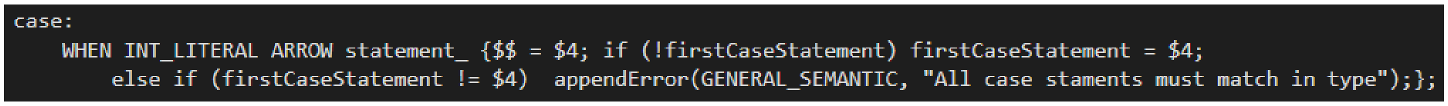
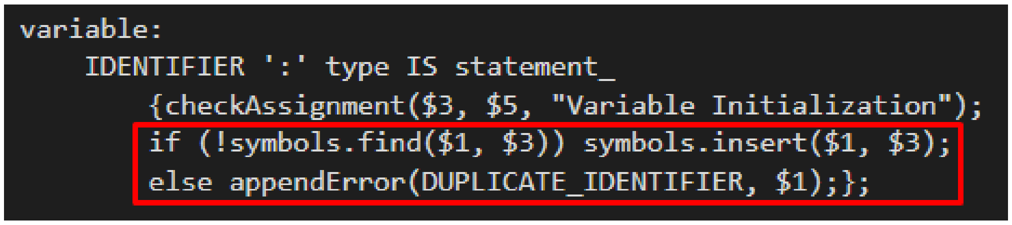
My approach to this project was to first look at types.cc to see how it worked. After going through types.cc and the new changes in parser.y I found how they worked in conjunction. In checkAssignment, I had to add a narrowing check for int to real, there is also a widening check but because that is legal there are no errors thrown. In checkArithemetic I had to add an else if statement that allows reals to have integer/real values. I did not touch checkLogical or checkRelational. To make sure the rem operator is only used with integers, I made the checkRemainder method which will return a mismatch if either left or right are not integers. The checkIfStatement method is one of the more complex ones. It checks the condition's type to make sure it is a boolean, and then checks each of the statements to make sure they also match in type. If the condition is not a boolean an error is appended, if the statement types don't match then an error is appended. Mismatch is not returned until after both checks are performed which means it can catch both errors in one go.

I broke some of the semantic checks for case statements up between parser.y and types.cc. checkCaseHead simply checks the head to see if it is an int, if it is not an error is appended. The rest of the semantic checks are performed in parser.y. It might be a bit more complicated, but I had fun trying to figure out how to make it work in there. I also further broke up the case statemen into two distinct parts, case_head and case_statement. The head was broken off so that the error appended would print on the line after "case identifier is". Next is case_statement which is a little more complicated. During the first case (if there is one), the type that each statement within the case statement will be is determined like this:

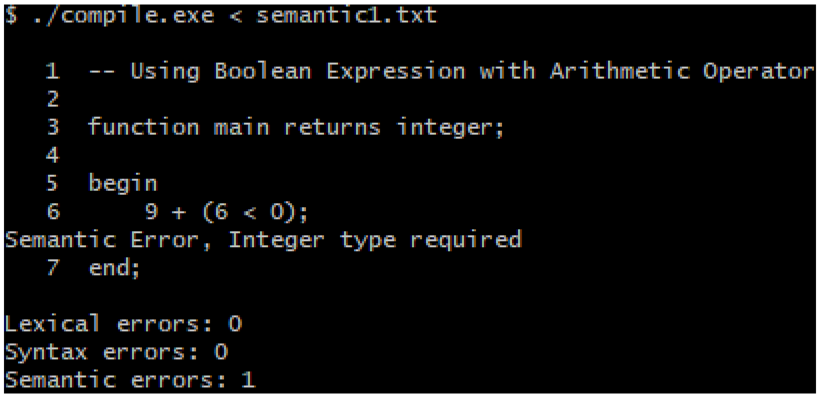

```
case:
    WHEN INT_LITERAL ARROW statement_ {$$ = $4; if (!firstCaseStatement) firstCaseStatement = $4;
        else if (firstCaseStatement != $4)  appendError(GENERAL_SEMANTIC, "All case staments must match in type");};
```

If the firstCaseStatement is null, then it will be set to the type of the current statement. If it is not empty (any statements after the first) then it will check to make sure that the type matches. The exact same type check is performed for the "others =>" portion as well.

All duplicate variable/parameters checks are also performed in the variable/parameter rather than outsourcing it. In other words, the variable/parameters checks assignment, if its IDENTIFIER is not in symbols then it is added to it, otherwise a DUPLICATE_IDENTIFIER is appended:


```
variable:
    IDENTIFIER ':' type IS statement_
        {checkAssignment($3, $5, "Variable Initialization");
        if (!symbols.find($1, $3)) symbols.insert($1, $3);
        else appendError(DUPLICATE_IDENTIFIER, $1);};
```

Throughout this course, I learned a lot about how compilers/lexers think and work. While I was quite intimidated at first and a bit overwhelmed by my unfamiliarity of bison/flex and Linux in general, I enjoyed the different types of troubleshooting involved over each of the four projects. This project felt the easiest because of how much easier it was to understand everything since I had already worked on three projects. Test cases in the form of screenshots begin below:

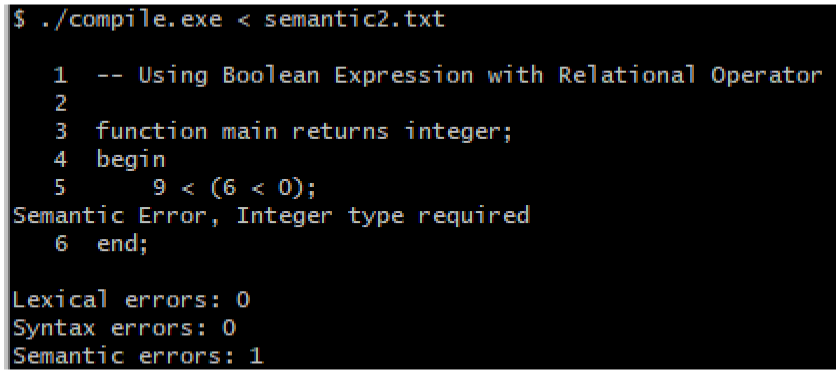


```
$ ./compile.exe < semantic1.txt

   1  -- Using Boolean Expression with Arithmetic Operator
   2
   3  function main returns integer;
   4
   5  begin
   6      9 + (6 < 0);
Semantic Error, Integer type required
   7  end;

Lexical errors: 0
Syntax errors: 0
Semantic errors: 1
```

Figure 1. Test case #1. Boolean expression with arithmetic operator.

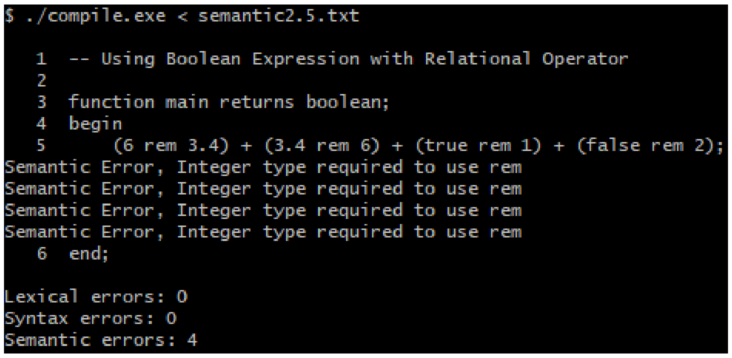


```
$ ./compile.exe < semantic2.txt

   1  -- Using Boolean Expression with Relational Operator
   2
   3  function main returns integer;
   4  begin
   5      9 < (6 < 0);
Semantic Error, Integer type required
   6  end;

Lexical errors: 0
Syntax errors: 0
Semantic errors: 1
```

Figure 2. Test case #2. Boolean expression with relational operator.

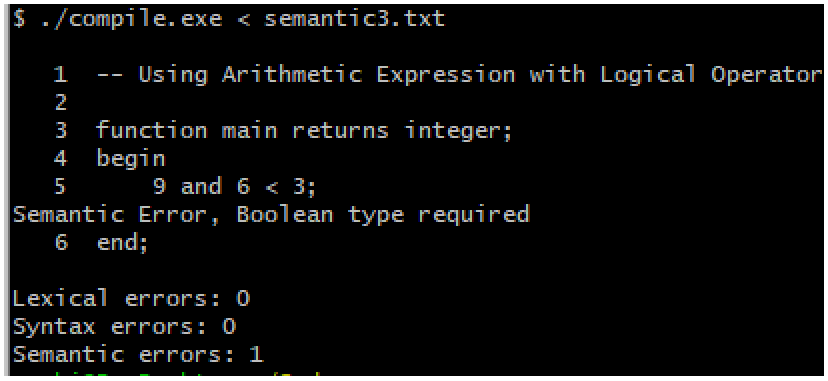


```
$ ./compile.exe < semantic2.5.txt

   1  -- Using Boolean Expression with Relational Operator
   2
   3  function main returns boolean;
   4  begin
   5      (6 rem 3.4) + (3.4 rem 6) + (true rem 1) + (false rem 2);
Semantic Error, Integer type required to use rem
Semantic Error, Integer type required to use rem
Semantic Error, Integer type required to use rem
Semantic Error, Integer type required to use rem
   6  end;

Lexical errors: 0
Syntax errors: 0
Semantic errors: 4
```

Figure 2.5. Test case #2.5. Rem operator being used without integer.

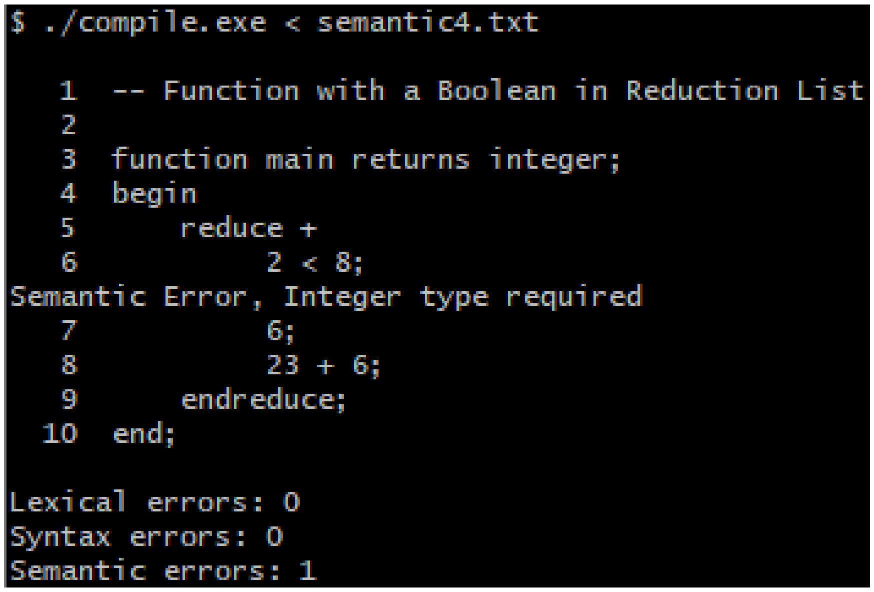


```
$ ./compile.exe < semantic3.txt

   1  -- Using Arithmetic Expression with Logical Operator
   2
   3  function main returns integer;
   4  begin
   5      9 and 6 < 3;
Semantic Error, Boolean type required
   6  end;

Lexical errors: 0
Syntax errors: 0
Semantic errors: 1
```

Figure 3. Test case #3. Boolean expression with logical operator.

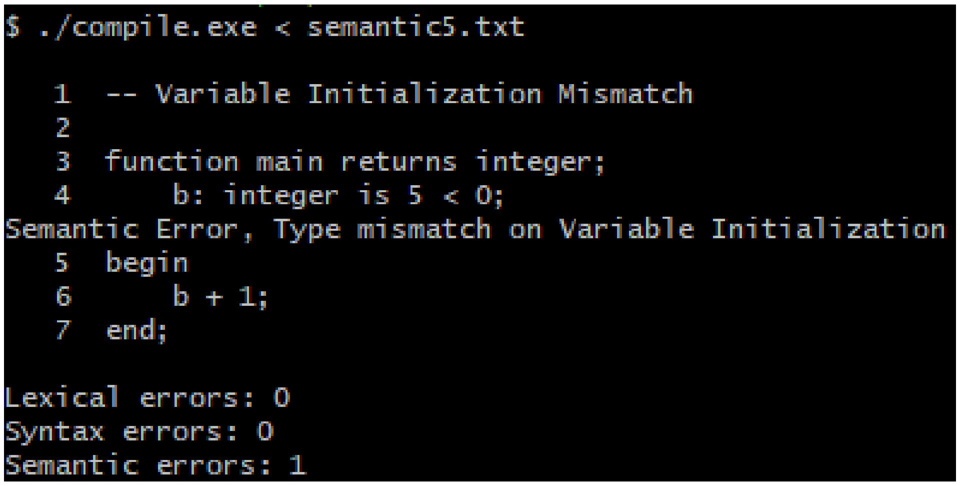


```
$ ./compile.exe < semantic4.txt

   1  -- Function with a Boolean in Reduction List
   2
   3  function main returns integer;
   4  begin
   5      reduce +
   6          2 < 8;
Semantic Error, Integer type required
   7          6;
   8          23 + 6;
   9      endreduce;
  10  end;

Lexical errors: 0
Syntax errors: 0
Semantic errors: 1
```

Figure 4. Test case #4. Boolean expression in reduction list.

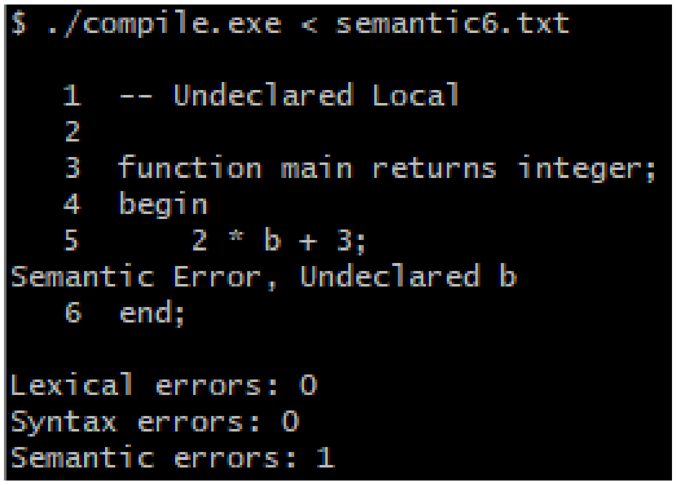


```
$ ./compile.exe < semantic5.txt

   1  -- Variable Initialization Mismatch
   2
   3  function main returns integer;
   4      b: integer is 5 < 0;
Semantic Error, Type mismatch on Variable Initialization
   5  begin
   6      b + 1;
   7  end;

Lexical errors: 0
Syntax errors: 0
Semantic errors: 1
```

Figure 5. Test case #5. Variable initialization mismatch.

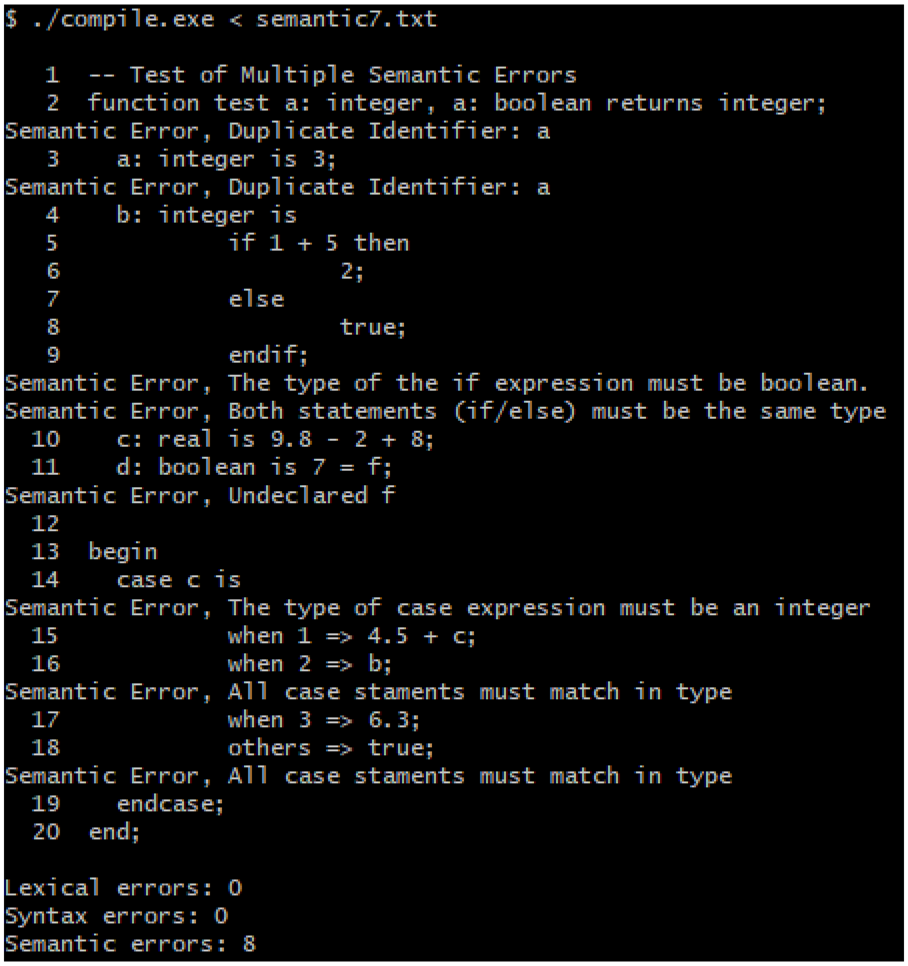


```
$ ./compile.exe < semantic6.txt

   1  -- Undeclared Local
   2
   3  function main returns integer;
   4  begin
   5      2 * b + 3;
Semantic Error, Undeclared b
   6  end;

Lexical errors: 0
Syntax errors: 0
Semantic errors: 1
```

Figure 6. Test case #6. Undeclared local variable.



```
$ ./compile.exe < semantic7.txt

   1  -- Test of Multiple Semantic Errors
   2  function test a: integer, a: boolean returns integer;
Semantic Error, Duplicate Identifier: a
   3    a: integer is 3;
Semantic Error, Duplicate Identifier: a
   4    b: integer is
   5            if 1 + 5 then
   6                    2;
   7            else
   8                    true;
   9            endif;
Semantic Error, The type of the if expression must be boolean.
Semantic Error, Both statements (if/else) must be the same type
  10    c: real is 9.8 - 2 + 8;
  11    d: boolean is 7 = f;
Semantic Error, Undeclared f
  12
  13  begin
  14    case c is
Semantic Error, The type of case expression must be an integer
  15            when 1 => 4.5 + c;
  16            when 2 => b;
Semantic Error, All case staments must match in type
  17            when 3 => 6.3;
  18            others => true;
Semantic Error, All case staments must match in type
  19    endcase;
  20  end;

Lexical errors: 0
Syntax errors: 0
Semantic errors: 8
```

Figure 7. Test case #7. Heavily modified version of program in handout.

Line 2 error: a was declared twice as a parameter.

Line 3 error: a was declared as a variable after already being declared as a parameter.

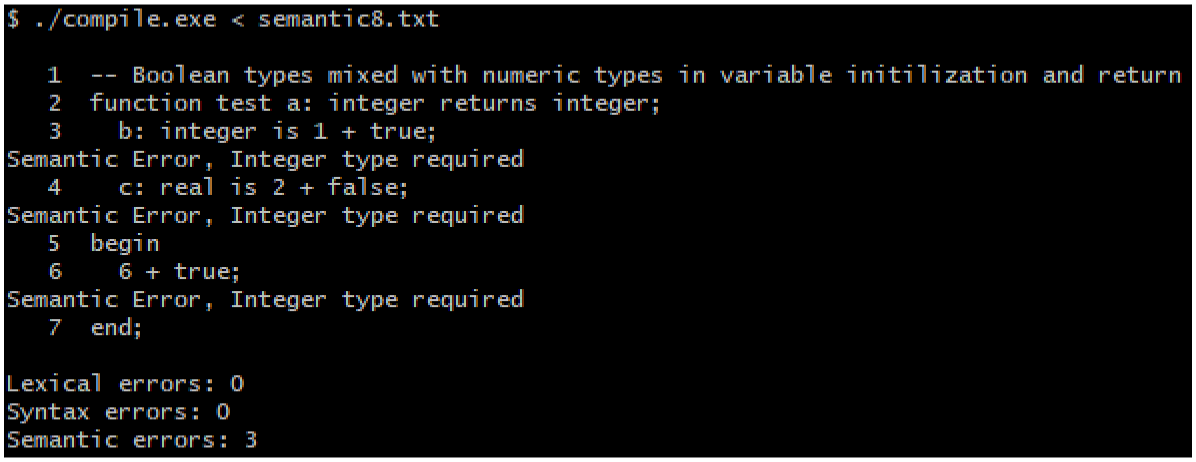
If statement error: Condition not boolean and statements don’t match in type.

Line 11 error: Self-explanatory, undeclared variable.

Line 14 error: Head of case statement must be an integer.

Line 16 error: Real was declared as the return type for the case statement in the previous line, but b is an int.

Line 18 error: Same error as line 16, but boolean rather than int.

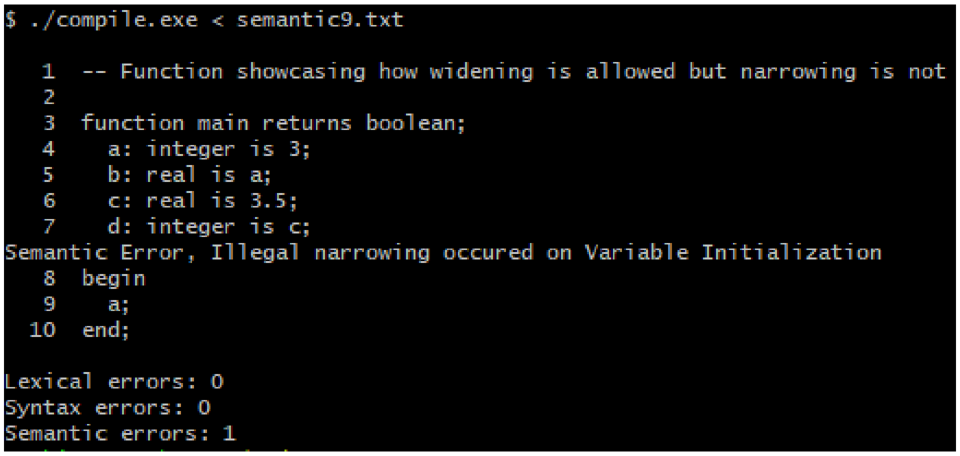


```
$ ./compile.exe < semantic8.txt

   1  -- Boolean types mixed with numeric types in variable initilization and return
   2  function test a: integer returns integer;
   3    b: integer is 1 + true;
Semantic Error, Integer type required
   4    c: real is 2 + false;
Semantic Error, Integer type required
   5  begin
   6    6 + true;
Semantic Error, Integer type required
   7  end;

Lexical errors: 0
Syntax errors: 0
Semantic errors: 3
```

Figure 8. Test case #8. Boolean types mixed with numeric types in variable initialization and return.

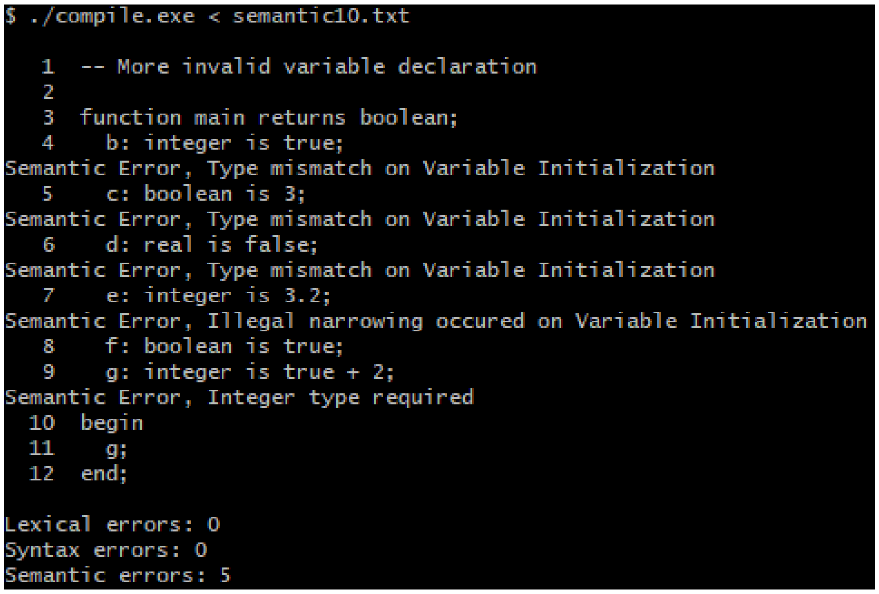


```
$ ./compile.exe < semantic9.txt

   1  -- Function showcasing how widening is allowed but narrowing is not
   2
   3  function main returns boolean;
   4    a: integer is 3;
   5    b: real is a;
   6    c: real is 3.5;
   7    d: integer is c;
Semantic Error, Illegal narrowing occured on Variable Initialization
   8  begin
   9    a;
  10  end;

Lexical errors: 0
Syntax errors: 0
Semantic errors: 1
```

Figure 9. Test case #9. Showcasing how widening is allowed but narrowing is not.

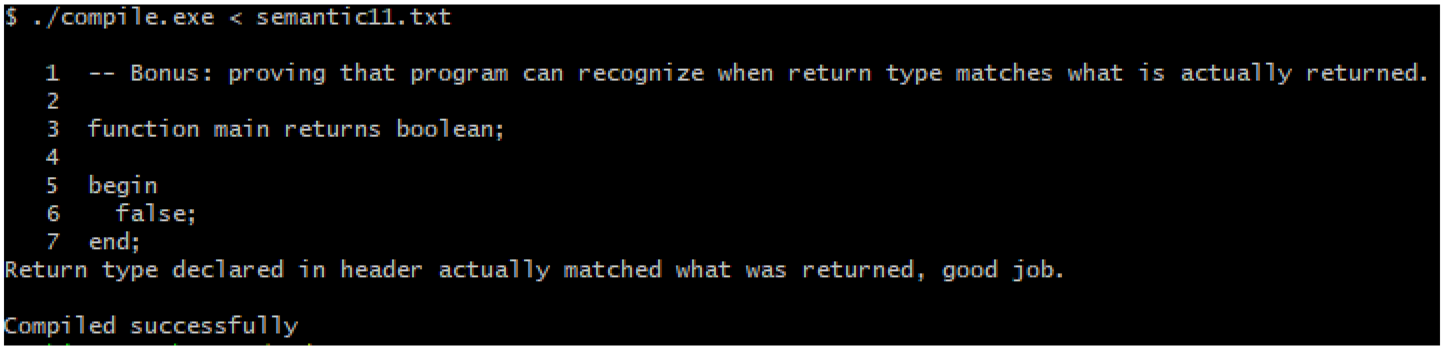


```
$ ./compile.exe < semantic10.txt

   1  -- More invalid variable declaration
   2
   3  function main returns boolean;
   4    b: integer is true;
Semantic Error, Type mismatch on Variable Initialization
   5    c: boolean is 3;
Semantic Error, Type mismatch on Variable Initialization
   6    d: real is false;
Semantic Error, Type mismatch on Variable Initialization
   7    e: integer is 3.2;
Semantic Error, Illegal narrowing occured on Variable Initialization
   8    f: boolean is true;
   9    g: integer is true + 2;
Semantic Error, Integer type required
  10  begin
  11    g;
  12  end;

Lexical errors: 0
Syntax errors: 0
Semantic errors: 5
```

Figure 10. Test case #10. Showcasing various invalid variable declaration.



```
$ ./compile.exe < semantic11.txt

   1  -- Bonus: proving that program can recognize when return type matches what is actually returned.
   2
   3  function main returns boolean;
   4
   5  begin
   6    false;
   7  end;
Return type declared in header actually matched what was returned, good job.

Compiled successfully
```

Figure 11. Test case #11. Logic to detect if return type matches what is returned.

For fun, I had implemented a logic check that would see if what was returned in the body matches what the function header said would be returned. It’s quite simple, there is a Types “returnType” that is initialized in the function header. Then, once the body is finished being parsed it checks to see if what is being returned is a match. However, I realized that most of the programs would append a return error (given that most are mismatches), so I changed the “if (returnType != $2)” to “if (returnType == $2)”. By doing this, it will only notify the user upon a successful match rather than adding an extra error to each program.