Here is the recommended approach for project 1.

1) First build the skeleton for project 1 as shown in part 5 of the video series on lexical analysis using the make file provided. Then run it on the test cases `test1.txt` – `test3.txt` that are provided in Project 1 Test Data and be sure that you understand how it works. Examine the contents of `lexemes.txt`, so that you see the lexeme-token pairs that it contains.

2) A good starting point would be item 1 in the requirements, which includes the additional reserved words of the language. Each of these is a separate token and requires a separate translation rule. Examine the existing translation rules for the reserved words as an example of how to proceed. In addition, add the token names for each one to the enumerated type `Tokens` in `tokens.h`. The order in which you add them is unimportant. Rebuild the program with the make file to ensure that it builds correctly.

Use `test4.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test4.txt

   1  // Function with All Reserved Words
   2
   3  function main returns character;
   4      number: real is when 2 < 3, 0 : 1;
   5      values: list of integer is (4, 5, 6);
   6  begin
   7      if number < 6.3 then
   8          fold left + (1, 2, 3) endfold;
   9      elsif 6 < 7 then
  10          fold right + values endfold;
  11      else
  12          switch a is
  13              case 1 => number + 2;
  14              case 2 => number * 3;
  15              others => number;
  16          endswitch;
  17      endif;
  18  end;

Compiled Successfully
```

You should receive no lexical errors. At this point, you should also examine `lexemes.txt` to see each new reserved word has a unique token number.

3) Adding all the operators as specified by items 2-8 in the requirements would be a good next step. Examine the existing translation rules for the existing operators as an example of how to proceed. As before, you must also add the token names for each new operator to the enumerated type `Tokens` in `tokens.h`.

Use `test5.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test5.txt

   1  // Program Containing the New Operators
   2
   3  function main b: integer, c: integer returns integer;
   4       a: integer is 3;
   5  begin
   6      if (a < 2) | (a > 0) & (~b <> 0) then
   7          7 - 2 / (9 % 4);
   8      else
   9          if b >= 2 | b <= 6 & !(c = 1) then
  10              7 + 2 * (2 + 4);
  11          else
  12              a ^ 2;
  13          endif;
  14      endif;
  15  end;

Compiled Successfully
```

As before, you should receive no lexical errors. At this point, you may again want to examine `lexemes.txt` to see that each new operator has the appropriate token number.

4) As the last modification to `scanner.l` and `tokens.h`, add the new comment, modify the identifier and character literal tokens and add the real literal, and hexadecimal integer literal tokens as specified by items 9-13 in the requirements.

Use `test6.txt` to test this modification. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test6.txt

   1  // Program Containing the New Comment, Modified Identifier
   2  //     and Real Literal and Hex and Character Literals
   3
   4  -- This is the new style comment
   5
   6  function main b: integer, c: integer returns integer;
   7       a: real is .3;
   8       d: real is 5.7;
   9       a__1: real is .4e2;
  10       ab_c_d: real is 4.3E+1;
  11       ab1_cd2: real is 4.5e-1;
  12       hex: integer is #2aF;
  13       char1: character is 'C';
  14       char2: character is '\n';
  15  begin
  16       hex + 2;
  17  end;

Compiled Successfully
```

As before, you should receive no lexical errors.

5) The final required change is to modify the three functions that generate the compilation listing as described in the requirements so that the number of errors are displayed at the end if any occur, or the message *Compilation Successful* is displayed if none occur. In addition, the modifications should ensure that all the error messages that have occurred on the previous line are displayed. Rerunning any of the previous test cases should confirm that the *Compilation Successful* is displayed.

Use `test7.txt` to test whether multiple lexical errors on the same line are displayed. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test7.txt

   1  // Function with two lexical errors
   2
   3  function main returns integer;
   4  begin
   5      7 $ 2 ? (2 + 4);
Lexical Error, Invalid Character $
Lexical Error, Invalid Character ?
   6  end;

Lexical Errors 2
Syntax Errors 0
```

6) As a final test, `test8.txt` contains every punctuation symbol, reserved word, operator and both valid and invalid identifiers. Shown below is the output that should result when using that test case as input:

```
$ ./compile < test8.txt

   1  -- Punctuation symbols
   2
   3  ,:;() =>
   4
   5  // Valid identifiers
   6
   7  name_1
   8  name_1__a2_ab3
   9
  10  // Invalid identifiers
  11
  12  name___2
Lexical Error, Invalid Character _
Lexical Error, Invalid Character _
Lexical Error, Invalid Character _
  13  _name3
Lexical Error, Invalid Character _
  14  name4_
Lexical Error, Invalid Character _
  15
  16  // Integer Literals
  17
  18  23 #3aD
```

```
 19
 20  // Real Literals
 21
 22  123.45 .123 1.2E2 .1e+2 1.2E-2
 23
 24  // Character Literals
 25
 26  'A' '\n'
 27
 28  // Logical operators
 29
 30  & | !
 31
 32  // Relational operators
 33
 34  = <> > >= < <=
 35
 36  // Arithmetic operators
 37
 38  + - * / % ^ ~
 39
 40  // Reserved words
 41
 42  begin case character else elsif end endcase endfold endif endswitch
 43  fold function if integer is left list of others real returns right
 44  switch then when

Lexical Errors 5
Syntax Errors 0
Semantic Errors 0
```

In addition to verifying that your lexical analyzer generates the same lexical errors, you should also examine the `lexemes.txt` file generated to be sure that every lexeme has the proper token associated with it.

All of the test cases discussed above are included in the attached .zip file. In addition, the `lexemes.txt` file from the final test case is included to compare with yours. Keep in mind that the actual token numbers will depend on the order of your `Tokens` enumerated type, so they may not exactly match.

You are certainly encouraged to create any other test cases that you wish to incorporate in your test plan. Keep in mind that your scanner should generate a lexical error for any input program that contains one, so it is recommended that you choose some different test cases as a part of your test plan. A comparable but different set of test cases may be used when testing your project.