Tom Caron Capstone Portfolio

Section 1: Program

Please include a zip file of the final repository in this directory.

Section 2: Teamwork

Team member 1 created three test cases for team member and team member 2 created three test cases for team member 1 The test cases were created to ensure that expressions and statements were implemented in a robust manner, each taking about a day to complete. Each group member, 1 and 2 were responsible for the documentation of each other’s code, which took approximately 1 day for each group member to complete. Each group member individually implemented their own code to ensure that parsing, evaluation, bytecode generation, and tokenization work correctly, this took approximately 3 and a half months to complete. Below are the tests I created for my partner:

```
@Test
public void parseListLiteralExpression() {
    ListLiteralExpression expr = parseExpression("[1, null, false]");
    assertEquals(3, expr.getValues().size());
    assertEquals(CatscriptType.INT, expr.getValues().get(0).getType());
    assertEquals(CatscriptType.NULL, expr.getValues().get(1).getType());
    assertEquals(CatscriptType.BOOLEAN, expr.getValues().get(2).getType());
    assertEquals(false, expr.hasErrors());
}

@Test
public void parseFunctionCallExpression() {
    FunctionCallExpression expr = parseExpression("foo(1, null, true)", false);
    assertEquals("foo", expr.getName());
    assertEquals(3, expr.getArguments().size());
    assertEquals(CatscriptType.INT, expr.getArguments().get(0).getType());
    assertEquals(CatscriptType.NULL, expr.getArguments().get(1).getType());
    assertEquals(CatscriptType.BOOLEAN, expr.getArguments().get(2).getType());
}


@Test
public void additiveExpressionsAreLeftAssociative() {
    AdditiveExpression expr = parseExpression("1 + 1 + 1");
    assertTrue(expr.getLeftHandSide() instanceof AdditiveExpression);
    assertTrue(expr.getRightHandSide() instanceof IntegerLiteralExpression);
```

```
        assertTrue(expr.getLeftHandSide() instanceof AdditiveExpression);
        assertTrue(expr.getRightHandSide() instanceof IntegerLiteralExpression);
        assertTrue(expr.getLeftHandSide() instanceof AdditiveExpression);
        assertTrue(expr.getRightHandSide() instanceof IntegerLiteralExpression);
    }
```

Section 3: Design pattern

By using the memoization design pattern in our code, we eliminate the need to check to see if a value is being used as an input, so we only execute the code once. It allows us to make calls recursively to use past results and inputs in the future to make new calculations without searching and or running a function with new values. The basic reason for using this design pattern is so that the values and input we use when running our code don't disappear, they are stored in a HashMap and are accessible throughout the execution of the program. The HashMap contains a CatScript Type as a key and a ListType as a value. The method checks to see if there is a mapping in the HashMap for a type, if so it returns ListType, else it will create and calculate a new ListType by calling new ListType(type); where type is the key, it then returns ListType.

Section 4: Technical Writing

CatScript Documentation

Features

For loops

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

For loops take a list of parameter expressions, and a variable identifier to be used as a local variable. The for loop then executes a body of statements, pushing the scope into the code block within the brackets. The local variable created in the parameters can be called to use whatever expression is currently being iterated over within the statement body.

If, Else Statement

```
if(true){ print(1) } else { print(2) }
```

If statements take a single parameter expression, and a body of statements to be executed if the expression evaluates to true, pushing a local scope. If the expression doesn't evaluate to true the else statement executes the body of statements and also pushes a local scope.

Print Statement

```
print(1)
```

Print statements take a statement or expression, and then outputs the evaluation of the statement or expression. Print can be done to Literals, expressions, and output of statements.

Variable Declaration Statement

```
var x = 42
```

Variable Declaration ('var') statements take an identifier and a right-hand side expression. The variable statement uses explicit typing, the new variable takes the explicit type of the expression value. It is executed by using a function in the runtime with HashMap of variable names to the value of the expression. Variables that are declared can only be accessed by scopes lower than their current scope on the stack with global being the top of the stack.

Assignment Statement

```
x = 25
```

Variable Assignment statements take an identifier of an already declared variable, and an expression value to insert into the identifier's variable memory slot. Unless the declared variable is an object the expression type must equal the variable's type, if they are not equal you will receive an incompatible type error.

Function Definition Statement

```
function foo() : int {
var x = 42
return x
}
```

Function definition statements take a function identifier, a list of parameter expressions, and an optional return variable return type. The body of the function definition statement is made up by a list of statements, with an optional return statement. If the return type is declared the return value must also have that type. This name and definition will be linked via a HashMap to the function name key and the definition value.

Function Call Statement

```
x = 25 + foo()
```

Function Call statements take an identifier and then a list of parameters expressions. It then takes the arguments and invokes the function definition and then returns the return value.

Return Statement

```
return "Hello  World"
```

Return calls have a 'return' keyword and a statement, the statement result is popped up 1 in the scopes stack.

Expressions and Operators:

Additive Expression:

Additive Expressions are comprised of 2 or more factor expressions that may be parenthesized, i.e., int, bool, or object types. Use the (+)add or the (-)minus operators to separate the factor expressions of valid type. The expression is parsed and returned as an additiveExpression type.

Additive Operator

```
"Hello" + "World" + 1
```

Additive Operators contain the additive operator and two expressions. The Additive Operator adds the values of the expressions to the left and right. Types must match you cannot add "int + null" or "int + boolean" this will result in an Incompatible Types Error.

Minus Operator (Additive Expression)

```
2-1
```

When minus operators have both a left and right expression and operator, they are evaluated under the Additive Expression as a subtraction. The operator will subtract the value of the right expression from the left expression.

Special Case

When Additive Operators are used with string variable types they act as a concatenate call for the expressions to the left and right of the operator. Types in this case do not need to match. For Integer values you will concat the .toString() value of the Integer to the String value. For Null values you will concat "null" to the String value. For Boolean values you will concat "true" or "false" to the String value.

Unary Expressions:

Unary operations parse unary expressions with a single operand. An expression such as not true would be returned as false, whereas an expression such as -1 would be returned as -1 in a non-boolean context and would be returned as true if it were to be used in context of an if statement it would be parsed as true. Logical negations such as "not" may be used with boolean types whereas unary operators such as (-) minus may not be used with boolean types as it would return a type error.

Minus Operator (Unary Expression)

```
-1
```

Minus operators contain a (-) minus operator and an expression. Unary operators negate the value of the expression to the right of the operator.

Not Operator (Unary Expression)

```
not true
```

The not operator takes a boolean expression to the right, any other type of expression will throw an incompatible type error. The operator acts as a ! operator flipping the boolean values of the left boolean value.

Type Expressions:

```
var x = 10 (implicit)
var x : int = 10 (explicit)
var x : list<int> = [1,2,3]
function x(a, b, c) {} (implicit)
function x (a : object, b : int, c : bool) {} (explicit)

types: 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' , type_expression, '>']
```

In the above examples of how type expressions may be used to define identifier types there are implicit and explicit types. In the first example expression (var x = 10), the type would be int because the parser reads an int (10) and infers the type to be int. On the other hand there are explicit types, in the second example (var x : int = 10) x is an int as well, but it is explicitly declared. If there were to be a statement such as var x : bool = 10, this would throw an INCOMPATIBLE_TYPES error as a bool cannot be defined as an int. When working with lists, the error that you may receive if not an INCOMPATIBLE_TYPES error will be an UNTERMINATED_LIST error signifying that you are missing a closing brace.

Literals

Literal Types: Int, String, Boolean, Null, Object, List

Integers

```
1
```

Integer Literals are Integer values, with Type INT (32 bit).

Strings

```
"Hola, como estas?"
```

String literals are the String values within the quotation marks. "java-style"

Boolean

```
true
```

Boolean literals can be true or false.

List

```
[1,2,3]
```

List literals take a list of expressions separated by commas. The list takes the type of whatever the elements are. If the list contains different types they must be as objects and not separate types, this will throw an incompatible type error. The list values are expressed as an Arraylist of the literal elements.

Null

```
null
```

Is a null value, there is nothing there, the memory spot is empty.

Object

```
x = <Any Literal Type>
```

Objects encapsulate the literal types to be used as a dynamic type that can be evaluated later. It will then give the value of the implicit type of the value assigned to it.

Factor Expressions

Factor Expressions are comprised of two or more unary expressions of int types. The expression uses either the (*) multiplication or (/) division operators to execute the code expression. The value is returned as a type of factorExpression.

Star Operators

```
2 * 3
```

The Star operator takes two expressions, a left and a right. It multiplies the expressions on the right and left.

Slash Operators

```
6/3
```

The slash operator takes two expressions, a left and right of the operator. The slash divides the left value by the right value.

Equality Expressions:

Equality Expressions compare two values of type int, bool, null, or object. These expressions use equality operators, (==) equal and (!=) not equal. The result of the expression is returned as an equalityExpression type.

Equality Operators

```
true == true
```

The equality operator takes two expressions and evaluates if their values are equal. The expression then produces a boolean value. True if they are equal, false if they are not.

Bang Equality Operators

```
true != true
```

The Bang Equality operator takes two expressions on the left and right and compares them. If the expressions are equal, it produces false. If the expressions are not equal it produces true.

Comparison Expressions

Comparison expressions encapsulate logic surrounding greater than, less than, and equal.

Greater than

```
5 > 2
```

The greater than operator takes two expressions and checks if the expression on the left is greater than the expression on the right. If the Left expression is greater than the right then it produces true, else false.

Less than

```
2 < 5
```

The Less Than operator takes two expressions and checks if the expression on the left is less than the expression on the right. If the Left expression is less than the right then it produces true, else false.

Greater than or Equal to

```
5 >= 2
```

The greater than operator takes two expressions and checks if the expression on the left is greater than or equal to the expression on the right. If the Left expression is greater than or equal to the right then it produces true, else false.

Less than or Equal to

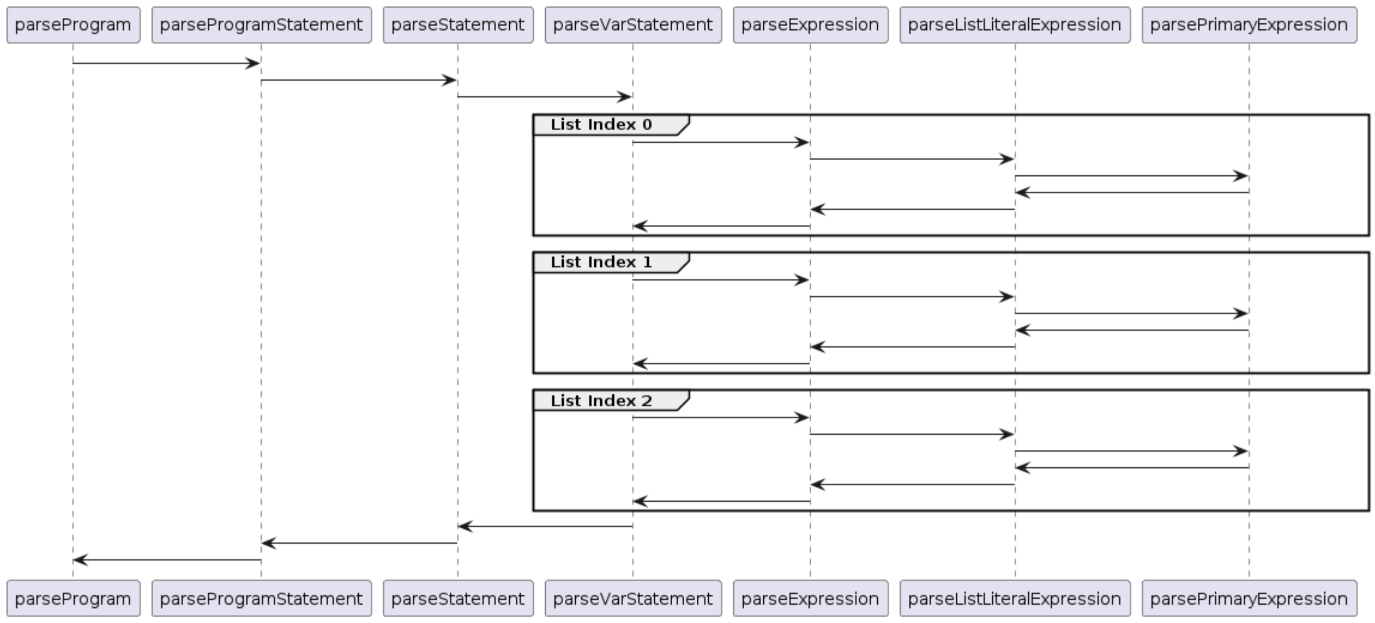
```
2 <= 5
```

The Less Than operator takes two expressions and checks if the expression on the left is less than or equal to the expression on the right. If the Left expression is less than or equal to the right then it produces true, else false.

Section 5: UML.

Sequence Diagram

Var Statement with type list consisting of 3 elements of type int:



The sequence diagram depicted above shows the parsing of a var statement with a list type. The diagram shows the recursive descent parsing taking place starting with the parseProgram class which looks for a statement to parse, it then goes on to parseStatement where it sees a "var" keyword is being tokenized. When it gets to the varStatement it creates an expression list and parses the primary expression where it then turns back around and returns the values to the varStatement where the element is added to the expression. it then repeats this process two more times for the remaining two elements. When all elements have been added to the expression the parser returns values to each recursive step in the program before it reaches the parseProgram class again, where it finally parses everything.

Section 6: Design trade-offs

Recursive Descent vs. Parser Generator

We used recursive descent to implement our parser. The benefit that was noticed in programming was that it was easy to understand how the program was working from a logical standpoint. The ability to customize the processes of specific parsing expressions and understand the processes that were running was great. When it came to debugging, I was able to see exactly where errors were coming from which made it easier to place problems and fix bugs that were in my code. A reason that the errors were so easy to read and understand was because recursive descent parser generation allows for custom errors for specific parsing errors. One of the most common errors that occurred was the unexpected token, when this happened it was always very easy to go back and find where there might have been a token that was parsed before checking to see if there were tokens left in the expression or if there was a specific keyword, identifier, or token that hadn't been required before parsing an expression or statement.

To compare with the parser generator method there are few things to look at. First the ability to code the parser directly into the code was a big plus opposed to having to code a grammar specific parser which is much more complex, although it can be a good way to code complex grammars, our grammar was a pretty simple grammar to work with so there was no need to use a grammar specific parsing method. Secondly, as I had mentioned before, the ability to customize error messages allowed for easy debugging. Overall recursive descent made for a good learning experience and allowed the group to fully understand the processes behind a parser.

Section 7: Software development life cycle model

Test Driven Development

We used test driven development to effectively implement our parser. Although we started with a skeleton of code to work with, the goal of our project was to ensure that the tests passed, confirming that our code worked properly. Traditionally the tests would be written before the code is written to ensure we are implementing the right features in a robust and optimized fashion. By coding in segments, it ensured that only the necessary code was being implemented. Instead of refactoring code because our code base was already created, I was able to just create the code needed and not worry about having to go back to change internal code operations because the external build of the program was already robust enough. For the team this development life cycle model was effective as it allowed the group to effectively make progress throughout the course of the semester. I like this approach to programming projects rather than just filling in skeleton code with a bunch of error messages. It makes programming a better learning experience, and it allows for the group and individuals to learn to debug as well as learn to work in a specific development lifecycle rather than just coding aimlessly.