Project 3

Approach to the project:

First step was to transfer the files from the previous projects to this one. Some corrections and additions were made to the scanner.l file because the result for true and false values was interpreted inverted for some reason (true = 0 and false = 1), so I changed the rule from:

```
28 bool          (true|false)
81 {bool}        { ECHO; return(BOOL_LITERAL); }
```

to

```
54 true          {ECHO;   yylval.value= true; return (BOOL_LITERAL);}
55 false         {ECHO;   yylval.value = false; return (BOOL_LITERAL);}
```

and now it works without problems (true = 1, false = 0)

Next, I added the code for each statement and added them to their respective production in bison.

Next, I added all the enum types that were necessary for the arithmetic and logic operations in values.h as well as the function declarations for the functions that were used in values.cc to calculate various results.

Finally, for the argument input I used a queue since I knew how it works from project 2 and I thought it is a very good and easy alternative from creating a static array. Also, because the command line arguments are strings we need to convert them to integers in order to do the arithmetic and logic operations so i used the atoi function.

For the program to choose between '*cases*' and '*other*', firstly I created three global variables needed for the following functions:

caseIdentifier() → save the value from the identifier token in a global variable for further use

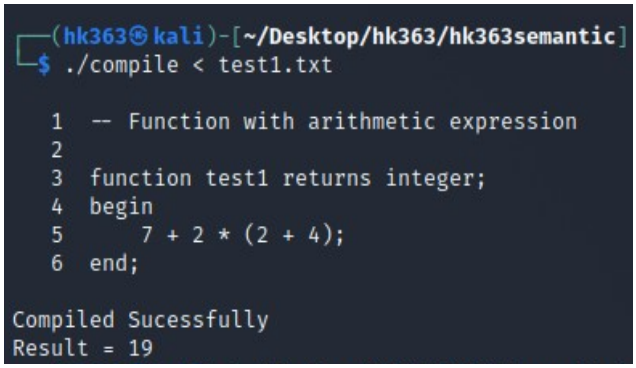
evaluateCase() → compares the literal with the value of the identifier. If the comparison is true returns the statement as a result.

mergeCases() → merges all cases that are not others into case_ (followed as an example by evaluate reduction)

chooseCaseorOther () → choose between the result of mergeCases() or other result.

Running the Tests:

Test 1:

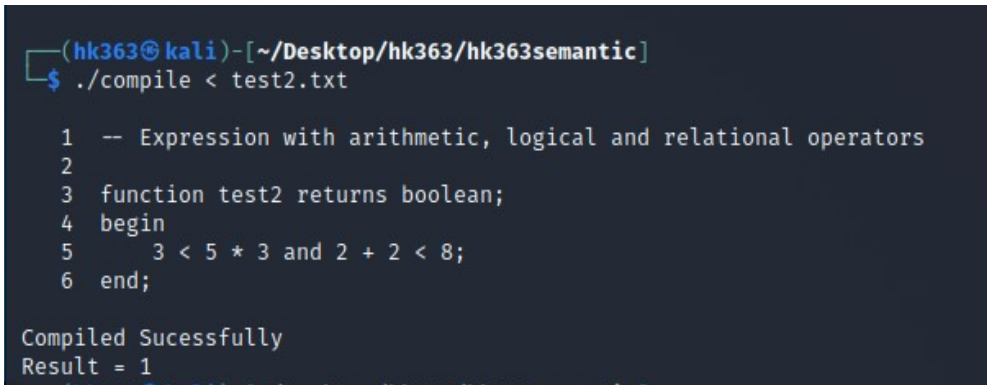


Compiled successfully, as it should.

The interpreted result is correct, due to precedence of arithmetic operations 7 + 2 * 6 = 7 + 12 = 19.

Result is correct

Test 2:



Compiled successfully, as it should.

The interpreted result is correct because:

- 3 < 15 (true)
- 4 < 8 (true)

true and true = true (true is interpreted as 1)

Result is correct

Test 3:

```
┌──(hk363㉿kali)-[~/Desktop/hk363/hk363semantic]
└─$ ./compile <  test3.txt

   1  -- Function with a Variable
   2
   3  function test3 returns integer;
   4      b: integer is 5 + 1 * 4;
   5  begin
   6      b + 8;
   7  end;

Compiled Sucessfully
Result = 17
```

Compiled successfully, as it should.

The interpreted result is correct because initially, b gets declared as 5 + 4 = 9, then b + 8 = 9 + 8 = 17.

Result is correct

Test 4:

```
┌──(hk363㉿kali)-[~/Desktop/hk363/hk363semantic]
└─$ ./compile < test4.txt

   1  -- Function with Nested Reductions
   2
   3  function test4 returns integer;
   4  begin
   5      reduce +
   6          2 * 8;
   7          reduce *
   8              3 + 4;
   9              2;
  10          endreduce;
  11          6;
  12          23 + 6;
  13      endreduce;
  14  end;

Compiled Sucessfully
Result = 65
```

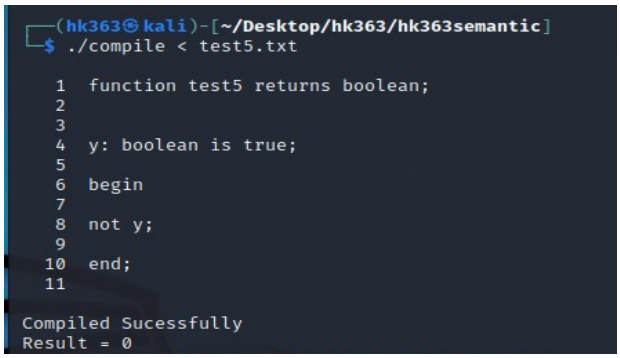
Compiled successfully, as it should.

The interpreted result is correct. First reduction is (3+4) * 2 = 14 and the second reduction is (2 * 8) + 14 + 6 + (23 + 6) = 65

Result is correct

Test 5:

This test was created to see if true, false values are interpreted correctly as well as the '**not**' boolean operator.



Compiled successfully, as it should.

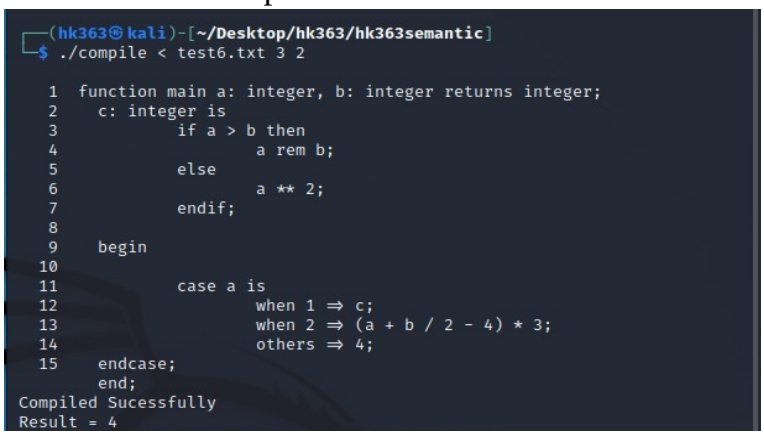
The result = 0 is correct because initially we set y to be true (1) and after we get to '*not y'*, it should become false (0).

Result is correct

Test 6:

We will split test 6 to subcases, depending on what input we will give as command line arguments. We will test all possible branches at the if and case statements.

- "a > b" with input 3 2:



The program will execute as follows:

Firstly, '**if (a > b)'** is true because 3 > 2 so '**c'** will be equal to **'a rem b'** which is 3 rem 2 = 1. Finally, the case which matches is '**case a is**' → '**others => 4**'. So the result will be **result = 4**.

Result is correct.

- “a < b” with input 1 2

```
┌──(hk363㉿kali)-[~/Desktop/hk363/hk363semantic]
└─$ ./compile < test6.txt 1 2

   1  function main a: integer, b: integer returns integer;
   2    c: integer is
   3            if a > b then
   4                    a rem b;
   5            else
   6                    a ** 2;
   7            endif;
   8
   9    begin
  10
  11            case a is
  12                    when 1 ⇒ c;
  13                    when 2 ⇒ (a + b / 2 - 4) * 3;
  14                    others ⇒ 4;
  15    endcase;
        end;
Compiled Sucessfully
Result = 1
```

The program will execute as follows:

Firstly, **‘if (a > b)’** is false because (1 < 2) so ‘**c**’ will be equal to ‘**a ** 2**’ which is $1^2 = 1$. Finally, the case which matches is ‘**case a is**’ → ‘**when 1 => c**’, So the result will be ‘**c**’ which has the value 1, so Result = 1.

Result is correct.

- Check “case a → when 2 => (a + b / 2 – 4) *3”, with input 2 4.

```
┌──(hk363㉿kali)-[~/Desktop/hk363/hk363semantic]
└─$ ./compile < test6.txt 2 4

   1  function main a: integer, b: integer returns integer;
   2    c: integer is
   3            if a > b then
   4                    a rem b;
   5            else
   6                    a ** 2;
   7            endif;
   8
   9    begin
  10
  11            case a is
  12                    when 1 ⇒ c;
  13                    when 2 ⇒ (a + b / 2 - 4) * 3;
  14                    others ⇒ 4;
  15    endcase;
        end;
Compiled Sucessfully
Result = 0
```

The program will execute as follows:

Firstly, **‘if (a > b)’** is false because (2 < 4) so ‘**c**’ will be equal to ‘**a ** 2**’ which is $2^2 = 4$. Finally, the case which matches is ‘**case a is**’ → ‘**when 2 => (a + b / 2 – 4) * 3**’, So the result will be (2 + 4/2 – 4) * 3 = (2 + 2 -4) * 3 = (4 – 4) * 3 = 0.

Result is correct.

Conclusion:

This phase of the compiler, the interpreter, was the hardest among first two. The actions which compute the semantic values of our language in many cases were not straightforward to write, although **[1]** provided a great guide with many examples.

The hardest rt of all was trying to write the actions and the functions for the '*case' statements*.

Also, I didn't create a test showcasing semantic errors, because the project didn't require to include semantic error recognition.

A next stage could include semantic error detection which is hard because although syntactically right, the results produced by the program could be wrong due to human error. One of the most common types of semantic error is the detection of uninitialized variables.

References:

[1] https://www.gnu.org/software/bison/manual/html_node/Actions.html