

CMSC 430 - Project 1

Lexical Analyzer and Compiler

Angel Lee

Oct 30, 2021

How I approached Project 1

Using Linux and developing arguments for id and the second comment

I have never coded on a Linux operating system before, and even though I worked with C and C++, I am not as fluent at it as Java. Hence, I had to review the flex and yacc tutorial by Niemann and a tutorial from the Oracle website to warm up. I had to figure out how to tell the c program (on the scanner lex file) what to do with the "secondcomment" and "id" expressions and how to put arguments for them.

For the *id*, when I used `[A-Za-z]_[A-Za-z0-9]*` as the argument, it ran fine until I tried other test cases, and realized I need to use parenthesis to let `[A-Za-z]` come first AND THEN look for an optional underscore in the middle, which HAS to be followed by characters in `[A-Za-z0-9]`.

Oracle source: <https://docs.oracle.com/cd/E19504-01/802-5880/lex-6/index.html>

Real literal token argument

Writing the argument for the "real" expression was the most complicated part of the project for me, and I had to be very slow and careful.

I came up with argument: `{digit}+(.){digit}*([eE][+-]?{digit}+)?`

`{digit}+` was written to mean a sequence of one or 'more' (signified by +) digits. I first used * for "zero or more" digits but then realized since the instructions asked for ONE or more copy, I switched from `{digit}*` to `{digit}+`.

Then I followed this with the decimal (.), making the argument `{digit}+(.)`, then followed by `{digit}*` to signify zero or more additional digits after the decimal.

Then I added `[eE]?` for an optional end with an exponent e or E. For the optional end, it starts with e or E but also must have the option for a "+" or "-" sign, so `[+-]?` was nested after `[eE]` inside the `()` creating `([eE][+-]?)?`

My first argument was `[eE]?[+-]?`, but then this read as "e/E is optional AND +/- is optional", instead of "if e/E option is picked, then +/- is an option". Hence, I corrected it to `([eE][+-]?)?`

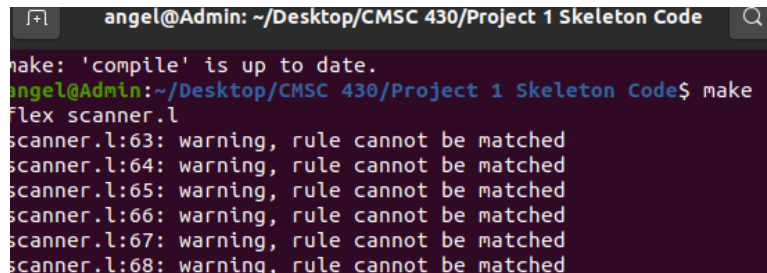
Then the instruction said `[+-]?` must be followed by "one or more" copies of digits, which is signified as `{digit}+`. Hence, `{digit}+` was nested next to `[+-]?` inside `([eE][+-]?)?`

Together, all these instructions combined made `{digit}+(.){digit}*([eE][+-]?{digit}+)?`

Even though I have used C before, I haven't had to write something as specific as the real literal token's argument.

Compilation warnings

Even though the code compiled, there were mentions of "warning, rule cannot be matched" in the compilation. I then reorganized the order of rules in order to avoid these errors. For example I noticed I need to put {int}, {real}, and {bool} before {id}. Hence, I learned that the order of rules matters in compiling. It is the same with Java, but Java is more forgiving.

A terminal window with a dark background and light-colored text. The title bar reads 'angel@Admin: ~/Desktop/CMSC 430/Project 1 Skeleton Code'. The terminal shows the command 'make' being executed, which outputs 'make: 'compile' is up to date.' followed by 'angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code\$ make'. Below this, it shows 'flex scanner.l' and then a series of six warning messages from 'scanner.l' at lines 63 through 68, each stating 'warning, rule cannot be matched'.

Methods or Functions amendment

The methods "lastLine" and "appendError" were amended in the listing.cc file to specify the types of errors that popped up during execution. C++'s grammar is very similar to Java's, so I wrote the methods as usual as in Java. I initialized three int variables to count the lexical, syntax, and semantic errors if there were any errors (meaning if not totalErrors == 0, then count the specific errors by category).

I noticed that ErrorCategories was already described as an enum category in the listing.h header file, so I just used that to let appendError method categorize the error types. Lexical errors would count as appendix 0 according to the ErrorCategories enum, so Syntax == 1, and Semantic == 2.

Queue

Another challenging part of the project was having each lexical error from the same line display their own error message. I noticed that the skeleton code only displays the error message for the last lexical error (for example when \$ and then ^ was read, only ^ would show). I created a String queue that all the error messages could be added to in appendErrors, so they can be displayed and then cleared afterwards in displayErrors. The queue should be "first in, first out".

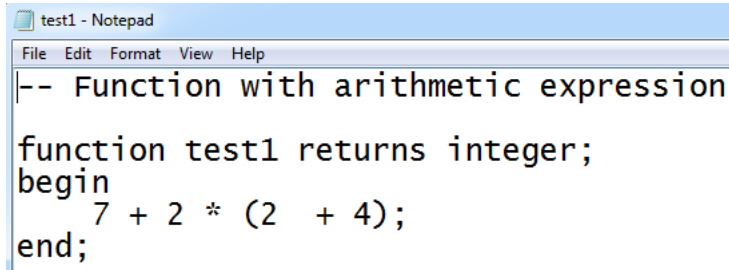
Test Plan:

I made 6 test files (included in the ZIP file): 3 from the video lecture, 4th as an edited version of test3.txt that tests all the additional modifications I made to the skeletal code files, and 5th and 6th from the examples in the Project Requirement PDF. To see if my modification of the skeletal codes worked, I compared my modified code to the skeletal code because the latter displays errors that the modified code doesn't, and vice versa.

I then ran "cat lexemes.txt" like shown in the class lecture video to show all the lexemes in each test file.

Test case	Expected output	Met expectation?
1, from video lecture	Compile successfully	Yes
2, from video lecture	Lexical error due to (^), describe the error under the line in question, and then a count of all types of the errors	Yes
3, from video lecture	Compile successfully	Yes
4, edited from test 3	Lexical error due to inappropriately placed underscores, describe each error under the line in question, and then a count of all types of the errors (total of 7)	Yes
5 from Project req. PDF	Compile successfully	Yes
6 from Project req. PDF	Lexical error due \$ and ^, describe each error under the line in question, and then a count of all types of the errors (total of 2)	Yes

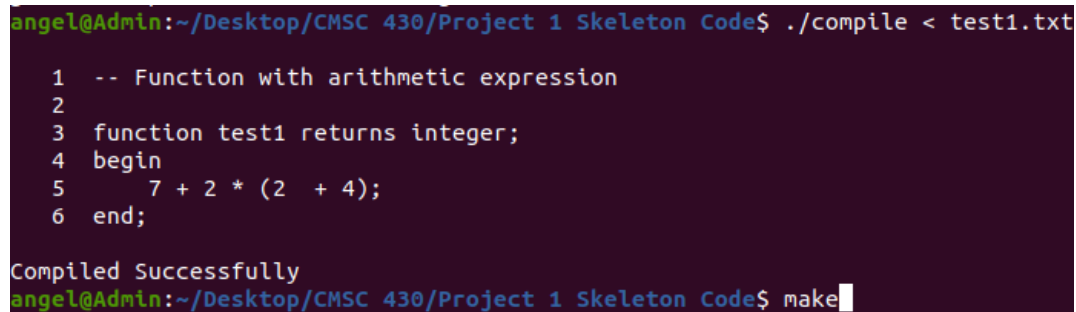
Test 1:



```
test1 - Notepad
File Edit Format View Help
|-- Function with arithmetic expression

function test1 returns integer;
begin
    7 + 2 * (2 + 4);
end;
```

Compilation for test 1:



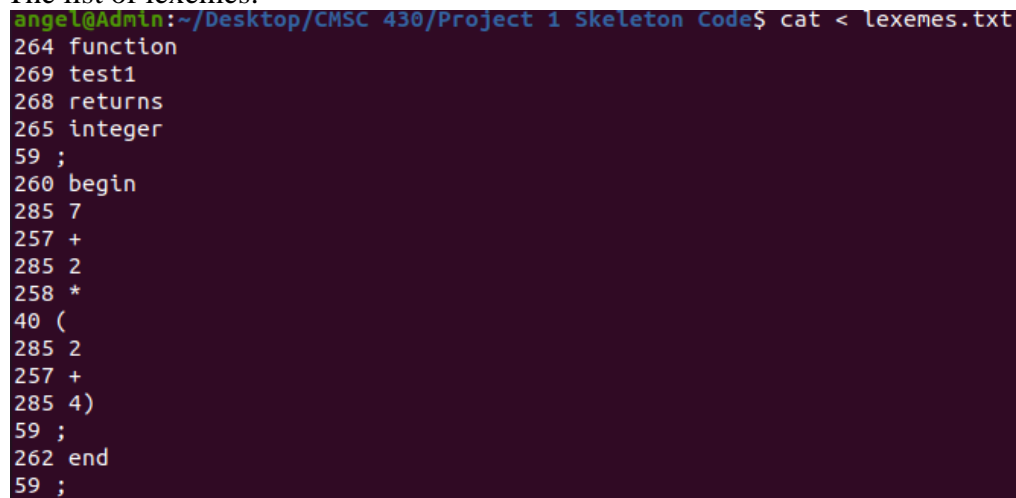
```
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ ./compile < test1.txt

1  -- Function with arithmetic expression
2
3  function test1 returns integer;
4  begin
5      7 + 2 * (2 + 4);
6  end;

Compiled Successfully
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ make
```

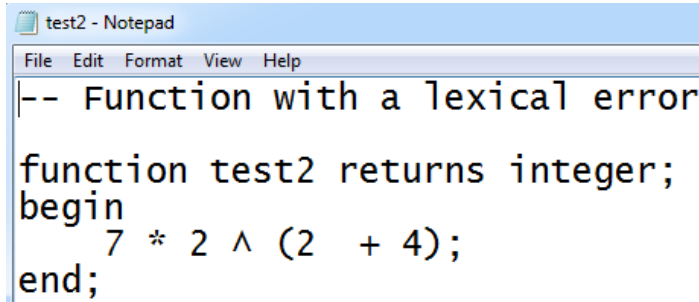
For test 1, this compiled successfully as there were no issues.

The list of lexemes:



```
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ cat < lexemes.txt
264 function
269 test1
268 returns
265 integer
59 ;
260 begin
285 7
257 +
285 2
258 *
40 (
285 2
257 +
285 4)
59 ;
262 end
59 ;
```

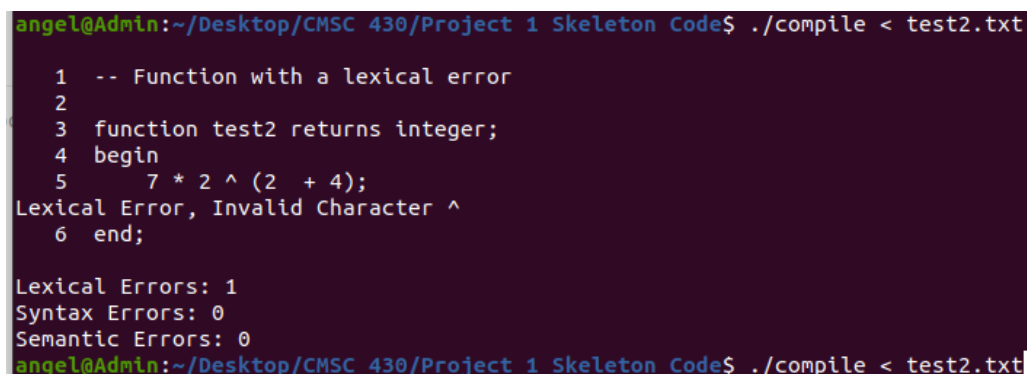
Test 2:



```
test2 - Notepad
File Edit Format View Help
-- Function with a lexical error

function test2 returns integer;
begin
    7 * 2 ^ (2 + 4);
end;
```

Compilation for test 2:



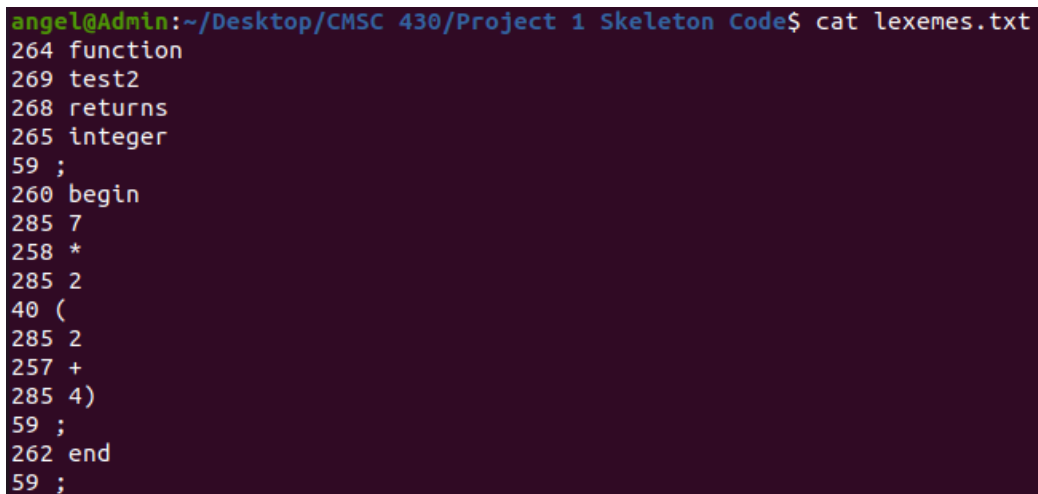
```
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ ./compile < test2.txt

1  -- Function with a lexical error
2
3  function test2 returns integer;
4  begin
5      7 * 2 ^ (2 + 4);
Lexical Error, Invalid Character ^
6  end;

Lexical Errors: 1
Syntax Errors: 0
Semantic Errors: 0
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ ./compile < test2.txt
```

There was one lexical error (^) which the program caught and printed out the list of errors, as expected.

List of lexemes:



```
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ cat lexemes.txt
264 function
269 test2
268 returns
265 integer
59 ;
260 begin
285 7
258 *
285 2
40 (
285 2
257 +
285 4)
59 ;
262 end
59 ;
```

Test 3:

```
test3.txt x
1  -- Punctuation symbols
2
3  , ; ( )
4
5  -- Identifier
6
7  name name123
8
9  -- Literals
10
11 123
12
13 -- Logical operators
14
15 and or not
16
17 -- Relational operators
18
19 <
20
21 -- Arithmetic operator
22
23 +
24
25 -- Reserved words
26
27 begin else end endif endreduce function if is integer reduce returns then
28
```

Compilation for test 3:

```
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ ./compile < test3.txt

1  -- Punctuation symbols
2
3  ,;()
4
5  -- Identifier
6
7  name name123
8
9  -- Literals
10
11 123
12
13 -- Logical operators
14
15 and or not
16
17 -- Relational operators
18
19 <
20
21 -- Arithmetic operator
22
23 +
24
25 -- Reserved words
26
27 begin else end endif endreduce function if is integer reduce returns then

Compiled Successfully
```

Test 3 compiled successfully as there were no characters out of bound.

List of lexemes:

```
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ cat < lexemes.txt

44 ,
59 ;
40 (
41 )
269 name
269 name123
270 123
259 and
281 or
282 not
256 <
257 +
260 begin
273 else
262 end
275 endif
263 endreduce
264 function
276 if
266 is
265 integer
267 reduce
268 returns
279 then
```


Test 4:

```
test4.txt x
1 // Testing comments with slashes instead of dashes
2
3 //more punct symbol
4
5 =>
6
7 // strings with underscores. has 7 lexical errors
8
9 test_1
10 t_est2
11 test__3
12 test___4
13 _test5
14 test6_
15
16 // more Relational operators
17
18 = /= > >= <=
19
20
21 // more Arithmetic operator
22
23 * - /
24
25 // remop and expop
26
27 rem **
28
29 // real literal token test
30
31 3.54 3.0e1 2.0e+12 3.5E45
32
33 // boolean literal token
34
35 true false
36
37 // more Reserved words
38
39 begin else end endif endreduce function if is integer reduce returns then
40
41 case endcase others real when
```

Compilation for test 4 (screenshot 1):

```
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ ./compile < test4.txt
1 // Testing comments with slashes instead of dashes
2
3 //more punct symbol
4
5 =>
6
7 // strings with underscores. has 7 lexical errors
8
9 test_1
10 t_est2
11 test__3
Lexical Error, Invalid Character _
Lexical Error, Invalid Character _
12 test___4
Lexical Error, Invalid Character _
Lexical Error, Invalid Character _
Lexical Error, Invalid Character _
13 _test5
Lexical Error, Invalid Character _
14 test6_
Lexical Error, Invalid Character _
15
16 // more Relational operators
17
18 = /= > >= <=
```

Compilation continued (screenshot 2):

```
18 = /= > >= <=
19
20
21 // more Arithmetic operator
22
23 * - /
24
25 // remop and expop
26
27 rem **
28
29 // real literal token test
30
31 3.54 3.0e1 2.0e+12 3.5E45
32
33 // boolean literal token
34
35 true false
36
37 // more Reserved words
38
39 begin else end endif endreduce function if is integer reduce returns then
40
41 case endcase others real when
Lexical Errors: 7
Syntax Errors: 0
Semantic Errors: 0
```

Test 4 created more errors with the original skeletal code due to the slashes (//) and underscores. For the modified code however, the slashes passed as I created another type of comment that can

accept slashes, and modified the "id" expression to accept just one underscore wedged between letters. I also added more lexemes as the project specifications requested, and these popped errors in the original skeletal code yet they passed in the modified one. Total errors were 7, which were all due to the duplicate underscores. As expected, the output had a separate error message line for each lexical error of the same line. Hence, my code passed my expectations.

List of lexemes:

```
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ cat lexemes.txt
271 =>
269 test_1
269 t_est2
269 test
285 3
269 test
285 4
269 test5
269 test6
256 =
256 /=
256 >
256 >=
256 <=
258 *
257 -
258 /
283 rem
284 **
285 3.54
285 3.0e1
285 2.0e+12
285 3.5E45
286 true
286 false
260 begin
273 else
262 endif
275 endif
263 endreduce
264 function
276 if
266 is
265 integer
267 reduce
268 returns
279 then
272 case
274 endcase
277 others
278 real
280 when
```

Test 5:

```
test5.txt x
1  (* Program with no errors *)
2
3  function test5 returns boolean;
4  begin
5      7 + 2 > 6 and 8 = 5 * (7 - 4);
6  end;
```

Compilation for test 5:

```
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ ./compile < test5.txt

1  (* Program with no errors *)
2
3  function test5 returns boolean;
4  begin
5      7 + 2 > 6 and 8 = 5 * (7 - 4);
6  end;

Compiled Successfully
```

Test 5 was copied from the project specification PDF, and as expected, it compiled successfully.

List of lexemes:

```
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ cat < lexemes.txt
40 (
258 *
269 Program
269 with
269 no
269 errors
258 *
41 )
264 function
269 test5
268 returns
261 boolean
59 ;
260 begin
285 7
257 +
285 2
256 >
285 6
259 and
285 8
256 =
285 5
258 *
40 (
285 7
257 -
285 4)
59 ;
262 end
59 ;
```

Test 6:

```
test6.txt x |
1  -- Function with two lexical errors
2
3  function test6 returns integer;
4  begin
5      7 $ 2 ^ (2 + 4);
6  end;
```

Compilation for test 6:

```
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ ./compile < test6.txt

1  -- Function with two lexical errors
2
3  function test6 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
```

Test 6 was copied from the project specification PDF, and as expected, the output had a separate error message line for each lexical error of the same line.

List of lexemes:

```
angel@Admin:~/Desktop/CMSC 430/Project 1 Skeleton Code$ cat lexemes.txt
264 function
269 test6
268 returns
265 integer
59 ;
260 begin
285 7
285 2
40 (
285 2
257 +
285 4)
59 ;
262 end
59 ;
```

Lessons Learned and any improvements that could be made

I wish we were provided more code samples in class, or taught better research skills with some external sources for c++ provided in class - similar to what the Oracle website is for Java programming. For some of the items, like the queue, I spent hours looking for something that worked for me before coming across the pre-made queue method and thinking “Oh. This was simple.” But, this is common with programming. For example, I often skim over 200+ lines of code to find out why the program isn't working (it compiles but is interpreted wrong, so often a semantic error), only to find out it was a misplaced curly brace or because an initialized variable was nested inside a method it shouldn't be in.

Sources for the queue from websites that might be good resources to learn more (like Oracle):

<https://en.cppreference.com/w/cpp/container/queue>

<https://www.cplusplus.com/reference/queue/queue/pop/>