Capstone

CSCI 468

Compilers Spring 2024

Members: Brianna Clark,

Megan Steinmasel

Section 1: Program

The repository of my code can be found inside of GitHub here: BriMarie933/csci-468-spring2024-private. Also, a zipped-up repo will be included in this directory.

Section 2: Teamwork

In our team, we had two members. I handled all the coding for my own repository and filled out all the necessary sections for testing. Meanwhile, my partner supplied documentation for my repo, viewable below (in section 4), and also included extra tests crucial for ensuring functionality within my repository. In return, I provided my partner with additional tests and crafted her documentation.

We managed to communicate effectively and arrange meet-ups to collaborate on these tests. Initially, there was some confusion regarding the project requirements and objectives. However, we supported each other in understanding the project and demonstrated how to create and write effective tests. Our meeting that day lasted approximately three hours.

Once my teammate grasped the assignment, she swiftly completed the tests, aided by my guidance on test creation. On the other hand, it took me a bit longer to develop my tests, but eventually, I comprehended the evaluation criteria. I spent about two additional hours writing the tests after we met up that day.

Here are my partner's tests:

```
public class PartnerTest extends CatscriptTestBase {

@Test
public void parseListLiteralExpression2() {
   ListLiteralExpression expr = parseExpression("[[1, 2], 2, 3, [], 4]");
   assertEquals(5, expr.getValues().size());
   ListLiteralExpression innerList = (ListLiteralExpression) expr.getValues().get(0);
   assertEquals(2, innerList.getValues().size());
   ListLiteralExpression innerList2 = (ListLiteralExpression) expr.getValues().get(3);
   assertEquals(0, innerList2.getValues().size());
}

@Test
public void ifStatementWithElseParsesWithAdditonStatement() {
```

```
   IfStatement expr = parseStatement("if(7 > 3){ var x = 11 } else { var x = 12 }",
false);
   assertNotNull(expr);
   VariableStatement stat1 = parseStatement("var x = 11");
   VariableStatement stat2 = parseStatement("var x = 12");
   assertTrue(expr.getExpression() instanceof ComparisonExpression);
   assertEquals(1, expr.getTrueStatements().size());
   assertEquals(1, expr.getElseStatements().size());
   assertEquals(stat1.getStart().getType(),
expr.getTrueStatements().get(0).getStart().getType());
   assertEquals(stat2.getStart().getType(),
expr.getElseStatements().get(0).getStart().getType());
}

@Test
void OrderOfOperationsExpressionEvaluatesProperlyWithFunctions() {
   assertEquals(-2, evaluateExpression("1 + 1 - 8 / 2"));
   assertEquals(-2, evaluateExpression("- 8 / 2 + 1 + 1"));
   assertEquals(-2, evaluateExpression("1 - 8 / 2 + 1"));
   assertEquals(-6, evaluateExpression("1 - 8 / 2 * 2 + 1"));  // 1 - 8 + 1
   assertEquals(20, evaluateExpression("8 / 2 + 2 * 8")); // 4 + 16 = 20
   assertEquals("20\n", executeProgram("function foo() : int { return 10 }" +
           "function fun() : int { return foo()+10 }" +
           "print(fun())"
   ));

}

@Test
public void returnIntExprInMultipleFunctionsAndOperate() {
   assertEquals("[1, 2, 3]\n[4, 5, 6]\n", executeProgram("function foo() : list<int> {
return [1, 2, 3] }" +
                   "function fun() : list<int> { return [4, 5, 6] }" +
                   "print(foo())" +
                   "print(fun())"
   ));
   assertEquals("700\n300\n400\n", executeProgram("function foo() : int { return 700
}" +
           "function fun() : int { return 300 }" +
           "print(foo())"+
           "print(fun())"+
           "print(foo() - fun())"
   ));
   assertEquals("700\n300\n1000\n", executeProgram("function foo() : int { return 700
}" +
           "function fun() : int { return 300 }" +
           "print(foo())"+
           "print(fun())"+
           "print(foo() + fun())"
   ));
   assertEquals("700\n300\n210000\n", executeProgram("function foo() : int { return
700 }" +
           "function fun() : int { return 300 }" +
           "print(foo())"+
           "print(fun())"+
```

```
            "print(foo() * fun())"
    ));
    assertEquals("600\n300\n2\n", executeProgram("function foo() : int { return 600 }"
+
            "function fun() : int { return 300 }" +
            "print(foo())"+
            "print(fun())"+
            "print(foo() / fun())"
    ));
}
}
```

Section 3: Design Pattern

Within our project, a prominent design pattern emerges in the CatscriptType.java file, particularly within the assignableFrom() method. This pattern embodies the concept of memoization, strategically applied to optimize the management of types within the Catscript type system.

The Catscript type system encompasses various fundamental types such as int, string, bool, object, null, void, and an array of list types derived from these fundamental types. However, the initial implementation encountered inefficiencies, especially evident in scenarios like List<int>, where multiple instances of the list type were needlessly created.

To address this inefficiency, we embraced the memoization pattern. By leveraging memoization, we aim to consolidate the creation of list types, ensuring that only one instance of List<type> exists for each distinct type, such as integers.

The essence of memoization lies in caching. Specifically, we cache instances of list types, allowing us to swiftly determine whether a particular type already exists or needs to be instantiated. This cache is typically implemented using a HashMap, enabling efficient lookup and management of type instances.

With memoization in place, our system gains the capability to maintain a singular instance of each list type, thereby minimizing resource wastage and enhancing overall performance. By intelligently managing type instantiation through memoization, we achieve a more streamlined and efficient implementation within the CatscriptType.java file.

Here is where the pattern is located:

- The implementation is highlighted in yellow.

```
public boolean isAssignableFrom(CatscriptType type) {
    if (type == VOID) {
        return false;
    } else if (type == NULL) {
        return true;
    } else if (this.javaClass.isAssignableFrom(type.javaClass)) {
        return true;
    }
    return false;
}

// TODO memoize this call? CHECK!!
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    // look in the map // if the thing is already there, return it
    if (cache.get(type) != null) {
        return cache.get(type);
    }
    // if the things isn't there then add it and then return it
    cache.put(type, new ListType(type));
    return cache.get(type); // get the value which is a ListType
}
```

Section 4: Technical Writing

Catscript Guide

Introduction

Catscript is a small statically typed programming language that supports a few Java-like features. Catscript is comprised of expressions, statements, and functions. Various expressions include comparison expressions, equality expressions, and more. Some statements include the

for-statement, if-statement, print-statement, and others. In addition to expressions and statements, Catscript also has the ability to invoke functions.

Types

Catscript is a statically typed programming language where the types of all variables and functions/parameters are known at compile time. The Catscript type system is shown below.

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

Catscript also has one complex type, the list type. You can declare a list of a given type with ‘list’. Example declarations of the list type are shown below.

- ist - list of integers
- list - list of objects
- list<list> - a list of lists of integers

Comments

Comments play a crucial role in Catscript as they allow developers to annotate their code for clarity and documentation purposes without affecting the program's functionality. Single-line comments, denoted by //, are perfect for adding brief explanations or notes to specific lines of code. Meanwhile, multiple-line comments, enclosed within /* */, provide the flexibility to include more extensive descriptions, comments spanning multiple lines, or even temporarily disable blocks of code during debugging or testing phases. Leveraging both single-line and multi-line comments, developers can enhance code readability, collaboration, and maintenance, ensuring that their Catscript projects remain well-documented and comprehensible for themselves and other team members.

```
// Single-line comment
/* Multiple line comment */
```

Print Statements

Catscript's print statements offer a straightforward way to output data to the console, facilitating debugging and program interaction. From basic greetings such as "Hello World" to intricate expressions involving variables and concatenation, Catscript's print statement adeptly manages a wide range of output scenarios. This simplicity makes Catscript a user-friendly choice for programmers accustomed to print statements similar to Python and Java.

```
print(“Hello World”)
```

```
var printNum = 1
print(printNum)
var printStr = "Student ID:"
print(printStr + 10001)
```

Variable Statements

In Catscript, the 'var' statement is used to declare variables, assigning them initial values if desired. In the provided example, a variable named 'x' is declared and initialized with the value 10. This statement signifies that 'x' is a variable that can hold numeric data. Developers can subsequently manipulate or access the value of 'x' throughout their code.

```
var x = 10
```

For Loops

For-loops in Catscript offer a robust method for iterating over collections of data or executing code repeatedly for a specified number of iterations. In the provided example, the loop traverses each element within the 'list' variable. Catscript's for-loop syntax closely resembles that of established languages like Java and Python, ensuring familiarity for programmers. Within the loop's block, developers have the flexibility to define operations to be performed for each iteration.

```
var list = [0, 0, 0, 0]
for (i in list) {
    print(i)
}
```

If Statements

The if-statement in Catscript allows developers to execute specific blocks of code based on conditions. It begins with the 'if' keyword followed by an expression in parentheses, evaluating whether the condition is true. If the condition is met, the code within the following curly braces executes. Optionally, 'else if' clauses can be added, each with its own expression and corresponding block of code to execute if its condition evaluates to true. This enables developers to test multiple conditions sequentially. Finally, an 'else' clause can be included to specify a block of code to execute if none of the previous conditions are met.

```
if (expression){
    \\ statement
} else if (expression){
    \\ statement
}else{
    \\ statement
}
```

Math Operations

Catscript supports fundamental mathematical operations such as addition, subtraction,

multiplication, and division. Developers can use these operators to perform arithmetic calculations within their code. In the provided examples, simple mathematical expressions demonstrate the usage of these operations. The addition operation (1 + 1) evaluates to 2, the subtraction operation (5 - 1) evaluates to 4, the multiplication operation (3 * 1) evaluates to 3, and the division operation (10 / 2) evaluates to 5.

```
print(1 + 1) // Will print out 2
print(5 - 1) // Will print out 4
print(3 * 1) // Will print out 3
print(10 / 2) // Will print out 5
```

Comparison

Catscript includes essential comparison operators, enabling developers to evaluate conditions and make decisions based on comparisons between values. These operators encompass less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=). In the provided examples, these operators are utilized to compare the value 10 with 0, showcasing their functionality. For instance, the expression "10 > 0" evaluates to true, indicating that 10 is indeed greater than 0. Similarly, "10 >= 0" evaluates to true, as 10 is greater than or equal to 0. Conversely, "10 < 0" evaluates to false, as 10 is not less than 0, and "10 <= 0" also evaluates to false since 10 is neither less than nor equal to 0.

```
10 > 0   // true
10 >= 0  // true
10 < 0   // false
10 <= 0  // false
```

Equality

Catscript offers functionality to assess equality and inequality between values using dedicated operators. The double equals sign (==) is employed to evaluate whether two values are equal, while the exclamation mark followed by an equals sign (!=) is used to determine if two values are not equal. In the provided examples, these operators are utilized to compare numerical values. For instance, the expression "1 == 1" evaluates to true, indicating that 1 is equal to 1. Similarly, "10 != 0" evaluates to false, signifying that 10 is not equal to 0.

```
1 == 1 // true
10 != 0 // false
```

Unary Expressions

Unary operators in Catscript allow for the negation of values, with two distinct approaches based on the type being operated on. For variables, the unary minus symbol (-) is used to negate numerical values. In the provided examples, variables 'x' and 'y' are declared with numeric values, and the unary minus operator is applied to 'y' to negate its value. Consequently, when 'x' is added to 'y' in the first print statement, the result is 11. However, when 'x' is added to '-y' in the second print statement, the negated value of 'y' causes the result to be -9.

```
var x = 1
var y = 10
print(x + y)  // Will print out 11
print(x + -y) // Will print out -9
```

For boolean values, the unary negation operator is denoted by the 'not' keyword. In the given example, variable 'x' is initialized as 'true'. When 'x' is printed, the output is 'true'. However, applying the 'not' operator to 'x' in the subsequent print statement results in its negation, changing the output to 'false'.

```
var x = true
print(x)      // Will print out true
print(not x)  // Will print out false
```

Function Definitions

In Catscript, developers can define functions using the 'function' keyword followed by the function name and parentheses containing any parameters. The function body, enclosed within curly braces, contains the code to be executed when the function is called. In the provided example, a function named 'helloWorld' is defined without parameters. Inside the function body, a print statement outputs the message "Hello World" to the console. To invoke the function and execute its code, the function name followed by parentheses is used. In this case, calling 'helloWorld()' prints "Hello World" to the console.

```
function helloWorld() {
    print("Hello World")
}
helloWorld()
```

Catscript Grammar

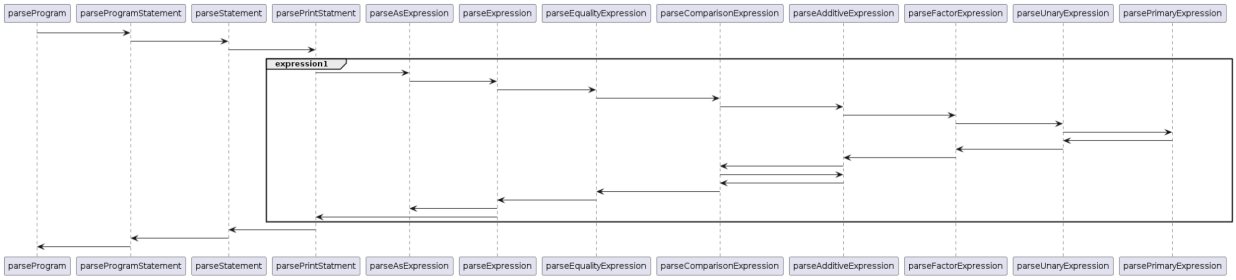
```
program_statement = statement |
                    function_declaration;

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 5: UML

Below is a UML diagram for parse elements. The expression being parsed is a print statement of a equality expression:

printStatement(expression + expression) or *print(1 + 1)*



This diagram illustrates the flow of parsing a Catscript statement, specifically parsing the expression print(1 > 2). The process begins with tokenizing the input string into individual tokens: "print", "(", "1", ">", "2", ")". From there, the parser transitions through various parsing functions, starting with parseProgram and progressing through parseProgramStatement, parseStatement, and finally reaching parsePrintStatement. At this point, the parser enters a group expression where it traverses deeper into the parsing hierarchy.

The parser sequentially calls functions to parse different components of the expression, such as parseAsExpression, parseExpression, parseEqualityExpression, parseComparisonExpression, parseAdditiveExpression, and parseFactorExpression. Each function handles a specific aspect of the expression parsing process, diving deeper into the structure until reaching the base case at parsePrimaryExpression, which handles individual tokens like "1" and "2". Once the base case is reached, the parser begins unwinding the function calls, evaluating conditions at the beginning of each function to determine the next step in the parsing process. Eventually, the parser identifies the ">" token and enters the parseComparisonExpression function, followed by consuming the right-hand side token "2" in the parseAdditiveExpression function.

After completing the parsing process, the parser unwinds back to the initial parsePrintStatement, signifying the successful parsing of the entire expression.

Section 6: Design trade-off

In the realm of parsing and compiler creation, two primary approaches stand out: recursive descent and parser generators. Parser generators are tools that take a language specification

and automatically generate a parser tailored to that specification. However, within this project, we opted for recursive descent. This decision was influenced by several factors, primarily the recursive nature inherent in the Catscript grammar.

Recursive descent proves to be an optimal design choice due to its alignment with the recursive structure of the Catscript grammar. For instance, consider an additive expression in Catscript; it essentially comprises two expressions separated by a "+" symbol. Each of these constituent expressions may further decompose into various types, such as factor expressions or additional additive expressions, thus exhibiting a recursive pattern.

The elegance of recursive descent lies in its ability to establish a structured code design wherein each function call leads to another, gradually traversing through the layers of the Catscript grammar until reaching the base functions. Subsequently, the process reverses, unraveling these nested calls and incomplete functions, meticulously examining if the succeeding tokens align with the requirements of each function.

Furthermore, recursive descent offers a notable advantage in terms of brevity and clarity. Compared to parser generators, which often result in lengthy and intricate code structures, recursive descent implementations tend to be more concise and straightforward. This characteristic facilitates easier debugging processes, as the codebase remains manageable and comprehensible.

In summary, while parser generators present an automated solution for generating parsers, the recursive descent approach emerges as the superior choice for the Catscript project due to its compatibility with the language's recursive grammar structure, as well as its concise and debug-friendly nature.

Section 7: Software development life cycle model

In our project, we embraced Test-Driven Development (TDD) as our chosen approach for software development. TDD follows a systematic process where tests are formulated before the actual implementation of code. These tests, typically provided by the instructor, serve as the guiding criteria for writing precise and functional code.

The essence of TDD lies in its proactive approach towards quality assurance (QA). By beginning with tests, developers are inherently compelled to consider QA aspects from the project's inception. This eliminates the need for extensive post-coding QA efforts and subsequent code refactoring or rework, as the code is iteratively refined to meet the predetermined test criteria.

One of the key advantages of TDD is its emphasis on incremental development. Developers incrementally write small, focused units of code to fulfill specific test cases. This iterative process not only ensures that the codebase remains well-tested throughout development but also facilitates early detection and resolution of issues, thereby enhancing overall code reliability and maintainability.

Moreover, TDD fosters a sense of confidence and clarity within the development team. By having a predefined set of tests, developers gain a clear understanding of the expected behavior and functionality of their code. This clarity minimizes ambiguity and fosters a more structured and disciplined development process.

Personally, I found TDD to be an enjoyable and effective approach. Its integration of QA from the outset streamlines the development process and alleviates the burden of retroactive testing and rework. By adhering to the TDD methodology, our project benefited from enhanced code quality, improved reliability, and a more systematic approach to software development.