Link to project source code - <https://github.com/grantdunbar8/csci-468-spring2023-private/blob/main/capstone/portfolio/source.zip>

Section 2: Teamwork

For our capstone project we split the work into one team member writing the compiler and the other team member writing 3 tests and documentation for the other team member's compiler. I spent about 85% of the time writing my compiler and my team member spent about 15% of the time writing 3 tests and documentation for my compiler. I then spent about 15% of the time my team member spent on their compiler writing 3 tests and documentation to outline the main features of their compiler. Overall, my team member and I worked well together, and I think having the team member helped me to get the most out of the project because we were able to talk through the compiler together. I also think that writing the tests for my team member's compiler forced me to learn more about my compiler because I needed to check for things that could break the compiler that were not already being checked.

The documentation that is below was written by team member two and the tests that they wrote to test the compiler that team member 1 wrote are stored in the PartnerTests.java file that is located here: `src/test/java/edu/montana/csci/csci468/capstone_tests/PartnerTests.java`.

Section 3: Design pattern

```
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new
HashMap<>();
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;
}
```

In the Catscript compiler I implemented memoization. Memoization allows for the program to run more efficiently because list types can be cached to be used later, rather than having to be redefined for every use. The way that memoization is implemented in Catscript is that when a new list is created it references the LIST_TYPES hash map to see if a list of this type has already been created. If it has been created, it will then use the predefined list, otherwise it will create a new one and add it to the LIST_TYPES hash map. This code is in the CatscriptType.java which is used during the validating and parsing process of the compiler.

There are pros and cons to using memoization rather than just directly coding the list types. One of the pros is that efficiency is increased if there are sufficient calls of the memoization algorithm. Memoization also helps to reduce redundancy. With efficiency up, memoization does not affect the amount of storage that the program requires to run. Another con to memoization is if the memoization algorithm is not called enough times. If the memoization is only used one or two times, it is less efficient than directly coding the list because it must check the hash map and create a new entry. As the number of lists increases, efficiency increases because it can reuse what it has created. The memoization could probably be expanded to the

Section 4: Technical Writing

CatScript Documentation

Introduction and Program Structure

CatScript is a simple functional programming language built with Java.

CatScript Variables

Variable Types

CatScript is a statically typed language that supports the following variable types:

- int: a 32 bit integer
- string: A Java style string
- bool: A boolean value (TRUE/FALSE)
- null: The null type
- object: A variable with no specified type

In addition, CatScript supports list variables composed of identically-typed elements with a type from the list above.

Variable Statements

Variables are initialized with the following grammar:

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

List variables in particular are initialized with the following grammar:

```
list_literal = '[', expression, { ',', expression } '];'
```

Here are example variable statements with:

- An int: `var myInt : int = 10`

- A string: `var myString : string = "Hello World"`
- A bool: `var myBool : bool = true`
- An object: `var myObject : object = 10`
- An list of ints: `var myIntList : list<int> = [1, 2, 3, 4, 5]`

Additionally, CatScript supports both explicit and implicit typing of variables. If an explicit variable type is not supplied, the default variable type is an `object`. Both of the following variable statements are valid in CatScript:

- `var myVariable : int = 10` (an integer type)
- `var myVariable = 10` (an object type)

CatScript Functions

CatScript supports function definitions and function calls in similar ways to analogous programming languages.

Function Definition

The CatScript syntax for a function definition is as follows:

```
function_definition = 'function', IDENTIFIER, '(', parameter_list, ')', [ ':' + type_expression, ]
```

Where `IDENTIFIER` is the name of the function, `parameter_list` is the list of input parameters to the function, `[':' + type_expression,]` is the return signature of the function, and is optional, and the `function_body_statement` is a sequence of program statements that collectively define the behavior of the function.

Here are several examples of function definitions:

```
function foo() { print("Hello World") }
function add(x : int, y : int) : int { return (x + y) }
function print(myString : string) { print(myString) }
```

Function Calls

Function calls can be created with the syntax:

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

Where `IDENTIFIER` is the name of the desired function to be called, and `argument_list` is the list of arguments supplied to the function, the necessity of which are determined by the respective function definition. Here are some example function calls of the example function definitions provided above:

```
foo()
add(5, 10)
print("Hello World")
```

Return Statements

If a function specifies a return type in it's function definition, CatScript requires a return statement after completion of the function body of the appropriate type. If a function has no specified return type in it's function definition, it is assumed that this function has a void return type and no return statement is required. Return statements can be created with the following syntax:

```
return_statement = 'return' [, expression];
```

CatScript Expressions

CatScript supports common unary, arithmetic, and comparative expressions.

Primary Expressions

Primary expressions in CatScript are supported with the following grammar:

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

Unary Expressions

CatScript supports the `not` (logical negation) and `-` (numeric negation) unary operations. Unary expressions are supported with the grammar:

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

Here are examples of valid CatScript unary expressions:

- not True
- - 100
- not 50
- -(10 + 20)

Additive Expressions

CatScript supports addition and subtraction operations on some expressions, depending on the type of the expression. The formal grammar for an additive expression is:

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

Addition is defined on `int`-typed expressions (conventional addition), `string`-typed expressions (string concatenation), and combinations of `string` and `int` expressions (string concatenation with the string value of the integer).

Some examples of valid CatScript additive expressions are:

- `2 + 2` (evaluates to 4)
- `"Hello " + "World"` (evaluates to "Hello World")
- `"There are " + "50" + " states in the US"` (evaluates to "There are 50 states in the US")

Subtraction is only defined on `int`-typed expressions.

Factor Expressions

CatScript also supports multiplication and division operations on `int`-typed expressions. The formal grammar for an additive expression is:

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

Some examples of valid CatScript factor expressions are: `- 5 * 7` (evaluates to 35) `- 100 / 10` (evaluates to 10)

It is worth noting that in CatScript, the order of operations follows traditional PEMDAS convention, where multiplication and division are evaluated first, from left to right as appearing in the expression, followed by addition and subtraction in a similar matter, and finally unary operation.

Comparison and Equality Expressions

CatScript supports the evaluation of equality expressions and comparative expressions among most variable types. These expressions evaluate to true or false, depending on the relationship between the expressions being compared. The formal grammar for an equality expression is:

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

Some examples of valid CatScript equality expressions are:

- `2 != 5` (evaluates to true)
- `true == true` (evaluates to true)
- `"Hello World" == "Hello Earth"` (evaluates to false)

The formal grammar for a CatScript comparison expression is:

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

Some examples of valid CatScript comparison expressions are:

- `10 <= 17` (evaluates to true)
- `5 >= 5` (evaluates to true)
- `(10 + 4) > (7 + 20)` (evaluates to true)

CatScript Flow Control

CatScript supports several flow control structures to manage the execution of code.

If Statements

CatScript supports standard if statements, as well as single else statements accompanying. The formal grammar for an if statement is:

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

If the initial condition supplied to the if statement evaluates to true, the statements contained in the "if" portion of the statement will be executed. Otherwise, the statements contained in the "else" portion of the statement will be executed.

A couple examples of valid CatScript if statements are:

- `if(true) { print("True") }`
- `if(a > b) { print("a is greater than b") } else { print("b is greater than a") }`

For Loops

CatScript supports for loops that loop based on a provided indexing variable and a list object of possible values of this indexing variable. The formal grammar for a for loop is:

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

A couple examples of valid CatScript for loops are:

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

```
for(x in ["a", "b", "c"]) {  
    for(y in [1, 2, 3, 4, 5]) {  
        print(x + y)  
    }  
}
```

Print Statements

CatScript supports basic print statements for expressions of any type. Supplying an expression to the `print()` will result in the printing of the string value of that expression to the console. The formal grammar for a print statement is as follows:

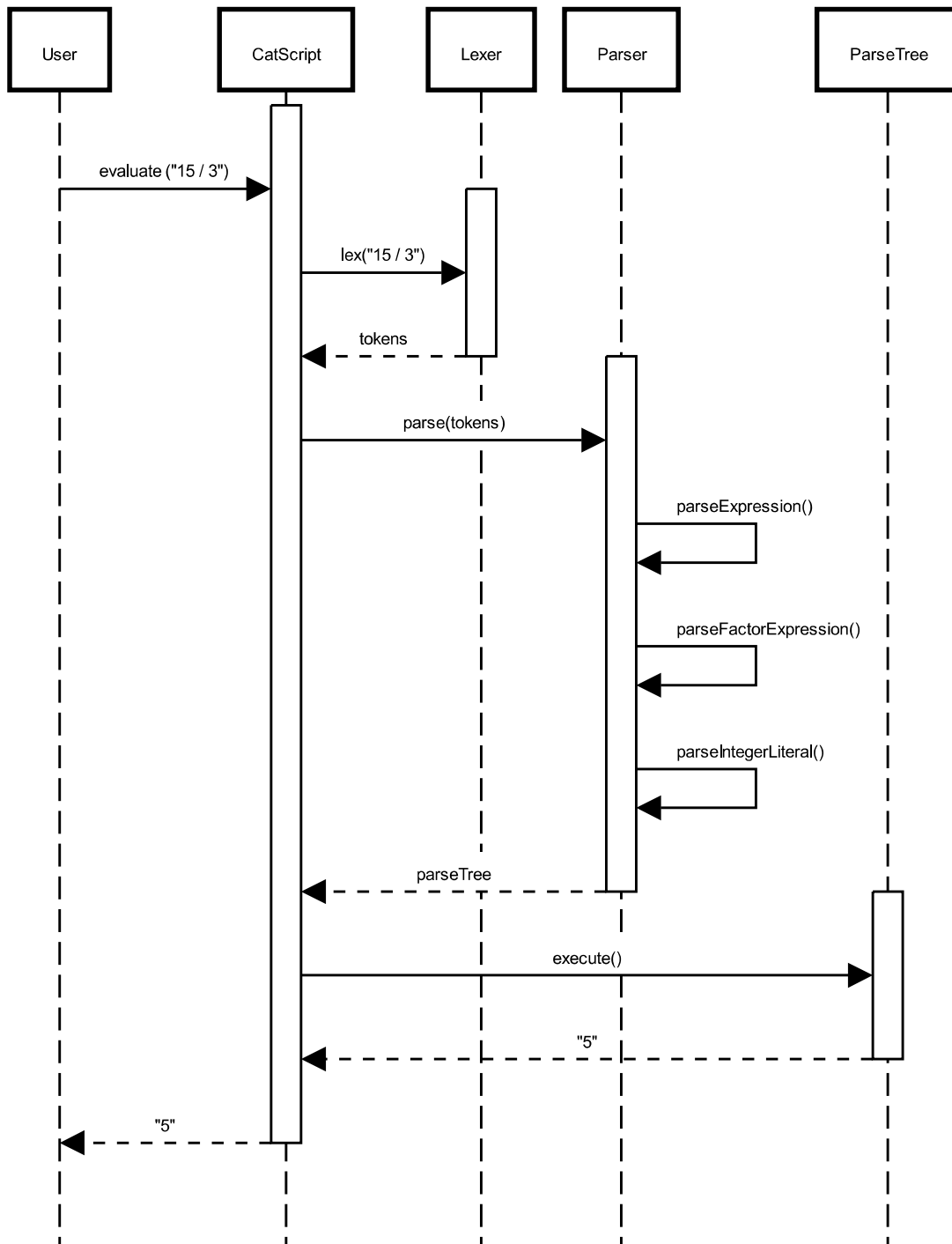
```
print_statement = 'print', '(', expression, ')'
```

Section 5: UML

On the next page is the sequence diagram for factor expressions. As you can see it starts with the user writing a factor expression by passing the expression into Catscript. Then, each part of the expression is tokenized using the lexer. After tokenization, the tokens are then parsed by the parser. Three parses happen in the parser. First the expression is parsed, then the factor expression is parsed, and then the integer literal is parsed. The parser will then return a parse tree that can be used later. After the parse tree is created, the expression can be evaluated, and the result of the expression can be computed. The result is then returned from the parse tree to Catscript and then Catscript returns the value to the user.

A similar sequence diagram can be created for the other expressions, but we thought that the factor expression would be a good example to show how the compiler should work. This sequence diagram helped us to be able to visualize what Catscript would be doing when a user gave it an expression and it helped us to better understand the parts of the compiler.

Catscript Factor Sequence Diagram



Section 6: Design trade-offs

One of the design trade-offs that we encountered in this project was to decide between using recursive descent or using a parser generator. In the end we decided to use recursive descent in the tokenizer because it would be faster and better at handling errors. We thought that this would be helpful in the design of this language to be able to have built-in error checking throughout the program. One of the cons of using recursive descent rather than a parser generator is that it required a lot more code to write because we had to develop it from scratch rather than using a prebuilt library.

Another reason for using recursive descent to create our parser is that we were able to learn how the parsing process worked as well as how to create a parse tree. When using a prebuilt library, it is easy to rely heavily on code that you do not understand how it works. By using recursive descent, we were forced to know exactly how my parser is working and it also allowed us to have flexibility with the elements that we wanted to include in Catscript. Overall, we thought that using recursive would be the best option for our compiler because it would allow us to learn the most about the parsing process as well as would allow us to have more flexibility in the features of our language.

Section 7: Software development life cycle model

For our development of the compiler, we used test driven development. We really enjoyed doing test driven development because it allowed us to ensure that each of the parts of the compiler worked and that each feature was working. Each part of the compiler, the tokenizer, evaluator, parser, and compiler required many parts to get them working and it would have been hard to ensure that we had every component if we did not use the tests to steadily get through each of the checkpoints. We also enjoyed using test driven development because it was able to quickly show the progress that we were making. It is a good feeling when you start a project with 160+ failing tests and then in the end have every test passing. We felt like that was a good motivator and allowed us to continue making progress as we made the red x's turn to green checkmarks.

Another perk of using test driven development was that we were able to create new tests to fill any gaps that we found in the compiler. We had never written tests for test driven development so it was interesting to learn how to write tests that would run properly. We thought that this would be a useful skill in the future and thought that it was a nice perk to using test driven development for the software development lifecycle of our compiler.