

CSCI 468 Compilers

Spring 2023

Michael Clearwater, Isaac Schmidt

Section 1: Program

See source.zip within this directory.

Section 2: Teamwork

Team member 1 and team member 2 fully implemented the Catscript compiler individually. Team member 1 completed the tokenizer, parser, and the compiler individually, and team member 2 did the same for thier capstone project. Team member 1 then wrote test cases and the documentation for team member 2, while team member 2 did the same for team member 1. Because of the way we completed the project, we both worked on 100% of our separte versions of Catscript. We had a 50/50 split for the percentage of time and work spent split between both of our projects, as we both did the same thing for each other.

Section 3: Design pattern

A design pattern that was used when implementing Catscript is memoization. I implemented memoization in `CatscriptType.java`, inside of the class's `getListType` method.

Below is the snippet of code where memoization was implemented:

```
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;
}
```

I used memoization to make the `getListType` method more efficient and faster. Previously, the method was creating a new `ListType` using the provided `CatscriptType` and returning it every time, creating a lot more unnecessary computation every time the method is called. With memoization implemented, we are only creating a new `ListType` when the `HashMap` hasn't seen the provided `CatscriptType`. If the `HashMap` has seen the provided type, we are just retrieving the `ListType` from the map and returning it, which is a lot faster in terms of time complexity than creating a new instance of `ListType`. This process decreases the time complexity of the `getListType` method, therefore it was implemented rather than just coding the method directly.

Section 4: Technical writing. Include the technical document that accompanied your capstone project.

This document is a guide for catscript, satisfying the technical document capstone requirement (section 4).

- [Catscript Guide](#)
 - [Introduction](#)
 - [Features](#)
 - [Type System](#)
 - [Print Statements](#)
 - [Functions](#)
 - [For Loops](#)
 - [Comparison Expressions](#)
 - [If Statements](#)
 - [Assignment](#)
 - [Operators](#)

Introduction

Catscript is a simple scripting language that is statically typed, with a limited selection of straightforward expressions and statements that can be executed.

Features

Type System

Catscript is statically typed, with these 6 types:

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

The assignability of int, string, bool, and object types are based on their corresponding Java types, where integers, strings, and booleans may be assigned to objects, but not the other way around. Null meanwhile may be assigned to any other type.

Print Statements

Catscript provides a simple built in print statement that takes 1 argument, to be put into the output stream. It also automatically comes with a newline character at the end of each print.

Print Statement Example:

```
print("Hello World!")  
// Will print "Hello World!\n" to the output stream.
```

Functions

Like in other languages, functions in Catscript act as a way for the control structure to associate a set of commands with an identifier, so that they can be executed as a group whenever needed.

Functions in Catscript are declared in the following grammar:

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

Here the function keyword indicates that a new function is being defined. The IDENTIFIER must then be unique from any previously defined. 'parameter_list' provides any parameters that may be needed to execute the function, and 'function_body_statement' is the collection of statements that are executed when the function is called. If there is a 'type_expression' included, then the function *must* return a value of the specified type, in every possible branch of the function. However, if there is no return type specified, then the function will just return void.

Functions are called using the following grammar:

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

The IDENTIFIER refers to the unique function that was previously declared, and the 'argument_list' is where the relevant (and required) parameters are supplied to the function as arguments.

Function Examples:

```
function myFun() {  
    print("Simple Function with no parameters or return type")  
}  
  
function addTwoNumbers(num1: int, num2: int): int {  
    return (num1 + num2)  
}  
  
myFun()  
print(addTwoNumbers(3,4))
```

For loops

For loops in Catscript are a critical piece of the control structure, as they are the only type of loop offered. In order to be executed, for loops must be supplied an iterator IDENTIFIER which will iterate over a given list. The type of the iterator is inferred based on the type of the list. For every item in the list, a set of statements is executed. The grammar of for loops is as follows:

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

For Loop Example:

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

Comparison Expressions

Comparison expressions are used to generate bool values based on a comparison between two values.

Catscript offers 6 different forms of comparison expressions:

- '=' - evaluates to true iff both values are the same, otherwise false.
- '!=' - evaluates to true iff the values are different, otherwise false.
- '<' - evaluates to true iff the left value is less than the right, otherwise false.
- '<=' - evaluates to true if the left value is equal to or less than the right; otherwise false.
- '>' - evaluates to true iff the left value is greater than the right, otherwise false.
- '>=' - evaluates to true if the left value is equal to or greater than the right; otherwise false.

Comparison Expression Examples:

```
0 == 0  
// evaluates to true  
  
0 != 0  
// evaluates to false  
  
0 < 0  
// evaluates to false  
  
0 <= 0  
// evaluates to true  
  
1 > 0  
// evaluates to true  
  
0 >= 1  
// evaluates to false
```

If Statements

If statements are part of the control structure that serve as a way to conditionally execute sections of code.

The grammar of if statements is as follows:

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

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

The statements in the body of the if statement are executed iff the expression in the first parentheses evaluates to true, otherwise they are skipped over. If the if statement has an 'else', then the second grouping of statements following the else are then executed iff the original condition was false.

Example of an If Statement:

```
if (true) {  
    print("this will get printed")  
} else {  
    print("this won't get printed")  
}
```

Assignment

Assignment in Catscript is used to assign human-readable identifiers to values. The 'var' keyword is used to indicate a new variable is being declared. New variables must be assigned a value upon declaration in Catscript. Variable types may be inferred or explicitly defined upon initialization. The grammar of assignment is as follows:

```
assignment_statement = IDENTIFIER, '=', expression;
```

Examples of Assignment:

```
var myString: string = "This string is explicitly declared as a string"  
myString = "can be assigned a new value"  
  
var myInt = 1  
// implicitly set to type int
```

Operators

There are many typical operators in Catscript, both for arithmetic and logical operations. Most evaluate from 2 parameters into 1 value, but some are unary and only take in 1 parameter. All operations in Catscript are left associative. Grammars associated with Catscript operators are as follows:

```
additive_expression = factor_expression { ("+" | "-") factor_expression };  
  
factor_expression = unary_expression { ("/" | "*") unary_expression };  
  
unary_expression = ( "not" | "-" ) unary_expression | primary_expression;
```

Here is a brief description of each operator:

- '+' - evaluates to the sum of the left and right
- '-' - evaluates to the value of the left minus the right
- '/' - evaluates to the value of the left divided by the right
- '*' - evaluates to the value of the left multiplied by the right
- 'not' - negates the value of the following boolean expression
- '-' negates the numerical value of the following int. Note this is distinguished from typical subtraction by the lack of a value on the left, thus forcing it to be a unary operator

Examples of Operators in Catscript:

```
1 + 2
// evaluates to 3

4 - 3
// evaluates to 1

4 / 2
// evaluates to 2

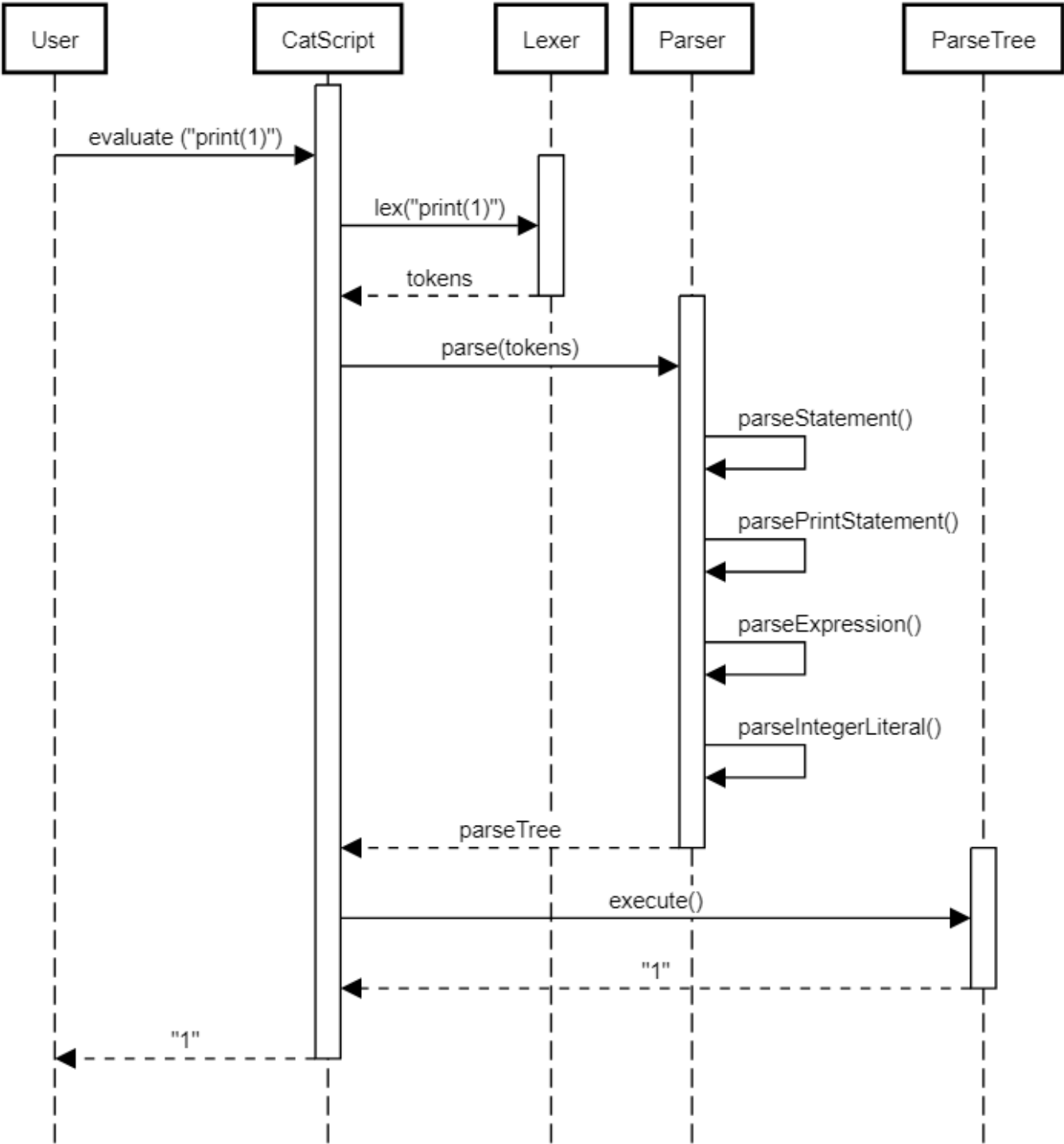
5 * 6
// evaluates to 30

not true
// evaluates to false

-(1)
// evaluates to -1
```

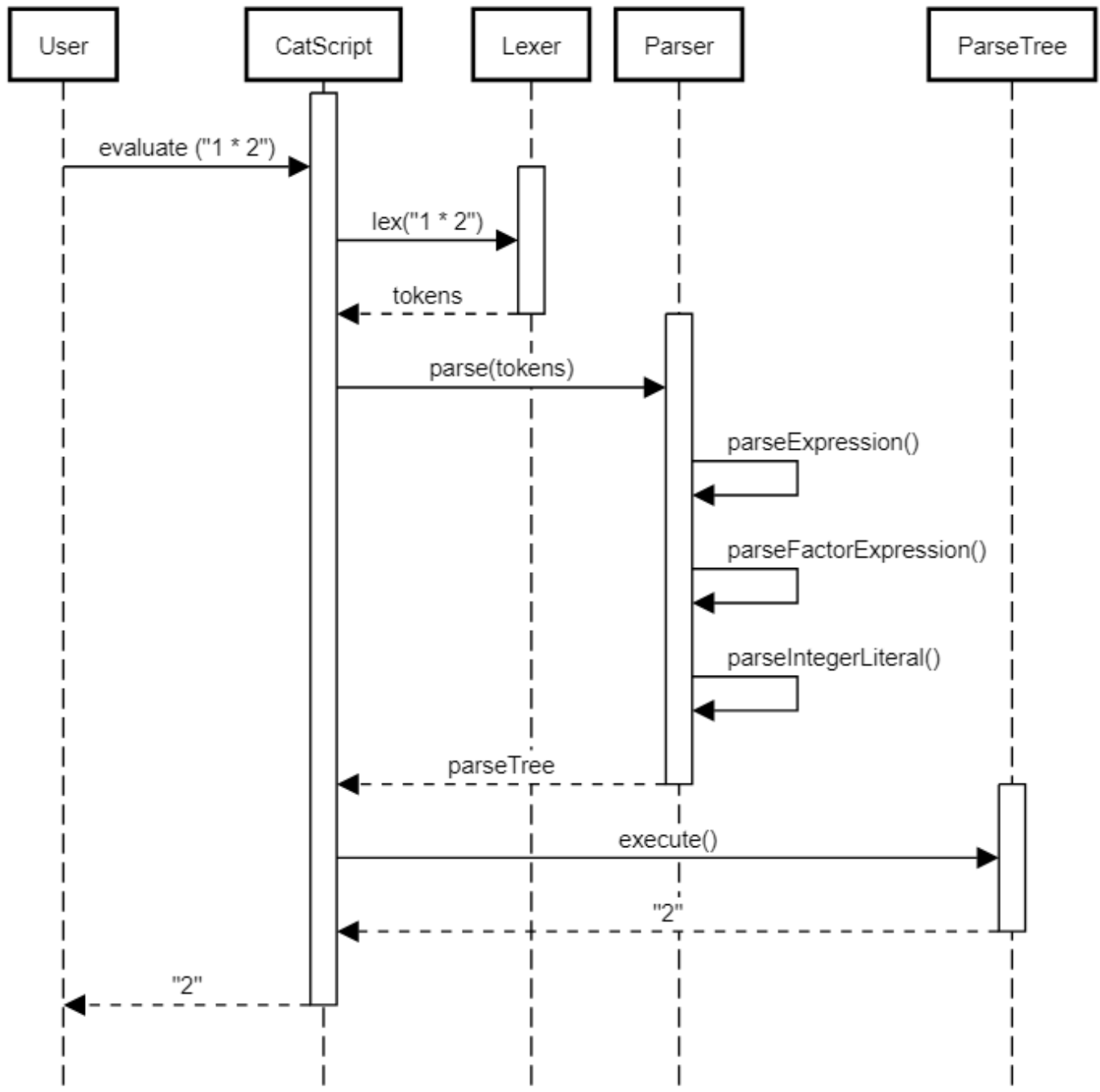
Section 5: UML.

Catscript Print Sequence Diagram



This UML sequence diagram shows the sequence of events that happen for a print statement. The user executes the print statement within CatScript. CatScript then tokenizes the given statement within the lexer, and the lexer returns the tokens to CatScript. CatScript then parses the tokens within the parser, and the parser parses every statement/expression individually and returns the parse tree. The parse tree is then executed and returns 1 back to the user.

Catscript Multiplication Sequence Diagram



This UML sequence diagram shows the sequence of events that happen for a factor expression. The user passes in the expression to CatScript. CatScript then tokenizes the expression within the lexer, and the lexer returns the tokens to CatScript. CatScript then parses the tokens within the parser, and the parser then parses every expression individually and adds it to the parse tree. The parse tree is then executed and returns 2,

Section 6: Design trade-offs

A design trade-off that is present within my implementation of CatScript was using the Recursive Descent Algorithm rather than using a parser generator. It would have been a lot easier to use a parser generator to implement CatScript, but I would not have gained the hands-on experience that I did by implementing the Recursive Descent Algorithm. By using the recursive descent algorithm, I was able to learn how to implement a compiler at every step of the way. The trade-off here was sacrificing simplicity and ease for the sake of gaining knowledge of how a parser works under the covers and how to implement a grammar.

Section 7: Software development life cycle model

The model that I used to complete my capstone project was Test Driven Development. Test-driven development allowed me to test my code as I went by executing some portion of CatScript and seeing if the output matched what was expected. I believe this model was extremely helpful in completing CatScript, as it made it a lot easier to find out why something was or wasn't working when using the debugger. This model also helped keep me on track with the order of things that needed to be completed as far as implementing the different expressions/statements before one another. I do not believe that Test Driven Development hindered my team in any way and only provided benefits.