Initially, our project had four shift/reduce conflicts in the yacc file. The first of these conflicts was caused by a typo in the second production of the Call nonterminal. The production was erroneously producing a left bracket, an Actual nonterminal and another left bracket. This was easily fixed by correcting the typo to produce the correct right brace token. The next shift reduce conflict was due to the If Stmt nonterminal. This nonterminal was responsible for parsing an if statement or an if else statement. This was resolved by creating two nonassociated tokens at the top of the yacc file which we called IFONLY and an else token. We then placed "%prec IFONLY" at the end of the production that created the if statement and the _else token at the end of the production that would produce the if else clause. The assigned precedence to the productions so yacc would no longer run into a shift/reduce conflict here. The next shift/reduce problem we encountered was in the Lvalue and Call nonterminals. These nonterminals both created several id tokens which would cause the shift/reduce conflict when the closure rule was applied on the table. To get around this, we created the LvalueX nonterminal so that the id token and the left recursive Lvalue production that produces any number of periods and id tokens would be interpreted differently by the parser. We also created a nonterminal ProdPDANDID that would produce the period token and an id token for the Call and Lvalue nonterminals. The last shift/reduce conflict we ran into is in the Variable and and Type nonterminals. From the Variable nonterminal, you can produce a Type nonterminal and an id token and in the Type token you can produce an id token only. When yacc does closure on the table containing Variable and Type, it cannot determine whether to shift to a different table on the id token or reduce on the nonterminal. We were not able to solve this shift/reduce conflict as any change to

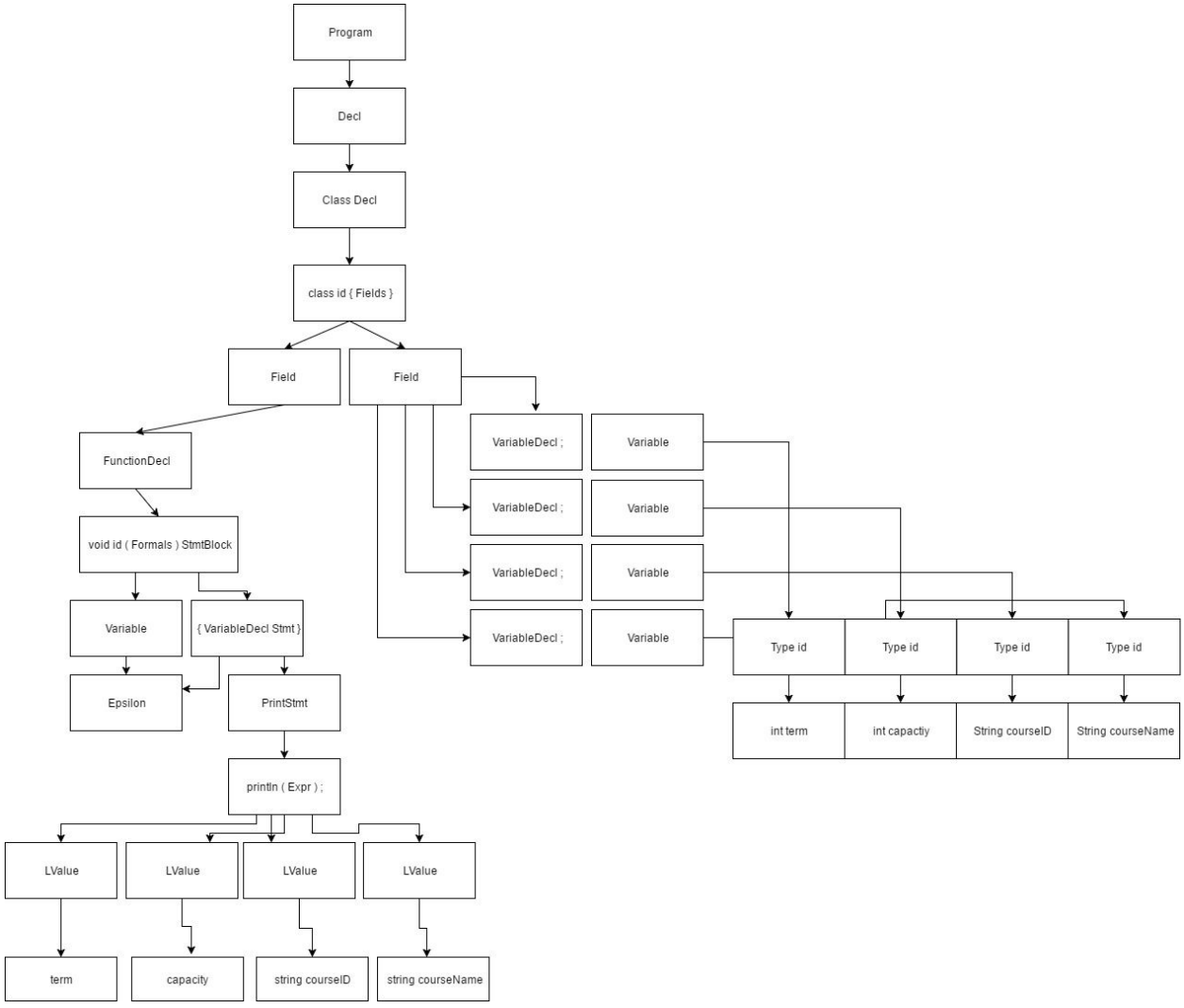
the grammar would change the way that input is parsed. We also tried to change the precedence of the tokens using %prec but it did not solve the conflict.

Our first input file was the project 1 test case. Although this test case looked syntactically correct, the parser ended up rejecting the input. This occurred after the for expression in the statement where total is set equal to total multiplied by x. This failed to parse because the for statement produces a Stmt nonterminal which is then derived into a StmtBlock nonterminal. This statement block produces our left brace terminal, a VariableDecls, a Stmts and a right brace. The VariableDecls nonterminal is then derived into a Variable nonterminal along with a semicolon. The Variable nonterminal then creates A Type and id token. This Type nonterminal is where the error arises as there is no correct way to parse a token that has already been declared from within the Type nonterminal.

Our next failing test was our sample input from the first project in a file called TestFromP1.toy. This test case was supposed to illustrate most of the capabilities of the language while still failing. The reason why it fails it because the lexer fails to parse a language correctly if a type is followed by an id and an assignment operator. This is a result of the grammar and how it parses the input after an interface declaration. The nonterminal InterfaceDecl handles the interface and creates the interface and id tokens along with a Prototype nonterminal surrounded in braces. This Prototype can then produce either a void token or a Type followed by an id token and a Formals nonterminal surrounded by parenthesis. The Type nonterminal can only produce the data type. The Formals nonterminal can produce a Variable nonterminal which then creates another Type nonterminal. Without a way to create an Expr nonterminal, the grammar cannot accept the variable assignment after the interface declaration.

Our first working test case was used to determine if it could correctly parse class and function declarations along with various variable declarations.

Working Test 1 Derivation Tree.



Our second working test program was intended to test the ability to create multiple lines of code inside of one function and nest function calls inside of println statements. The various functions also test out the different operators that can be chosen in the Expr nonterminal as well as the return statement.

Working Test 2 Derivation Tree.