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 full new grammar:
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
6 - params -> param-list | void
7.1 - param-list -> param param-list-tail
7.2 - param-list-tail -> , param param-list-tail | ε
8.1 - param -> type-specifier ID param-tail
8.2 - param-tail -> ε | [ ]
9 - compound-stmt -> {statement-list}
10.1 - statement-list -> empty statement-list-tail
10.2 - statement-list-tail -> statement statement-list-tail | ε
11 - statement -> assignment-stmt | compound-stmt | selection-stmt | iteration-stmt
12.1 - selection-stmt -> if (expression) statement selection-stmt-tail
12.2 - selection-stmt-tail -> else statement | ε
13 - iteration-stmt -> while (expression) statement
14 - assignment-stmt -> var = expression
15.1 - var -> ID var-tail
15.2 - var-tail -> [expression] | ε
16.1 - expression -> additive-expression expression-tail
16.2 - expression-tail -> relop additive-expression expression-tail | ε
17 - relop -> <= | < | > | >= | == | !=
18.1 - additive-expression -> term additive-expression-tail
18.2 - additive-expression-tail -> addop term additive-expression-tail | ε
19 - addop -> +|-
20.1 - term -> factor term-tail
20.2 - term-tail -> mulop factor term-tail | ε
21. mulop -> * | /
22. factor -> (expression) | var | NUM

Implementation Issues and Decisions

To represent each token type, we declare an enumerated type TokenType in a scanner.h file. This file is included in both the parser.cpp and scanner.l so that both the scanner and parser have access to it.

A struct, Token, is used to store the tokens. It has an attribute of TokenType for the type, a string representing its value, and a line to store on what line that token was encountered.

To generate readable error messages, an array (tokenTypeNames[]) maps each token's enum value to its string name. This allows for descriptive error messages.

The parser and scanner are integrated together by making every token produced in the scanner be returned directly by the Flex-generated function (using return <TOKEN> in scanner.l). The parser (in parser.cpp) declares an external function:
extern "C" int yylex();

```
extern "C" int yylex();
```

and uses a function getToken() that calls yylex() to retrieve the next token. So, tokens produced by the scanner are immediately consumed by the parser.

When faced with an error, the program prints the error and then stops running. The error function is used to display a detailed error message, including the type of error and the lookahead token that caused it in contrast to what was expected, as well as 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.

If the error is from the lexical analyzer where it encounters a wrong number format, a wrong identifier format, or an unknown character, then the lexical analyzer prints a representative error and exits directly.

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

Below are a number of test cases and their outputs demonstrating the parser's output when an input program is valid and other cases where there are issues with it.

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
    }
}.
```

Output:

```
zein@Zein:/mnt/c/Users/zeinn/Downloads/Assignment2$ ./parser < test1.txt
Parsing completed successfully!
```

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
    }
}.
```

Output:

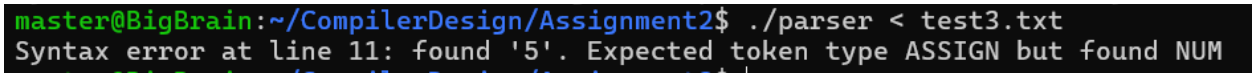
```
zein@Zein:/mnt/c/Users/zeinn/Downloads/Assignment2$ ./parser < test2.txt
Syntax error at line 3: found 'integer'. Expected 'int' or 'float' keyword
```

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:



Test case 4:

```
Program programWithLexicalErr {
    /* Declaration List */
    int 23e+sds;
    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:

```
zein@Zein:/mnt/c/Users/zeinn/Downloads/Assignment2$ ./parser < test4.txt
Lexical Error: Wrong number format at line 13, column 9: 23e+sds
```