Design Report

CS3361 Spring '22

Dylan Sheehan, Dan Limbu, Edward Grady

INTRODUCTION

The goal of the project was to create a parser from the scanner in project 1. We edited the scanner so that it also output the value of the token, i.e. id value is A, so that we could make a completed XML type tree. This was done with using arrays and having a token value as the amount of tokens we can store in each of these arrays. Once the scanner creates the values of tokens and holds those tokens it read, it passes it to the parser. The parser is then used to create the XML type tree as it shows the flow of recursive descent to each token value. In order to assist us with this project, we took advantage of using the Context Free Grammar provided and sketched out a plan. Then we also used the pseudocode from the book for assistance in determining what should go in each method/function. Then we took our plan and created our own pseudocode that represents an object-oriented view so we can portray the values that represent each token in the XML type tree.

Each class (Scan and Parser) has their own constructors in order to set each field variable to their correct token in Scan and call the recursive descent in Parser. Notice that Scan has a accessor methods in order to access the amount of tokens we have in the array and the tokens themselves to be used for the Parser. The Parser uses these methods to correctly create the XML type tree.

DATA STRUCTURE

The data structures we used in Project 2 were arrays. We needed to create 3 new arrays in our project. The first array was an edited version in Scan to store not just the tokens, but the values of the tokens in order to be used in the Parser. Then we crated 2 more arrays in our Parser. The first array was the token array that was edited to add the "$$" to the end in the Parser constructor in order to signify that there are no more tokens remaining. Then we used a second array to store the values of tokens in the Parser array. Because we can't use the Scanner's array directly we needed to hold these separate values in the Parser's array for values.

TEST CASES

1. The first test case we did is represented in "test1.txt" and what it shows is valid calculator language. This is the correct grammar so that the Parser and Scanner will not output an error. As you can see we assigned 2 variables (ids) to 1 large number with a decimal and the other to an integer value with parenthesis. When you run this case, you will notice that the XML tree is very long and you are able to trace the value down using the pseudocode as guidance.
2. The second test case represents a better real example when someone decides to use the scanner and the parser. This is reflected in "test2.txt". Here notice we read a variable 'var' and

then we perform some arithmetic on the variable and when it is done we use 'write' to write it back. This descent is also very large because we have many tokens in each of the files.

3. The last test case is supposed to represent an error in the syntax to test and make sure our parser and scanner output the correct error when the user types invalid grammar into the text file. If you see the text file we read a variable called 'count' and then perform arithmetic operation using count on the operator (another id) and the operand (also another id). Notice that this violates the Context Free Grammar of our calculator language that was provided because we can't perform arithmetic without assigning the value of the arithmetic to another variable (id). This is why we will get an error here.

ALGORITHMS

The algorithms for the scanner have remained the same with only minor adjustments to account for the value of the tokens along with the tokens themselves. The following are the algorithms for the parser class and main class:

Main.java Class

Data: file name argument from console

Input: file name

Output: complete XML parse tree using production rules and grammar or error if encountered

Invariant: file name must be valid

Main(String[] args)

Create new scan object from console input of test file name

Create new parser object using tokens and token values from scan object

Parser.java Class

Data: String[] array tokens, String[] array tokenValue, int counter = 0

Input: tokens from scanner, token values from scanner, amount of tokens from scanner

Output: completed xml parse tree or error if error encountered

Parser(String[] tokens, string[], int counter)

Copy tokens and tokenValue array into new arrays such that the end of array contains no null values

Place "$$" at end of arrays to indicate end of program

Call program()

```
Program()
        Case id, read, write: print “program”, call stmt_list()
        Case $$: match $$
Stmt_list()
        Case id, read, write: print smt_list, call stmt(), call stmt_list()
        Case $$: break
        Print error and exit otherwise
Stmt()
        Case id: match id, match :=, call expr()
        Case read: match read, match id
        Case write: match write, call expr()
        Print error and exit otherwise
Expr()
        Case id, number, lparen: call term(), call term_tail()
        Print error and exit otherwise
Term()
        Case Id, number, lparen: call factor(), call factor_tail()
        Print error and exit otherwise
Factor()
        Case id, number: match( id or number)
        Case lparen: match( ( ), call expr(), match ( ) )
        Print error and exit otherwise
Fact_tail()
        Case plus, minus,rparn,id,read,write,$$: skip
        Case times, div: call multi_op, factor(), fact_tail()
         Print error and exit otherwise
Term_tail()
        Case rparen, id, read, write, $$: skip
        Case plus, minus: call add_op, term(), term_tail()
```

Print error and exit otherwise

Add_op()

Case plus, minus: match plus or minus

Print error and exit otherwise

Multi_op()

Case times, div: match times or div

Print error and exit otherwise

Match(string value, int index)

For each valid rule, print token type and token value, then increment index + 1

ACKNOWLEDMENTS

Dan Limbu helped with report and pseudocode. He made sure that the pseudocode matched the source code in the most structured way as possible. Dylan Sheehan typed the parser and each method to represent the XML type tree of recursive descent. Dylan also fixed the scanner so that it output token values along with the token itself and proofread the report. Edward Grady wrote report and the pseudocode. Edward also assisted Dylan with writing and debugging code and came up with a clever way to properly indent the XML type tree so that it looks neat and we can see what the tokens of each value are as it goes through it.