CSCI 468

Compilers

Spring 2024

Ryan Armstrong

Justin Smith

Section 1:

The final repository is in a zip file called source in the portfolio folder. The source code in that source folder is a complier for a language CatScript. It includes a tokenizer, parser, and bytecode generator. The following sections explain the functionality and process used to create the parser.

Section 2

For teamwork we split the jobs into two main parts. The first partner was responsible for creating a document to allow people who have never used catscript to read the documentation and write a program in CatScript. For this they had to document the different expressions and statements included in CatScript and give a description of each one as well as how to use it. The first partner was also responsible for additional testing. For this the partner had to go through the tests already provided to us and find things that were untested. Once they found things that were untested they had to write test cases to ensure proper functionality of the CatScript compiler. Below are the additional test he contributed.

```
@Test
public void ifStatementWithElseWithNestedIfElseParses() {
    IfStatement expr = parseStatement("if(x > 10){ print(x) } else { if(x ==
5){ print(x) }"
            + "else { print(3) } }", false);
    assertNotNull(expr);
    assertTrue(expr.getExpression() instanceof ComparisonExpression);
    assertEquals(1, expr.getTrueStatements().size());
    List<Statement> elseStatements = expr.getElseStatements();
    assertEquals(1, elseStatements.size());
    Statement stmt = elseStatements.get(0);
    assertTrue(stmt instanceof IfStatement);
    IfStatement ifStmt = (IfStatement) stmt;
    assertTrue(ifStmt.getExpression() instanceof EqualityExpression);
    assertEquals(1, ifStmt.getTrueStatements().size());
    assertEquals(1, ifStmt.getElseStatements().size());
}

@Test
public void functionDefWithParamsWithTypesParses() {
    FunctionDefinitionStatement expr = parseStatement("function x(a :
list<int>, b : list<string>, c : list<object>)"
            + "{ print(1) }");
    assertNotNull(expr);
    assertEquals("x", expr.getName());
    assertEquals(3, expr.getParameterCount());
    assertEquals("a", expr.getParameterName(0));
    assertEquals("b", expr.getParameterName(1));
    assertEquals("c", expr.getParameterName(2));
    assertEquals(CatscriptType.getListType(CatscriptType.INT),
expr.getParameterType(0));
    assertEquals(CatscriptType.getListType(CatscriptType.STRING),
expr.getParameterType(1));
    assertEquals(CatscriptType.getListType(CatscriptType.OBJECT),
expr.getParameterType(2));
    assertEquals(1, expr.getBody().size());
}

@Test
```

```
void varMutationsInsideFunctionWorksProperly() {
    assertEquals("36\n", executeProgram("function foo() : int {\n" +
            "  var x = 42\n" +
            "  x = x - 10\n" +
            "  x = x + 4" +
            "  return x\n" +
            "}\n" +
            "print( foo() )\n"));
}
```

The second partner was responsible for creating the code for the compiler. This included creating a tokenizer to break the CatScript into smaller parts and put them into a certain format. Then they had to parse the tokens produced by the tokenizer to create expressions and statements that can be evaluated. Then they had to generate the code so that the expressions and statements evaluate properly. Lastly the second partner coded bytecode generation so that the code was broken down into something machine understandable.

Section 3

The design pattern that we implemented for this project was memoization. Memoization is where you cache the results of operations and reuse them when needed in the future. We implemented this design pattern to improve runtime. This strategy improves runtime by storing already computed variables in a map and returning the desired values instead of reevaluating every time. This optimization would greatly reduce runtimes due to avoiding redundancies in evaluation.

This pattern is implemented in the CatScriptType.java file from line 36-46.

The CatScriptType.java is located in this directory
Source.zip\src\main\java\edu\montana\csci\csci468\parser

Section 4

Introduction

Catscript is a simple statically typed scripting language.

Types

Catscript supports null, bool, string, integer, object, and even a list containing any of these. Catscript also supports the void type for function declaration statements.
Whenever a type is defined in Catscript a colon always precedes it. In Catscript lists are declared with the syntax list<type>. Types are known at compile time in Catscript.

Print Statement

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

Example:

This will print foo to the console.

Behavior:

In Catscript print statements take any expression and print the result to the console. Print statements support null, bool, string, integer, object, and lists of any of the former types. You can also use print on a variable and the expression that the variable points to will be printed to the console. You can also use print to directly print out what is returned from a function if the function has a return value. For example, if function foo returns a string then I could say print(foo(x)) and whatever string is returned will get printed to the console.

For Loops

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

Example:

This will print 123 to the console.

Behavior:

These statements take any list of items and iterate over them. The for loop creates a new variable within it's scope, in the example above the variable is x. On each iteration x points to the next item in the list. The body of the for statement can hold any number of other statements you wish. In this way you can do a specific task for every item in a list. Inside of the for loop you can also declare new variables to assist with the functionality of the loop, but you must be careful with these variables because they are only able to be used in the

scope of the current iteration of the loop. Something you cannot do inside of a for loop is creating new functions. To use a function in a for loop the function must be declared in the scope in which the loop was called from and then the function can be called in the loop.

If Statement

```
if(true) {
	print(“This is true!”)
} else if(1 < 5) {
	print(“This is false!”)
} else {
	print(“It is bigger!”)
}
```

Example:

prints this is true! if true is true.

prints this is false! if 1 is less than 5 and true is not true.

prints It is bigger if 1 is not less than 5 and true is not true.

Behavior:

The if statement evaluates a boolean expression and if it’s true it will run the code in the body. Otherwise, it will continue to the next instruction. After the if statement you can use an else statement indicating that since the first expression was false you now want this other set of statements to run.
If there is an if following the else statement then it will check the next condition and if the condition is true it will execute that code. If there is no else statement and the boolean expression is false then the code will skip the if’s body and continue as if it was not there.
The expression provided to the if statement must evaluate to a boolean, providing an if statement some other kind of literal expression will cause an error.
In if statements you are allowed to use variables declared in the scope where the if statement was called from as well as new variables you can create inside of the if. Although variables declared in the if are only able to be used inside the if, once the if statement finishes execution, it's scope is removed from the stack making the variables declared inside unusable.

Variable Statement

```
var x : int = 0
var str = “I’m a string”
var y : list<bool> = [true, false, true]
```

Example:

Instantiates a string with the value "I'm a string", an integer with the value 0, and a list containing booleans with the value true, false, true.

Behavior:

Catscript also gives the user the ability to specify what the variable’s type is as shown in the first example where there is a semi-colon followed by int. That tells the compiler the type of var x needs to be int. If the type is not specified the created variable will infer the type from what follows the equals sign, as shown in example 2. In that case the type of var str is inferred to be String based on what follows the equals. If the type is specified but does not match the type of what follows the equals, it will throw an error. By assigning values to identifiers, we can save any number of values for future use. Variable statements support all types available in Catscript and their type cannot be changed once they are created, due to CatScript being a statically type language. One way to store values of different types together is to create a list of objects. You can put multiple objects in the list that could hold different types, but lists cannot be changed once declared.

Assignment Statement

```
x = 1
```

Example:

This changes the value to 1 for the variable x.

Behavior:

Assignment statements are a way to change what is held in a variable statement. When assigning a new value to the variable the new value’s type must match the type that was used when creating the variable or be assignable to that type. This means that if you declared the original variable as an integer the new value must be an integer. The types String, List, and boolean also follow the same rule as described for integer, if a variable is declared with one of these types, they can only be assigned a value of the same type. Null expressions are assignable to everything whereas everything is assignable to an object type in Catscript. All other types in Catscript cannot be assigned to each other. It is also important to note that lists cannot be modified once created. This means that values inside of a list cannot be reassigned, you must create a new list with the updated values every time you want to reassign values.

Function Definition

```
function foo(a : int, b : int) : int {
	int result = a + b

	return result
}

function bar(d : string) {
	print(d)
}
```

Example:

When the function foo is defined the arguments a and b will have integer types. The return type will be an integer.

When the function bar is defined the argument d will have the type string. The return type will be void.

Behavior

The function definition statements make code re-usability very simple in Catscript. A function definition is composed of an identifier. This acts as a name for the function. Arguments that belong to the function. These are enclosed in parentheses within the function's declaration after the name. These are variables for the function and these variables can only be used within the function's scope. Much like variable statements, arguments can be of any type available in Catscript. A return type for the function is signified by a type outside the parentheses. This type acts as the type for the result of the function. If this type is not specified, the function's type will default to void, which is a special type exclusively for a function's return type. If the function has void for its return type you do not have to use a return statement within the function's body. If you choose to, or you want to terminate execution early within the function. You can add a return statement without an expression following it. Finally the functions body will contain every line of code you want to execute within that particular function.

Function Call

```
foo(1, 2, true)
bar("foo " + "bar")
```

Example:

When the function foo is called it will instantiate a new integer called result. It will add a and b and put that result in the variable result and return it.

When the function bar is called it will pass in the value "foo bar" and the function will print out foo bar and it will return nothing at the end of the function.

Behavior:

This statement is made up of the identifier associated with the function's declaration and arguments that match the arguments within the function's declaration. The arguments that are placed into the parentheses following the function's identifier need to match the types specified in the function declaration and the order of values does matter. The function call will pass in these arguments to the function and run the code that we put in the function's declaration. Something to not with function calls is that functions can be called within functions allowing for some more advanced coding, such as recursion. This allows the function you are currently in to call itself, this can be useful when trying to code a repetitive task and provides similar functionality to loops with some ability to code some more advanced things.

Return **Statements**

```
return
return x
```

Example:

int x = 1

function foo(x) : int { return x + 1 }

int y = foo(x) - y would be 2 since foo would take the value in x then add one and return the new value.

Behavior:

Return statements are used at the end of a function to end the function and return a value if the function declaration has a return type. If a function does not have a return type specified, a return statement can still end the function. This would be best used inside of an if statement to terminate a function if a certain criteria is met. A return is also used to specify what value to return if the function has a return type specified in the declaration. The value being returned must match the type specified in the declaration of the function.

Equality Expression

```
2 == 2
2 != 3
```

Example:

```
2 == 2 is true
2 != 3 is false
```

Behavior:

Equality expressions are used to check if values are equal or not equal. Equality expressions evaluate to true or false. Equality expressions can also handle any type in

CatScript. You can check the equality of values of the same type, but you can also check the equality of values that are of different types. true == 1 is a valid equality expression that would evaluate to false. Equality expressions are most useful when used in an if statement as the entry condition.

Comparison Expression

```
5 < 6
7 <= 10
4 > 120
140 >= 20
```

Example:

```
5 < 6 is true
7 <= 10 is false
4 > 20 is false
140 >= 20 is true
```

Behavior:

Comparison expressions are used to check if variables are less than each other, if variables are less than or equal, if variables are greater than each other, or even if variables are greater than or equal. Similar to equality expressions, comparison expressions are most useful as entry conditions to an if statement. But comparison expressions in CatScript can only be used to compare integers. Trying to compare any other type will result in an incompatible type error.

Additive Expression

```
5 + 6
2 - 1
-2 + 4
“foo ” + “bar”
```

Example:

```
5 + 6 is 11
2 - 1 is 1
-2 + 4 is 2
"foo " + "bar" is "foo bar"
```

Behavior:

Additive expressions are used to do mathematical operations on integers. The additive expression adds the left hand side to the right hand side and returns the result. Catscript can also subtract or even add negative numbers together. In Catscript the additive expressions also support the ability to concatenate two strings together. The additive expression is left associative, meaning it will execute from left to right. Something to note is that concatenation will happen if either side of the addition operator is a string. For example, "number " + 4 will result in the string value "number 4".

Factor Expression

```
8 * 5
4 / 2
```

Example:

```
8 * 5 is 40
4 / 2 is 2
```

Behavior:

Factor expressions will either multiply the right hand side by the left hand side or divide them depending on if a * or / is used. Factor expressions are also left associative. It is worth noting that factor expressions override the left associativity of additive expressions. For example, 5 + 5 * 6 will evaluate to 35 since the factor expression takes priority over additive expressions. Integers are the only value that can be used with a factor expression, the use of other types in factor expressions will result in an error.

Unary Expression

```
--4
not true
```

Example:

```
--4 is 4
not true is false
```

Behavior:

Unary expressions are used to negate a value. In additive and factor expressions this involves making positive values negative and negative values positive. Whereas in equality and comparison expressions it will switch the result from true to false if the result was true or from false to true if the result was false. Unary expressions are right associative.

Primary Expression

```
4 + 5 + (6 - 3)
[“foo”, “bar”]
```

Example:

```
4
"foo"
true
["foo", "bar"] is a list holding the strings foo and bar
```

Behavior:

Primary expressions are any single token expression. This can be an identifier which is simply a name that is assigned to any variable. A string, this is a string of characters or numbers that is surrounded by quotes. An integer which is a number without decimal points. A boolean this can either hold the value true or false. Null this represents the absence of any value. A list literal, this is comprised of any type of variable surrounded with square brackets.

Parenthesized Expression

```
4 + 5 + (6 - 3) is 9 + 3 is 12
Not (5 == 5)
```

Any expression can also be parenthesized in Catscript with parentheses. This changes the associativity of any expression. This means that the expressions that are inside of the parentheses take priority in the order of execution. This can be used to ensure proper execution of mathematical expressions as well as using nots on comparison and equality expressions as shown above.

Section 5

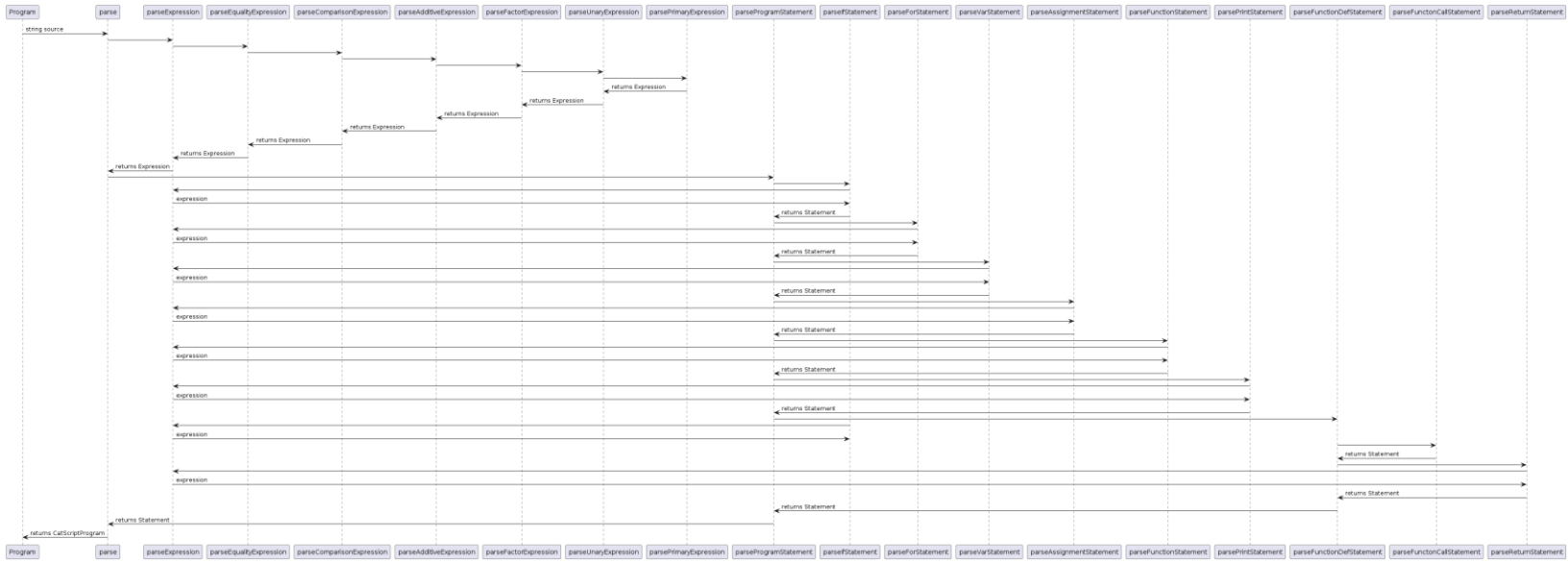

Figure 1: UML/sequence Diagram

Above is a sequence diagram showing the flow of the parser and how it turns a string that was passed in and turns it into a program that can be evaluated. It shows the recursive decent design, shows how when the parse function is called it descends through all the expressions type and parses the string if the input met the criteria for the expression type. Then the pares function evaluates the next steps after the parser descent is done and if needed parses the input into statements and returns the resulting program.

For example, I will pass in a simple instruction set and walk through the steps of the parser. The set of instructions is shown below.

Var x = “Word”

Print(x)

First parse program is called which calls parse expression on the input. The start of the input does not meet any of the requirements to parse an expression so after descending down to parsePrimaryExpression it returns null back to parse which then calls parseProgramStatement, which finds the var indicator so it calls parseVarStatement. Which then finds and expression (“Word”) and calls parseExpression which descends down to parsPrimaryExpression which creates a string literal expression that then returns that StringLiteralExpression all the way back to parseVarStatement and assigns that variable identifier that StringLiteralExpression. Then once the VarStatement is finished it

returns back to parse which is called again with a shorter set of instructions this time due to the variable being parse. It then parses the print statement. This parse reruns until the entire input string has been handled and returns the final program.

The image might show up a little blurry in the pdf format so there is a sequenceDiagram.png in the portfolio folder that provides a clearer picture if needed.

Section 6

In this class we built a recursive decent parser, another option for creating a parser is parser generator. In this section I will compare the two options.

First, we will go over parser generators. Parser Generators take grammar specifications and automatically generate a parser to mee the specifications. This sounds like a preferable way to create a parser. But it does come with some pros and cons. The pros are it is easy and efficient due to an automatic program doing the generation for you. But it does come with some cons. The cons of a parser generator are issues with adding things and changing things about the parser due to not having coded it and not having a great understanding of the code. The generators also adhere strictly to the grammar initially provided, so again making changes could be hard due to potential issues with strict initial specifications.

Building a recursive decent parser, this is a parser built and coded by the person that uses recursion to follow the specified grammar. There are pros and cons to this strategy as well. Some pros are that the code is going to be easier to understand due to the programmer having made the parse themselves, which will also make future updates easier to implement. Another pro is debugging, if something is going wrong it will be much easier to debug something you wrote and understand well. But there are also some cons, the most obvious con is the initial time and effort needed to build a recursive decent parse is much greater than having it automatically generated. Another con is efficiency, recursive decent parser might not be as efficient as a generated parse due to likely not following as strictly to the specified grammar.

Overall, I would prefer to build a recursive decent parser, because one of the most time-consuming parts of programming is debugging. If anything were to ever go wrong with the generated parser it would be incredibly challenging to track down the issue and fix it. It would also be much more challenging to add functionality in the future. While more time and effort would be needed up front, in the long run it will save time and money to have a parser you know and understand.

Section 7

In this project we used TDD (Test Driven Development). That means that we had a certain set of tests that sends inputs to our code and has an expected output, if the output matches, then the test passes and if it doesn't the test fails. The goal being to get all your test passing, which in theory means that your code is functioning properly. I believe the process we used is both good and bad. For this class doing test driven development helped because it gave us a clear set of goals for completing this project as well as giving us a good idea of the goal of the code we were writing. In general, I have mixed feelings about test driven development. While it is nice having a set of tests that are the benchmark for determining the functionality of the code and can help confirm that your code is working properly. It can also hurt the coding process, if you are only focusing on getting your tests to pass the code then you might miss something that the test might not be catching or something you didn't test for. With test driven development where you create test cases after the code is finished, you know how it is supposed to work in your head and will tend to test to confirm your thoughts and not give the code a more challenging test to ensure proper functionality. Test driven development is a positive strategy, there are issues with it as well, it would be better used in combination with other development strategies.