Montana State University
Gianforte School of Computing

CSCI: 468 Compilers,
Spring 2024

Colton Parks, Nicholas Weigand

Section 1: Program

Source.zip for the Catscript compiler is located in the same directory in this pdf.

Section 2: Teamwork

My team is composed of 2 people, myself and Partner 1. Each of us finished all checkpoints within parser, expression, bytecode, and tokenizer individually. The total workload for the class was split 90/10, with myself being the primary programmer. For this final capstone project, we both exchanged 3 tests with the “PartnerTest” file found in src>test>java>demo. Partner 1 also helped a lot debugging my tests and finding new unused ones for use.

```
public class PartnerTest extends CatscriptTestBase {


    void divisionByZeroThrowsError()
    {
        assertThrows(ArithmeticException.class, () -> executeProgram("var x =
1 / 0"));
    }


    public void parseListLitExpr()
    {
        FunctionCallExpression expr = parseExpression("plug([3,5,6] ,
[2,4,9])", false);
        assertEquals("plug", expr.getName());
        assertEquals(2, expr.getArguments().size());
        LinkedList<Expression> list = (LinkedList<Expression>)
expr.getArguments();
        assertTrue(list.get(0) instanceof ListLiteralExpression);
        assertTrue(list.get(1) instanceof ListLiteralExpression);
    }
    @Test
    public void parseEqualityNestUnary()
    {
        EqualityExpression expr = parseExpression("not false != not not
true");
        assertEquals(false, expr.isEqual());
        assertTrue(expr.getRightHandSide() instanceof UnaryExpression);
        assertTrue(expr.getLeftHandSide() instanceof UnaryExpression);
    }

}
```

Section 3: Design pattern

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type)
{
	 if (cache.containsKey(type))
	{
		return cache.get(type);
	}
ListType listType = new ListType(type);
cache.put(type, listType);
return listType;
}
```

A design pattern we used for memoization can be found in src>main>java>parser>statements>catScriptType.java which is given above. We don't want to include the code directly here to avoid multiple inputs. To store the results from the specific input, we can put them in a static hashmap. Before running an input through this given method, we can use a hashmap to check for duplicates of the input using variable cache. If the results are not found there, then we don’t need to run the method a second time and we will just return the results. If we were to code directly on this doc, we would run the method on the same input, which is very redundant.

Section 4: Technical writing

Expressions

Equality Expression: Evaluates comparison expressions, compares them equal each other. Uses equal or not equal using symbols “==” and “!=”.

Ex. name != false;   name = true;

Comparison Expression: Evaluates two mathematical expressions and declares them as greater, less, greater than or equal to, or less than or equal to each other using“>,<, >=, <=” symbols.

Ex. 2 > 1;   val >= 1;   2 = 2;

Additive Expression: Evaluates the addition or subtraction between two expressions using “-,+”. Can be used in conjunction, but order does matter, using parentheses symbols as precedence

Ex. 2 + 1;   1 + (2 + 5)   (2 - 1) + (5 - 1)

Factor Expression: Evaluates the multiplication or division between two expressions using “*, /”. These symbols take precedence over additive expressions unless parentheses symbols are involved

Ex.   1 * 2;   1 * 2 + 3;   1 * (3 + 5);

Unary Expression: Not the same as a comparison or equality expression. This function evaluates whether an expression has a “! , not” label that is attached to it.

Ex. not false;   !false;

Function Call Expression: Evaluates if the expression starts with a call or identifier and is trailed by arguments within parentheses.

Ex. Add(5, 6, 0);   Pop(value);

Statements

For Statement: Operates like any other mainstream programming language(Java, Python, C) for loop. Includes an identifier variable, its incrementation, and its arguments expression with a statement within its block. The identifier represents the count of the loops iteration or how many times its inside expression is executed. The incrementation variable determines down the code increments, whether it moves down, up, when it stops, or at what rate. For loops can be nested in one another as well.

```
Ex.  for(i: list)
        { remove(i); }
```

Ex.

```
for(i in firstLoop)
{
        for(a in namesList)
        {
                print(names);
        }
}
```

If Statement: Like For Statement, operates like a Java if statement that executes the function upon the conditions provided are met. The” if” is followed by parenthesis with a condition statement inside, which determines whether the program continues to whatever is inside the block of code. If met, the statement or expression inside is executed. You can stack If statements in conjunction, or even nest them inside on another alongside else statements.

```
Ex.    if (value == 1)
       {
           return true;

       }
       else
         { return false; }
```

Print Statement: Is responsible for displaying expressions onto the terminal for the user to see. Any text within the print statement will be outputted, but variables that are included in the print statement might not. They must have some type of text returned

when called, or no output will be given. In some cases, indexes, or spaces in memory might be printed instead of a desired text. These variables must have some text associated to them beforehand, since Catscript does not understand what needs to be printed.

Ex. print("hello world ");   output: hello world

Ex. val = “one”   print(val);   output: one

Assignment Statement: Assigns a value to a variable, but only after that variable has been created. You cannot create a variable and give it a value in one line of code. One can modify or override previous assignments to protect it. Catscript also prevents reassigning its value if the type is different, so an int variable cannot be given a bool later on. This applies when the variable has not had a value assigned to it and afterwards.

Ex. Name = “John”;
Ex. cannot do: var Name = new var “John”

Function Call Statement: Executes an expression function call. The function called would already have expressions, conditions, etc. built beforehand. Like assignment statement, instantiating and assigning the functions content cant happen in one step. When calling this function, the parameters it includes are required to me included as well.

Ex. print(Name, name2);   cannot do: print(Name); , if the function requires both parameters

Return Statement: Can be executed to send a program back to its original caller or is part of a body of code. This is done when a program is completed or used to break it when conditions fail. Also is used to return a value back to the place that called upon it. They can return any type of value, as long as the function type matches the type of value being returned.

Ex: print(“hello”)
        Return;

Ex. val = 1;

```
	return val;
```

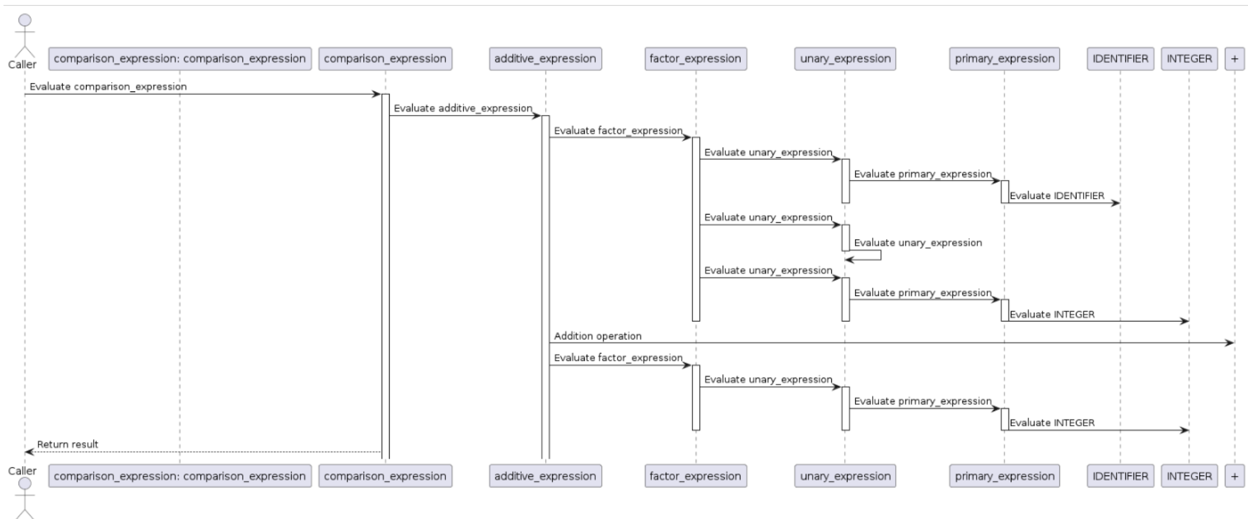
Function Definition Statement: Creates a new function. Must include an identifier name and a block of code within it with some type of executable code or expression. This is required as the call to this function must return something. You can also include parameters after its identifier to take in variables to be performed on or evaluate them in a condition statement.

```
String whatsUp(name)
{
	Return name + “, how are you doing today?”
}
```

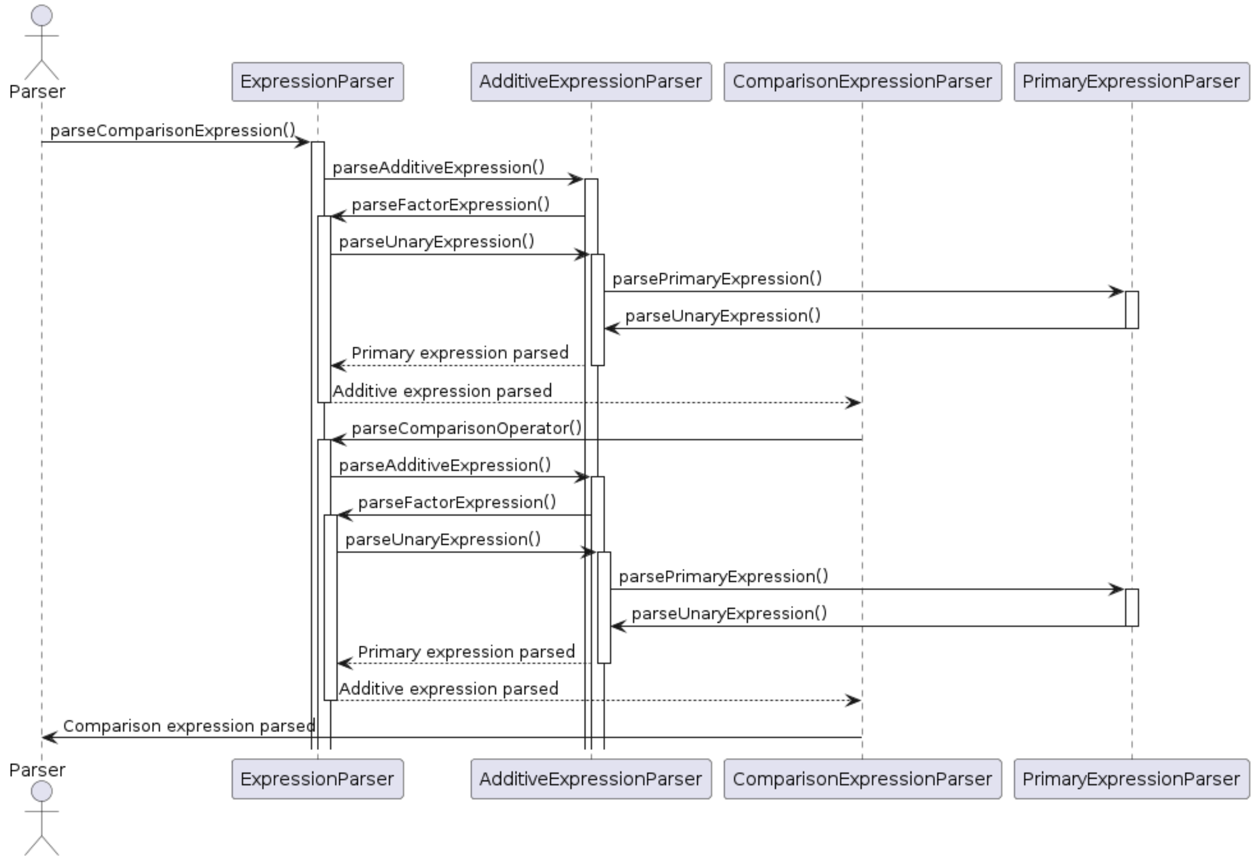
Section 5: UML

Summary: The sequence diagrams illustrate the evaluation process of a comparison expression within the Catscript language. It begins with the caller initiating the evaluation of the comparison expression. The comparison expression then proceeds to evaluate the additive expression, which involves addition operations and the evaluation of factor expressions. Each factor expression undergoes evaluation of unary expressions and primary expressions, which include identifiers and integers. Once the additive expression is evaluated, the result is returned to the caller.

<u>Evaluating comparison_expression:</u>



<u>Parsing comparison_expression:</u>


Parser
ExpressionParser
AdditiveExpressionParser
ComparisonExpressionParser
PrimaryExpressionParser
parseComparisonExpression()
parseAdditiveExpression()
parseFactorExpression()
parseUnaryExpression()
parsePrimaryExpression()
parseUnaryExpression()
Primary expression parsed
Additive expression parsed
parseComparisonOperator()
parseAdditiveExpression()
parseFactorExpression()
parseUnaryExpression()
parsePrimaryExpression()
parseUnaryExpression()
Primary expression parsed
Additive expression parsed
Comparison expression parsed
Parser
ExpressionParser
AdditiveExpressionParser
ComparisonExpressionParser
PrimaryExpressionParser

Section 6: Design Trade-offs

Recursive descent, which is used in Catscript, is something that is very profound yet simple and easy to understand. It is a type of Top-Down Parser that builds the parse tree from the top to down that gives the start symbol to the entire program.

Parser generators, on the other hand, are less superior to recursive decent. It can be less written code than doing it by hand as it has certainly less infrastructure.However, it is more difficult to understand. It has obscure syntax for things that are obvious compared to when you do it by hand and does not give a good feel of the recursive nature of grammars.

Section 7: Software development life cycle model

Test driven development(TDD) is a type of software development process of design that includes writing tests before writing the code. The principle is that writing tests first will guarantee that the code written afterwards will meet the requirements given and allow for more efficient progress. If we can focus on the function and goal of the code, the developers can focus on what matters and will reduce the amount of unnecessary or complex code. The tests allow the programmers to find bugs and errors in the development process.

Our code followed a similar path, as the tests were given to us beforehand, and chunks of the code completed beforehand. Although the code given to us by the professor already works flawlessly, he took out chunks of it for the class to finish. These TODOS, given throughout the code, were key pieces of the code that allowed all tests to pass. Although TODOS won't be given in a real world programming project, it gave us pieces of the puzzle. Removing them didn't give any compile errors, but completing each piece would result in more test passing. By doing this, Prof. Gross tailored what and where he wanted us to fill in the blanks to learn specific programming skills.

The cool part of TDD in this class was completing the code and being able to run tests right after at any time. The errors given to us also pointed us in the right direction, as the errors told us where there were issues in specific files and methods, so anyone would know where they need to work on. It was almost fun to work on a TODO in one method and discover that one seemingly simple method would grant 10 tests to work that you were stuck on for hours.