Original Grammar:

```
program → declaration-list
declaration-list → declarartion-list declaration | declaration
declaration → var-declaration | fun-declaration
var-declaration → type-specifier ID ; | type-specifier ID [ NUM ];
type-specifier → int | void
fun-declaration → type-specifier ID ( params ) compound-stmt
params → param-list | void
param-list → param-list , param | param
param → type-specifier ID | type-specifier ID [ ]
compound-stmt → { local-declarations statement-list }
local-declarations → local-declarations var-declaration | empty
statement-list → statement-list statement | empty
statement → expression-stmt | compound-stmt | selection-stmt
            | iteration-stmt | return-stmt
expression-stmt → expression ; | ;
selection-stmt → if ( expression ) statement
                  | if ( expression ) statement else statement-list
iteration-stmt → while ( expression ) statement
return-stmt → return ; | return expression ;
expression → var = expression | simple-expression
var → ID | ID [ expression ]
simple-expression → additive-expression relop additive-expression
                  | additive-expression
relop → <= | < | > | >= | == | !=
additive-expression → additive-expression addop term | term
addop → + | -
term → term mulop factor | factor
mulop → * | /
factor → ( expression ) | var | call | NUM
call → ID ( args )
args → arg-list | empty
arg-list → arg-list , expression | expression
```

EBNF Grammar:

```
program → declaration-list
declaration-list → declaration {declaration}
declaration → var-declaration | fun-declaration
var-declaration → type-specifier ID [ [ NUM ] ] ;
type-specifier → int | void
fun-declaration → type-specifier ID ( params ) compound-stmt
params → param-list | void
param-list → param { , param }
param → type-specifier ID [ [ ] ]
compound-stmt → { local-declarations statement-list }
local-declarations → [ local-declarations var-declaration ]
statement-list → {statement}
statement → expression-stmt | compound-stmt | selection-stmt
            | iteration-stmt | return-stmt
expression-stmt → [expression] ;
selection-stmt → if ( expression ) statement [ else statement-list ]
iteration-stmt → while ( expression ) statement
return-stmt → return [expression] ;
expression → var = expression | simple-expression
var → ID [ [ expression ] ]
simple-expression → additive-expression [ relop additive-expression ]
relop → <= | < | > | >= | == | !=
additive-expression → [ additive-expression addop ] term
addop → + | -
term → [ term mulop ] factor
mulop → * | /
factor → ( expression ) | var | call | NUM
call → ID ( args )
args → [ arg-list ]
arg-list → expression { , expression }
```

Left Recursion Removal of EBNF Grammar:

```
program → declaration-list
declaration-list → declaration {declaration}
declaration → var-declaration | fun-declaration
var-declaration → type-specifier ID [ [ NUM ] ] ;
type-specifier → int | void
fun-declaration → type-specifier ID ( params ) compound-stmt
params → param-list | void
param-list → param { , param }
param → type-specifier ID [ [ ] ]
compound-stmt → { local-declarations statement-list }
local-declarations → {var-declaration}
statement-list → {statement}
statement → expression-stmt | compound-stmt | selection-stmt
            | iteration-stmt | return-stmt
expression-stmt → [expression] ;
selection-stmt → if ( expression ) statement [ else statement-list ]
iteration-stmt → while ( expression ) statement
return-stmt → return [expression] ;

expression → ( expression ) simple-expression' | NUM simple-expression'
            | ID expression'
expression' → expression | ( args ) simple-expression'
            | [ expression ] expression'' | simple-expression'
expression'' → expression | simple-expression'
var → ID [ [ expression ] ]
simple-expression → additive-expression [ relop additive-expression ]
simple-expression' → additive-expression' [ relop additive-expression ]
relop → <= | < | > | >= | == | !=
additive-expression → [ additive-expression addop ] term
additive-expression' → term' { addop term }
addop → + | -
term → [ term mulop ] factor
term' → { mulop factor }
mulop → * | /
factor → ( expression ) | var | call | NUM

call → ID ( args )
args → [ arg-list ]
arg-list → expression { , expression }
```