CAPSTONE PORTFOLIO

Keegan Gaffney

Fletcher Phillips

capstone/portfolio/keegan_gaffney_portfolio.md

Overview

This is a capstone project for the CSCI 468 class.

Section 1: Program

The source code to my compiler can be found in this directory, in the source.zip file.

Section 2: Teamwork

I worked on this project with Fletcher Phillips. He wrote the below tests and Catscript documentation for my compiler, and I did the same thing for him. The documentation he wrote can be found in section 4. As for these tests, they can be found and run at `src/test/java/edu/montana/csci/csci468/demo/PartnerTest.java`. Here are the tests he wrote for me:

1. Test nested statements and expressions function properly.

   ```
   @Test
   public void nestedIfStatementWithAdditiveExpression() {
     ForStatement statement = parseStatement("for (value in [1, 2, 3, 4, 5]) { if((value + 1) == 2
     assertNotNull(statement);
     assertTrue(statement.getExpression() instanceof ListLiteralExpression);
     assertEquals(1, statement.getBody().size());
     assertTrue(statement.getBody().get(0) instanceof IfStatement);
     IfStatement ifStatement = (IfStatement) statement.getBody().get(0);
     assertNotNull(ifStatement.getExpression());
   }
   ```

   This test checks that my compiler can parse the following Catscript code:

   ```
    for (value in [1, 2, 3, 4, 5]) {
      if((value + 1) == 2) {
        print(value)
      }
    }
   ```

   This code snippet does a good job of testing the parser's ability to handle nested statements and expressions. The for loop contains an if statement that first evaluates an additive expression, then checks if the result is equal to 2. If it is, it prints the value of the variable `value`. This test is important because it ensures that the parser can handle complex nested statements and expressions.

2. Test complex uses of parenthesized expressions.

   ```
   @Test
   public void parenthesizedExpressionWithLHSandRHS() {
     CatScriptParser catScriptParser = new CatScriptParser();
     CatScriptProgram program = catScriptParser.parseAsExpression("((12434543 * (54332 + 321312))
     Expression expression = program.getExpression();
     assertTrue(expression instanceof AdditiveExpression);
     AdditiveExpression subtractionExpression = (AdditiveExpression) expression;
     assertTrue(subtractionExpression.getLeftHandSide() instanceof ParenthesizedExpression);
     assertTrue(subtractionExpression.getRightHandSide() instanceof ParenthesizedExpression);
   }
   ```

   This test checks that my compiler can parse the following Catscript expression:

   ```
   ((12434543 * (54332 + 321312)) / 212312) - (3437 * (544 - (455432 + 43351)))
   ```

   This test is important because it ensures that the parser can handle complex parenthesized expressions with multiple levels of nesting. The expression contains multiple levels of nested parentheses and mathematic operations. This test verifies that the parser can handle these complex expressions calculate the correct result.

3. Test function declaration with if statement.

   ```
   @Test
   public void functionDeclarationWithIfStatement() {
     FunctionDefinitionStatement statement = parseStatement("function returnTest(x, y) : int { if
   ```

```
    assertNotNull(statement);
    assertEquals("returnTest", statement.getName());
    assertEquals(1, statement.getBody().size());
    assertTrue(statement.getBody().get(0) instanceof IfStatement);
    IfStatement ifStatement = (IfStatement) statement.getBody().get(0);
    assertEquals(1, ifStatement.getTrueStatements().size());
    Expression ifStatementExpression = ifStatement.getExpression();
    assertTrue(ifStatementExpression instanceof EqualityExpression);
    EqualityExpression equalityExpression = (EqualityExpression) ifStatementExpression;
    Expression lhs = equalityExpression.getLeftHandSide();
    assertTrue(lhs instanceof ParenthesizedExpression);
    ReturnStatement returnStatement = (ReturnStatement) ifStatement.getTrueStatements().get(0);
    assertNotNull(returnStatement);
    assertTrue(returnStatement.getExpression() instanceof AdditiveExpression);
    AdditiveExpression additionExpression = (AdditiveExpression) returnStatement.getExpression();
    assertTrue(additionExpression.getLeftHandSide() instanceof ParenthesizedExpression);
    assertTrue(additionExpression.getRightHandSide() instanceof IntegerLiteralExpression);
}
```

This test checks that my compiler can parse the following Catscript code:

```
function returnTest(x, y) : int {
  if ((2+1) == 3) {
    return (2 + 5) + 2
  }
}
```

This test is important because it ensures that the parser can handle function declarations with a return statement, that is nested in the function rather than directly in the functions scope. The function declaration contains an if statement that first evaluates an equality expression, then checks if the result is equal to 3. If it is, it returns the result of an additive expression. This test actually required a few changes to the parser to handle the nested return statement, so it was good that Fletcher chose to test this.

Section 3: Design pattern

The design pattern used in this project is called memoization. Memoization is the process of storing the results of complex calculations, so that you don't need to compute it multiple times. This pattern is used in the Catscript type system, specifically for retrieving the listType of a list literal. The code that accomplishes this can be seen below.

src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java:

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();

public static CatscriptType getListType(CatscriptType type) {
    return cache.computeIfAbsent(type, ListType::new);
}
```

This code snippet shows the very basic implementation of memoization that we are using in this compiler project. First, we created a new HashMap called cache that indexes CatscriptType objects, and stores the associated ListType objects. The getListType function takes in a CatscriptType, and performs the computeIfAbsent function on the cache. This function will check if the CatscriptType is already in the cache, and if not, will create a new instance of the ListType class, and store it. If the CatscriptType is found in the cache, the function will immediately return the related ListType object. This is the textbook definition of memoization, where instead of recomputing, we simply return the cached result.

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:

Type Literal Expressions:

Catscript has the following types

- bool - booleans are conditional true or false values
  - Example: `true, false`
- int - integers `Numbers such as 1, 2, 3, 443543, 2626`
- list - a list of values (can be any type or generic) enclosed by a left and right brace `[1,2,3,4] [true, false] [“String1”, “String2”]`
- string - strings surrounded by quotations `”Hello World”`
- object - any type
- null - null value

```
null
var x = null
```

Statements

For Loop:

For loops in Catscript are loops that will loop through a list of values. As the loop iterates over the values, the code in between the braces will run. The loop will traverse over each value in the list provided, therefore allowing an action to be performed for that value in the list. A for loop can iterate over any type in a list. Variables that are defined inside of the scope of the for loop can only be used in the for loop scope. Look at the example below for iterating through a list of strings. Example:

```
for(value in [“String1”, “String2”, “String3”]){
   print(value)
}
```

Output:

```
String1
String2
String3
```

Print Statement:

A print statement is a statement in Catscript that is used to output information to the console. Whatever is in the parentheses will be evaluated and displayed. Print statements are good for displaying useful information to the console that is relevant to the program. Here is a simple example of printing a string to the console. You can start a print statement with the "print" token. You can have any type of expression inside of the parentheses of the print statement.

```
print(“Hello World”)
```

Output:

```
Hello World
```

Variable Statement:

Variable Statements in Catscript are statements used to declare a variable. In Catscript a variable can be explicitly or implicitly declared for its type. Having the flexibility of implicit and explicit declarations will allow a developer to write code that is better suited to the program. With the ability to represent and store different types, variable statements are crucial to the Catscript programming language. Variables are only accessible in the scope that they were defined in. Those that are globally defined can be used across the entire program, unlike those that are defined inside of a different statement such as a function.

```
// explicit type declaration
var explicit: string = “Hello World”
print(explicit)


// implicit type declaration
var implicit  = “Hello World”
print(implicit)
```

Output:

```
Hello World
Hello World
```

Assignment Statement:

Assignment Statements in Catscript are statements that are used to assign/store a value to a variable. Whatever follows the = sign on the right side will be the value assigned to the variable on the left side of the =. You cannot assign a new type of variable to an already assigned variable. You can assign a new value of the same type to the variable as seen in the example code below. The variable you are assigning a value to must also be inside of the scope the variable was defined in, except for global variables.

```
var number = 231564
number = 465132
print(number)
```

Output:

```
465132
```

If Statement:

An if statement in Catscript is a control flow statement that will evaluate a condition inside of its parentheses. After it evaluates the condition in the parentheses, it then executes the code inside its braces if the condition is evaluated to be true. It may also be followed by an else statement if the condition evaluates to false. Using if statements allows the developer the ability to control the execution of their program.

```
var x = 1
if(x == 1){
 print(x)
}else{
 print(“Not 1”)
}
```

Output:

```
1
```

Function Declarations:

Function Declarations in Catscript declare functions. These functions contain blocks of code that can be called/referenced in another spot of the code. By having a function declaration a developer may be able to create more modular programs. Function declarations have no direct output. In the example below you can see the syntax of saying function then the function name for our case "func". Then we put in our optional parameters. After the parenthesis, you can optionally include a return type with a ':'. The code you want to be invoked on the call of this function is what will be in between the curly braces.

```
function func(parameter1: string){
   print(parameter1)
}
```

Function Call Statement:

Function Call Statements in Catscript invoke a function that was previously declared using a function declaration statement. When called it will execute the code of the functions code block. You can pass parameters to function call statements if they are required. Being able to pass different parameters to these function call statements ensures the modularity of the program and the reusability of code.

```
function func(parameter1: string){
   print(parameter1)
}
func(“Hello World”)
```

Output:

```
Hello World
```

Return Statements

Return Statements in Catscript are statements that will return data to the program. A return statement will end the execution of a function code block. After a return statement is reached in the flow of the program, the code will resume outside of function and continue its execution. Return statements can only be used within the scope of the function declaration.

```
function func(parameter1: string) : string {
   return(parameter1)
}
print(func(“Hello World”))
```

Output:

```
Hello World
```

Expressions:

Additive Expression:

Additive expressions in Catscript are used to add or subtract numbers or concatenate string values together.

```
2 + 2
2 – 1
“Hello ” + “World”
```

Output:

```
4
1
Hello World
```

Factor Expressions:

Factor Expressions in Catscript are used to divide or multiply values.

```
2 ∗ 2 (multiplication)
2 / 1 (division)
```

Output:

```
4
1
```

Equality Expressions:

Equality Expressions in Catscript are used to compare values to each other and will be evaluated as either true or false.

```
true == true
false == true
false != true
4 == null
```

Output:

```
true
false
true
false
```

Comparative Expressions:

Comparative Expressions in Catscript compare values and will be evaluated to be either true or false.

```
4 > 2 (greater)
4 < 2 (less than)
4 >= 4 (greater than or equal)
4 <= 4 (less than or equal)
```

Output:

```
true
false
true
true
```

Unary Expressions:

Unary Expressions in Catscript negate a value.

```
not true
-1
```

Parenthesized Expressions:

Parenthesized Expressions in Catscript are used to group and give precedence to math expressions.

```
(2 + 2) - 2
```

Output:

```
2
```

Identifier Expressions:

An expression that uses a certain name to refer to an entity in the program.

Function Call Expressions:

Function call expressions are the use of a function within an expression.

Section 5: UML

 Parser Sequence Diagram The above diagram shows a sequence diagram for parsing the below Catscript code:

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

The diagram seems a bit complex for how simple the code is, but it is actually not too bad when you take a closer look. This compiler uses a parsing method called recursive descent parsing. This means that the parser will start at the top layer of parsing, and will recursively break down the statement into it's smallest components. For this example, we start by parsing the for loop, removing the related tokens for that, getting the identifier, then recursively parsing the list literal expression. This recursive decent processes starts by calling parseExpression, then recursively calling the next expression parsing function until the correct tokens are found. The parser attempts to parse an equality expression, then comparison, then additive, then factor, then unary, then finally the primary expressions. After the parser has determined that the expression is a list literal expression, it then has to loop through the list and perform the recursive descent parsing on every element in the list. After the list literal is complete, the parser will then move onto the for body which has a print statement in it. The parser then calls the parsePrintStatement function, which again recursively calls the parseExpression function to get the expression inside of the print statement. After this the parser returns the final parsed statement back to the top parse function. This sequence diagram shows just how complex the recursive descent parsing can be, however, while it may be complex, it also shows that it is very powerful and can efficiently parse complex statements.

Section 6: Design trade-offs

The main design tradeoff we made for this project was to use a recursive descent parser rather than a parser generator. Recursive descent parsers work by parsing the input from the top down, recursively. You typically have some sort of main parse function, that first attempts to parse all the different statements, then each of those functions will attempt to parse the different expressions. A typical run of this may go from the parse function, to parsePrintStatement, to parseExpression, then through all the different expressions until it is able to parse the entire statement. Recursive descent parsers are very powerful, and surprisingly intuitive. You get a really good sense of the recursive nature of these grammars, and gain a much better understanding of how the parser actually works by writing it yourself. While recursive descent parsers are great and what we ended up going with for this project, they aren't perfect. Recursive descent parsers can quickly grow in complexity when the grammar it is parsing gets more complex. It also tends to be slightly slower than parser generators.

Parser generators are the other possible option we could've chosen for writing our parser. Parser generators are separate tools that take in a grammar, and generate the parser code for you. This sounds like a great tool on the surface, and it can be, but it isn't perfect. When you use a parser generator, you give up a lot of control over the code you are using. Code generators tend to generate very large and complex code that can be extremely difficult to read and debug. This is a big problem because if you run into any issues, you are going to spend much longer trying to

figure out what went wrong and where. By having the code generated for you, you also miss out on the experience of writing and understanding the parser yourself. In a school setting, this is a huge downside because you are missing out on one of the most important parts of the class. Again, parser generators can be a great tool for these compilers. The ease of initial use where you can simply write the grammar and have everything else done for you is a huge advantage. They also tend to be faster than recursive descent parsers. However, for this project, it was more important to focus on the ease of debugging, and the learning experience that comes with it. That is why we chose to use a recursive descent parser for this project.

Section 7: Software life-cycle

The software development life-cycle we used for this project was test based development (TBD)/test first development (TFD). TBD/TFD is a method of software development where you start by writing all of the tests to make sure all of the planned functionality works correctly. It isn't until after all the tests are written where you actually start implementing the planned functionality. Personally, I am pretty divided on this method of software development. I feel that in a perfect world, this is a great idea. Once you have all your tests, implementation to make them all pass becomes easy. It seems great! In reality however, I feel it is unrealistic for any moderately complex project. In my experience, it is extremely difficult to write tests for code that does not exist. I also feel that if you have put enough planning in to be able to perfectly test all the code before it is written, you have probably done too much. Software development is an iterative process, and there is always going to be something that doesn't go to plan. With TBD/TFD, if you run into any of these problems, you have to go back and rewrite the test again. This means you essentially have to write the tests twice. My preference is to write the tests as you go. It is too challenging and can be a waste of time to do it first, and you will almost never get to it if you wait till the end. Writing tests as you go however seems to be a good middle ground. This way you make sure to stay on top of it, and can write the tests while you are still fresh on the code you are testing. This is the method we are using at my internship for Land id and I feel it is a good balance between the two.

The interesting thing about this project however, is that I feel TBD/TFD was actually a great choice. In a school setting, I know it can be extremely challenging to have students know exactly what they are supposed to do, without being given the exact code. Even when students fully understand what they are doing, it is so easy for them to miss something. This doesn't necessarily mean their code is wrong, just not what the grader is expecting. I have experienced this first hand as a TA for the Web Design class last year. I feel that TBD/TFD is the perfect solution to this problem. Carson already needed to have a working compiler done to be able to effectively teach this class. This made it easy to write the tests, then remove the important parts of the code that students need to figure out themselves. In this way, it doesn't matter if students write almost identical code to what the professor has, all that matters is that the tests pass. This gave us students a great opportunity to implement a working compiler, while still going through the process of writing the code ourselves in our own style. That is why I feel TBD/TFD was a great choice for this project, and am glad we used it.