Assignment 2

The Grammar After Removing Left Recursion and Common Prefixes

Here is the grammar before removing the left recursion and common prefixes:

1. program -> Program ID **{**declaration-list statement-list**}**.
2. declaration-list -> declaration-list declaration | declaration
3. declaration -> var-declaration
4. var-declaration -> type-specifier **ID ;** | type-specifier **ID [NUM] ;**
5. type-specifier -> **int** | **float**
6. params -> param-list | **void**
7. param-list -> param-list **,** param | param
8. param -> type-specifier **ID** | type-specifier **ID** [ ]
9. compound-stmt -> **{**statement-list**}**
10. statement-list -> statement-list statement | ε
11. statement -> assignment-stmt | compound-stmt | selection-stmt | iteration-stmt
12. selection-stmt -> **if (** expression **)** statement | **if (** expression **)** statement **else** statement
13. iteration-stmt -> **while (** expression **)** statement
14. assignment-stmt -> var **=** expression
15. var -> **ID** | **ID [** expression **]**
16. expression -> expression relop additive-expression | additive-expression
17. relop -> **<=** | **<** | **>** | **>=** | **==** | **!=**
18. additive-expression -> additive-expression addop term | term
19. addop -> **+**|**-**
20. term -> term mulop factor | factor
21. mulop -> ***** | /
22. factor -> **(** expression **)** | var | **NUM**

The rules highlighted in pink are the ones that have either left recursion or a common prefix and are altered in the new grammar.

The new rules added become:

1- program -> Program ID {declaration-list statement-list}.
2.1 - declaration-list -> declaration declaration-list-tail
2.2 - declaration-list-tail -> declaration declaration-list-tail | ε
3 - declaration -> var-declaration
4.1 - var-declaration -> type-specifier ID var-declaration-tail
4.2 - var-declaration-tail -> ; | [NUM] ;
5 - type-specifier -> int | float
7 - params -> param-list | void
8.1 - param-list -> param param-list-tail
8.2 - param-list-tail -> , param param-list-tail | ε
9.1 - param -> type-specifier ID param-tail
9.2 - param-tail -> ε | [ ]
10 - compound-stmt -> {statement-list}
12.1 - statement-list -> empty statement-list-tail
12.2 - statement-list-tail -> statement statement-list-tail | ε
13 - statement -> assignment-stmt | compound-stmt | selection-stmt | iteration-stmt
15.1 - selection-stmt -> if (expression) statement selection-stmt-tail
15.2 - selection-stmt-tail -> else statement | ε
16 - iteration-stmt -> while (expression) statement
18 - assignment-stmt -> var = expression
19.1 - var -> ID var-tail
19.2 - var-tail -> [expression] | ε
20.1 - expression -> additive-expression expression-tail
20.2 - expression-tail -> relop additive-expression expression-tail | ε
21 - relop -> <= | < | > | >= | == | !=
22.1 - additive-expression -> term additive-expression-tail
22.2 - additive-expression-tail -> addop term additive-expression-tail | ε
23 - addop -> +|-
24.1 - term -> factor term-tail
24.2 - term-tail -> mulop factor term-tail | ε
25. mulop -> * | /
26. factor -> (expression) | var | NUM

The full new grammar:

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

To run the program, run the following commands:

```
flex scanner.l
gcc -c lex.yy.c -o lex.yy.o
g++ parser.cpp lex.yy.o -lfl -o parser
./parser < test.txt
```

Implementation Issues and Decisions

The program prints the error and then stops running when faced with an error. The error function is used to display a detailed error message, including the type of error and the line number at which it occurred.

Assignments in the language are not terminated by semicolons. This decision was based on the provided grammar rules. The parser properly processes assignment statements per this rule and does not expect semicolons after assignments, ensuring that the program runs according to the expected syntax.

The parser was designed to handle a variety of language constructs such as declarations, assignments, conditional statements (if-else), loops (while), and expressions.

Handling Syntax Errors

The error message is displayed in the format:
"Syntax error at line X: found 'TOKEN'. Expected 'EXPECTED_TOKEN'."
where X is the line number of the token, TOKEN is the token that caused the error, and EXPECTED_TOKEN is the expected token type according to the grammar rules.

Once an error is detected, the parser stops further execution. This behavior is intentional and ensures that no further incorrect processing occurs once a syntax error is encountered.

Sample Runs of Test cases

Test case 1:

```
Program validProgram {
    /* Declaration List */
    int x;
    float y;
    int z[5];

    /* Statement List */
    x = 10
    y = 3.14

    if (x < 100) {
        x = x + 1
    } else {
        x = x - 1
    }

    while (x <= 200) {
        x = x + 10
    }
}.
```

Test case 2:

```
Program invalidProgram {
    /* Declaration List */
    integer x;
    float y;

    /* Statement List */
    x = 10
    y = 3.14

    if (x < 100) {
        x = x + 1
    } else {
        x = x - 1
    }
```

```
    while (x <= 200) {
        x = x + 10
    }
}.
```

Test case 3:

```
Program invalidAssignment {
    /* Declaration List */
    int x;
    float y;

    /* Statement List */
    x = 10
    y = 3.14

    /* Invalid assignment due to missing operator */
    x 5

    if (x < 100) {
        x = x + 1
    } else {
        x = x - 1
    }

    while (x <= 200) {
        x = x + 10
    }
}.
```

Output:

```
master@BigBrain:~/CompilerDesign/Assignment2$ ./parser < test1.txt
Parsing completed successfully!
master@BigBrain:~/CompilerDesign/Assignment2$ ./parser < test2.txt
Syntax error at line 3: found 'integer'. Expected 'int' or 'float' keyword
master@BigBrain:~/CompilerDesign/Assignment2$ ./parser < test3.txt
Syntax error at line 11: found '5'. Expected token type ASSIGN but found NUM
```