Here is the approach I recommend 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 Skeleton Test Data file under the Week 2 videos 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 items 1 and 2 in the requirements, which include the arrow symbol and the additional reserved words of the language. Each of these is a separate token and require 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  -- Program containing arrow symbol and new reserved words
   2
   3  function main b: integer returns integer;
   4       a: real is 3;
   5  begin
   6      if a < 2 then
   7          7 + 2 * (2  + 4);
   8      else
   9          case b is
  10              when 1 => a * 2;
  11              when 2 => a + 5;
  12              others => a + 4;
  13          endcase;
  14      endif;
  15  end;
```

You should receive no lexical errors. At this point, you should also examine `lexemes.txt` to see each new reserved word and the arrow symbol have unique token numbers.

3) Adding all the operators as specified by items 3-8 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: real is 3;
 5  begin
 6      if (a < 2) or (a > 0) and (b /= 0) then
 7          7 - 2 / (9 rem 4);
 8      else
 9          if b >= 2 or b <= 6 and not(c = 1) then
10              7 + 2 * (2 + 4);
11          else
12              a ** 2;
13          endif;
14      endif;
15  end;
```

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 add the remaining literals as specified by items 9-12 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 comment, modified identifer and new literals
 2
 3  // This is the new style comment
 4
 5  function main b: integer, c: integer returns real;
 6       a: real is 3.;
 7       d: real is 5.7;
 8       a_1: real is 4.e2;
 9       ab_c_d: real is 4.3E+1;
10       ab1_cd2: real is 4.e-1;
11  begin
12      if (a < 2) or (a > 0) and (b /= 0) or false then
13          7 - 2 / (9 rem 4);
14      else
15          if b >= 2 or b <= 6 and not(c = 1) and true then
16              a_1 + ab_c_d * (ab1_cd2 + 4);
17          else
18              a ** 2;
19          endif;
20      endif;
21  end;
```

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
Semantic 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 _
  13  _name3
Lexical Error, Invalid Character _
  14  name4_
Lexical Error, Invalid Character _
  15
  16  -- Literals
  17
  18  123 123.45 123. 1.2E2 1.e+2 1.2E-2 true false
  19
  20  -- Logical operators
  21
  22  and or not
  23
  24  -- Relational operators
  25
```

```
  26  = /= > >= < <=
  27
  28  -- Arithmetic operators
  29
  30  + - * / rem **
  31
  32  -- Reserved words
  33
  34  begin boolean case else end endcase endif function if is integer real
returns then when

Lexical Errors 4
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 I recommend that you choose some different test cases as a part of your test plan. I may use a comparable but different set of test cases when I test your projects.