CSCI 468 Compilers Report

Spring 2021

Partner: Daniel Vinogradov

Micheal Wetherbee

Section 1: Program

Project for this capstone project can be found at the file path csci-468-spring2021-Compilers\csci-468-spring2021-private\capstone\portfolio

The Github page for the project is publicly available at - https://github.com/micmicmw/csci-468-spring2021-private

Section 2: Teamwork

In this project Team member one did all of the work considering each student in this class had their own compiler to work on. Team member 2 did contribute in the sense of helping team member one with certain coding issues once Team Member one reached out for help. Team member one completed the majority of the assigned tests including the tokenizer, parser, and a portion of the eval tests as well as a couple of the bytecode tests. Overall completing 90 out of 148 tests. Team member one was also given 5 tests from team member two to pass. He was able to get all 5 tests passing.

Section 3: Design pattern

In this project we all used memorization. This process is an optimization technique in which a cache stores data types so it can access it later instead of computing those data types again. This leads to the compiler running more efficient. This optimization can be found from lines 36 to 47 in the CatscriptType.java file.

Section 4: Technical writing

In this project I have worked on I have completed a good number of tasks laid out for myself. I can separate these tasks into checkpoints. The four checkpoints I will be delving into include the bytecode checkpoint, the evaluation checkpoint, the parser checkpoint, and the tokenizer checkpoint. I will start from high level to low level in the mechanics of how this project works.

I was unable to complete the bytecode portion of the compiler. Bytecode is an instruction set designed for efficient running by a software interpreter. This portion of the project would have been the top most layer of the project. In the case that it would streamline the below parts in the use of the compiler.

The next part to talk about is the evaluation part. In this portion we patch up portions of the expression code that we were given by the professor.

```
@Override
public Object evaluate(CatscriptRuntime runtime) {
    int right = (int) rightHandSide.evaluate(runtime);
    int left = (int) leftHandSide.evaluate(runtime);
    if(isLessThan()){
        return left < right;
    }else if(isGreater()){
        return left > right;
    }else if(isGreaterThanOrEqual()){
        return left >= right;
    }else{
        return left <= right;
    }

}
```

The above code is an example of what we needed to patch here we see the evaluation of comparing expressions. In this case we compare the right and left side set as integers. If the first is less then the second we return left < right. We use if else statements to do this for is greater, is greater than or equal, and less then or equal to. With this we can pass the test for comparison operators working correctly.

```
@Override
public Object evaluate(CatscriptRuntime runtime) {
    if(isMultiply()){
        return ((Integer)leftHandSide.evaluate(runtime)) * ((Integer) rightHandSide.evaluate((runtime)));
    }else{
        return ((Integer)leftHandSide.evaluate(runtime)) / ((Integer) rightHandSide.evaluate((runtime)));
    }
}
```

In this example we can see the same theme repeated as above except we are seeing if we can multiply the numbers together or not sending back the correct answer for each. This same idea can be seen throughout the rest of the expressions.

Next, we can look at the statements we completed in the parser checkpoint. This portion of the code involves figuring out whether a string of code is a statement such as a when or if statement.

```
Statement printStmt = parsePrintStatement();

if (printStmt != null) {
    return printStmt;
} else {
    if (currentFunctionDefinition != null) {
        return parseReturnStatement();
    }else {
        Statement forStmt = parseForStatement();
        if (forStmt != null) {
            return forStmt;
        } else {
            Statement ifStmt = parseIfStatement();
            if (ifStmt != null) {
                return ifStmt;
            } else {
                Statement varSmt = parseVarStatement();
                if (varSmt != null) {
                    return varSmt;
                } else {
                    Statement funSmt = parseFunctionDeclaration();
                    System.out.println(funSmt);
                    if (funSmt != null) {
                        return funSmt;
                    } else {
                        Statement assiSmt = ParseAssignmentStatement();
                        if (assiSmt != null) {
                            return assiSmt;
                        }
                    }
```

Above we can see where we are figuring which statement, we might be dealing with in this function. It had to be set up this way to avoid a bug that was overriding true Statement variables with other Statement variables in the same scope so I had to nest them like shown above.

```
private Statement parsePrintStatement() {
    if (tokens.match(PRINT)) {

        PrintStatement printStatement = new PrintStatement();
        printStatement.setStart(tokens.consumeToken());

        require(LEFT_PAREN, printStatement);
        printStatement.setExpression(parseExpression());
        printStatement.setEnd(require(RIGHT_PAREN, printStatement));

        return printStatement;
    } else {
        return null;
    }
}
```

In this above Statement we are looking to see if the statement is a print statement. We deduce this by looking through the string token by token setting a start end and the expression. This same theme is repeated with the other functions used to identify statements.

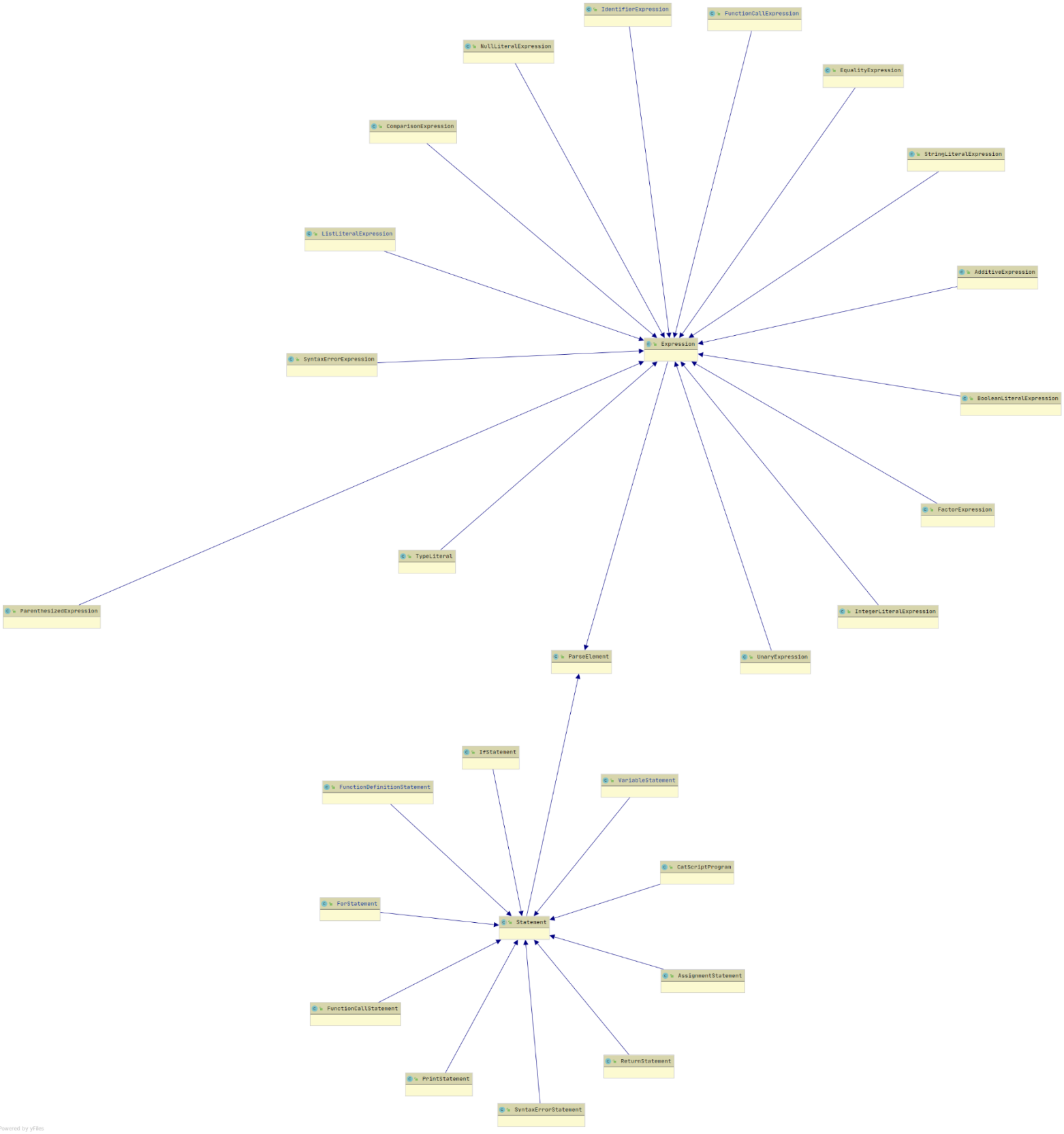
Lastly, we have the tokenizer checkpoint which involved the creation and implementation of the tokenizer that all other aspects of the compiler are built on.

```
} else if(matchAndConsume( c: '-')) {
    tokenList.addToken(MINUS, stringValue: "-", start, postion, line, lineOffset);
} else if(matchAndConsume( c: '/')) {
    if (matchAndConsume( c: '/')) {
        while (peek() != '\n' && !tokenizationEnd()) {
            takeChar();
        }
    } else {
        tokenList.addToken(SLASH, stringValue: "-", start, postion, line, lineOffset);
    }
} else if(matchAndConsume( c: '=')) {
    if (matchAndConsume( c: '=')) {
        tokenList.addToken(EQUAL_EQUAL, stringValue: "==", start, postion, line, lineOffset);
    } else {
        tokenList.addToken(EQUAL, stringValue: "=", start, postion, line, lineOffset);
    }

}else if(matchAndConsume( c: '(')) {
    tokenList.addToken(LEFT_PAREN, stringValue: "(", start, postion, line, lineOffset);
}else if(matchAndConsume( c: ')')) {
    tokenList.addToken(RIGHT_PAREN, stringValue: ")", start, postion, line, lineOffset);
}else if(matchAndConsume( c: '}')) {
    tokenList.addToken(RIGHT_BRACE, stringValue: "}", start, postion, line, lineOffset);
}else if(matchAndConsume( c: '{')) {
    tokenList.addToken(LEFT_BRACE, stringValue: "{", start, postion, line, lineOffset);
}else if(matchAndConsume( c: '[')) {
    tokenList.addToken(LEFT_BRACKET, stringValue: "[", start, postion, line, lineOffset);
}else if(matchAndConsume( c: ']')) {
    tokenList.addToken(RIGHT_BRACKET, stringValue: "]", start, postion, line, lineOffset);
}else if(matchAndConsume( c: '*')) {
    tokenList.addToken(STAR, stringValue: "*", start, postion, line, lineOffset);
}else if(matchAndConsume( c: ':')) {
    tokenList.addToken(COLON, stringValue: ":", start, postion, line, lineOffset);
}else if(matchAndConsume( c: ',')) {
```

In this portion of the project, we identify the tokens of the language. This is done by adding tokens to the project using given code. Label the token with its sign and character for instance left bracket is LEFT_BRACKET and has the char [given to it as seen above. This same theme is repeated with every other character that the language will recognize as a significant symbol

Section 5: UML



In this Diagram we can see how each function made for identifying a certain statement connects into the statement class and links back into the function parsing the current element. We can see on the other side how each function made for detecting expressions link back into the function expression and runs back to the function that is parsing the current element.

Section 6: Design Trade offs

So, the design that I used in this project was recursive decent parsing. I will compare it to parse generators. Overall, I prefer the recursive decent parsing that I used in this project compared to other methods.

My reasoning for this is that recursive descent parsing was incredibly easy for me to wrap my head around. They are more flexible in recognizing multiple data types compared to the generators. The down side to them is that they cannot do parses that require arbitrarily long lookaheads, it is tricky to get really good error messages with this method, and this method is slower than other methods but is faster than the generator.

The generator doesn't need that much code to set up and run but it is less flexible in the sense that it doesn't take in ambiguous grammar very well. Along with this your code will get harder to read if you need to implement many different types of grammar.

Section 7: Software Development Life cycle model

In this project I got very well acquainted with the test-based way of software development. This method involved us as a class completing a series of tests before a checkpoint due date to get credit for the code we completed by that date. I thought this method had its pros and cons as with any system.

The pros to this system are the black and white expectations of what you need to have done before the due date. There is no mistaking what needs to be done. You must get this input to lead to this output. I liked the way that each test was the same weight at the others leading to an equal importance being placed on all the tests at any checkpoint with the exception of the last checkpoint in which the bytecode tests were weighted more to give inceptive to finish those tests. The adding of the tests due at the last checkpoint to the next one lead to a double edge sword. The sword being that in completing the 16 tests at the beginning you had 16 tests completed at all future checkpoints. The other side being that if you are having trouble with a test and are unable to complete it till after the next point you will be marked down on each of those checkpoints instead of just the first one. The only other con that I can find with this system is not understanding how the tests were implemented in code as I did not write the whole program. This took a bit of rooting around to figure it all out. Overall, I would

recommend the tests system for future compilers courses. I felt that this way of measuring progress is best for the situation.