

Capstone Portfolio

Jacob Clostio

Compilers

Carson Gross

Due Date: 5/5/2023

Section 1:

Source code included in portfolio submission.

Section 2 Teamwork:

When working on this project, I took the role of the developer in charge of writing code to ensure all of the pieces of the compiler were correct, while my team member took the role of the testing and documentation engineer. The main focus was getting a functional compiler, so that was by far the most time consuming part, taking up approximately 90% of the time of the project. The last 10% was the documentation and testing that my team member did for the project. The tests that they wrote are included here:

```
@Test
void ifInsideForLoopWithListAndElse() {
    assertEquals("Larry (0) Found\nNot Larry (0)\nNot Larry (0)\n", executeProgram(
        "for(x in [0, 5, 25] ) { \n" +
            "if(x == 0 ) { \n" +
            "print(\"Larry (0) Found\") \n" +
            "} else { \n" +
            "print(\"Not Larry (0)\") \n" +
            "} \n" +
            "} \n"
    ));
}

@Test
void nestedIfsAndFors() {
```

```

assertEquals("16\n35\n30\n25\n20\n42\n36\n30\n24\n49\n42\n35\n28\n",
executeProgram(
    "function foo() { \n" +
        "if (23 >= 23) { \n" +
        "for (x in [4,5,6,7]) { \n" +
        "if ( x == 4) { \n" +
        "print(x*x) \n" +
        "} else { \n" +
        "for (y in [7,6,5,4]) { \n" +
        "print(x*y) \n" +
        "} \n" +
        "} \n" +
        "} \n" +
        "} \n" +
        "} \n" +
        "foo()"
));

}

```

@Test

```

void functionReturnWithLists() {
    assertEquals("[YOU, PASSED, !]\n", executeProgram(
        "function foo(x : int) { \n" +
            "if(x == 10) { \n" +
            "for(y in [1,2,3]) { \n" +
            "return [\"YOU\", \"PASSED\", \"!\"] \n" +
            "} \n" +
            "} \n" +
            "} \n" +

```

```
        "print(foo(10))"  
    ));  
}
```

Overall, due to the nature of this project, and the learning outcome of each person needing to be responsible for developing their own working compiler, my team member did well in documenting the code, as well as providing some more high level tests to ensure that the compiler was working as intended.

Section 3 Design pattern:

Design pattern. Identify one design pattern that was used in your capstone project and describe exactly where in the code it is located. Highlight the design pattern in yellow. Explain why you used the pattern and didn't just code directly.

In our project we used the Memoization design pattern in CatscriptType.java as follows:

```
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;  
}
```

The Memoization design pattern is used to optimize the performance of functions by caching the results of function calls and returning the cached value when the same inputs occur again. This helps to avoid repeated computations and improve the overall efficiency of the program. In our code, the Memoization pattern was implemented by defining a Hashmap of LIST_TYPES. When we called `getListType(CatscriptType type)`, we checked to see if the list type was already in our Hashmap. If it was, we returned it. If not, we created a new type, added it to the map, and then returned it. This improved the performance of the compiler, as we didn't have to create a new `listType` every time, we were able to cache some of them for use later.

Section 4 Catscript Documentation:

CatScript Documentation

Kelby Abel

Introduction

CatScript is a simple scripting language that has six data types.

It is statically typed and defined by a set of grammar rules.

Here is an example of CatScript:

```
var x = "foo"
if (42 >= 20) {
  print(x)
}
```

The types include:

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

Features

CatScript Program

The CatScript Program is defined by:

```
catscript_program = { program_statement };
```

This grammar defines the CatScript program as program statement(s).

Program statements are then used to define the syntax of the language.

Program Statement

The Program statement is defined by:

```
program_statement = statement | function_declaration;
```

This grammar defines the Program Statement as statement(s) or function declaration(s).

Statement

The Statement is defined by:

```
statement = for_statement | if_statement | print_statement |  
            variable_statement | assignment_statement |  
            function_call_statement;
```

This grammar defines the Statement as either a for statement, if statement, print statement, variable statement, assignment statement, or function call statement.

This grammar is the main starting point for all existing syntax in the programming language.

For Statement

The For Statement is defined by:

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

Here is an example:

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

In this example, an identifier `x` and an expression `[1,2,3]` are defined inside parenthesis. Then a statement is defined inside the body of the for loop. More than one statement can be included inside of the body.

If Statement

The If Statement is defined by:

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

Here is an example:

```
if (10 == 12) {  
    print(10+12)  
} else if (10 == 10) {  
    print(10+10)  
} else {  
    print(0)  
}
```

In this example there is an expression inside of parenthesis, `(10 == 12)`, this can be any expression that returns a boolean value.

Any If Statement can contain any number of else ifs, as well as an optional else. In this example there is one else if and an optional else. The else if has the required expression `(10 == 10)` and both contain a body.

Print Statement

The Print Statement is defined by:

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

Here are three examples:

```
print("Hello There")  
print("General Kenobi")  
print(x)
```

In these examples there are expressions inside of parenthesis.

Variable Statement

The Variable Statement is defined by:

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


Here are two examples:

```
var senate = "Execute Order"  
var palpy : int = 66
```

In this example, there are the required identifier `senate`, `palpy` and expression `"Execute Order"`, `66`.

The second example includes the optional type of the expression, and the correct expression type.

Any CatScript variable can be any of the available types without specification.

Function Call Statement

The Function Call Statement, Function Call, and Argument List are defined by:

```
function_call_statement = function_call;  
  
function_call = IDENTIFIER, '(', argument_list, ')'  
  
argument_list = [ expression, { ',', expression } ]
```

Here is an example:

```
executeOrderFunction(66)
```

In this example the function `executeOrderFunction` is called with the argument `66`.

Argument lists are optional and are expression(s) separated by commas.

Function Declaration and Function Body Statement

The Function Declaration, Parameter List, Parameter, Type Expression, Function Body Statement and Return Statement are defined by:

```
function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +  
                      [ ':' + type_expression ], '{', { function_body_statement }, '}'  
  
parameter_list = [ parameter, { ',', parameter } ];  
  
parameter = IDENTIFIER [ ':' + type_expression ];  
  
type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [ '<', type_expression, '>' ]  
  
function_body_statement = statement | return_statement;
```

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

Here is an example:

```
function executeOrderFunction(order : int): string {  
    var cody = "Yes my Lord"  
    return cody  
}
```

In this example the required identifier `executeOrderFunction` is followed by a parameter list inside of parenthesis.

The parameter list is a list of parameters separated by commas. In this example `order : int`. Parameters include an identifier, a colon, and a type expression.

A type expression is any of the included data types. A list type expression has an optional list data type type expression. This requires `<>` around the type expression immediately following the type expression list, ex:
`list<int>`.

In this example, the function body statement includes a statement `var cody = "Yes my Lord"` and a return statement `return cody`.

A return statement is only required if a return type is specified in the function declaration statement, otherwise it is void. The return statement requires the `return` keyword and then an optional expression.

Expressions

Expressions and List Literal are defined by:

```
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 } '];
```

These grammars are setup using a hierarchy where primary expression is the lowest level and the highest is equality expression. The complete order is primary expression, unary expression, factor expression, additive expression, comparison, expression, equality expression .

The primary expression includes the most primitive expressions including IDENTIFIER , STRING , INTEGER , true , false , null , list_literal , function_call . These are used in many statements.

The unary expression includes unary operators and not negation. Ex: -501, not true

The factor expression includes multiplication and division operators. Ex: 7567*1138, 5555/99

The additive expression includes addition and subtraction operators. Ex: 3263827+2187, 5127-327

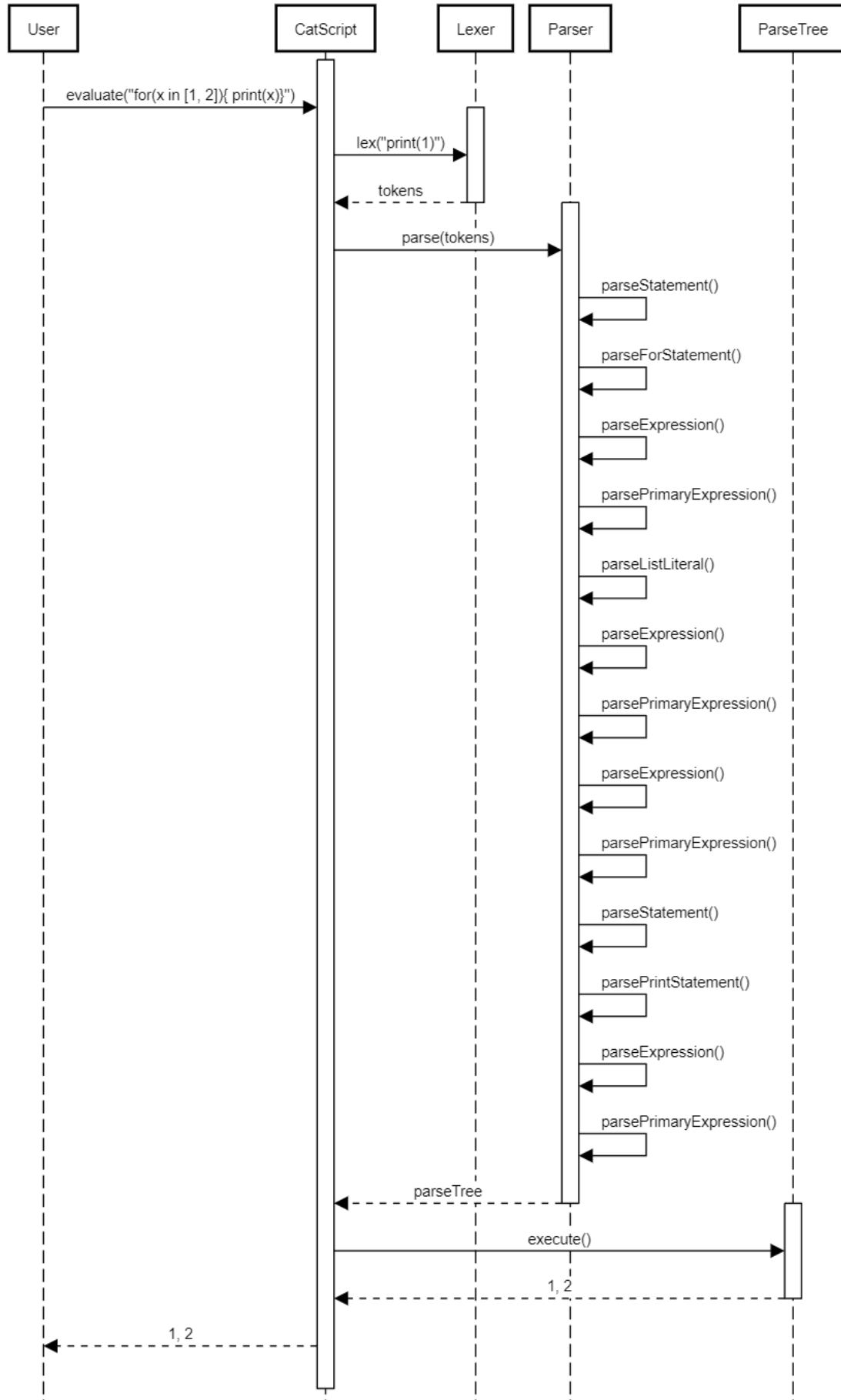
The comparison expression includes relation operators. Ex: 1 > 2, 3 < 4, 5 >= 6, 501 <= 66

The equality expression includes equality operators. Ex: 15 != 11, 421 == 421

Section 5 UML:

Found below is the UML sequence diagram for a For loop statement (that includes a print statement), in Catscript. We first evaluate the string “for(x in [1, 2]) {print(x)}” in Catscript. From there Catscript needs to lex the input and return a list of tokens. These tokens are then parsed in the parser (parsing the for statement, expressions, and the print statement). A parseTree is returned to Catscript, which then calls execute(), and the parseTree returns the printed numbers 1 and 2 to the user.

Catscript For loop Statement (with print) Sequence Diagram



Section 6 Design Trade-offs:

In our project, one of the design trade-offs we faced was whether to write a parser by hand or use a parser generator tool. While using a parser generator tool (such as lex or yacc) could have saved us time in development and made it easier to add features to the language, it could have also increased runtime performance as well as give higher complexity of the codebase.

Therefore, we made the decision to write the parser by hand, despite the increased development time required. This allowed us to have more control over the parsing process (strictly following the grammar of Catscript) and the option to optimize the code for better performance. By writing the parser ourselves, we gained a deeper understanding of the more intuitive recursive descent style algorithm as well as how the Catscript works and how it can be extended in the future. This trade-off allowed us to create a more flexible parser for the creation of Catscript.

Section 7 Software Development Life Cycle:

The software development life cycle model that we used for our project was Test-driven Development. Test-driven Development is a software development approach where tests are written before the actual code, with the goal of ensuring that the code is correct and reliable. This allowed us to pass tests for each feature of the language and establish that they worked as intended before moving on to the next feature. It also provided a clear structure for development and testing, ensuring that our code was always in a working state. However, one potential drawback of Test-driven Development was that it required a significant amount of time and effort upfront to write comprehensive tests for each feature. Furthermore, It may be challenging to write tests for certain types of features, such as those that involve complex user interactions or higher level interactions. Overall, Test-driven Development was a nice way of breaking up the project into smaller chunks that were easily quantifiable, as a checkpoint could subsist of a certain number of tests that needed to pass; this made the whole project more manageable to code and develop.