CMSC 430 - Project 2

Syntax Analyzer and Compiler

Angel Lee

Nov 3, 2021

How I approached Project 2

First I took notes of what the project requires, for example the order of precendence. I took note that all operators are left associative except EXPOP (exponent).

I copy pasted the scanner.l and listing.cc files from Project 1 because it already described the arguments for a lot of the lexemes and tokens that will be used for Project 2, but had to delete the main method in that file so it didn't collide with the main method in Project 2's parser.

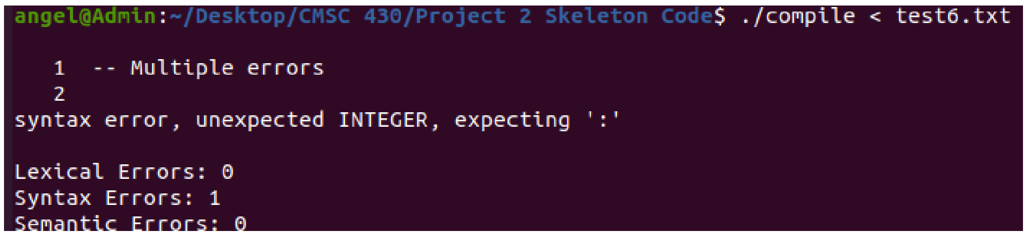
I removed the %token's in parser.y and tried #include "tokens.h" instead, but it wasn't pulling in. I suspected it had something to do with how the makeFile file was set up, but decided it's easier to transfer all the tokens from tokens.h and manually write them into the parser.y file instead.

The order of precedence for the tokens/operators should be: EXPOP, MULOP, REMOP, ADDOP, RELOP, ANDOP, OROP. I edited the parsing methods according to this.

REMOP is the remainder operator so I nested it with MULOP so it can be used after an integer division.

Adding error production to function_header

When I first tested Test6.txt which had the test case from the Project requirement PDF, the parsing stopped after catching the first error on Line 2:

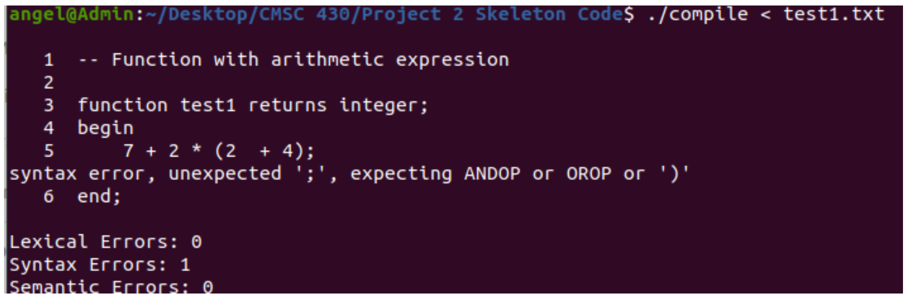


```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test6.txt

   1  -- Multiple errors
   2
syntax error, unexpected INTEGER, expecting ':'

Lexical Errors: 0
Syntax Errors: 1
Semantic Errors: 0
```

I looked at "function main a integer returns real" and cut it. "A integer" in this case would be an optional parameter, and there should have been ':' between a and integer. I noticed that the parser keeps going when there is an error in the *body*, so I looked at the *body* method, saw *statement_* had **error ';' ;** and placed that in the function_header method so it can recover and keep parsing after hitting an error.

Scanner problems with the real literal token argument

When I wrote out the methods for parsing, there was an issue where the ';' in the statements (statement_) were not caught by the parser for some reason. Test6 came out fine, but the other tests did not. Here is an example:



```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test1.txt

   1  -- Function with arithmetic expression
   2
   3  function test1 returns integer;
   4  begin
   5      7 + 2 * (2  + 4);
syntax error, unexpected ';', expecting ANDOP or OROP or ')'
   6  end;

Lexical Errors: 0
Syntax Errors: 1
Semantic Errors: 0
```

I spent hours trying to figure out what was wrong with the parser, and compared it to the original skeletal file which didn't have any issues parsing test cases 1-3 (test cases came from the lecture). I realized the parsing problem might have something to do with the scanner.l file while comparing it with the original skeletal scanner. I edited and tested each line, that found it was the "real" token and its argument that was off.

I have no idea why the real literal token would have anything to do with the statement_ method missing its semicolon, but on the argument I originally wrote for it which was ***{digit}+(.){digit}*([eE][+-]?{digit}+)?*** I switched the (.) to **\.** and it worked. It was such a simple bug and it took too long to find it. I still don't understand why it was giving issues reading the semicolon. It is now ***{digit}+\.{digit}*([eE][+-]?{digit}+)?***

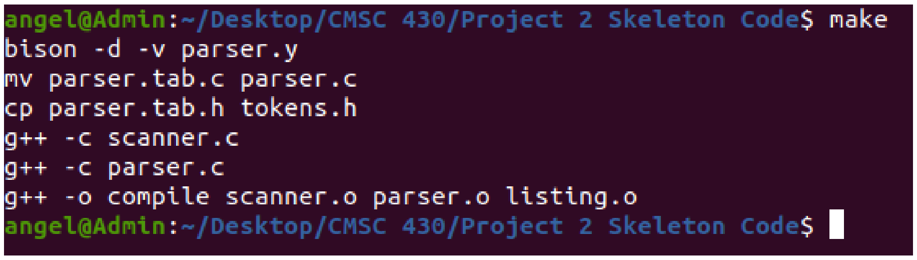
Optional variable on top of optional variable

I noticed that for the test case from the PDF, there were TWO optional variable lines instead of just one. Hence, when I corrected the test case from the PDF (like in Test case 12 in my project), it still gave me an error. I don't know if I am supposed to remove the 2nd variable line but in case that the program is supposed to allow the user to input more than one variable line, I updated the optional_variable method to "optional_variable variable" so the first expression can multiply on itself and allow the user to input more variable lines.

Shift/reduce or reduce/reduce errors

Sometimes as I was trying to write out my parsing methods, there were warnings during compilation that would mention them - yet not tell them where the issues were. Most of the errors happened while trying to write out how the expression statements should be parsed.

Eventually with meticulous fixing, there were no errors when my files were compiling such as shift/reduce or reduce/reduce errors:



```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ make
bison -d -v parser.y
mv parser.tab.c parser.c
cp parser.tab.h tokens.h
g++ -c scanner.c
g++ -c parser.c
g++ -o compile scanner.o parser.o listing.o
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$
```

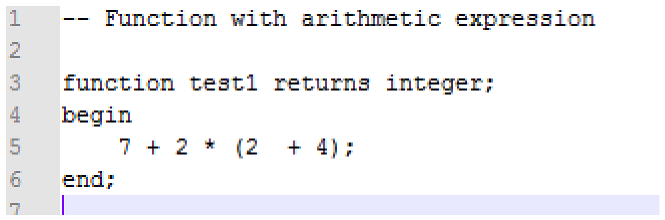
Test Plan:

I made 12 test files (included in the ZIP file): 5 from the video lecture, 6th from the Project Requirement PDF, and 7-12th are variations of the previous 6 files to test more of the program. To see if my modification of the skeletal codes worked, I compared my modified code to the skeletal code.

Test case	**Expected output**	**Met expectation?**
1, from video lecture	Compiled Successfully	Yes
2, from video lecture	Compiled Successfully	Yes
3, from video lecture	Compiled Successfully	Yes
4, from video lecture	Compiled Successfully	Yes
5, from video lecture	Syntax error output due to missing operator	Yes
6 from Project req. PDF	4 syntax errors, exactly like the one from Project req PDF	Yes
7, based off test 1	Syntax error output due to an extra ')'	Yes
8, based off test 2	Syntax error due to a missing ':' in the optional variable line	Yes
9, based off test 3	Syntax error due to invalid return type	Yes
10, based off test 4	A syntax error and a lexical error due to invalid identifiers and characters in the wrong places	Yes
11, based off test 5	Compiled Successfully	Yes
12, based off test 6	Compiled Successfully	Yes

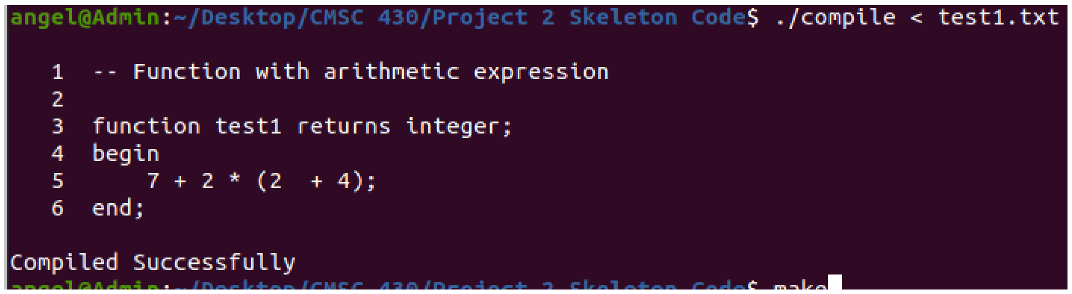
Test1:

This tests if the parser reads and passes valid arithmetic expressions and operators.



```
-- Function with arithmetic expression

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

```

Compilation for test 1:

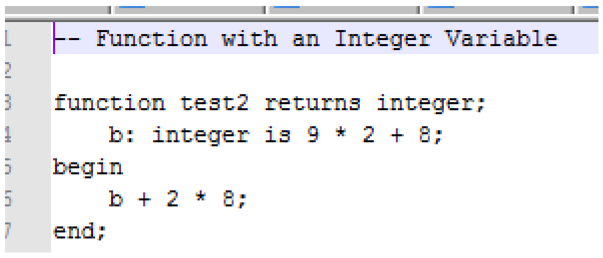


```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test1.txt

   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 2 Skeleton Code$ make
```

This compiled successfully as there were no issues due to no syntax errors.

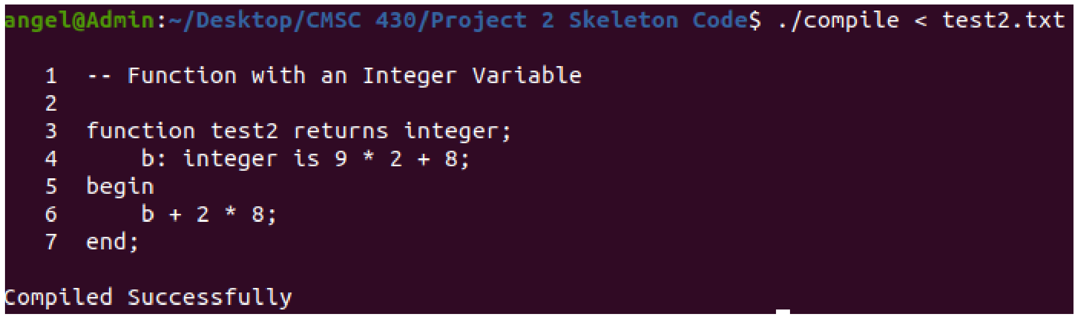
Test 2:

This tests whether the method to parse the optional variable line wedged between the function header and the rest of the body works. It should be in the form of [identifier] : [type] is [expression or statement].



```
-- Function with an Integer Variable

function test2 returns integer;
    b: integer is 9 * 2 + 8;
begin
    b + 2 * 8;
end;
```

Compilation for test 2:

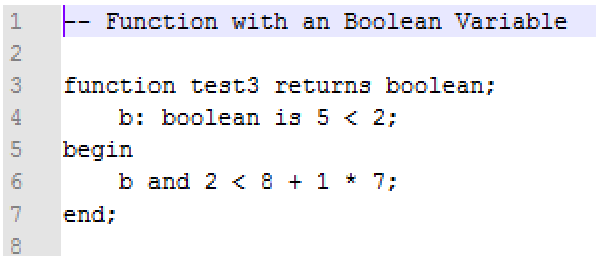


```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test2.txt

   1  -- Function with an Integer Variable
   2
   3  function test2 returns integer;
   4      b: integer is 9 * 2 + 8;
   5  begin
   6      b + 2 * 8;
   7  end;

Compiled Successfully
```

This compiled successfully as there were no issues due to no syntax errors.

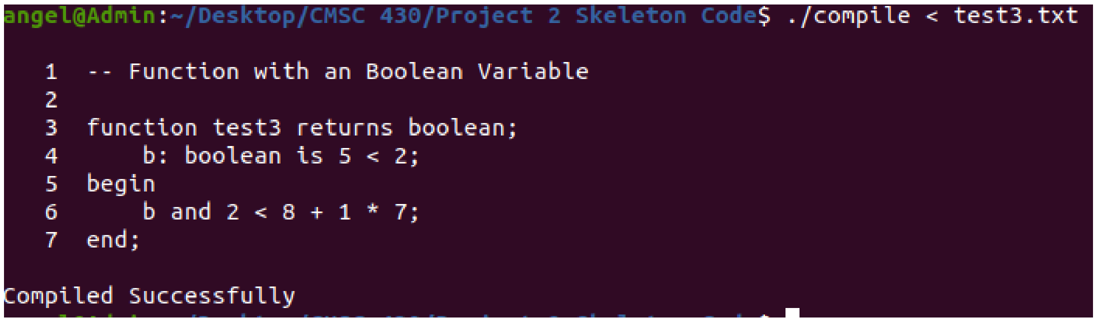
Test3:

Similar to test 2, but returns type boolean instead of integer, and should compile with no issue



```
-- Function with an Boolean Variable

function test3 returns boolean;
    b: boolean is 5 < 2;
begin
    b and 2 < 8 + 1 * 7;
end;
```

Compilation for test 3:

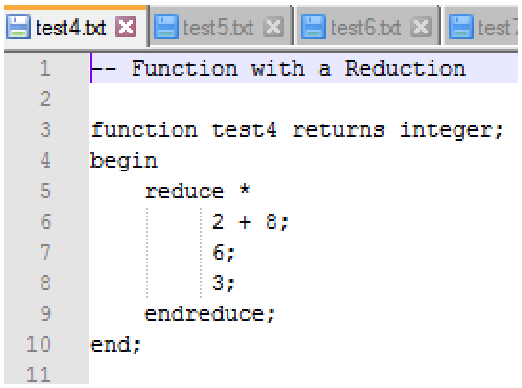


```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test3.txt

   1  -- Function with an Boolean Variable
   2
   3  function test3 returns boolean;
   4      b: boolean is 5 < 2;
   5  begin
   6      b and 2 < 8 + 1 * 7;
   7  end;

Compiled Successfully
```

This compiled successfully as there were no issues due to no syntax errors.

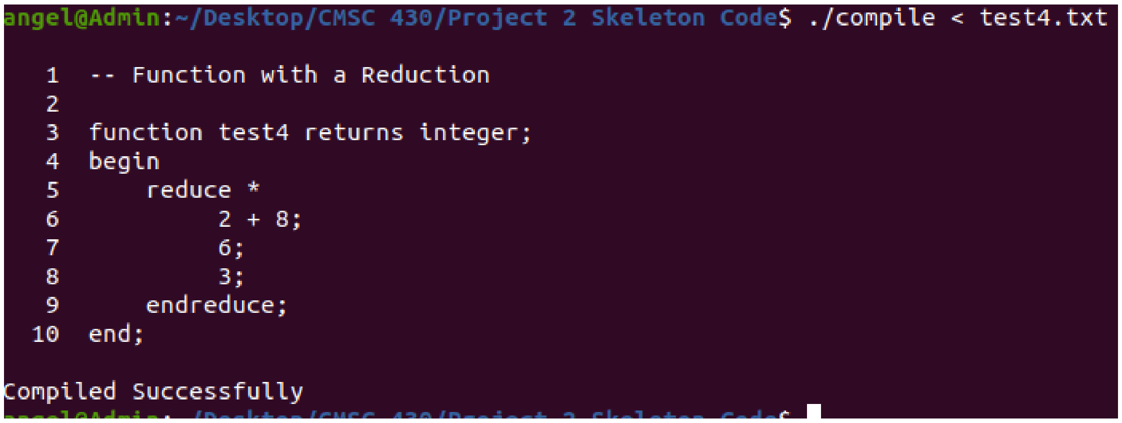
Test 4:

This tests the reduction method, and since all the syntax is correct with no lexical issues, it should compile.



```
-- Function with a Reduction

function test4 returns integer;
begin
    reduce *
        2 + 8;
        6;
        3;
    endreduce;
end;
```

Compilation:

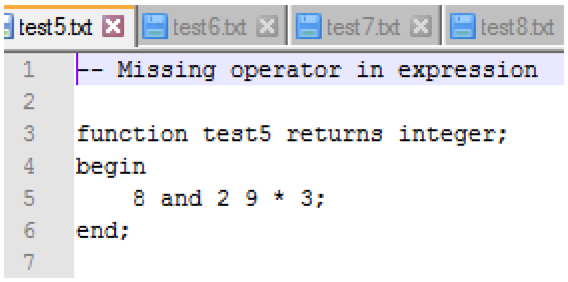


```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test4.txt

   1  -- Function with a Reduction
   2
   3  function test4 returns integer;
   4  begin
   5      reduce *
   6           2 + 8;
   7           6;
   8           3;
   9      endreduce;
  10  end;

Compiled Successfully
```

This compiled successfully as there were no issues due to no syntax errors.

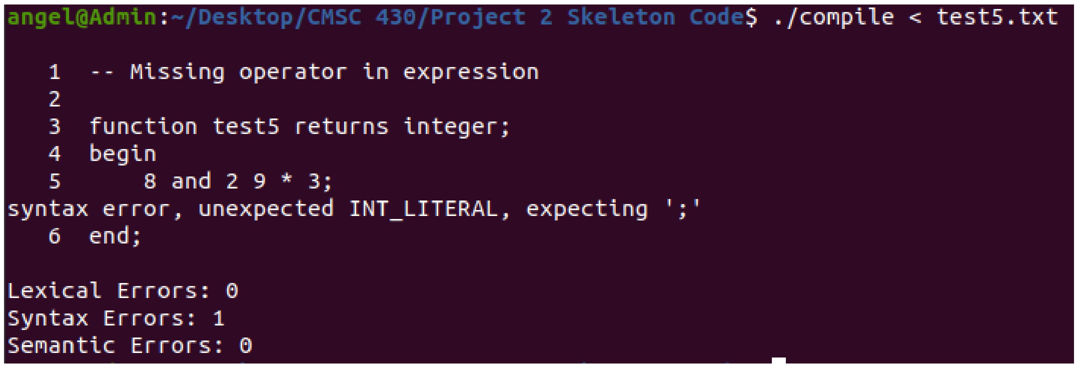
Test5:

This should catch the missing operator in the expression/statement between 2 and 9



```
test5.txt  test6.txt  test7.txt  test8.txt

-- Missing operator in expression

function test5 returns integer;
begin
    8 and 2 9 * 3;
end;
```

Compilation:

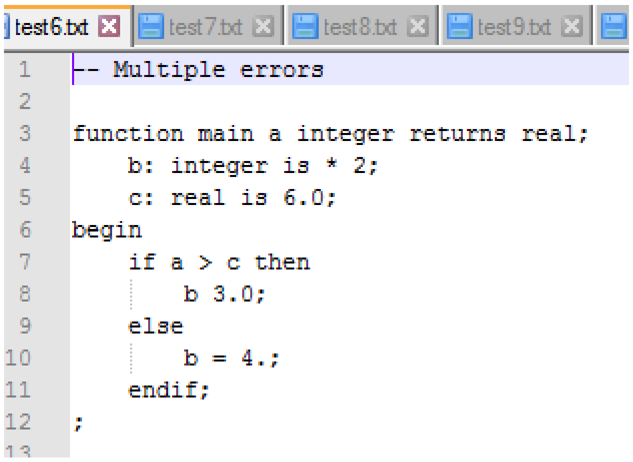


```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test5.txt

   1  -- Missing operator in expression
   2
   3  function test5 returns integer;
   4  begin
   5      8 and 2 9 * 3;
syntax error, unexpected INT_LITERAL, expecting ';'
   6  end;

Lexical Errors: 0
Syntax Errors: 1
Semantic Errors: 0
```

It caught the issue, and when it hit 9, it threw a suggestion.

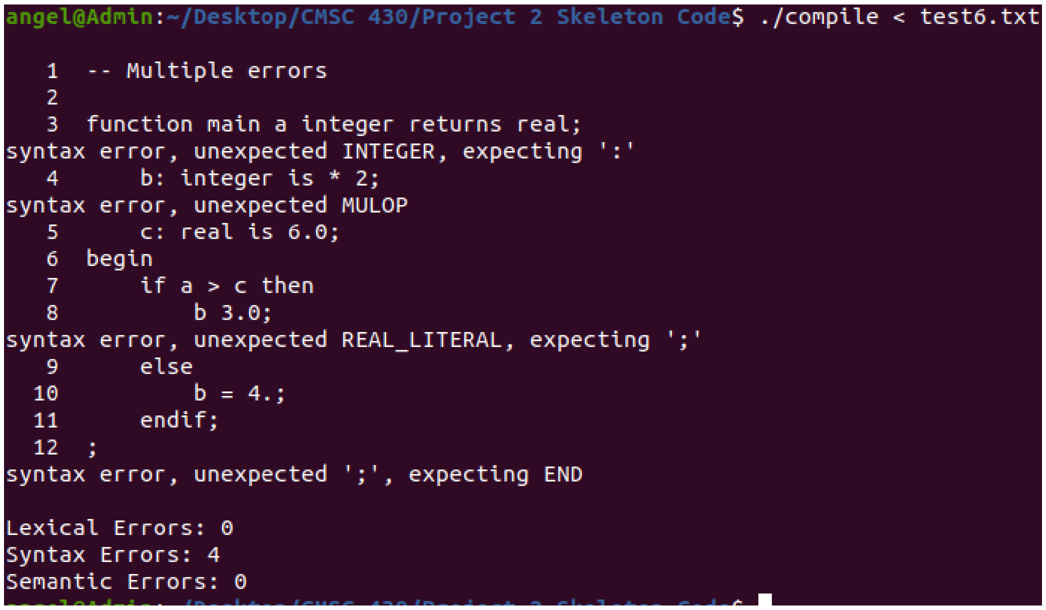
Test6:

The test case sample from the PDF, and the compilation and its errors should look like the one from PDF


test6.txt test7.txt test8.txt test9.txt

```
-- Multiple errors

function main a integer returns real;
    b: integer is * 2;
    c: real is 6.0;
begin
    if a > c then
        b 3.0;
    else
        b = 4.;
    endif;
;
```

Compilation:

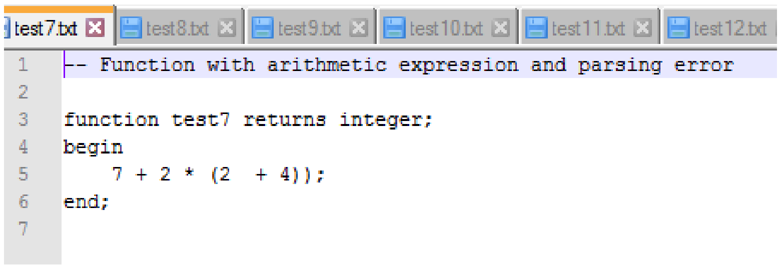


```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test6.txt

   1  -- Multiple errors
   2
   3  function main a integer returns real;
syntax error, unexpected INTEGER, expecting ':'
   4      b: integer is * 2;
syntax error, unexpected MULOP
   5      c: real is 6.0;
   6  begin
   7      if a > c then
   8          b 3.0;
syntax error, unexpected REAL_LITERAL, expecting ';'
   9      else
  10          b = 4.;
  11      endif;
  12  ;
syntax error, unexpected ';', expecting END

Lexical Errors: 0
Syntax Errors: 4
Semantic Errors: 0
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$
```

It catches all four errors with the same error messages as the PDF

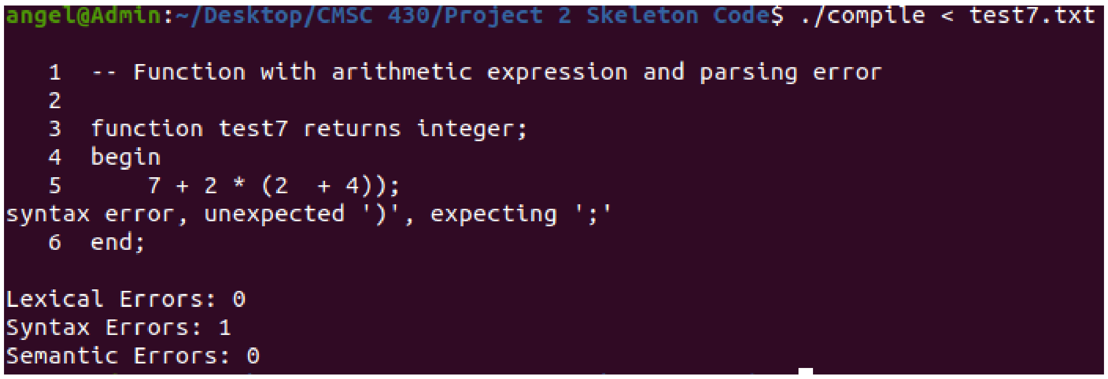
Test7:

Based off test1, sees if the error is caught in the expression or statement. The problem here is the extra ')' at the end of the statement.



```
-- Function with arithmetic expression and parsing error

function test7 returns integer;
begin
    7 + 2 * (2  + 4));
end;
```

Compilation:

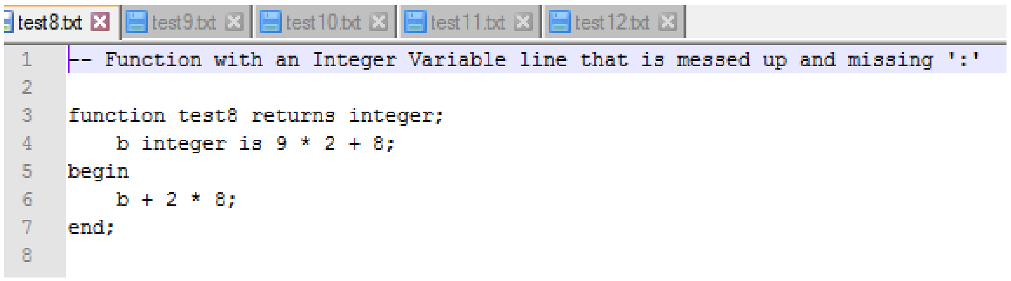


```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test7.txt

   1  -- Function with arithmetic expression and parsing error
   2
   3  function test7 returns integer;
   4  begin
   5      7 + 2 * (2  + 4));
syntax error, unexpected ')', expecting ';'
   6  end;

Lexical Errors: 0
Syntax Errors: 1
Semantic Errors: 0
```

It caught the ')' and threw a suggestion.

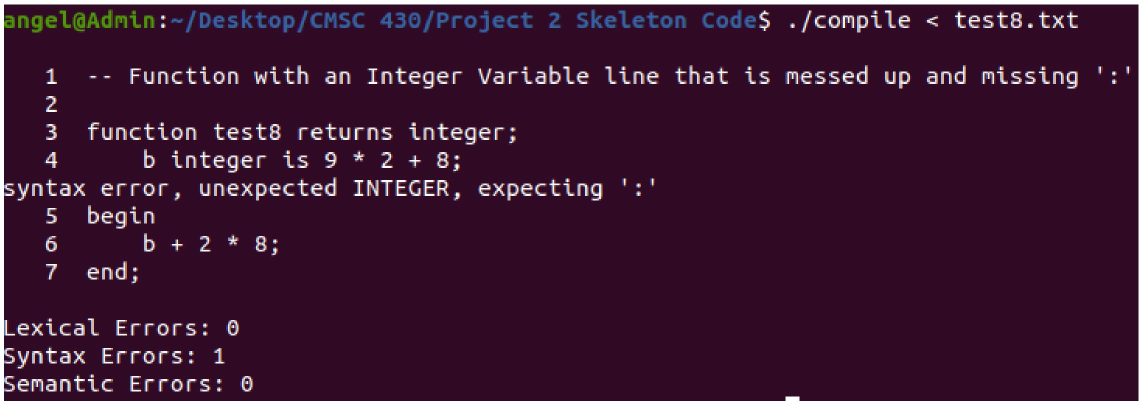
Test 8:

Based off test 2, catches the missing ‘:’



```
-- Function with an Integer Variable line that is messed up and missing ':'

function test8 returns integer;
    b integer is 9 * 2 + 8;
begin
    b + 2 * 8;
end;
```

Compilation:

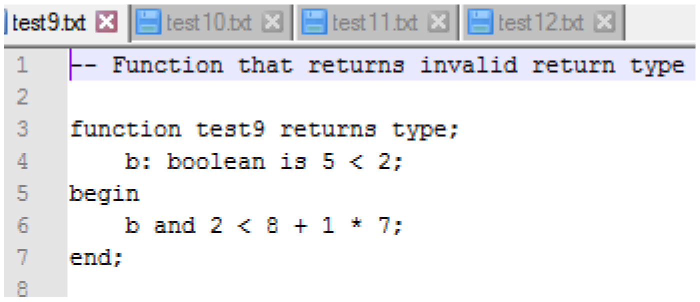


```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test8.txt

   1  -- Function with an Integer Variable line that is messed up and missing ':'
   2
   3  function test8 returns integer;
   4      b integer is 9 * 2 + 8;
syntax error, unexpected INTEGER, expecting ':'
   5  begin
   6      b + 2 * 8;
   7  end;

Lexical Errors: 0
Syntax Errors: 1
Semantic Errors: 0
```

Caught the missing ':' and gave an error when it hit the type 'integer' instead, and tossed a suggestion.

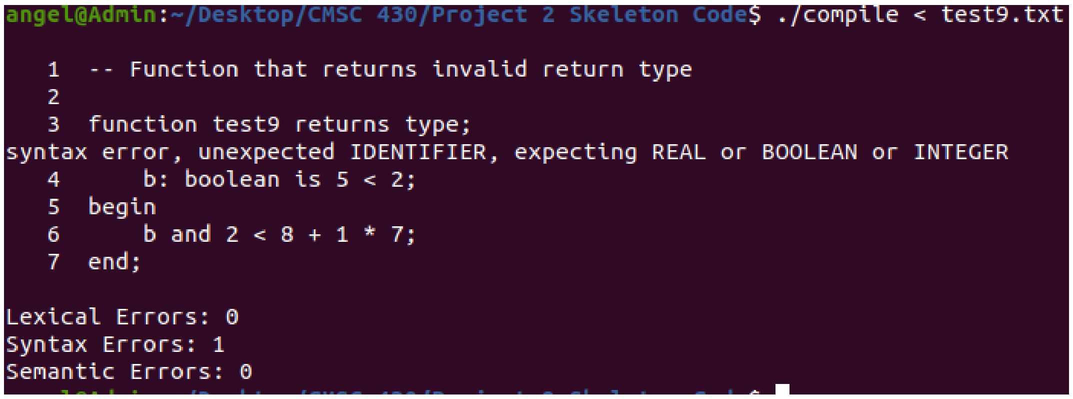
Test 9:

Based off test 3, catches invalid return type other than real, boolean, or integer.



```
-- Function that returns invalid return type

function test9 returns type;
    b: boolean is 5 < 2;
begin
    b and 2 < 8 + 1 * 7;
end;

```

Compilation:

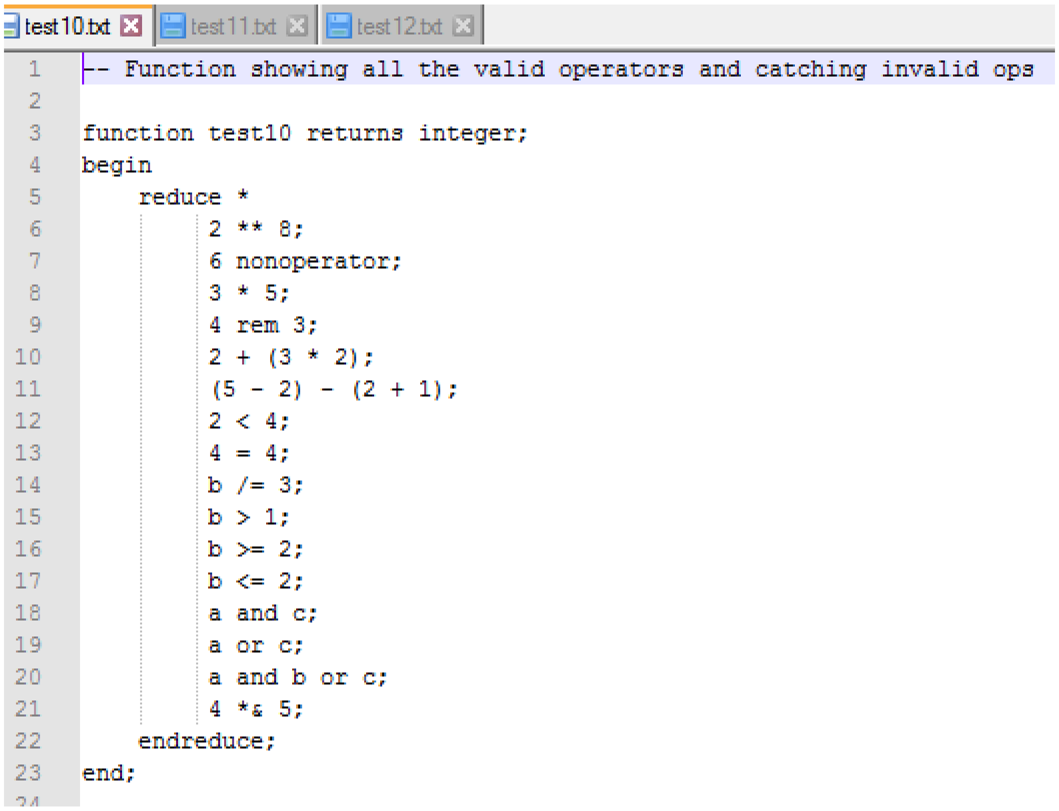


```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test9.txt

   1  -- Function that returns invalid return type
   2
   3  function test9 returns type;
syntax error, unexpected IDENTIFIER, expecting REAL or BOOLEAN or INTEGER
   4      b: boolean is 5 < 2;
   5  begin
   6      b and 2 < 8 + 1 * 7;
   7  end;

Lexical Errors: 0
Syntax Errors: 1
Semantic Errors: 0
```

Caught the invalid return type, as type should only be real, integer, or boolean.

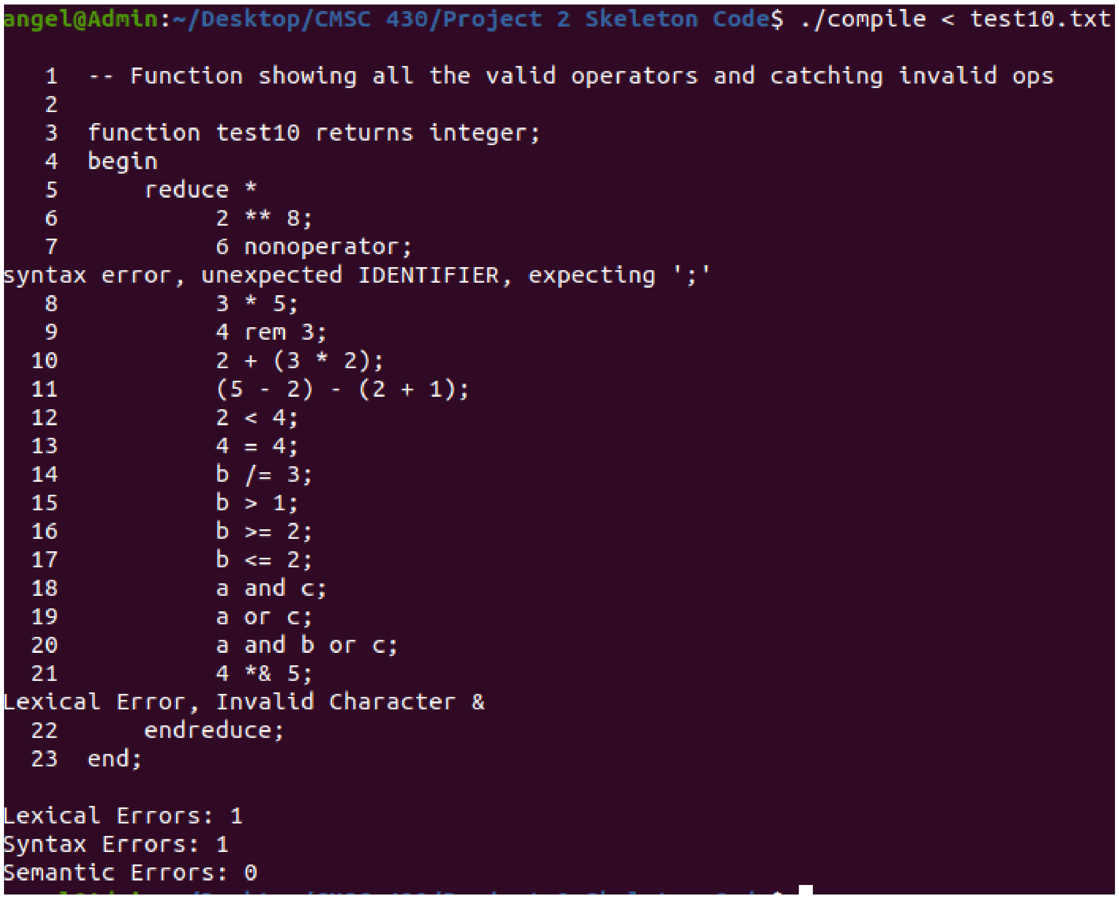
Test 10:

Based off test 4, tests all the operators (EXPOP, MULOP, REMOP, ADDOP, RELOP, ANDOP, OROP) and catches invalid operators or identifiers such as "nonoperator" on line 7 and '*&' on line 21 which is a lexical error.



```
1   -- Function showing all the valid operators and catching invalid ops
2
3   function test10 returns integer;
4   begin
5       reduce *
6           2 ** 8;
7           6 nonoperator;
8           3 * 5;
9           4 rem 3;
10          2 + (3 * 2);
11          (5 - 2) - (2 + 1);
12          2 < 4;
13          4 = 4;
14          b /= 3;
15          b > 1;
16          b >= 2;
17          b <= 2;
18          a and c;
19          a or c;
20          a and b or c;
21          4 *& 5;
22      endreduce;
23  end;
```

Compilation:

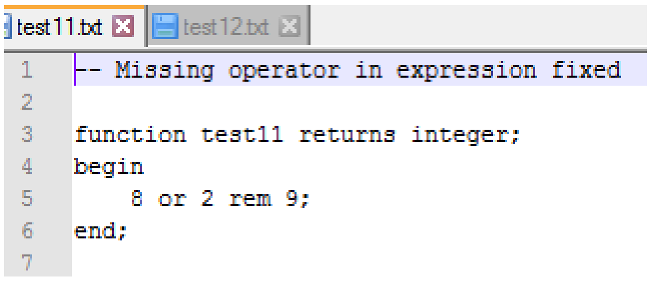


```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test10.txt

   1  -- Function showing all the valid operators and catching invalid ops
   2
   3  function test10 returns integer;
   4  begin
   5      reduce *
   6           2 ** 8;
   7           6 nonoperator;
syntax error, unexpected IDENTIFIER, expecting ';'
   8           3 * 5;
   9           4 rem 3;
  10           2 + (3 * 2);
  11           (5 - 2) - (2 + 1);
  12           2 < 4;
  13           4 = 4;
  14           b /= 3;
  15           b > 1;
  16           b >= 2;
  17           b <= 2;
  18           a and c;
  19           a or c;
  20           a and b or c;
  21           4 *& 5;
Lexical Error, Invalid Character &
  22      endreduce;
  23  end;

Lexical Errors: 1
Syntax Errors: 1
Semantic Errors: 0
```

It passed all the valid operators (including all relational operators) and caught 'nonoperator' and '*&'.

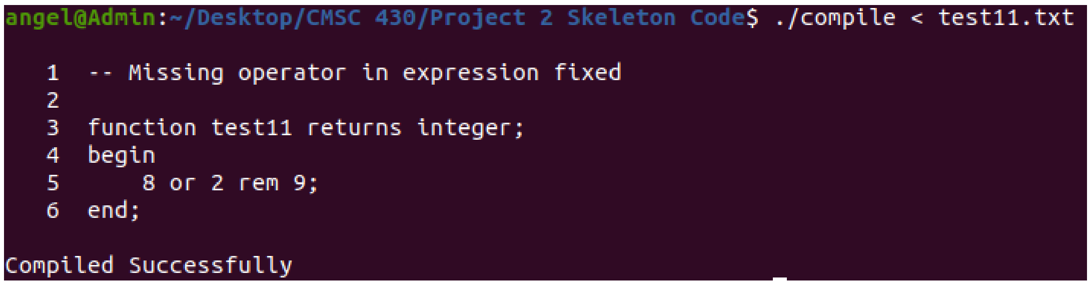
Test 11:

Based off test 5, should test if the “or” binary operator is valid, and if remainder operator “rem” is valid. They should both pass.



```
-- Missing operator in expression fixed

function test11 returns integer;
begin
    8 or 2 rem 9;
end;
```

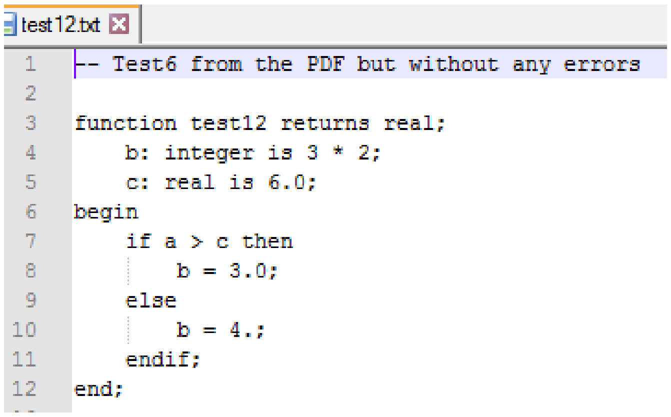
Compilation:


```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test11.txt

   1  -- Missing operator in expression fixed
   2
   3  function test11 returns integer;
   4  begin
   5      8 or 2 rem 9;
   6  end;

Compiled Successfully
```

This compiled successfully as or and rem are valid and in the right place, and there were no issues due to no syntax errors.

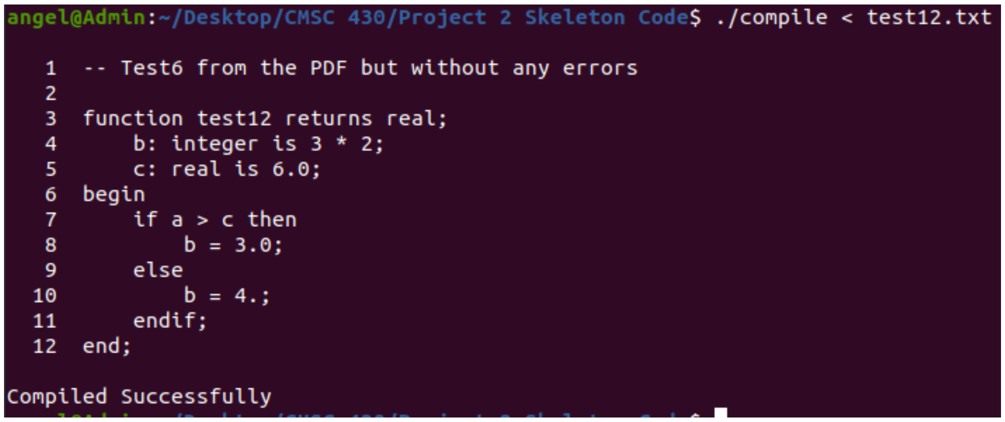
Test 12:

Like the test case from the PDF, but without the errors. It should compile with no issues.


test12.txt

```
-- Test6 from the PDF but without any errors

function test12 returns real;
    b: integer is 3 * 2;
    c: real is 6.0;
begin
    if a > c then
        b = 3.0;
    else
        b = 4.;
    endif;
end;
```

Compilation:



```
angel@Admin:~/Desktop/CMSC 430/Project 2 Skeleton Code$ ./compile < test12.txt

-- Test6 from the PDF but without any errors

function test12 returns real;
    b: integer is 3 * 2;
    c: real is 6.0;
begin
    if a > c then
        b = 3.0;
    else
        b = 4.;
    endif;
end;

Compiled Successfully
```

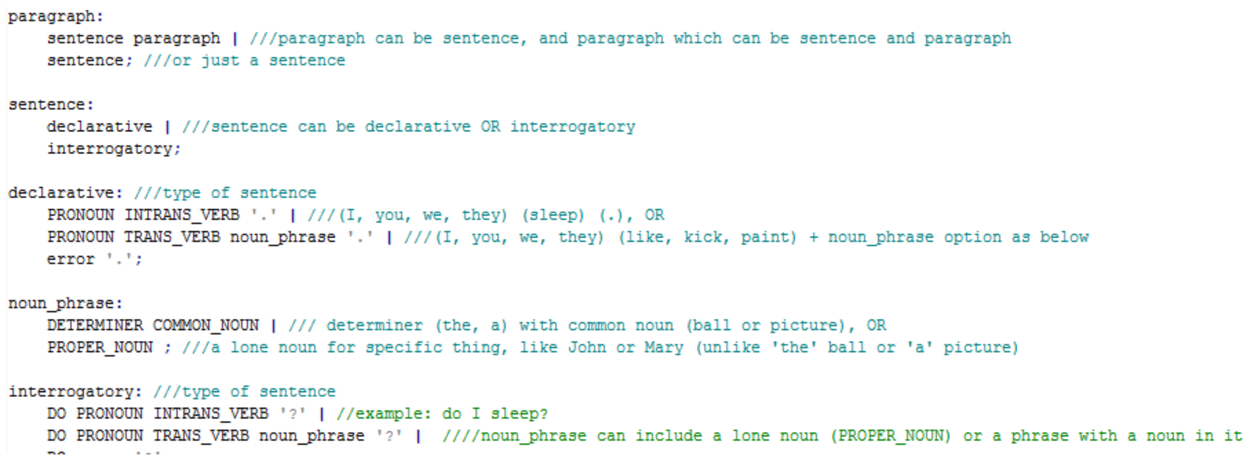
This compiled successfully as there were no issues due to no syntax errors.

Lessons Learned and any improvements that could be made

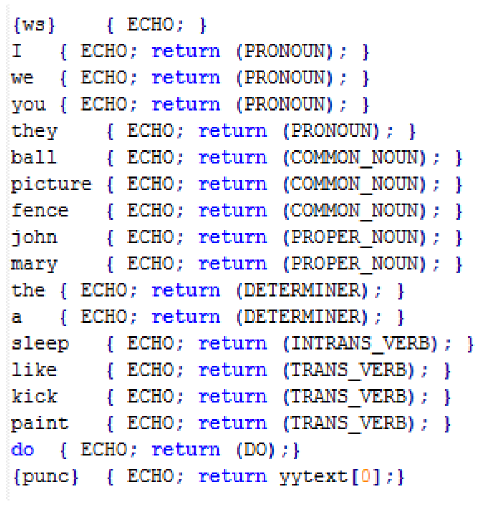
This video was helpful for this Project:
https://learn.umgc.edu/d2l/le/content/615097/viewContent/22459901/View

I wish this video was provided on first or second week because the english grammar is something everyone in this class is familiar with (since the class is done in English, at UMGC), and it made the whole "lexemes, tokens, expressions, etc" make more sense.

What helped me make sense of the parsing for Project 2 was combing through the files for the English parser and taking notes. Before that, the parsing was hard to wrap my head around.



```
paragraph:
    sentence paragraph | ///paragraph can be sentence, and paragraph which can be sentence and paragraph
    sentence; ///or just a sentence

sentence:
    declarative | ///sentence can be declarative OR interrogatory
    interrogatory;

declarative: ///type of sentence
    PRONOUN INTRANS_VERB '.' | ///(I, you, we, they) (sleep) (.), OR
    PRONOUN TRANS_VERB noun_phrase '.' | ///(I, you, we, they) (like, kick, paint) + noun_phrase option as below
    error '.';

noun_phrase:
    DETERMINER COMMON_NOUN | /// determiner (the, a) with common noun (ball or picture), OR
    PROPER_NOUN ; ///a lone noun for specific thing, like John or Mary (unlike 'the' ball or 'a' picture)

interrogatory: ///type of sentence
    DO PRONOUN INTRANS_VERB '?' | //example: do I sleep?
    DO PRONOUN TRANS_VERB noun_phrase '?' |  ////noun_phrase can include a lone noun (PROPER_NOUN) or a phrase with a noun in it
```



```
{ws}     { ECHO; }
I    { ECHO; return (PRONOUN); }
we   { ECHO; return (PRONOUN); }
you { ECHO; return (PRONOUN); }
they     { ECHO; return (PRONOUN); }
ball     { ECHO; return (COMMON_NOUN); }
picture { ECHO; return (COMMON_NOUN); }
fence    { ECHO; return (COMMON_NOUN); }
john     { ECHO; return (PROPER_NOUN); }
mary     { ECHO; return (PROPER_NOUN); }
the { ECHO; return (DETERMINER); }
a    { ECHO; return (DETERMINER); }
sleep    { ECHO; return (INTRANS_VERB); }
like     { ECHO; return (TRANS_VERB); }
kick     { ECHO; return (TRANS_VERB); }
paint    { ECHO; return (TRANS_VERB); }
do   { ECHO; return (DO);}
{punc}   { ECHO; return yytext[0];}
```