

CSCI 468: Compilers

Senior Team Portfolio

Devon Salveson & Samuel Ertischek

Spring 2023

Part 1: Program

Program Zip

See the source.zip in this directory

Part 2: Teamwork

We both did equal amounts of work on our capstone project. We split the tasks equally between us and frequently collaborated with each other on tasks. For the capstone writeup, Teammate 1 did the documentation of the program while teammate 2 did the other sections.

Part 3: Design Pattern

The design pattern that we used in the project was memoization. Memoization is used for optimization to make the program more efficient and thus faster. This is done by storing results in the cache and pulling that same result from the cache when it is next needed instead of computing it again. An example from our project can be seen in the code below.

```
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;
}
```

Catscript Documentation - CSCI 468

Table of Contents

1. [Introduction](#)
2. [Types](#)
3. [Expressions](#)
4. [Statements](#)
5. [Functions](#)

Introduction

CatScript is a programming language that compiles to JVM Bytecode. It's main features are:

1. A [Type System](#)
2. [Functions](#)
3. Lists
4. For Loops

CatScript can be used for a variety of different purposes, but its main strength is in its ability to be used as a scripting language. An ideal use case would be to use CatScript in the browser to evaluate problems that require loops and lists, rather than taking the time to use JavaScript or Python.

Types

CatScript has 5 main types. They are:

1. Integer

`int`: A signed integer.

2. String

`string`: A string of characters.

3. Boolean

`bool`: A boolean value. `true` or `false`.

4. Object

`object`: A generic object. Can be used to represent any type. The default type for variables initialized without a type.

5. List

`list` or `list<type>`: A list composed of either objects, or the specified type. Lists can be created with a composite type that refers to the type of each value in the list.

The CatScript type system is very simple. Types can be defined when initializing a variable or defining a function. They can be utilized to set the type of a variable, function parameters, and the return type of a function.

Example:

```
var x: int = 5

func add(y: int, z: int): int {
    return y + z
}

z = add(x, 5)
```

Expressions

CatScript has 7 types of expressions. They are:

1. Equality Expression

`comparison_expression { ("!=" | "==") comparison_expression };` Used to check if two values are equal. Returns a boolean.

Example:

```
5 == 5 // true
5 != 5 // false

7 == 5 // false
7 != 5 // true
```

2. Comparison Expression

`additive_expression { (">" | ">=" | "<" | "<=") additive_expression };` Used to compare two values. Returns a boolean.

Example:

```
5 > 5 // false
5 >= 5 // true

7 < 5 // false
7 <= 5 // false
7 > 5 // true
```

3. Additive Expression

`factor_expression { ("+" | "-") factor_expression };` Used to add or subtract two values. Returns an integer or a string.

Example:

```
5 + 5 // 10
5 - 5 // 0

"Hello " + "World" // "Hello World"

5 + 5 - 5 // 5

"I have " + 5 + " apples" // "I have 5 apples"
```

4. Factor Expression

`unary_expression { ("/" | "*") unary_expression };` Used to multiply or divide two values. Returns an integer.

Example:

```
5 * 5 // 25
5 * 0 // 0

10 / (2 * 5) // 1
```

5. Unary Expression

`("not" | "-") unary_expression | primary_expression;` Used to negate a value. Returns a boolean or an integer.

Example:

```
-5 // -5

not true // false
```

6. Primary Expression

`IDENTIFIER | STRING | INTEGER | "true" | "false" | "null" | list_literal | function_call | "(" , expression , ")"`: Used to represent a value. Can return any type.

7. Type Expression

See: [Types](#)

Statements

CatScript has 7 types of statements. They are:

1. For Statement

`for', '(', IDENTIFIER, 'in', expression ')', '{', { statement }, '}'`;: Used to iterate over a list.

Example:

```
var x = [1, 2, 3]

for (y in x) {
  print(y * 2)
}

// 2
// 4
// 6
```

2. If Statement

`'if', '(', expression, ')', '{', { statement }, '}' ['else', (if_statement | '{', { statement }, '}')]`;: Used to execute code based on a condition.

Example:

```
var x = 5
var y = 10

if (x > y) {
  print("x is greater than y")
} else {
  print("x is less than y")
}

// x is less than y
```

3. Print Statement

`'print', '(', expression, ')'`: Used to print the value of an expression.

Example:

```
print("Hello" + " World")
// Hello World
```

4. Variable Statement

`'var', IDENTIFIER, [':', type_expression,] '=', expression;` Used to initialize a variable. Can be initialized with or without a type.

Example:

```
var x = 5
var y: int = 5
var z: string = "Hello"

var comp = z + " World"
```

5. Assignment Statement

`IDENTIFIER, '=', expression;` Used to update a value stored in a variable that has already been initialized with a variable statement.

7. Function Call Statement

`function_call;` Used to call a function that has been defined in the program.

Example:

```
add(5, 5)
```

6. Return Statement

`'return' [, expression];` Used to return a value from a function. Can only be used inside of a function body.

Functions

CatScript has the ability to define and call functions. Functions can be defined with types assigned to the parameters and return type, or without. Functions return values which can be assigned to variables or used in expressions. Recursive functions are supported.

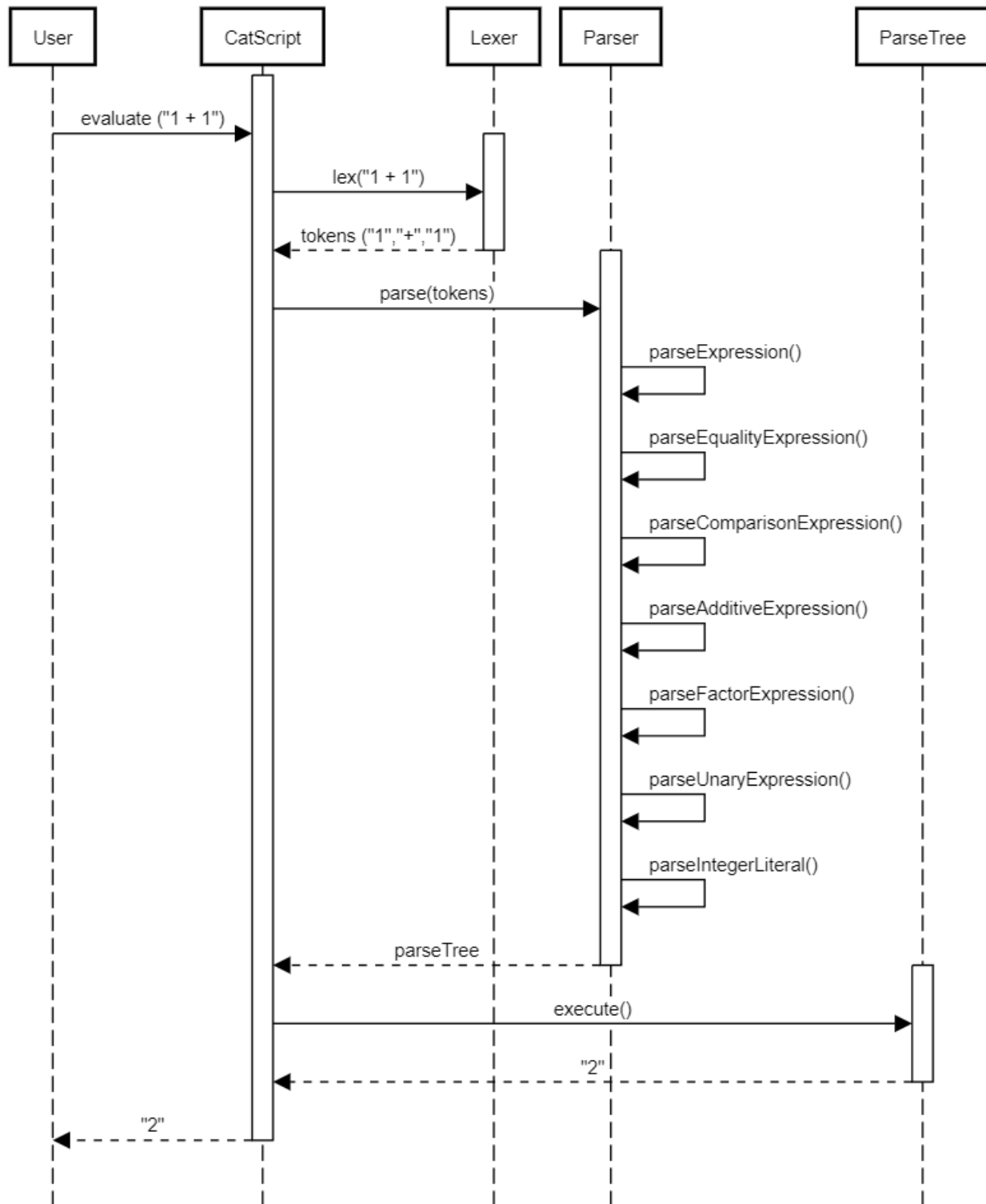
Example:

```
function add(val1: int, val2: int): int {
    return val1 + val2
}

var x = add(5, 5)
// the value of x is 10
```

Part 5: UML

Catscript Addition Sequence Diagram



Part 6: Design Trade-offs

The design trade off that we had in our project was the decision to use the recursive descent algorithm for creating parsers instead of a parser generator. Parser generators are programs that take a language specification and generates a parser for that specification. Whereas recursive descent is a parser that uses recursion to process the input without backtracking. We decided to use the recursive descent model due to it being easier to understand, it letting us understand the recursive nature of grammars and the fact that recursive descent is used way more in the industry.

Part 7: Software development life cycle model

For our Software development life cycle model we used Test Driven Development. This means that tests were created for each functionality before hand and if our code fails the test then new code is written until it passes the test. This was very useful for our project because it allows us to avoid duplication of code and minimize the amount of code needed to check functionality. It also ensures that we are meeting the requirements needed for out project.