CSCI 468 Compilers, Spring 2024

Dustin Edgerton

Robert Van-Spyke

Section 1: Program

Source code can be found in source.zip.

Section 2: Teamwork

For this project, Partner 1 did the coding, and Partner 2 did the documentation and designed the tests. Partner 1 implemented the parser throughout the semester, see source.zip for the code. See below for Partner 2's tests. Partner 1 did approximately 90-95% and Partner 2 did approximately 5-10 %, as designed by the course instructor. Additionally, Partner 2 wrote their own program, and Partner 1 did the documentation and tests for them.

```
public class PartnerTest extends CatscriptTestBase {
  @Test
  //Test comparison operators for proper symbol recognition and return type
  public void testParseComparisonOperators() {
    //Testing less than operator
    ComparisonExpression lessThanExpr = parseExpression("10 < 5"); //Parse expression
    assertTrue(lessThanExpr.isLessThan()); //Check if symbol recognized
    assertEquals(CatscriptType.BOOLEAN, lessThanExpr.getType()); //Check if our return type is boolean
    //Testing less than equal operator
    ComparisonExpression lessThanEqualExpr = parseExpression("10 <= 5"); //Parse expression
    assertTrue(lessThanEqualExpr.isLessThanOrEqual()); //Check if symbol recognized
    assertEquals(CatscriptType.BOOLEAN, lessThanExpr.getType()); //Check if our return type is boolean
    //Testing greater than operator
    ComparisonExpression greaterThanExpr = parseExpression("10 > 5"); //Parse expression
    assertTrue(greaterThanExpr.isGreater()); //Check if symbol recognized
    assertEquals(CatscriptType.BOOLEAN, greaterThanExpr.getType()); //Check if our return type is boolean
    //Testing greater than equal operator
    ComparisonExpression greaterThanEqualExpr = parseExpression("10 >= 5"); //Parse expression
    assertTrue(greaterThanEqualExpr.isGreaterThanOrEqual()); //Check if symbol recognized
    assertEquals(CatscriptType.BOOLEAN, greaterThanEqualExpr.getType()); //Check if our return type is boolean
  }
  @Test
  //Test print statement parsing and evaluation
  public void printStatementTesting() {
    //Testing parsing
    PrintStatement expr = parseStatement("print(5)"); //Parse int 5
```

```
        assertNotNull(expr); //Check if null
        //Testing closure error catch
        PrintStatement errorExpr = parseStatement("print(5", false); //Test unclosed print statement
        assertNotNull(errorExpr); //Check if null
        assertTrue(errorExpr.hasErrors()); //Assert that an error was thrown
        //Print type checking
        assertEquals("5\n", executeProgram("print(5)")); //Int
        assertEquals("-5\n", executeProgram("print(-5)")); //Negative Int
        assertEquals("true\n", executeProgram("print(true)")); //Boolean true
        assertEquals("false\n", executeProgram("print(false)")); //Boolean false
        assertEquals("stringTest\n", executeProgram("print(\"stringTest\")")); //Basic string test
        assertEquals("string test with spaces\n", executeProgram("print(\"string test with spaces\")")); //String with spaces
    }
    @Test
    //Testing factor expression parsing and evaluation
    public void factorExpressionTesting(){
        //Multiplication
        FactorExpression expr = parseExpression("10 * 5"); //Parse expression
        assertTrue(expr.isMultiply()); //Check if a multiplication type
        assertEquals(50, evaluateExpression("10 * 5")); //Check for proper evaluation
        //Division
        FactorExpression divisionExpr = parseExpression("10 / 5"); //Parse expression
        assertFalse(divisionExpr.isMultiply()); //Check if multiplication type
        assertEquals(2, evaluateExpression("10 / 5")); //Check for proper evaluation
        //Precedence testing
        AdditiveExpression  precedenceTestExpr = parseExpression("10 * 5 + 10 / 5"); //Parse expression
        assertTrue(precedenceTestExpr.isAdd()); //Check for additive expression
        assertTrue(precedenceTestExpr.getLeftHandSide() instanceof FactorExpression); //Check that left side of additive is a factor
        assertTrue(precedenceTestExpr.getRightHandSide() instanceof FactorExpression); //Check that right side of additive is a
factor
        //Left association testing
        FactorExpression associationTestExpr = parseExpression("10 * 5 * 10"); //Parse expression
        assertTrue(associationTestExpr.isMultiply()); //Check for multiplication expression
        assertTrue(associationTestExpr.getLeftHandSide() instanceof FactorExpression); //Check that left side is a factor
        assertTrue(associationTestExpr.getRightHandSide() instanceof IntegerLiteralExpression); //Check that right side is integer
        //Evaluation edge case testing
```

```
        assertEquals(100, evaluateExpression("10 * 5 * 2")); //2 Multiplications
        assertEquals(1, evaluateExpression("10 / 5 / 2")); //2 Divisions
        assertEquals(1000000, evaluateExpression("1000 * 10 * 10 * 10")); //3 Multiplications
        assertEquals(1, evaluateExpression("1000 / 10 / 10 / 10")); //3 Divisions
        assertEquals(25, evaluateExpression("10 * 5 / 2")); //Symbol mixing
        assertEquals(4, evaluateExpression("10 / 5 * 2")); //Symbol mixing
    }
}
```

As shown above, Partner 2 provides tests for parseComparisonOperations, factorExpressionTesting, as well as printStatementTesting.

Section 3: Design Pattern

We call this design pattern the memoization pattern, (spelled incorrectly on purpose because of a long held MSU tradition.) This design pattern maps the getListType to the cache. Why we use this pattern is because we don't want to create two different instances of list type. This would occur if we had two different Lists of the same type in our program. Instead, we add them to the HashMap of <Catscript Type, ListType>, and check if the cache has a specific type already in it. Then, if the cache has the same type, we return that type, and if the cache does not have the same type, we add it to the map. This pattern allows us to be more efficient in our coding, and saves memory.

Section 4: Technical Writing

CatScript Documentation

Below is documentation for the CatScript program, this should serve as guide for new users in implementing and deploying CatScript code.

Features [Expressions]

Additive/Subtractive Expressions

This is support for basic addition, the CatScript parser ensures left association with additive expressions ensuring the program follows basic mathematically standards. Parenthesizing is also supported and can be used to give higher precedence than the left association rule that is default. To use in the program use the convention expression, + or -, expression. Examples of basic syntax, left association, and parenthesized expressions can be seen below.

```
//Basics
x = 1 + 1
x = 1 - 1

//Left association takes affect
x = 1 + 2 - 4

//Can be parenthesized
x = 1 + (3 + 5)
```

Multiplication/Division Expressions

CatScript also supports basic multiplication and division. CatScript follows the same standards in mathematics where multiplication and division have precedence over additive expressions. Parenthesising is also supported with multiplication as it was wtih additive and subtractive expressions.

```
//Basics
x = 1 * 1
x = 1 / 1

//Left association rule
x = 2 * 2 / 4

//Can be parenthesized
x = 1 * (6 / 3)
```

```
//Higher precendence than additive expresions
5 * 1 + 10 / 2
```

Equality Expressions

These expressions compare two expressions and determines whether the values are equal or not (true if equal). The expression value is what would be compared so values declared as identifiers on the table can be evaluated to its literal value as seen below. The convention used is: expression, equal, equal, expression.

```
if (x == 0){
    print("hello world")
}
```

Comparison Expressions

There is a set of 4 comparative operators that can be used in CatScipt, those being: Greater (>), Greater Equal(>=), Less (<), Less Equal (<=). The CatScript supports the comparison operator between two expressions. The convention therefore would be expression, operator, expression. Below is a use case in an if statement and all 4 symbols used in print statements comparing an identifier with an integer value of 4.

```
//Example use case
if (x > 4){
    print("over limit")
}

//Comparitive symbols that are supported
print (x > 4)
print (x >= 4)
print (x < 4)
print (x <= 4)
```

Not Equal Expressions

This expression is the counterpart for the equal equal expression returning false if the expressions are equal and true if they are not equal. Similar to equal equal expressions, the value of the expression is evaluated before comparison so identifier can be used to compare to an explicit value. A similar convention is used: expression, bang, equal, expression.

```
//Returns false
1 != 1

//Returns true
2 != 1
```

Literal Expressions

These expressions in CatScript come in 6 differing types making up the CatScript type system. The "basic types" also known as "primitive types" that are supported by CatScript are boolean values, strings, integers, and the null value. Those can be declared using the conventions below.

CatScript along with these 4 primitives also supports Lists and Objects. For lists in CatScript both implicit and explicit typing is permitted. If a list is declared with no type the compiler will infer the type based on the type of the right side. Lastly the object type is the last literal supported in the CatScript typing system and can be utilized as a component in lists. The conventions for implicit and explicit type lists can be seen below as well as the object type being used as the component type.

```
//Basic literals
var booleanValue = true
var stringValue = "hello world"
var intValue = 1
var nullValue = null

//Lists
var listWithImplictType = [1,2,3]
var listWithExplicitType : list <int> = [1,2,3]

//Objects
var listWithObjectType : list<object> = [object1, object2, object3]
```

Identifier Expressions

Identifiers allow programmers to create unique names for variables. This is utilized by the var expression to create variables. Below shows the identifier expression used within a variable expression. See variable expressions to understand how these are utilized. It should also be noted that identifier follow similar scoping rules as java.

```
var identiferExampleX = 10
var identiferExampleY = true
```

Unary Expressions

These expressions allow programmers to negate a value. There are two supported unary operations the first is the minus operator which can negate an integer value. The other supported operation is the not which will negate an expression. The convention used is: not or minus, expression.

```
//Unary "-" on integer value
```

```
x = -1

//Unary "not" on boolean value
y = not true

//Nested unary
z = not not true
```

Features [Statements]

For loops

For loops allow users to specify an identifier to match in a provided expression and then execute the following statement enclosed in the braces. The below example takes the identifier x and matches it in the list expression. The statement to be executed is the print statement which in this case print out the integer values 1, 2, 3.

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

Print Statements

These statements allow users to print out to the console. The expression that is used will be evaluated and that value would be added to the stack. The print identifier followed by an opening paren, and expression, and a closing paren.

```
//Print statement with string value
print("hello world")

//Print statemnet with integer value
print(2002)

//Print statement with comparison expression
print(x > 100)
```

Function Calling

Function calling allows users to call functions that are previously defined in the program. Function calls must call a defined function in the scope and the proper arguments must be provided as seen below with the two functions. The convention used is: identifier, open paren, argument list, close paren.

```
function foo(a,b,c){
    print (a)
    print (b + c)
```

```
}

fucntion bar(){
    print("hello world")
}

foo(1,2,3)
bar()
```

If and Else Statements

If and else statements can be used in conditional programming. The else statement can only be used in an if statement but the use of the else statement in a if statement is optional in CatScript. The following convention is used for the if else statements: if, open paren, expression, close paren, open brace, statement, close brace, optional else with either an if statement or open brace, statement, close brace.

```
//If with an else branch
if (x >= 93){
    print("Recieved an A")
}
else{
    print("Did not recieve an A")
}

//If statement only
if(x >= 70){
    print("You passed!")
}
```

Var Statements

These statements allow for variables to be declared and assigned a value. There are two types of variable statements, implicit and explicit. Implicit will infer type based on the type of the right side, explicit will take in a defined type. In CatScript a value must be assigned when a variable statement is constructed. The following convention is used: var, identifier, optional (colon, type), equal, expression.

```
//Implicit typing
var x = 5

//Explicit typing
var x : int = 10
```

Assignment Statements

Assignment statements allow variables in the scope to be reassigned values. The new value must be assignable from the defined type. The following convention is used: identifier, equal, and expression.

```
//Integer assignment
var x = 5
x = 10

//Boolean assignment
var y = true
y = false
```

Function Definition Statements

These statements define a set of instructions that can be called in the program through function calls. The function must have a name in the form of an identifier. Functions in CatScript may take in a list of parameters, but it is not required. Function definitions can also have an explicit return type defined as well.Function definitions have function bodies which can either be an statement or a return. Function definitions follow the convention: function, identifier, open paren, parameter list (can be empty), close paren, optional (colon, type), open brace, function body, close paren.

```
function foo(a,b,c){
    print (a)
    print (b + c)
}

fucntion bar(){
    print("hello world")
}


foo(1,2,3)
bar()
```

Return Statements

These statements are to be used in function definitions to return a value or to execute the function at its current state. Return statements can either return nothing as seen with the second example or they can return an expression. The convention used is: return, optional (open paren, expression, close paren).

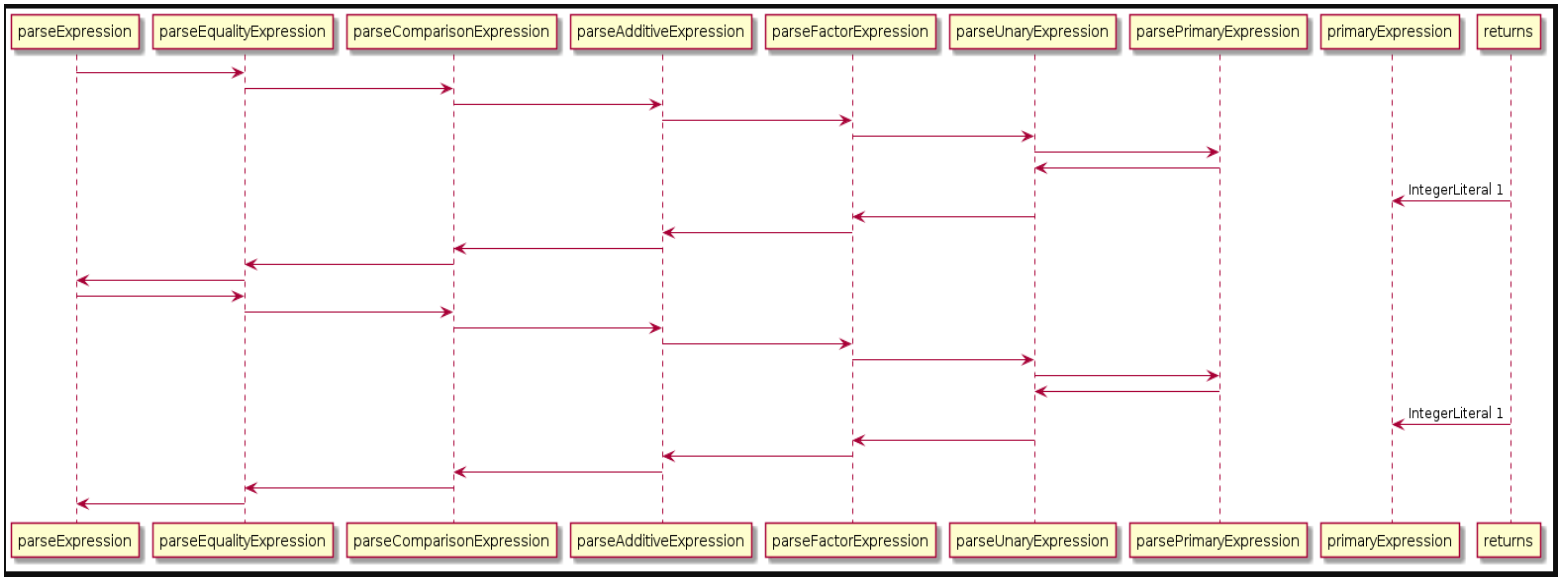
```
//Function with expression return
function foo(a){
    return (a * a)
}

//Function with both return types
```

```
function bar(a){
    if (a == 1){
        return (a)
    }
    else{
        return
    }
}

//Function calls
foo(10)
bar(10)
```

Section 5: UML



Above is a sequence diagram example of parsing 1 + 1 in CatScript using recursive decent. The CatScript parser uses recursive decent to parse by recursively going down the grammar and calling each expression. If the parser came across an additive expression such as 1+1, it would call parseExpression. ParseExpression would call parseEqualityExpression, which would then call parseComparisonExpression, which would call parseAdditiveExpression, which would call parseUnaryExpression, which would call parsePrimaryExpression, which would return the primary expression integer 1. Then the parser would go back up the grammar until it got to parseExpression again. After it got back to Parse Expression, it would once again go back down the grammar tree until it got another primary expression, also integer 1. Because additiveExpression requires a + or –, after the program goes back up the grammar to parseExpression, you would be left with 1+1.

Section 6: Design Trade Offs

For this project we decided to use a handwritten recursive decent parser instead of a parser generator. First off, CatScript is a fairly basic grammar and recursive decent gave us the ability to fine tune our parsing logic. The recursive decent allowed us to work through the grammar line by line and build the parser around the specific needs of CatScript. Additionally, the simple nature of recursive decent allowed us to easily comprehend and debug what our parser was doing. This allowed us as students to better understand how our compiler worked, and how the parser played a role in compiling the CatScript language.

```
private Statement parseAssignmentOrFunctionCallStatement(){
    if (tokens.match(IDENTIFIER)){
        Token identifier = tokens.consumeToken();
        if(tokens.match(EQUAL)) {
            tokens.consumeToken();
            AssignmentStatement assignmentStatement = new AssignmentStatement();
            Expression expression = parseExpression();
            assignmentStatement.setVariableName(identifier.getStringValue());
            assignmentStatement.setExpression(expression);
            return assignmentStatement;
        }else {
            FunctionCallExpression functionCallExpression = parseFunctionCallExpression(identifier);
            return new FunctionCallStatement(functionCallExpression);
        }
    }
    return null;
}
```

In the CatScript grammar, there is some ambiguity. This is because Assignment Statement and Function call Statement both start with IDENTIFIER. I chose to parse Assignment Statements and Function Call Statements at the same time to avoid the parser from calling the wrong method. As you can see from the code snippet above, in parseAssignmentOrFunctionCallStatement(), I solve the problem of ambiguity by consuming the first token IDENTIFIER, and then looking at the next token to determine if it is a function call or assignment. The alternative would be to match the IDENTIFIER, and then call a different method. I chose to make my method a little more complicated rather than having two separate methods, one for parseAssignmentStatment, and another for parseFunctionCallStatement.

Section 7: Software Life Cycle

For our project we used test driven development. We had an extensive test base of approximately 148 tests. These tests ranged from simple parsing tests, to advanced bytecode generation tests. Because of test driven development, we were able to debug a lot easier. The tests allowed you to compare what the code should be producing, vs what the code was actually producing. Then as the developer, you could step through each line of code and find exactly where your code was erroring or producing the wrong value.

In the beginning, I loved the test driven development that was implemented. It gave me a great way to measure my success and to see my progress. It also allowed me to easily understand the inner workings of my code. For example, I could debug a test and watch it step line by line through an additive expression and see where it was adding the left side to the left side instead of left side to right side. However, as we moved into bytecode, the test based development became less useful, and brought errors to light that had previously been unnoted.

The problem with test based development is that you can write code that is good enough for that test, but will not be good enough for future tests. For example, I wrote code that passed all the tokenizing and parsing tests for if statements. However, when I got to code execution, I spent hours trying to get the ifStatement tests passing. Eventually I narrowed it down to a parsing error. This happened because I assumed that my parsing was working correctly because I was passing all the parsing tests, when in fact, I was adding my else statements to the wrong list, which was causing the code execution to error out.

Overall, test based development is by far my favorite, and even though it caused me some time and heartache later in the project, it was only because of test based development that I was able to debug and narrow down exactly where the error was occurring in the parser. For the future, I wish to lean more heavily on test based development for all coding projects after graduation.