

CSCI 468

Spring 2023

Joshua Fried, Jeremy Heng

Section 1: Program

A zip folder, source.zip, is attached to the project containing the source code for the Catscript Compiler. This is the technical language that Catscript follows:

- 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 2: Teamwork

Team member 1 was the primary engineer for the Catscript Compiler. They wrote all the code for the compiler. They made the methods and completed every test for the compiler. Team member 2 was the testing engineer. They provided tests to ensure the completeness of the compiler. They also wrote the technical documentation for the Catscript language. Team member 1 contributed about 80% of the time and effort in the overall project and team member 2 contributed 20% of the time and effort.

Section 3: Design pattern

The **memoization design pattern** was used in the development of the compiler. The memoization design pattern is used to remember the results of a function, so that they don't need to be called again. This pattern improves the compiler because it only has to call a function one time and saves time by remembering the result.

Section 4: Technical writing

Technical Documentation:

Introduction

Catscript is a simple scripting language that has basic features found in contemporary programming languages. It is a statically typed language and has several design choices that are influenced by features present in Java. However, Catscript is a functional scripting language and doesn't support classes. The compiler utilizes recursive descent as the technique for parsing, making it intuitive and easy to understand. The compiler was written in Java and has four different parts: tokenization, parsing, evaluation, and bytecode. Tokenization breaks input code into tokens that are utilized by the parser. The parser creates a parse tree and ensures the tokens follow the correct syntax. The evaluation phase makes sure the input code is correct for the language itself. The final phase converts the input code into Java bytecode to be used

by the Java Virtual Machine. An example of a Catscript program is:

Type System

```
function example(x : list) {  
  
    print(x)  
  
}  
example([1, 2, 3, 4])
```

Catscript is statically typed and the types of all variables, literals, parameters, functions, etc. are known at compile time. Despite being statically typed, the compiler can infer types (implicit declaration) in addition to users being able to explicitly declare types. The type system is quite similar to that of Java. Catscript has a basic type system that consists of int, string, bool, null, void, and object. It also has a type called list that is an iterable collection of one other type or a mix of other types. If a list contains a mix of types, the list itself will be of type object in order to maintain a collection of mixed types. Lists are immutable to avoid a weakness in the Java type system. Everything in Catscript is assignable from null. Types are covariant and can only be assigned to other types if that is the case in Java. Types are determined in the validation phase of the compiler. Symbol tables are used to track variables and to track various scopes that could exist within a program.

Features

Catscript provides basic features that greatly increase the functionality and capability of the language. All Catscript features are implemented using expressions, statements, or a mix of both. This language does not support comments. Variables are available through declarations and assignments, like variables in other contemporary languages. As mentioned earlier, variable types can be implicitly or explicitly declared. Here are a couple of examples:

```
var x = 1  
var x : int = 2  
var x : string = "Hello"
```

It supports for loops and if/else statements that are implemented similarly to other contemporary languages. Comparisons like ==, <, >, <=, >=, and != are available for use in the for loops or if/else statements. The following are some examples of those features:

```
for (x in [1, 2, 3]) {  
  
    print(x)  
  
}  
  
if (x > 0) {  
    for (y in [1, 2] {  
  
        print (x + y) }  
  
}  
  
if (x != y) {  
  
    print(x)  
  
} else {  
  
    print(y)  
  
}
```

Catscript also has print statements that allows users to print specific messages, or other things, to a terminal output. Here are some examples of the print feature:

```
print("Hello World")  
  
var x : list<int> = [1, 2, 3] print(x)
```

Although classes were not implemented for Catscript, functions were implemented. Functions can be declared, defined, and called in programs. As mentioned earlier, types can be implicitly or explicitly declared which greatly increases convenience to users.

Functions are declared using the keyword 'function' followed by the name of the function. Parentheses after the function name define any parameters for the function. Using ':' followed by a variable type specifies if/what a function will return. The following are some examples of Catscript functions and their respective function calls:

```
function helloWorld {  
  
    print("Hello World")  
  
}  
  
helloWorld  
  
function printList(x : list<int>) {  
  
    for(y in x) {  
  
        print(y)  
  
    }  
  
}  
printList([1, 2, 3, 4])  
  
function incrementByOne(x : int) : int {  
  
    x=x+1  
  
    return x  
  
}  
  
var x : int = incrementByOne(4)
```

The following are some examples of Catscript programs that demonstrate most of the features of Catscript:

```
function exampleFunction1(x : int) {
```

```
    print(x)

    if (x > 0) {
        for (y in [1, 2, 3]) {

            print (y)

        }

    }

}

example1(1)
function exampleFunction2(x : int) : int {

    var y = x + 1

    return y

}

for (z in [1, 2, 3]) {

    print(example2(y))

}

for (x in [1, 2, 3]) {
    for (y in [1, 2, 3]) {

        print (x + y)

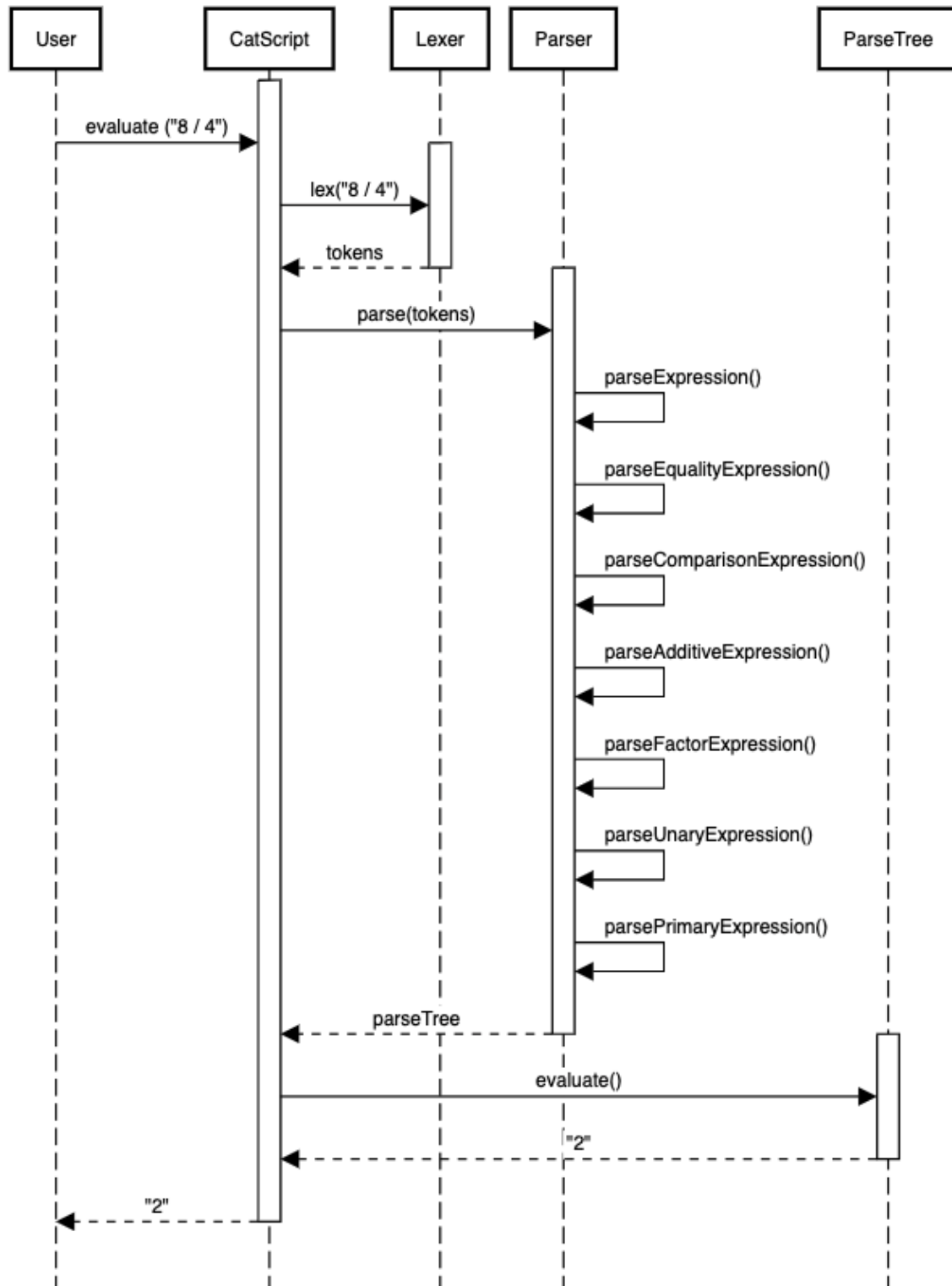
    }

}
```

Section 5: UML

This UML sequence diagram was used to properly develop classes and methods for the parser.

Catscript Factor Expression Evaluation Sequence Diagram



Section 6: Design trade-offs

A design trade-off for this project is not using parse generators. A parser generator takes a grammar or language as input and is able to generate source code that can follow the grammar to parse the right characters. This would have been a simpler way to do the parser for the Catscript Compiler, but it would have significantly reduced the students' learning and understanding of parsing. Instead, we wrote the source code for the parser by hand.

Section 7: Software development life cycle model

A test driven development model was used for software development when implementing the compiler. This model helped break down the daunting task that was this compiler into smaller and more manageable steps. The tests help define the scope of each section of the compiler and make implementing each section more clear. Every method that needed to be implemented was clearly defined by the tests. The test helped drive the work that needed to be accomplished to get a working compiler.