This grammar is lightly adapted from the provided grammar. I refactored both declaration-list and param to convert this from a LL3 grammar to a LL1 grammar. (Before, the rule was param := type-specifier ID [] | type-specifier ID ε. By refactoring, we no longer have too look ahead three items to tokens determine which rule applies here).

1. program := declaration-list | ε
2. declaration-list := declaration-list type-specifier ID declaration | type-specifier ID declaration
3. declaration := var-declaration | fun-declaration
4. var-declaration := ; | [ NUM ] ;
5. type-specifier := int | void
6. fun-declaration := ( params ) compound-stmt
7. params := param-list | void | empty
8. param-list := param-list , type-specifier ID param | type-specifier ID param
9. param := [] | ε
10. compound-stmt := { local-declarations statement-list }
11. local-declarations := local-declarations var-declaration | var-declaration
12. statement-list := statement-list statement | statement
13. statement := expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt | io-stmt
14. io-stmt := input-stmt | output-stmt
15. input-stmt := input ( STRING )
16. output-stmt := output ( STRING ) | output ( expression )
17. expression-stmt := expression ; | ;
18. selection-stmt := if ( simple-expression ) statement | if ( simple-expression ) statement else statement
19. iteration-stmt := while ( expression ) statement
20. return-stmt := return ; | return expression ;
21. expression := var = simple-expression | simple-expression
22. var := ID | ID [ simple-expression ]
23. simple-expression := additive-expression relop additive-expression | additive-expression
24. relop := <= | < | > | >= | == | !=
25. additive-expression := additive-expression addop term | term
26. addop := + | -
27. term := term mulop factor | factor
28. mulop := \* | /
29. factor := ( simple-expression ) | var | call | NUM | input-stmt
30. call := ID ( args )
31. args := arg-list | ε
32. arg-list := arg-list , expression | expression