

CSCI 468

Spring 2023

Emry Krems

Peyton Dorsh

Section 1: Program

See source.zip in the directory.

Section 2: Teamwork

The teamwork was split into two sections. The first being the development of the software, the second being SQA services. The first responsibility was handled by team number one and the second by team number two. Unit tests were written by team number two after which team number one needed to write source code to make the tests pass. An additional SQA service provided was documentation of the CatScript programming language which is featured in section four.

Section 3: Design Pattern

The **memoization pattern** was used in this project. It is implemented in `src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java`. The memoization pattern was used increase the runtime performance for converting a basic type into a list type. Without using this pattern, every time a list type was needed, it would need to be created from scratch resulting in excess space. In addition, creating a class can be an expensive operation. For this reason, the memoization pattern was chosen to store previously create *ListTypes* in order to eliminate redundant calls to the *ListType* constructor. To implement the pattern, a hash map data structure was utilized. Upon request to get a list type, the map was queried in $O(1)$ time to check if it was already instantiated. If so, the result was returned with no extra space needed. Otherwise, a new list type would be created and stored (memoized) for future

Section 4: CatScript Documentation

Statements

The for Statement

The for statement provides a way to iterate over a set of values. Given a set of values, the for statement will loop over each value and then execute the body within the loop on that value. The for statement exits once all values in the set have been visited.

```
for (num in numbers) {  
    print(num);  
}
```

The if Statement

The if statement provides a way to execute a body of code, only when a certain condition is met. The if statement also provides an optional else statement that has a body of code that executes if the condition for the if statement is not met. If an else statement is not provided and the condition for the if statement is not met, execution will drop down to the code following the if statement and skip executing the if statement's body.

```
var num = 5  
if (num > 3) {  
    print("The number is greater than three");  
} else {  
    print("The number is not greater than three");  
}
```

The print Statement

The print statement allows the representation of data values to be printed to the console.

```
print("String value to be printed");
```

The variable Statement

The variable, var, statement allows a data value to be assigned a reference value with a name/identifier. The value of the variable is saved in memory under the reference of the assigned name. There is an optional field to add a type identifier that specifies what type the data value is. This identifier can be omitted.

```
var one = 1;  
var two : int = 2;
```

The function call Statement

The function call statement is how a list of parameters (which can also be empty) is passed into a

function to call execution on that function. The function is called by using the name of the function declaration.

```
myFunctionWithParams(1, 2, 3);  
myFunctionWithoutParams();
```

The function declaration Statement

The function declaration statement is the signature of a function. It consists of the function name, the list of parameters, an optional return type, and the function body. This information follows the function keyword.

```
function myFunction(one: int, two: int, three: int) {  
    //function body code  
}  
  
function myFunctionWithReturnType(one: int, two: int, three: int): int {  
    //function body code  
}
```

The function body Statement

The function body statement is any statement that lives inside a function declaration statement. The body statements are executed when a function call is made to its respective function declaration. There is an optional return statement that can return a data value from the function.

```
function myFunctionWithReturnType(one: int, two: int, three: int): int {  
    print(one);  
    print(two);  
    return three;  
}
```

The return Statement

The return statement consists of the keyword return followed by the data value(s) to be returned. The type of the data value(s) after the return keyword must match the return type of the function declaration.

```
function myFunctionWithReturnType(str: string): string {  
    return str;
```

```
}
```

Expressions

The equality Expression

The equality expression provides the ability to compare if two values are equal or if they are not equal. The equality expression is often used within an if statement as a condition. To test if two values are equal, use `==`. To test if two values are not equal, use `!=`.

```
var num1 = 1
var num2 = 2
if (num1 == num2) {
    print("equal")
}
if (num1 != num2) {
    print("not equal")
}
```

The comparison Expression

The comparison expression provides the ability to compare two values on whether one is greater than (`>`), greater than or equal to (`>=`), less than (`<`), or less than or equal to (`<=`). The equality expression is often used within an if statement as a condition.

```
var num1 = 1
var num2 = 2
if (num1 < num2) {
    print("num 1 is less than num 2")
}
```

The additive Expression

The additive expression evaluates the addition or subtraction of two values.

```
var add = 1 + 1
var sub = 2 - 1
```

The factor Expression

The factor expression evaluates the multiplication or division of two values.

```
var mul = 2 * 3
var div = 6 / 2
```

The unary Expression

The unary expression negates the value of an expression.

```
var unary1 = -1
var unary2 = not true
```

The primary Expression

The primary expression consists of the different primary values that the other expressions are acting on. A primary expression can be an identifier, string, integer, "true", "false", "null", list, function call, or another expression.

```
X           : IDENTIFIER
"String"    : STRING
2           : INTEGER
"True"      : true
"False"     : false
"Null"      : null
[1, 2, 3]   : list_literal
myFunction() : function_call
```

The list literal Expression

The list literal expression consists of one or more expressions/values inside of brackets. A list can be used to iterate over a for loop for example.

```
var intList = [1, 2, 3]
for (x in intList) {
    //do something
}
```

The function call Expression

The function call expression is similar to the function call statement. The statement ultimately turns

into the function call expression. It has the same form as the function call statement.

```
myStringFunction("A", "B");
```

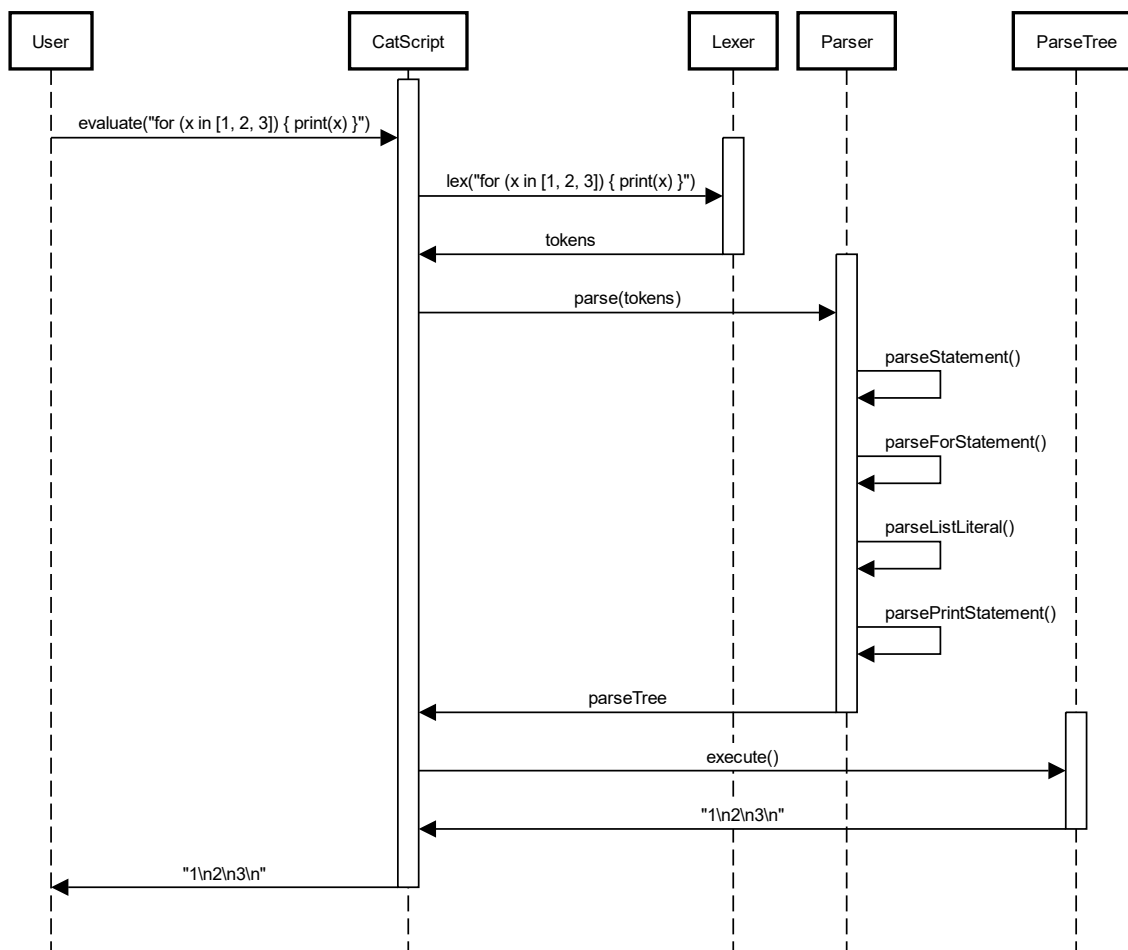
The type Expression

The type expression is what specifies the type of data object. These types can be int, string, bool, object, and list with a type expression which is indicated after the keyword list in angled brackets.

- int
- string
- bool
- object
- list<int>

Section 5: UML

CatScript For Loop Sequence Diagram



This sequence diagram demonstrates the *recursive* nature of recursive descent for a *for loop*. First the user provided source is tokenized. Then statements and expressions are recursively parsed using the CatScript grammar. After the statement is parsed, it is executed, and the result is returned to the user.

Section 6: Design Trade-Offs

The biggest design decision was to use design a recursive parser instead of utilizing a parser generator. Parser generators are the traditional choice when building a compiler. It uses less written code to use than utilizing recursive descent which could lead to the parser being built fast. However, the use of a generator was rejected because of its complexity. From a pedagogical perspective, recursive descent elegantly took advantage of the recursive nature of the language's grammar leading to much deeper understanding of the CatScript language. Another pedagogical advantage of recursive descent is that most languages designed in industry also take advantage of it. Because of these factors, we decided to write a recursive descent parser instead of using a parser generator.

Section 7: Software Development Lifecycle Model

This project utilized test driven development. From my standpoint as the developer, I was provided with unit tests which I needed to write code in order to make the tests pass. Using test driven development was a pleasant experience as it provided me with clear indications of the progress of my code. It also gave clear expectations as to what needed to function in order to create an MVP. Tests acted as guidelines towards which I could focus my efforts. Just as importantly, test driven development provided easy and routine regression testing. This ease of regression testing greatly increased the maintainability of the source code. Whenever changes were made to the source code, all previously passed tests were run to guarantee there were no breaking changes made. After a specific test case was passed, I looked for opportunities to refactor the code in order to improve readability and performance. Overall, test driven development acted as insurance guaranteeing the software I wrote was always of good quality.