Capstone Portfolio

Lucas Rowsey
May 1,2024

Section 1: Program

My source code for this project is attached as a zip. It is a compiler that tokenizes, parses, evaluates expressions, evaluates statements, and creates bytecode.

Section 2: Teamwork

To facilitate collaboration on our project, my partner and I devised a series of tests designed to challenge each other's programs beyond the scope of previously passed assessments. These tests aimed to uncover any latent errors and assess the robustness of our implementations against diverse scenarios.

During the evaluation of my partner's project, we encountered several issues related to the if-else statement. Through collaborative efforts utilizing debugging tools and thorough code comparison, we dissected the codebase to discover the causes of these issues. We were able to find that the issues stemmed from the way that the if-else was being evaluated. Working together, we were able to get the tests working by editing the code they had used without directly copying mine because we had small differences in the way that we wrote it.

The tests that my partner gave to me tested statement functionality, expressions working together and variables. I was able to discover two issues with my code from these tests.

```java
@Test
public void ifStatementFunctional(){
    // Create
    IfStatement ifStat = parseStatement( source: "if (x == 5) { print(x) }");
    // tests if the statement can be properly parsed
    assertNotNull(ifStat);
    // checks if the conditional statement is recognized as an equality expression
    assertTrue(ifStat.getExpression() instanceof EqualityExpression);
    // checks if the if statement executes as expected
    assertEquals( expected: "5\n", executeProgram( src: "if (x == 5) { print(x) }"));
}

new *
@Test
public void expressionTester(){
    // these tests assess the working state of expressions
    // does multiplication work as expected?
    assertEquals( expected: "21\n", compile( src: "7 * 3"));
    // do comparative expressions evaluate properly?
    assertEquals( expected: "true\n", compile( src: "5 > 3"));
    // can parentheses be used in factor expressions
    assertEquals( expected: "25\n", compile( src: "5 * (10 / 2)"));
    // does the additive expression properly handle string cases?
    assertEquals( expected: "1a\n", compile( src: "1 + \"a\""));

}

new *
@Test
public void variables(){
    // create a new variable statement
    VariableStatement newVar = parseStatement( source: "var x : int = 40");
    // check if parses correctly
    assertNotNull(newVar);
    // does the value get stored properly?
    assertEquals( expected: "40", newVar.getExpression().toString());

    // error checking
    VariableStatement newVar2 = parseStatement( source: "var x : string = \"I am a string", verify: false);
    assertNotNull(newVar2);
    assertTrue(newVar2.hasErrors());
    // does the parser recognize the expression even with an error?
    assertTrue(newVar2.getExpression() instanceof StringLiteralExpression);
}
```

Along with the testing we wrote and then edited documentation for each other. My partner's documentation is included in section 4 of this document.

Section 3: Design Pattern

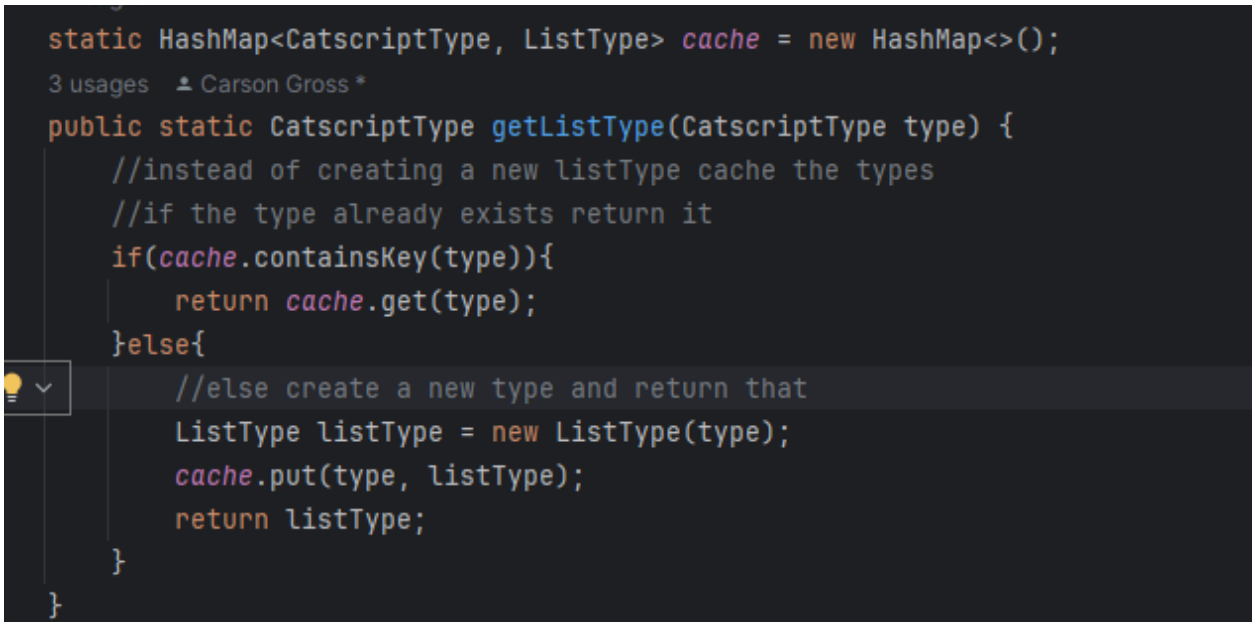
In my project, I employed the memoization design pattern to optimize performance by reducing redundant method calls. Memoization leverages data structures such as hashtables, hashmaps, or arrays to store previously computed results, thereby minimizing computation overhead.

For my implementation, I opted for a Hashmap to cache instances of ListType, ensuring efficient retrieval without the need for repetitive instantiation. The crux of the method lies in its utilization of a static hashmap, serving as the repository for cached ListType instances.

When the getListType method is invoked, it first queries the cache to determine if the requested ListType exists. If a cached instance is found, the method promptly returns it, obviating the need for redundant instantiation. However, in the absence of a cached instance, the method proceeds to create a new ListType and stores it in the cache for future access.

By employing this memoization strategy, the getListType method optimizes performance by circumventing unnecessary computations and leveraging precomputed results when available. This not only enhances the efficiency of the method but also contributes to overall system performance by minimizing resource utilization.

In essence, the memoization design pattern serves as a cornerstone of optimization in my project, exemplifying a proactive approach to performance enhancement through intelligent caching and reuse of computed results.


static HashMap<CatscriptType, ListType> cache = new HashMap<>();
3 usages Carson Gross *
public static CatscriptType getListType(CatscriptType type) {
//instead of creating a new listType cache the types
//if the type already exists return it
if(cache.containsKey(type)){
return cache.get(type);
}else{
//else create a new type and return that
ListType listType = new ListType(type);
cache.put(type, listType);
return listType;
}
}

Section 4: Technical Writing
Catscript Guide

This document should be used to create a guide for catscript, to satisfy capstone requirement 4

Introduction

Catscript is a simple scripting language. Here is an example:

```
var x = "foo"
print(x)
```

Features:

Statements

Catscript features a wide variety of statements and function declarations, including for loops, if-else conditionals, print statements, variable declarations, assignment statements, and function calls.

For loops

- For loops are an iteration statement that is able to execute an expression or statement as many times as the user desires.
- Each for loop starts with the keyword "for", then takes an identifier and expression as flow control, and contains a statement body with all the necessary code to execute.

```
for (x in [5, 6, 7]){
   x = x + 1
   print(x)
}
```

If-Else conditional

- The if-else conditional is a statement that can execute specific blocks of code depending of if a certain condition is met.
- if the condition is evaluated to be true, the block will run.

- if the condition is evaluated to be false, the entire statement within the if will skip.
- else statements can follow if statements, and, should the user desire, can include another conditional to be satisfied or skipped, or just a plain else.

```
if (true) {
   print("I am a true statement")
} else {
   print("I am a false statement")
}
```

Print Statements

- A print statement is a simple statement that outputs the value of an expression.

```
print("I am a print statement")
```

Variable Declaration

- Catscript variables are declared with the "var" keyword, followed by an identifier, an optional type, and an initialization value.

```
var identifier : type = expression
```

```
var name : string = "name"
```

Variable Assignment

- Catscript variables can be given new values using the equals (=) operator.

```
name = "A new name"
```

Function Call Statements

- Function calls in catscript consist of the function identifier, followed by a list of arguments that the function requires.

```
exampleFunc(25, 15, name)
```

```
simpleFunc(userInput)
```

Function Declaration

- Catscript features the ability to declare unique functions
- Functions are declared with the "function" keyword, followed by an identifier or function name, then a list of parameters, an optional return type, and a statement body.

```
function calculateSquare(userInput){
   return userInput * userInput
}

function introduction(name){
   print("Hello, my name is " + name)
}
```

Return Statements

- Return statements begin with the "return" keyword, followed by an expression.
- In catscript, return statements are found at the end of function bodies.
- A return statement can pass the specified expression or return value to the code that called the function.

```
function foo() : {return true}

return 40
```

Expressions

Expressions in CatScript can be equality expressions, comparison expressions, additive expressions, factor expressions, unary expressions, or primary expressions

Equality Expression

- An equality expression compares two given values to each other.
- Depending on the operator (==) or (!=), the expression will evaluate if values are the same or if they are not the same.
- Equality expressions evaluate as a boolean, returning true or false.
- The equals (==) expression evaluates if given values are the same, returning true if so, or false if not.
- The not equals (!=) expression evaluates if the values given are different, returning true if they are different, and false if not.

```
25 == 25
10 != 15
```

Comparison Expression

- Comparison expressions, like equality expressions, compare two values to each other.
- Evaluates as a boolean
- There are 4 different operators for comparison expressions:
- Greater Than: The greater operator (>) checks if the left hand side holds a greater value than the right hand side
- Greater Than / Equal To: The greater than / equal to operator (>=) checks if the left hand side holds an equal or greater value than the right hand side
- Less Than: The less operator (<) checks if the left hand value is less than the right hand value
- Less Than / Equal To: The less than / equal to operator (<=) checks if the lesft hand side value is either equal to or less than the right hand value.

```
5 > 1
2 < 7
5 <= 5
8 >= 3
```

Additive Expression

- Additive expressions add or subtract left hand side and right hand side values.
- When using the (+) operator, the left and right hand side values will be added together.
- Secondly, the (+) operator can also combine string values.
- When using the (-) operator, the right hand side will be subtracted from the left hand side.

```
2 + 2
4 - 3
"foo" + "bar"
```

Factor Expression

- Factor expressions are used to multiply and divide left and right hand side values
- Evaluates as an integer.
- The (*) operator multiplies left and right hand side values
- The (/) operator divides the left hand value by the right hand value

```
4 * 12
```

10 / 2
2 * 2 * 2
20 / (1 * 2)

Unary Expression

- Catscript unary expressions change values.
- The not operator flips boolean values
- (-) changes positive integers to negative.

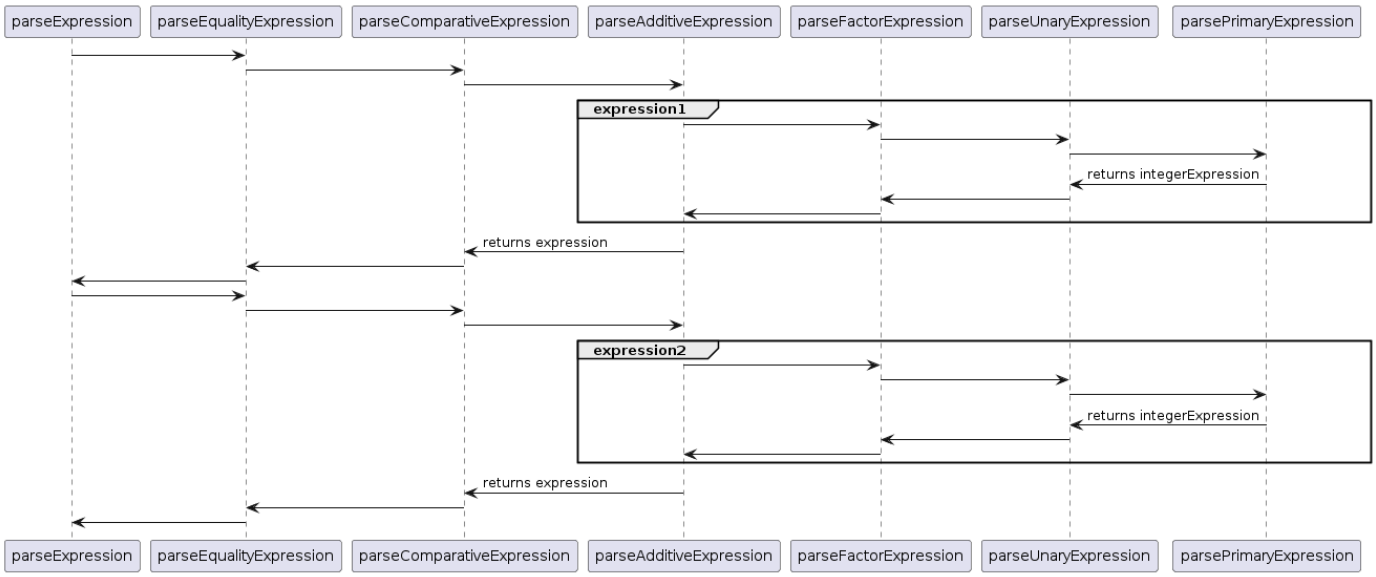
not true
not false
-30

Type Expression

- Catscript type expressions display the data type of a value.
- Type expressions in catscript include int, string, bool, object, and list.

Section 5: UML

For my UML diagram I chose to show recursive descent using a sequence diagram. This is the diagram for parsing an additive expression.



Section 6: Design Trade-offs

Recursive descent parsing is a top-down parsing technique commonly used in compiler construction. It involves recursively traversing the input source code and constructing a parse tree based on production rules defined by a formal grammar. This method offers simplicity and ease of understanding, as developers can visualize the parsing process through the parse tree. Additionally, recursive descent parsers are straightforward to debug and modify, facilitating rapid development and iteration. However, a notable drawback is its lack of scalability, as the recursive nature of the algorithm can lead to exponential runtime in certain cases.

On the other hand, parser generators automate the process of generating parsers from formal grammar specifications. By taking input in the form of a grammar definition, parser generators produce executable code or parser modules capable of parsing input according to the specified grammar rules. The primary advantage of parser generators lies in their scalability and efficiency. Generated parsers are optimized for performance and can handle complex and large-scale grammars with ease.

In summary, while recursive descent parsing excels in simplicity and ease of use, it may struggle with scalability due to potential runtime issues. In contrast, parser generators offer scalability and efficiency but may require a learning curve to master and configure effectively.

I really liked this design when dealing with order of operation for expressions.

```
private Expression parseAdditiveExpression() {
    //call parse factor before checking for +-
    Expression expression = parseFactorExpression();
    while (tokens.match(PLUS, MINUS)) {
        //while there are more signs parse
        Token operator = tokens.consumeToken();
        //set the right hand side of the expression
        final Expression rightHandSide = parseFactorExpression();
        //create an additive expression with the operator expression from factor and the right hand value
        AdditiveExpression additiveExpression = new AdditiveExpression(operator, expression, rightHandSide);
        //set start and end
        additiveExpression.setStart(expression.getStart());
        additiveExpression.setEnd(rightHandSide.getEnd());
        //set the expression value to be returned
        expression = additiveExpression;
    }
    //returns the expression the value will be changed if there was a - or +
    return expression;
}
```

This code snippet is my parseAdditiveExpression. This will set the value of an expression to parseFactorExpression, meaning if there is a factor expression, it will be evaluated before the addition. This will be the same for factor and parenthesized expressions.

```
}else if(tokens.match(LEFT_PAREN)) {
    // if there is a paren consume paren then parse expression
    Token parenToken = tokens.consumeToken();
    //create expression
    Expression expression = parseExpression();
    //make a paren expression with the parsed expression
    ParenthesizedExpression parenthesizedExpression = new ParenthesizedExpression(expression);
    parenthesizedExpression.setToken(parenToken);
    parenthesizedExpression.setEnd(require(RIGHT_PAREN,parenthesizedExpression));
    return parenthesizedExpression;

}
```

In this snippet, you can see that if a parenthesized expression is found, it will take that expression and run it through the parser again, then return the value. This means that no matter what, the parenthesized expression will be run before factor, and factor will run before addition, keeping the correct order of operations. It also means that I am able to reuse code instead of having to rewrite or have a bunch of different function calls.

Section 7: Software Development Life Cycle Model

In our project, we embraced Test-Driven Development (TDD), a methodology where automated tests are crafted before writing any software code. This approach ensured that our development process was guided by clear objectives defined by the tests.

In a classroom setting, TDD proved to be a highly effective model. With tests outlining specific requirements, I always had a tangible goal to work towards. For instance, if a test specified an integer value of 3, I could immediately pinpoint the necessary modifications in the code to meet that expectation. This direct correlation between tests and code facilitated a structured and focused approach to development.

Moreover, the grading system based on the percentage of passing tests provided a transparent and objective measure of progress and proficiency. It incentivized thorough test coverage and robust code implementation, contributing to a deeper understanding of software quality principles.

However, in real-world scenarios, the practicality of TDD may vary. While it promotes disciplined development practices, there are potential pitfalls to consider. One concern is the possibility of "gaming" the system by modifying code to pass tests without truly addressing underlying issues. This highlights the importance of complementing TDD with code reviews, peer feedback, and other quality assurance measures to ensure the integrity of the software.

Additionally, pre-writing tests in TDD may inadvertently lead to oversight or incomplete coverage of edge cases and real-world scenarios. While tests provide valuable insights into expected behavior, they may not capture the full spectrum of user interactions

and system complexities. Therefore, it's essential to supplement TDD with exploratory testing, user acceptance testing, and continuous monitoring to identify and address potential blind spots.

In conclusion, while TDD offers numerous benefits, its effectiveness depends on the context and the diligence with which it's implemented. In both educational and professional settings, TDD serves as a valuable tool for promoting code quality, but it should be supplemented with comprehensive testing strategies and ongoing evaluation to ensure robust and resilient software solutions.